Support for an integrated approach to program understanding: an application of software visualisation

Chan, Pui Shan

Support for an Integrated Approach to Program Understanding: An Application of Software Visualisation

Pui Shan Chan

Ph.D. Thesis

Centre for Software Maintenance
Department of Computer Science
University of Durham

May 1998

Abstract

Program Comprehension is a key factor in providing effective software maintenance and enabling successful evolution of software systems. The objective of this research is to provide a framework and mechanism to facilitate the understanding of large software systems.

There exist a number of theories and models of Program Comprehension where each favours a different approach to comprehension. It is evident that there is no real consensus on how maintainers understand software systems. The disparities in the comprehension strategies are largely dependent on the personal and circumstantial factors. Factors such as the level of technical competence of the maintainers, the size and complexity of the piece of software, and the types and goals of the maintenance activities can influence the process of comprehension.

This research proposes an alternative approach to Program Comprehension. It acknowledges that the process of comprehension is opportunistic, and that the current comprehension theories are inadequate in addressing this. There is a need for a more flexible approach towards comprehension, and the Integrated Approach proposed in this thesis provides a way for the utilisation of the various comprehension theories under a single environment. It recognises that any one of the comprehension theories may become active during comprehension. Under the Integrated Approach, maintainers have the option of selecting and executing the various comprehension strategies as they see fit.

The Integrated Approach to comprehension is based on a matrix of Program Relationships between Program Elements of a programming language. In this thesis, these Program Relationships are derived for the C programming language constructs.

This work also involves the investigation of the roles of both textual and graphical representations during the comprehension process. Both representations are commonly used to alleviate the problem of information overloading when maintainers trying to understand and maintain a software system.

The Integrated Approach is realised in a tool named PUI (*P*rogram *U*nderstanding *I*mplements) which provides an environment enabling the utilisation of various comprehension theories.

Acknowledgements

The author would like to acknowledge the University of Durham for the award of a research studentship. Special thanks are due to my supervisor Mr. Malcolm Munro for all his help and guidance throughout the course of this research. Many thanks must also go to my colleagues at the Centre for Software Maintenance and especially to Miss Elizabeth Burd for her help in the writing of this thesis. Finally, I would like to thank my parents and Eng Tiong for their continuous support and encouragement especially during the past three years.

Copyright

Declaration

No part of the material provided has previously been submitted by the author for a higher degree in the University of Durham or in any other University. All the work presented here is the sole work of the author and no-one else.

Table of Contents

List of Figures

List of Tables

Chapter One

Introduction

1.1 Software Engineering

1.1.1 The Software Crisis

The term Software Engineering was first introduced in the late 1960s to address the Software Crisis. Thirty years on, the Software Crisis still has not been resolved [Pres92, Somm96, Vlie93]. Programming techniques have lagged behind the developments in software both in size and complexity. Traditional techniques such as programming languages, tools and methods are primarily developed to support programming-in-the-small. Transferring these techniques directly to the development of large programs therefore proved unfruitful.

The use of computers has now become an integral part of our lives. People are becoming more dependent on channels of communication, more reliant on the vast traffic in the invisible data and more connected to the computers that manage it. The following examples illustrate the scale of some software development projects:

- the Dutch KLM airline reservation system contains two million line of (assembler) code [Vlie93]
- the UNIX operating system comprises over 3 700 000 lines of source code (System V release 4.0, including Xnews and the X11 window system) [Vlie93]
- the NASA Space Shuttle software counts 40 million lines of object code (this is 30 times as much as the software for the Saturn V projects from the 1960s) [Boeh81]
- the IBM OS360 operating system look 5000 man years of the development effort [Broo75]

1.1.2 The Software Process Model

The evolution of the Software Process Model [Royc70] was one of the results after the identification of the Software Crisis. The process model (the Waterfall model) reflected the view that software development should be perceived as an engineering discipline. This was warmly welcomed by software project management as it offered a means of making the development process more visible and manageable.

There are a distinguishable number of phases in the Waterfall model, namely Requirement analysis, Design, Implementation, Testing and Maintenance. Each phase can be divided into a number of different activities [Somm96]:

- *Requirements analysis and definition*: The system's functionalities, constraints and goals are established by consultation with the system users. They are defined in a manner which is understandable by both users and the development staff.

- *System and software design*: The system design process partitions the requirements to either hardware or software systems, and it also establishes an overall system architecture. Software design involves representing the software system functions so that they may be transformed into one or more executable programs.

- *Implementation and unit testing*: During this stage, the software design is realised as a set of programs. Unit testing involves verifying that each program meets its specification.

- *Integration and system testing*: The individual programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

- *Operation and maintenance*: Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

This is a general model rather than a detailed process model. A number of different general models or paradigms of software development can be derived from this such as the Prototyping model [Fair85] and the Spiral model [Boeh86, Boeh88]. The Waterfall model puts the emphasis on the importance of the careful analysis and planning before any major decision is committed, and thus avoid wasting the extraneous effort to re-develop a system. Management generally found this model useful for planning

and reporting. However, for a given project these activities are not necessarily separated as strictly as indicated above. Iterations and overlapping of activities may arise.

1.1.3 Definition

The definition of Software Engineering is given in [ANSI83]:

> *Software Engineering is the systematic approach to the development, operation, maintenance, and retirement of software.*

Software Engineering is concerned with systems developed by teams that collaborate over periods spanning from months to years. It also encompasses both technical and non-technical (managerial) issues. Sommerville [Somm96] points out that software is not just a collection of computer programs. It includes the documentation necessary to install, use, develop and maintain these programs. For large software systems, the effort required to write this documentation is sometimes as great as developing the systems themselves.

1.2 Software Maintenance

Studies have shown that organisations spend on average over half of their resources on software maintenance activities [Alkh92, Dekl92, Lien80, Lien81]. Indeed, it is impossible to build software systems which do not require some kind of maintenance effort. Over the lifetime of a system, its original requirements will be modified to reflect changing needs and enhancements requested by users, the system's environment may change and errors, undiscovered during system validation, may emerge [Schn87].

Both the following definitions for Software Maintenance:

> *Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the project to a changed environment.* [ANSI83]

> *Software Maintenance is the set of activities (both technical and managerial) necessary to ensure that software continues to meet organisational needs.* [CSM]

sum up the importance of careful planning and well-organised management to Software Maintenance.

Maintenance activities can be broken down into four main categories [Lien78, Lien80]:

- **Perfective maintenance** involves improving the functions of the software by responding to user defined changes.
- **Corrective maintenance** involves the correction of processing, performance or implementation failures. It includes activities such as bug fixing and correction of software errors.
- **Adaptive maintenance** involves modifying the software in order to keep up with environmental changes. It may involve changes in hardware or data.
- **Preventive maintenance** involves updating software in order to forestall future problems and to increase maintainability.

Whether it is for corrective, adaptive, perfective or preventive maintenance, the key to all of these activities is Program Comprehension [Mayr95, Oman90a].

1.3 Program Comprehension

Understanding how a program or an application is constructed and its underlying intent is essential to the task of enhancing and/or the maintaining of the program. Research has shown that maintainers spend a considerable amount of time studying programs, especially when engaged in maintenance activities. This figure can be as high as three-and-a-half times as long as they studied the documentation [Litt86].

One way of acquiring information about a program is from the documentation. It is widely understood that good documentation can aid the process of understanding programs. However, the problem for most maintainers is they have to maintain unfamiliar code that has been modified and the accompanying documentation is usually out of date, inadequate, inconsistent or sometimes non-existent. In the case where documentation does exist, maintainers may find it difficult to acquire sufficient information from the document because it is not produced with their needs in mind. As a last resort, they have to rely on the source code in order to gain an understanding of the programs. Sometimes, the source code may be the only reliable documentation available to them.

Program Comprehension plays an important role in Software Maintenance as well as other activities in Software Engineering. It can be used as an aid to Reverse Engineering [Robs91], Testing and Debugging [Weis82, Weis84, Weis86], Reuse [Stan84], Redocumentation [Basi82, Youn93], and Learning, as shown in Figure 1-1. A good understanding of the source code is required before the commencement of any of the activity mentioned above. For a maintainer, the primary desire is the ability to decipher the source code accurately, quickly and efficiently.

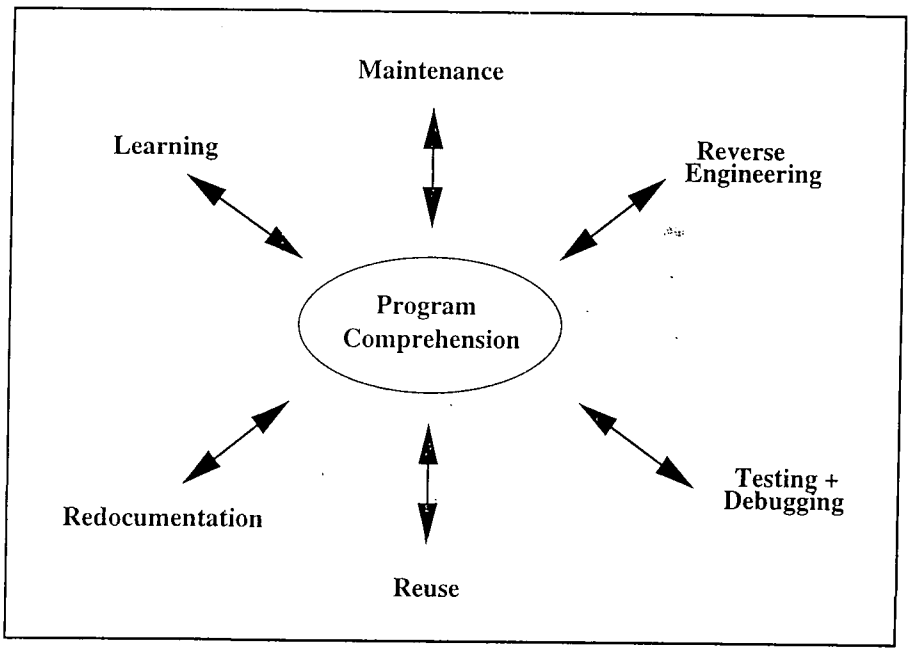

Figure 1-1 Program Comprehension in relation to other activities in the context of Software Engineering

There are a number of theories and models of Program Comprehension advocated by psychologists who are interested in studying the behaviour of programmers. Most of the work has been carried out by observational studies, where typically, programmers are given a task to complete within a time limit. Some of the programmers were tested against their understanding, while others were encouraged to think out loud so that their thoughts could be recorded. The results indicate that comprehension is performed in either a top-down or a bottom-up manner. However, Chan [Chan97], Letovsky [Leto86a] and von Mayrhauser and Vans [Mayr95] believe that the process of comprehension is an opportunistic approach where maintainers utilise both the top-down and the bottom-up approaches whenever additional information is encountered.

There are a number of academic and commercial software maintenance tools available but most of them are not powerful enough for use on a large scale as they offer limited analysing power. Most of them are developed under the influence of a particular Program Comprehension theory which may not be sufficient to cope with the diversity of the software maintenance activities. It is unlikely to find an existing tool which has the capability to assist all activities which are encompassed by the various cognition models for Program Comprehension.

1.4 Research Problem

The research program to be addressed in this thesis is to provide a framework that enables the utilisation of various Program Comprehension theories and models in the same environment.

Programs are complex, abstract objects which include many components with many different attributes that are interrelated in complicated ways. Maintainers may find it difficult to understand and navigate through these complex interrelationships among different parts.

An important aid to the problems of Program Comprehension has been the use of static and dynamic analysis tools which can provide useful and up to date information of a program. Through providing different views such as call graphs, control flow graphs, data flow information, program slices and cross references, a maintainer can utilise this information to gain a better understanding of a program. This information is mostly presented in textual and two-dimensional form at present, which can lead to problems of layout and display for large amounts of information generated by the analysis tools.

The intention of this research is to investigate what can be done when a maintainer is overloaded with too much information to deal with. The roles of both the textual and graphical representations will also be investigated.

Graphical representations are useful for exploring relationships. For modern large-scale problems, which require maintainers to understand large collections of information, solutions must be found for managing these complex interrelationships. This problem can be decomposed into three subproblems:

- how to make a meaningful visualisation of a single object
- how to make a meaningful visualisation of a collection of objects
- how to allow the users to control the selection of the visualisation efficiently

The essence of the problem to be researched is that of providing a mechanism for maintainers to achieve an understanding of a program by using the Program Comprehension theories which are suitable to the task at hand.

1.5 Criteria for Success

A In order to facilitate the process of Program Comprehension, a maintainer needs to have access to different kinds of information concerning a piece of source code. This can be in textual and/or graphical forms. Hence:

- maintainers should have easy and quick access to information at different levels of abstraction during various stages of comprehension
- support should be provided for maintainers with various degrees of experience and abilities
- support should be provided for the different types of maintenance activities that they may engage in

B There are a number of theories and models of Program Comprehension. Some researchers argue that it is done in a top-down fashion, whereas others advocate that it should be conducted in a bottom-up manner. There is no real consensus on how maintainers should perform comprehension. Moreover, most maintainers may prefer to employ the use of a mixture of strategies when the situation arises. Hence:

- any alternative approach to Program Comprehension proposed should address the need for a more flexible approach

C The feasibility of the Integrated Approach proposed needs to be examined. Hence:

- it needs to be demonstrated that it is feasible to realise the Integrated Approach in a physical form which can be executed with minimal difficulty

D The size of a software system should not be a hindrance to the process of Program Comprehension. Much research effort has been devoted to the development of techniques which support understanding-in-the-small. Hence:

- the Integrated Approach should be equipped with the capability to support understanding-in-the-large

In the context of this thesis, the term understanding-in-the-small is used to refer to the set of activities that are associated with the understanding of small programs which are relatively simple. The term understanding-in-the-large refers to the understanding of larger programs which contain more complex program relations.

E The usability and practicality of the Integrated Approach and of the implementation needs to be examined. Hence:

- both the Integrated Approach and the implementation should be measured against a set of criteria, which should lead to an objective evaluation

1.6 Thesis Overview

The remainder of this thesis is organised as follows.

Chapter Two reviews two areas of Program Comprehension. In the first part of this chapter, the theoretical background of the comprehension process and different Program Comprehension theories and models will be discussed. This is followed by a review of the common techniques and practices used by the maintainers during the comprehension process. It will concentrate on the use of visualisation techniques. A number of strategies which can be used to improve the complexity of the graphical representations will be presented, together with a survey of a number of Program Visualisation systems.

Chapter Three describes a framework for the evaluation of the Integrated Approach outlined in Chapter Four, the implementation outlined in Chapter Five and the Case Studies outlined in Chapter Six. The first part of the chapter explores the use of research methods such as Surveys, Formal Experiments and Case Studies. The second part of this chapter describes a set of objective criteria for the evaluation. The set of criteria is divided into two branches. The first branch is intended to capture various comprehension theories such as the top-down and bottom-up approaches. The other branch addresses the cognitive issues of a maintainer while he browses and navigates the visualisation of the program structures.

Chapter Four introduces an alternative approach to Program Comprehension. The Integrated Approach addresses the need for a more flexible approach to comprehension, and it provides a framework and mechanism to facilitate the understanding of large software systems. In particular, it discusses the use of Program Relationships through carrying out a systematic analysis of Program Elements.

Chapter Five describes how the various Program Comprehension theories and models can be realised by a simple browsing tool named PUI (***P***rogram ***U***nderstanding ***I***mplement), which allows maintainers to understand the relationships between Program Elements. The tool is based on a matrix of Program Elements and Program Relationships which are designed to reflect the multi-dimensional nature of programs.

Chapter Six demonstrates the principal use of the prototype by way of Case Studies. Demonstrations of how both the top-down and the bottom-up approaches to Program Comprehension can be utilised by using PUI is presented in this chapter. It shows how maintainers can use the prototype to recover information as they browse through the various parts of a program representation.

Chapter Seven presents an evaluation of the work undertaken. It is evaluated against the existing Program Comprehension theories and models, the prototype implementation and the results of the Case Studies. They are evaluated against a hierarchy of cognitive issues raised in Chapter Three. This is followed by a discussion on the requirements for automation.

Chapter Eight presents a summary of this research and evaluates the success of the research against the criteria defined in section 1.5. An indication on the directions for further work is also presented.

Chapter Two

Theories and Practices of Program Comprehension

2.1 Introduction

This chapter reviews two areas of Program Comprehension. In the first part of this chapter, the theoretical background of the comprehension process and different Program Comprehension theories and models are discussed. This is followed by a review of the common techniques and practices used by the maintainers during the comprehension process. It will concentrate on the use of visualisation techniques. A number of strategies which can be used to improve the complexity of the graphical representations will be presented, together with a survey of a number of Program Visualisation systems.

2.2 Theories and Models of Program Comprehension

Program Comprehension plays a critical part in all aspects in Software Engineering, and especially in software maintenance. Activities such as Reverse Engineering, Reuse, Testing and Software enhancement will require a good understanding of the source code before any modification is to take place. Research has shown that maintainers spend a considerable amount of time studying programs. This figure can be as high as three-and-a-half times as long as they studied the documentation [Litt86]. In the absence of a complete and consistent documentation, the source code may be the only information the maintainers have. Hence, there is a strong desire for strategies and techniques which can be utilised to facilitate the comprehension process. The following is a review of the literature on the theories and models of Program Comprehension.

2.2.1 Syntactic/Semantic Knowledge

Shneiderman [Shne80] conjectures that the information chunking process is used in understanding programs. Programmers abstract the information in the programs into chunks which are then built into an internal semantic structures representing the programs.

Further, he suggests that programs are not understood on a statement by statement basis unless a statement represents a logical chunk. Shneiderman and Mayer [Shne79] identify three types of knowledge used in Program Comprehension:

- **Syntactic Knowledge** is the language dependent detail used for carrying out actions or defining objects. For example, the use of semi-colons to terminate or separate statements or the use of iteration words (DO, FOR, LOOP or REPEAT) is language-dependent and arbitrary. This knowledge must be frequently rehearsed to preserve retention.

- **Semantic Knowledge of Software Engineer** is meaningfully acquired by reference to previous knowledge by example or by analogy. There is a logical structure to semantic knowledge that is independent of the specific syntax used to record it.

- **Semantic Knowledge of task-related knowledge** is the domain knowledge about the real-world in which the program operates. For example, it may be the knowledge of accountancy practices or air traffic control procedures. Any knowledge of business rules defined by the users of the program also fits into this category.

Information regarding a program is categorised into different levels of representation ranging from high to low. High level representation allows a top-down comprehension approach to be used and low level representation favours the bottom-up approach. Shneiderman and Mayer believe that the semantic knowledge is acquired by experience and through active learning where new information is consciously integrated with existing semantic structures. They found that the major difference between novice and expert programmers lies in the type of knowledge they possess. Experts tend to concentrate on building a semantic representations of the programs, whereas novice programmers rely more upon the retention of specific code.

2.2.2 Systematic/As-needed Approach

Littman *et al.* identify the strategies that programmers were observed to use when studying small programs [Litt86]. They believe that there are two basic approaches to Program Comprehension. They are the systematic approach and the as-needed approach.

- **Systematic approach:** When using this strategy, a maintainer examines the entire program and works out the interactions among various components within the program. This is completed *before* any attempt is made to modify the program. This usually involves identifying the data flow and control flow between subroutines.

- **As-needed approach:** In contrast to the systematic strategy, a maintainer understands and studies only *parts* of a program which need to be modified. Program reading time is thus minimised. Once the maintainer has gained enough information, the modification is commenced.

Littman *et al.* suggest that the approach a programmer uses to study a program strongly influences the knowledge acquired. This knowledge directly determines whether the programmer can perform a successful modification. They also identify two types of knowledge: static and causal knowledge. To perform a successful modification, a programmer must be able to detect the static and causal interactions among the functional components. Together, they enable the programmer to create a strong mental model. This process corresponds to the systematic approach to Program Comprehension. They believe if only the static knowledge is gathered, it will eventually lead to a weak mental model and the programmer may fail to perform a successful modification. This process corresponds to the as-needed approach. The study shows that the systematic approach has been proven superior to the as-needed approach on small programs. The authors argue that the use of systematic approach may not be feasible on large programs. Programmers may be forced to use studying methods similar to the as-needed strategy. It is therefore necessary to augment the as-needed strategy so that additional information can be acquired.

2.2.3 Hypotheses Verification

Brooks [Broo83] believes that the theory of Program Comprehension is based on the reconstruction of mappings between the problem and the programming domain. In his model, comprehension is an iterative process of hypotheses verification and modification. A maintainer begins with an initial hypothesis about the behaviour of a function which is generated from documentation or from sources such as a program name or a variable name. This initial hypothesis leads the maintainer to expect certain structures and operations in the program. These expectations form a second level of more specific hypothesis about the function. Brooks calls these expectations 'beacons'. An example which

he proposes is the swapping of values, which he believes is a beacon for a sorting routine. Once the relatively specific hypothesis is established, the maintainer then tries to verify the hypothesis from information in the code by refining or rejecting it iteratively until the hypothesis matches the actual code. This process is repeated until sufficient information is obtained. Brooks argues that when maintainers try to verify hypotheses, it is not by line-to-line examination of source code but rather by a series of increasingly specific hypotheses about the program functions. Knowledge of the problem domain also plays a critical role in making hypotheses. The ability of making appropriate hypotheses lies in the experience of a maintainer in a particular domain.

2.2.4 Beacons

Brooks introduces the concept of beacons [Broo83] and this concept is further explored by Wiedenbeck [Wied86, Wied91]. In the study [Wied86], Wiedenbeck investigates whether beacons exist as a focus for program understanding. Programmers were given a short time to both memorise and understand a program. The results support her hypothesis about beacons. She concludes that the process of comprehension is not linear, and each statement in the source code does not play an equal role. She believes that beacons can give high level overviews of programs. However, these overviews are not sufficient for debugging or modification purposes which may require a deeper level of understanding.

In another study [Wied91], Wiedenbeck suggests that Program Comprehension is a gradual process of assimilation through study. Beacons can be described as idiomatic or stereotypical elements in the source code. She points out that most maintainers seem to have a tendency to refer to the source code to develop an overview of a program. This orienting phase is important because it allows a mental map of the program to be developed. The mental map includes the basic goals and operations which can be later used to build a deeper understanding of the programs. The results of this study have shown that:

- Programs with beacons were understood more accurately than those without. Further, beacons can aid Program Comprehension even in unfamiliar programs.

- Beacons have the power to aid Program Comprehension when they appear in the appropriate context; they also have the power to depress Program Comprehension or even lead to 'false comprehension' (wrong hypothesis) when used inappropriately.

Wiedenbeck concludes by stating that beacons do play a large role in the initial high-level comprehension of a program. They can help the programmers to gain overviews of programs with minimal effort.

2.2.5 Program Plans

Much of the research effort has been devoted to Program Plans [Leto86b, Solo84, Solo86]. Soloway and Ehrlich [Solo84] suggest that experts have and use two types of programming knowledge in the process of comprehending programs:

- **Programming Plans** are program fragments that represent stereotypical action sequences in programming such as a RUNNING TOTAL LOOP PLAN or an ITEM SEARCH LOOP PLAN.

- **Rules of Programming Discourse** are rules that specify the conventions in programming. For example, the name of a variable should usually agree with its function. These rules set up expectations in the mind of the programmers about what should be in the program. They are analogous to discourse rules in conversation.

Soloway and Ehrlich argue that programs are composed from a number of programming plans that have been adapted to fit the needs of specific problems. The composition of such plans are governed by the rules of programming discourse. They believe that if the rules of discourse are violated, it can make a program much more difficult to comprehend. For example, the use of a variable name **`MAX`** in a function would lead the programmer to expect the variable to hold the maximum value of some numbers, instead of expecting it to hold the minimum. If the latter is true, then the programmer would need to employ additional processing techniques in order to reach the correct conclusion. The authors conclude that programming plans and the rules of programming discourse do play a powerful role in Program Comprehension. Experts have strong expectations about what programs should look like, and it would be a real hindrance to programmers when those expectations are violated.

Letovsky and Soloway [Leto86b] suggest that the goal of program understanding is to recover the intentions behind the source code. A maintainer may use the following to achieve this:

- **Goal** is used to denote the intentions
- **Plan** is used to denote the techniques for realising the intentions

In their model, program understanding is viewed as a process of recognising plans in the source code. Program plans are conceptually distinct from algorithms and functions. The essential property of program plans is that they can be composed in complex ways. For example, plans can be abutted, interleaved, nested or merged. Algorithms are simply the compositions of program plans. The recognition of plans may be complicated by *delocalised plans*, where statements within a plan are

scattered throughout the whole of a program. Letovsky and Soloway [Leto86b] argue that when a maintainer tries to perform a modification within a short time, he often forms a local and partial understanding of the program by focusing his attention on the parts of the code which would be affected. When neither the program nor the documentation reveals that certain pieces of code are interdependent and that they are some distance away, the formation of a purely local understanding can lead to an inaccurate understanding of the program as a whole. This in turn can result in incorrect or inefficient comprehension. Letovsky and Soloway believe that the tendency of programmers to make plausible but incorrect assumptions is considered as a fundamental problem for Software Maintenance.

2.2.6 A Cognitive Model

Letovsky describes an empirical study of the cognitive process of Program Comprehension [Leto86a]. In the study, programmers were given a program to modify and were encouraged to think out loud so that their thoughts could be recorded.

Based on the analysis on the empirical results, Letovsky develops a cognitive model of the subjects' understanding processes. He views programmers as Knowledge Base Understander. A Knowledge Based Program Understander consists of the following:

- **A knowledge base**. It encodes the expertise and background knowledge which a programmer brings to the comprehension process.
- **A mental model**. It encodes the programmer's current understanding of the target program. This model evolves in the course of the understanding process.
- **An assimilation process**. It interacts with the stimulus materials (target program code and documentation) and the knowledge base to construct the mental model.

Base on the results of the empirical study, it is suggested that a mixture of top-down and bottom-up strategies are employed in both the mental model and the assimilation process. Letovsky believes that the human Understander is best viewed as an opportunistic processor capable of exploiting both bottom-up and top-down cues as they become available.

2.2.7 Stimulus Structures and Mental Representations

Pennington [Penn87] believes that comprehension involves detecting or inferring different kinds of relations between parts of a program. Based on the results from two studies, Pennington suggests that comprehension leads to two different mental representations:

- **A Program model**. In this model, Pennington found that the mental representation built is a procedural one (control-flow program abstraction) when a piece of source code was shown to the programmers the first time. This representation is built from bottom-up via the identification of beacons and programming plans.

- **A Situation model**. This model requires knowledge of the real world domains and it tries to relate representations in the program model to the domain. The situation model is complete once the program goal is reached.

Pennington uses text structure [Basi82, Broo83] and programming plan knowledge [Solo84] to explain the program model development. It is created via chunking and cross-referencing. In the situation model, the matching process takes information from the program model and builds hypothesised higher order plans. These new plans are chunked to create additional higher order plans. The program model can change even after the situation model construction has begun. Pennington believes that programmers use the plans as input to the program model comprehension process. They allow a cross-reference map to be built which is aimed to establish direct mappings from procedural and statement-level representations to the functional and abstract program views. Higher order plans may cause a programmer to enhance the program model.

2.2.8 An Integrated Metamodel

von Mayrhauser and Vans [Mayr94, Mayr95] express the view that none of the existing theories and models for Program Comprehension can account for all the different behaviours of the programmers when they try to understand unfamiliar source code. The Integrated Metamodel is formulated in order to reflect the cognition needs for large software systems. It addresses some of the shortcomings of the existing theories and models, and tries to piece together the relevant portions of the strategies in a single model.

The integrated code comprehension model has four major components:

- Top-down structures
- Situation model
- Program model
- The knowledge base

This model combines the top-down approach (top-down structures and the knowledge base) proposed by Soloway and Ehrlich [Solo84] and the bottom-up approach (situation and program models) proposed by Pennington [Penn87]. The knowledge base is needed in order to build the other three components successfully. Each of the first three components are involved in the internal

representation of a program. The knowledge base furnishes the process with information related to the comprehension task and will store any new and inferred knowledge. For large systems, a combination of approaches to Program Comprehension becomes necessary. Based on the results of the paper [Mayr94], both the authors believe that any three of the approaches may become active at any time during the comprehension process.

2.3 Current Techniques and Practices

The following sections discuss the common techniques and practices often used by the maintainers during the comprehension process.

2.3.1 The Concept Assignment Problem

Biggerstaff *et al.* believe that concept assignment is closely linked to Program Comprehension [Bigg93, Bigg94]. The authors explain that a person understands a program because he is able to relate the structure of the program and its environment to the human oriented conceptual knowledge about the world. The problem of discovering individual human oriented concepts and assigning them to their implementation oriented counterparts for a given program is the concept assignment problem.

Through the use of Case Studies, Biggerstaff *et al.* [Bigg93] has found that there is no definite solution to this problem. Automation of the process is difficult and it would require architectures that process a range of information types varying from formal to informal. The study has also found that understanding is derived from a process that relies strongly on plausible inference. They believe that better understanding of programs relies on an *a prior* knowledge base that is rich with expectations about the problem domain and the program architecture typical of that problem domain. Their views echo with those suggested by Shneiderman and Mayer [Shne79] where the process of Program Comprehension is built upon a knowledge base consisting syntactic, semantic and domain knowledge.

2.3.2 Modularisation

Wirth introduces a technique for program development by stepwise refinement [Wirt71] and Parnas suggests the use of modularisation [Parn72]. Both of the approaches are aiming at improving the understandability of the source code by hiding information at various levels of development. Other techniques such as Jackson's Structure Programming, or JSP [Jack85], and Object-Oriented Design, or OOD [Booc91], are aiming at developing programs which have a specific structure and design in order to improve reliability and maintainability.

Most maintenance activities are a cognitive skill. It is therefore subjected to the limitation of the human brain, i.e., only a limited amount of information can be studied at a time. Shneiderman [Shne80] conjectured that the information chunking process is used in understanding programs.

Maintainers abstract the information in the program into chunks which are then built into an internal semantic structure representing the program. Complex problems are usually decomposed into sub-problems until these 'chunks' are reduced to manageable sizes. This echoes with the views expressed by both Wirth and Parnas.

JSP [Jack85] comprises three principles of structured program design. They are stepwise refinement, the use of three structured control constructs, namely, sequence, iteration and selection, and finally, data structure-based design. This design method is based upon a hierarchical view of the data processed by a program. Jackson's contention is that program designs should be dictated by the characteristics of the data being processed. It is these characteristics which later determine the structures of the programs.

Booch suggests that when designing a software system of any complexity, it is essential to decompose it into smaller and smaller parts so that each one may be refined independently [Booc91]. Booch believes that the use of OOD not only helps to organise the inherent complexity of software systems, but it also supports software reuse directly. Booch believes that Object-oriented systems are more resilient to changes because their designs are based upon stable intermediate forms, and hence better able to evolve over time. Under OOD, software systems are viewed as collections of objects, where each object manages its own state information. An object may comprise the data structure and operations which it inherits from a class plus any other attribute which uniquely defines the object. Conceptually, an object communicates by exchanging messages with other objects.

2.3.3 Program Slicing

The concept of Program Slicing was first introduced by Weiser [Weis82]. Weiser's original version of program slicing is classified as static slicing. Another type of program slicing was introduced in the papers [Weis84, Weis86], which is known as dynamic slicing. Apart from these two types of program slicing, techniques such as quasi-static slicing [Venk91], conditioned slicing [Luci96] and amorphous slicing [Harm97], have also received a lot of attention. The following discussion will concentrate on static and dynamic program slicing.

In his paper [Weis82], Weiser defines program slicing as:

> *The process of stripping a program of statements without influence on a given variable at a given statement.*

Weiser believes that program slices are most useful for understanding a program when they are considerably smaller than the original program. He believes that experienced programmers are mentally slicing and decomposing the program while debugging. Weiser introduces a formal

definition of slices and a mechanism to extract slices in programs in the paper [Weis84]. He believes that slices have a very clear semantics based on the projections of behaviour from the program being decomposed. Program slicing is a method of minimising the amount of code to be studied when debugging or understanding programs [Weis86].

An application of data flow and control flow analysis can be used to extract the slices from the program which contain only those statements relevant to the computation of a given output [Weis86]. This technique is known as Dynamic slicing. A dynamic program slice is an executable subset of the original program that produces the same computation on a subset of selected variables and inputs. In other words, it consists of all statements that actually affect the value of an instance of a variable for a given input. This technique has been further developed and reported in their work [Arga90, Kore88, Kore97]. Dynamic slicing differs from the static slicing [Weis82] in that it is entirely defined on the basis of a computation. The main advantage of dynamic slicing is that data structures such as arrays and pointers can be handled more precisely and the size of the slice can be significantly reduced, leading to a finer localisation of the fault.

The as-needed approach proposed by Littman *et al.* [Litt86] can be facilitated by the technique of program slicing. Under the as-needed approach, a maintainer understands and studies only parts of a program which need to be modified. Once the maintainer has gained enough information, the modification will commence. Weiser advocates that programmers do not have to waste time learning about irrelevant details, they can concentrate on the program slices instead which he believes can shorten the comprehension time [Weis84, Weis86]. However, the statements in a slice may be scattered throughout the code of the larger program. Some crucial elements in a program may be left out using this technique which may affect the correct behaviour of the program.

2.3.4 Source Code Presentation

The problem for most maintainers is they have to maintain unfamiliar source code that has been modified and the accompanying documentation is usually out of date, inadequate, inconsistent or sometimes non-existent. Improving ways of abstracting relevant information from the source code is therefore much needed. This issue can be tackled in a number of ways:

- the use of natural naming
- the use of comments
- pretty-printing the source code

I Natural Naming

The development of high level languages such as Pascal and C was an important step towards increasing source code readability and understandability. When a maintainer first encounters an identifier, he would invariably try to infer a meaning from its name [Broo83]. The use of appropriate naming for variables, functions and program files is thus essential to bridge the gap between the programs and the semantics of the problem domain.

Laitinen [Lait95] believes that the objective of using natural names in source code is to increase program understandability, which in turn facilitates software development and maintenance. Having natural words in source code brings these documents terminologically closer to other types of documents such as written English texts and graphical-textual models (software designs). Using natural names in source code should therefore make the entire documentation simpler.

If there are two versions of a functionally equivalent program where each has a different visual appearance, it is very likely that each may evoke a different mental model in an observer's mind. This coincides with the results found in another study. Teasley has found that naming style is an important factor in comprehension of programs written in high level procedural languages [Teas94]. The results also show that experienced programmers are better at finding cues present in the textual material to gain an understanding of the programs than the novice programmers.

II Comments

It is common consensus that the use of suitable comments can be an invaluable aid to the comprehension process. The use of appropriate comments can be very powerful when used in conjunction with suitable naming of identifiers used in source code. From the name of an identifier, a maintainer form an assumption about the functionality of that identifier [Broo83]. The Maintainers then search for extra cues from the source code in order to justify this assumption. Comments can be valuable and effective in providing these extra cues.

III Pretty-printing

Pretty-printing, which was introduced by Ledgard [Ledg75], has gained much attention since the 1970s. It describes the use of indentation, spacing and layouts to make source code more presentable and readable to the programmers. The principle behind pretty-printing is that the appearance of the source code can affect the comprehension process.

Miara *et al.* [Miar83] have conducted a comprehension experiment and they report that the use of indentation and block-structured source code can facilitate the comprehension process. They conclude that if no indentation is used in a large program, it would be a real hindrance to the comprehension process and the program would be difficult to follow. The idea of pretty-printing is further explored by

Baecker and Marcus [Baec90]. They developed a system, SEE, which can take the listings of a C program and produce a book-like layout. Oman and Cook [Oman90b] have conducted several empirical studies and they believe that the book paradigm is superior to traditional methods such as natural naming and the use of comments. They argue that the book paradigm provides a method of formatting which is consistent with the comprehension theories and models. By providing visual cues and different ways to organise the source code, typographical formatting can reflect the underlying structures of the source code and aid the comprehension process.

2.3.5 Visualisation

Purely textual source code is far from matching the maintainer's cognitive model of a program, though it may be the case that a maintainer will use the relative locations of program constructs within the source code as a basis for the cognitive model. Formatting or pretty-printing of the source code using techniques such as indentation and spacing can give the code some visible structures. However this can only be viewed in small portions; the maintainer must still navigate the source code to construct an overall model.

All the theories and models of Program Comprehension discussed in the section 2.2 agree that the comprehension process involves an abstraction process and the construction of a cognitive model during different stages of the comprehension process. The abstraction process works hand in hand with the cognitive model. During the abstraction process, maintainers would look for various cues from the source code and try to extract relevant information from them. A cognitive model is then constructed which will later guide the maintainer to follow and understand the interrelationships between the program constructs.

Different levels of abstraction can be displayed using graphical representations. They can take on a number of forms and can represent various views of the programs. The most commonly used graphical representation of a program is the call graph which shows the functions as nodes and the function call relations as directed arcs depicting which functions are called and from where [Ryde79]. Other graphical representations used are the control flow graph, module dependencies, file inclusion hierarchies, hybrid call/control flow graphs, data flow and message passing. Each of these graphical representations provides the maintainers with a different perspective on the software system though none of them can give the full picture.

The use of visualisation techniques to facilitate the comprehension process can be an important step forward. The ultimate goal of Program Visualisation is to help maintainers to form clear and correct mental images of a program's structure and functions. Graphical representations are useful in that they are easy to understand and manipulate. These representations are a natural way to depict relationships.

I Problems in Laying Out Graphs in Two-dimensions

It is widely acknowledged that there are problems in laying out graphs [Carp80, Gans88, Gans93, Mess91, Rein81, Sugi81, Tama88, Walk90, Warf77, Weth79]. The layout of graphs are governed by both the aesthetic features and the semantic constraints of the drawings of graphs. Most of the aesthetic features and constraints are incompatible in nature and trade-offs have to be made in order to produce drawings that can convey the appropriate meanings. A considerable effort is required to select criteria to suit the needs of a particular type of graph.

Batini *et al.* [Bati85] have analysed and compared two hundred different diagrams in order to find out how the layout of the diagrams can be affected by the different aesthetic features and how these features can affect viewers to perceive the diagrams. The sources of these diagrams were selected from scientific papers, technical publications and industrial project documentation. Batini *et al.* believe that the difference between the human and automatic approaches in the layout problem lies in how the conflicts between aesthetic features and semantic constraints are resolved. They found that automatic tools usually adopt fixed weights (trade-offs) in solving the incompatibilities, while human designers tend to choose different weights for each application, thus reaching better results. They believe that the key to alleviate the layout problems is to:

- find out as many layout criteria as possible
- find out the ranges of the weights usually adopted by designers in solving the conflicts between such criteria

Each of the aesthetic features and semantic constraints which governs the readability of the drawings may be:

- local or global
- hierarchic or flat

A feature or constraint is local when it refers only to a part of the drawing, it would be global otherwise. In the same vein, a feature or constraint is hierarchic when it concerns the relative positions of a set of symbols, it would be flat otherwise.

II Strategies for Improving Graphical Representations

Studies have shown that most aesthetic features are incompatible in nature [Supo83, DiBa94]. Conflicts have to be resolved and trade-offs have to be made in order to produce drawings that can convey the relevant information to the viewers. Moreover, the problem of layout for large amounts of information generated by the static analysis tools is still left unresolved. It is widely acknowledged

that humans cannot handle highly complex systems. The systems are repetitively broken down until they are divided into parts which can be handled with ease. Techniques such as graph simplification and graph reduction are frequently used to managing the highly complex graphs.

It is widely accepted that graphical representations can offer better insights into a program when compared to the textual representations. Call graphs, control flow graphs and data flow diagrams are the most frequently used graphical representations. However, while graphical representations are an improvement upon textual ones, they still have a tendency to provide maintainers with too much information. For this reason, the Visualisation Research Group in Durham has carried out a number of Case Studies to investigate the use of visualisation techniques [Burd96]. The Group has also suggested a number of strategies which can be applied in order to improve the readability of call graphs. The work concentrates mainly on the C programming language, but other languages such as COBOL have also been investigated. The suggestions are:

- simplification involving the hiding of nodes
- clustering involving the grouping of nodes
- slicing involving the extraction of nodes
- presentation

Burd *et al.* [Burd96] maintain that the strategies identified are not intended to form a rigid method, rather they provide a selection of strategies which the maintainers can select in order to produce the best results for an application under maintenance.

A Graph Simplification

For a small and simple program, the global program behaviour can be examined and studied thoroughly. However, as programs grow in size and complexity, the task may no longer be trivial. Burd *et al.* [Burd96] have identified five graph simplification strategies:

- to number arcs
- to isolate subgraphs
- to hide third party libraries
- to hide ANSI C standard libraries
- to hide external function calls to the application's libraries

One major cause of clutter in call graphs is multiple calls of one function to another, which leads to multiple directed arcs between the nodes. These arcs can be combined and replaced by a number which denotes the number of function calls made. As a result, the number of directed arcs is reduced and no information is lost.

Another strategy is to isolate any unconnected graphs. The rest of the strategies involve the hiding of certain library functions. Obviously, if the aim is to investigate the behaviour of the source code which relates to those libraries such as memory management, then the hiding of the libraries may not be a sensible approach.

The authors have observed that even after applying these strategies, one may still be left with a complex relationship. The approach of information clustering may be more useful if the interactions among the user defined functions are low and the interactions among the library routines are high.

B Clustering

Information clustering is the process whereby information is abstracted from the call graph and represented as 'common nodes'. The information clustering principle can be used in a number of ways:

- grouping of function calls to other source code files
- grouping of function calls to other libraries
- grouping nodes into groups where nodes have a high degree of fan-in or fan-out

Burd *et al.* have again observed that the grouping of some nodes may increase the complexity of the call graph in some cases. Nevertheless, it is possible to analyse and identify nodes which may benefit from clustering.

C Graph Slicing

Graph slicing is another way of reducing complexity. Contrary to the technique of graph simplification, the attention is given to a small number of nodes and their connecting nodes. By concealing the rest of the nodes present in a graphical representation, a small section of the representation can be studied with more attention. The slicing principle can be used in a number of ways:

- to investigate the characteristics of function calls
- to investigate the characteristics of library function calls
- to investigate the ripple effect after a modification

D Presentation

Apart from the graph layout strategies, a number of other approaches to support the understanding process have also been investigated by Burd *et al.*:

- the use of colour
- hierarchical views

One use of colour is to indicate clustering and information encapsulation. It can also be used to indicate connectivity. Conversely, colour can be used for concealment. To prevent distraction by the appearance of certain nodes, these nodes can be set to the same colour as the background. This in effect is similar to the hiding principles described above, but leaving the nodes on the graph.

As the authors have pointed out, colour can also be used to identify a program's hierarchical composition. In directed graphs, the nodes in each level are traditionally laid on one horizontal line, and the levels are stacked vertically [Mess91]. The primary goal of the hierarchical layout is to try to reveal the ancestral relationship among nodes clearly and unambiguously. In a perfect hierarchy, all the nodes predecessors appear physically above, and all of the nodes successors appear physically below it. However, rarely are such perfect hierarchies achieved, and thus using colour to represent hierarchical levels is a more flexible approach.

III Program Visualisation

Programs are built by many functional components and they are often related in complicated ways. In the paper [Fitt79], Fitter and Green try to identify some of the principles that the designer of a graphical notation should be aware of and they also highlight some of the problems associated with the present notations. They point out that the use of diagrams has often been proven successful and many of the graphical conventions can be learnt very quickly. They can reveal the structures inherent in the underlying data or process by which entities are manipulated and so graphical representations make an excellent communication medium.

Fitter and Green propose that a good graphical notation should:

- present relevant information in a perceptual form
- restrict viewers to objects that can readily be understood
- reveal the underlying mechanisms and be responsive to manipulation
- allow easy and accurate revision

Both authors admit that it is impossible to lay down principles that would ensure a good fit for a graphical representation for a given set of aesthetic features and semantic constraints. All that can be done is to eliminate the misfits.

Messinger *et al.* [Mess91] point out that many people still find it is difficult to lay out graphs with many vertices and edges. For example, a viewer may find a reduction from 20 edge crossings to 10

improving readability whereas a reduction from 2000 to 1990 edge crossings is not likely to have the same effect. To produce a graphic layout from application-generated data such as a parse tree generated by a compiler, is also considered laborious. Not surprisingly, a lot of effort has been focused on reconciling aesthetics features and semantic constraints of graphical representations such as minimising edge crossings and balancing distribution of graph elements. Messinger *et al.* argue that present technologies still do not allow large graphs, one with thousands of vertices and edges, say, to be displayed in their entirety, and so some sort of display/browser interface must be employed. It is important to provide a mechanism which can offer overviews, multiple views and hierarchical abstractions of graphs.

The goal of Program Visualisation is to help maintainers form clear and correct mental images of a program's structure and functions. When combined with the abstraction power of human vision, the interactive power of graphics environments will remain central to the efforts of harnessing computing power.

Visualisation is often widely understood as comprising only of visual images. However, Price *et al.* [Pric93] emphasise that the term Visualisation conveys more meaning than this restricted view. In their opinion, visualisation is 'the power or the process of forming a mental picture or vision of something not actually present to the sight'. They argue that programming is visual because it involves programmers reading textual information (source code) instead of reading serially a stream of ones and zeros in the way an interpreter or a compiler does.

The idea of using visual representations to aid Program Comprehension is not new. In the 1950s, flow charts were first introduced to present diagrammatic forms of the source code. In the 1970s, pretty-printers (the use of spacing, indentation and layout) were employed to facilitate Program Comprehension. Today, window interface techniques are gaining popularity. These techniques which allow direct manipulation of objects on screens, take the full advantage of large-screen graphics and windowing-based computer systems.

The use of visualisation techniques is particularly suitable to be used in conjunction with the Design and the Maintenance phase of the Software Maintenance process model. Visualisation is used in the Maintenance phase in two significant ways: for code comprehension and for impact analysis. Price *et al.* believe that traditional use of call graphs, control flow graphs and entity-relation diagrams also fits comfortably well inside the area of Program Visualisation.

IV Definitions

There is yet to exist an agreement on the definition of the term Program Visualisation, a list of definitions are presented below:

> *Program Visualisation refers to the use of graphics to illustrate some aspects of the program or its run-time execution. The original program is usually specified in a conventional, textual manner.* [Myer90]

> *Program Visualisation, in the general sense, is the use of various techniques to enhance the human understanding of computer programs.* [Pric93]

> *Program Visualisation is a mapping, or transformation, of a program to a graphical representation.* [Roma93]

Price *et al.* believe that Program Visualisation consists of several components. They are:

- **Code Visualisation:** It illustrates the actual program code by adding graphical marks to it or by converting it to a graphical form, such as flow charts.
- **Data Visualisation:** It shows graphical forms of the actual data of the program.
- **Algorithm Visualisation:** It uses graphics to show *abstractly* how the program operates.

These components can also be incorporated in the static or dynamic analysis of programs.

Algorithm Visualisation is different from Data and Code Visualisation. It is the visualisation of a *high level description* of program code and the graphics may not correspond to a specific piece of code, whereas implemented code is visualised in Code or Data Visualisation. Dynamic Visualisation systems can show the animation of the programs' behaviour when they are executing. Static visualisation systems, on the other hand, are limited to show the analysis of programs prior to execution.

The term Visual Programming is often confused with Program Visualisation. Myers [Myer90] refers Visual Programming to any system that allows the user to specify a program in a two- or three-dimensional fashion whilst Price *et al.* [Pric93] prefer a more general definition. They consider that Visual Programming is the use of visual techniques to specify a program.

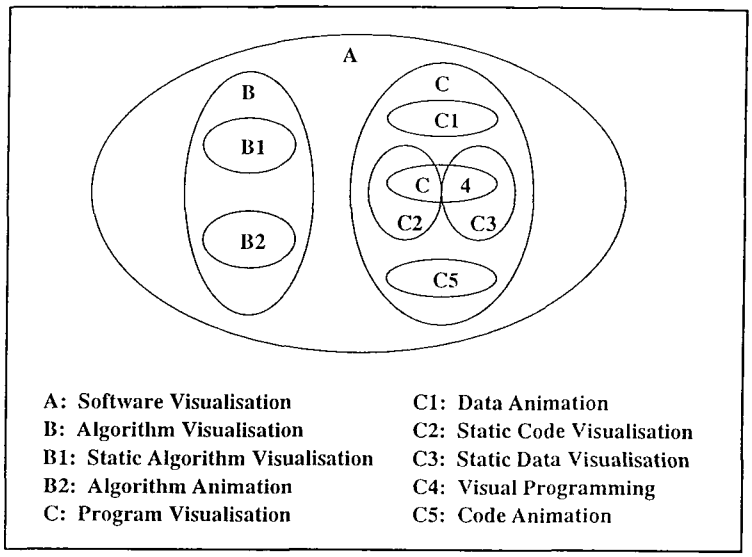

Figure 2-1 A Venn diagram showing the relationships among the terms

Price *et al.* try to clarify the confusion by proposing the model as shown in Figure 2-1. They suggest using the term Software Visualisation to encompass all the activities. They define Software Visualisation as the use of the crafts of typography, graphic design, animation and cinematography together with modern human-computer interaction technology to facilitate the understanding of software systems.

V Survey of Program Visualisation Systems

A number of taxonomies on Program Visualisation have been carried out over the years. Most of them try to identify the characteristics of the visualisation systems and classify them into different categories.

In her book [Shu88], Shu focuses on the increasing degree of sophistication exhibited by Program Visualisation systems ranging from pretty-printing to complex algorithm animation. Myers [Myer90] proposes to classify the systems along two axes: whether they illustrate the code, data or algorithm of the program, and whether they are dynamic or static. Stasko and Patterson [Stas92] introduce scaled dimensions in their four-category scheme covering Aspect, Abstractness, Animation and Automation. Price *et al.* [Pric93] try to categorise the systems in a systematic way. They establish a taxonomy hierarchy so that the taxonomy can be expanded and revised. The taxonomy comprises six basic categories: Scope, Content, Form, Method, Interaction and Effectiveness. Roman and Cox [Roma93] emphasise that their model of visualisation is based on formally well-understood areas. Their model is a mapping that leads to a classification of systems based on the Scope, Abstraction, Specification method, Interface and Presentation of the systems.

The following is a survey of some Program Visualisation systems. This includes systems that are of historic importance and systems that illustrate a diversity of approaches to Program Visualisation.

A Sorting Out Sorting

Sorting Out Sorting [Baec81] is the first major software visualisation work (data visualisation) of the 1980s. It is a 30-minute video which uses animated computer graphics to explain how nine different sorting algorithms manipulate their data.

Sorting Out Sorting begins by introducing the concept of sorting data and goes on to explain the nine sorting algorithms, namely linear insertion, binary insertion, shell sort, bubble sort, shaker sort, quicksort, straight selection, tree selection and heap sort. It shows a race of all nine algorithms running in parallel on large data sets at the end.

B BALSA

BALSA, which stands for *B*rown *U*niversity *A*lgorithm *S*imulator and *A*nimator, is the first major interactive software visualisation (both data and algorithm visualisation) system [Brow84, Brow85]. BALSA is written in the C programming language but the algorithms it animates are in Pascal. Figure2-2 shows a screen from the system BALSA.

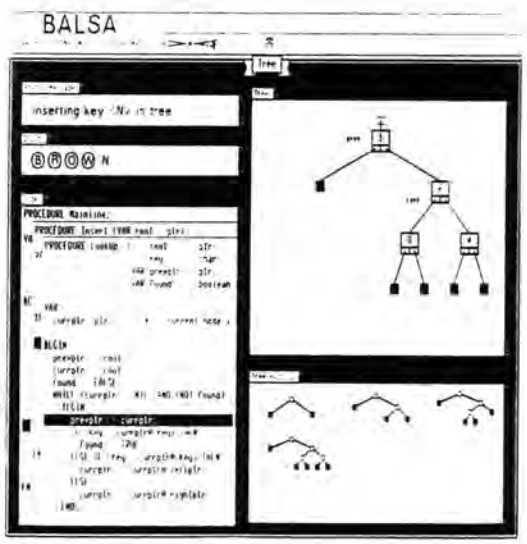

Figure 2-2 A screen showing the code view, input, status message and graphical representations of a data structure

BALSA takes the advantage of windowing techniques and large screen graphics to support multiple simultaneous views of the running algorithms. It can display the multiple views of the same data

structure. It is the first system that can show algorithms racing with each other in the same display. BALSA also provides a code view showing the pretty-printed listing of the current function.

C VIFOR

VIFOR, which stands for *V*isual *I*nteractive *FOR*tran, is a software tool oriented towards the maintenance of medium-to-large Fortran 77 programs [Rajl90, Rajl96]. The tool itself is implemented in the C programming language.

Within VIFOR, programs can be displayed in a code and/or a graphical representations. It also provides transformation in both directions, from code to graph and from graph to skeletons of code. An abstraction facility is available for discarding unrelated information. Rajlich *et al.* [Rajl90] believe that VIFOR can be of use in both Maintenance and Re-engineering activities. Figure 2-3 shows a screen shot of VIFOR.

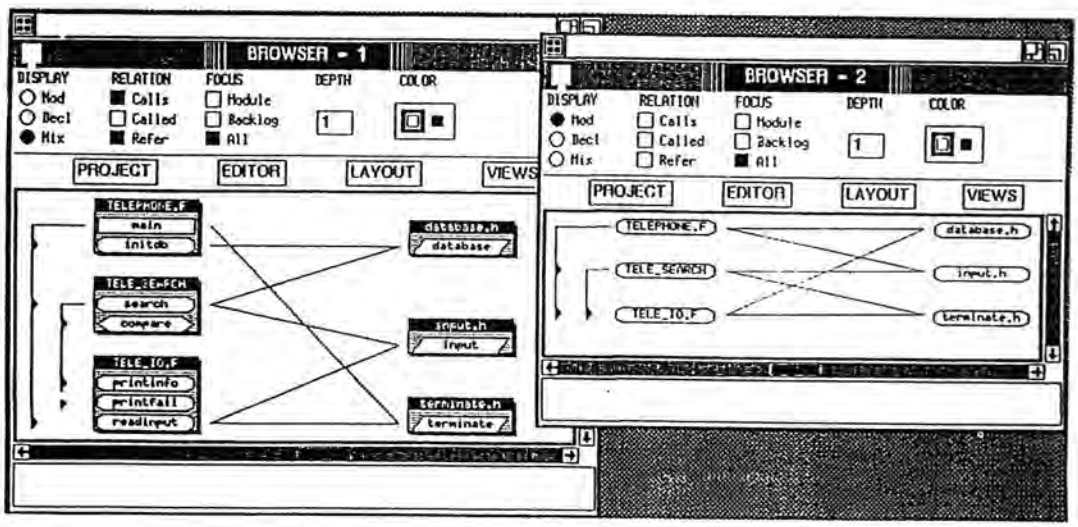

Figure 2-3 A screen shot of VIFOR

In the graphical form, a program is represented by a graph consisting of icons and lines between these icons. The two-column graph as shown in Figure 2-3 is an original layout that was specifically developed for VIFOR. The left column consists of processes (main program, subroutines, and functions), and the right column consists of commons (global data elements). Arrows on the left represent the call relations among the processes (the call graph). The lines in between the two columns represent the reference relations (the reference graph). This is an attempt to try to combine the function call relationship and the data dependency in a single representation.

D Dependency Analysis Tool

The Dependency Analysis Tool is developed to capture and analyse the program dependencies from C programs [Oman90a, Wild91].

Wilde and Huitt [Wild91] maintain that the use of dependency graphs is an advantage because:

- users of the toolset can acquire the information they need without listing all the dependencies surrounding their enquiries
- the dependency graph view is not bound by any language or environment
- indirect dependency can be found easily
- false dependency can be filtered out

The tool uses the concept of a dependency graph as a basic abstraction to simplify the understanding of program relationships of which definitional, calling, functional and data flow dependencies are analysed. Wilde and Huitt believe that this toolset can be either used directly or it can be used to provide a base for constructing other maintenance aids.

E CARE

CARE, which stands for *C*omputer-*A*ided *RE*-engineering, is a software tool that attempts to facilitate the comprehension of C programs [Lino93, Lino94]. The tool itself is implemented in the C programming language.

This code visualisation tool uses windows and browsers to display the data flow and the hierarchy control flow of the C programs. CARE maintains a repository of structural and functional dependencies for programs. Visualisation of such dependencies is accomplished by using a presentation model which combines the data flow (called colonnade graphs) and the control flow (the call graphs) information. A colonnade is an extension of the two-column display used by VIFOR and it has been formally defined as a m-column graph. CARE also emphasises on the additional facilities it provides: the partitioning (abstraction) techniques and the transformation mechanism.

Within the environment, a user can obtain either the colonnade representation of the data flow or the hierarchy representation of the control flow from the source code of a program. The reverse operations are also supported. In addition, colonnade graphs can be transformed into call graphs or vice versa. Graphical or textual slices can also be created from these representations. A summary is shown in Figure 2-4.

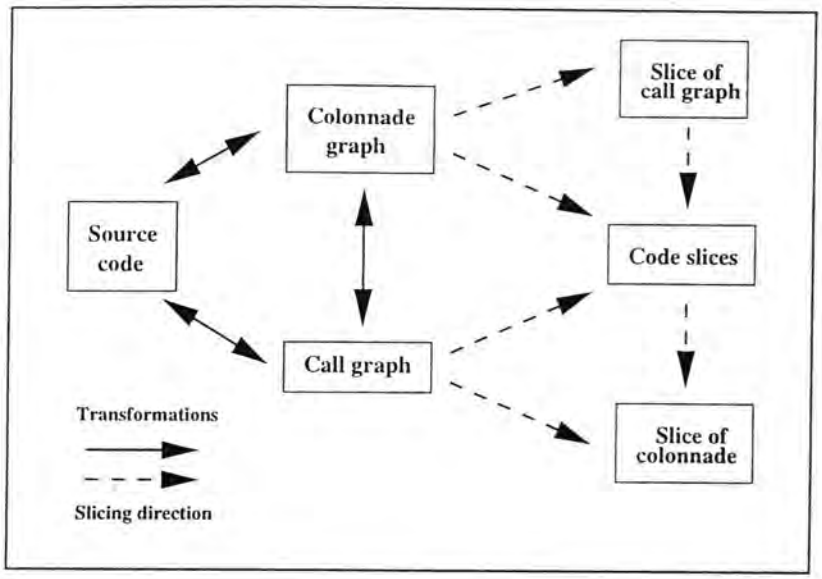

Figure 2-4 The transformation and slicing mechanism provided by CARE

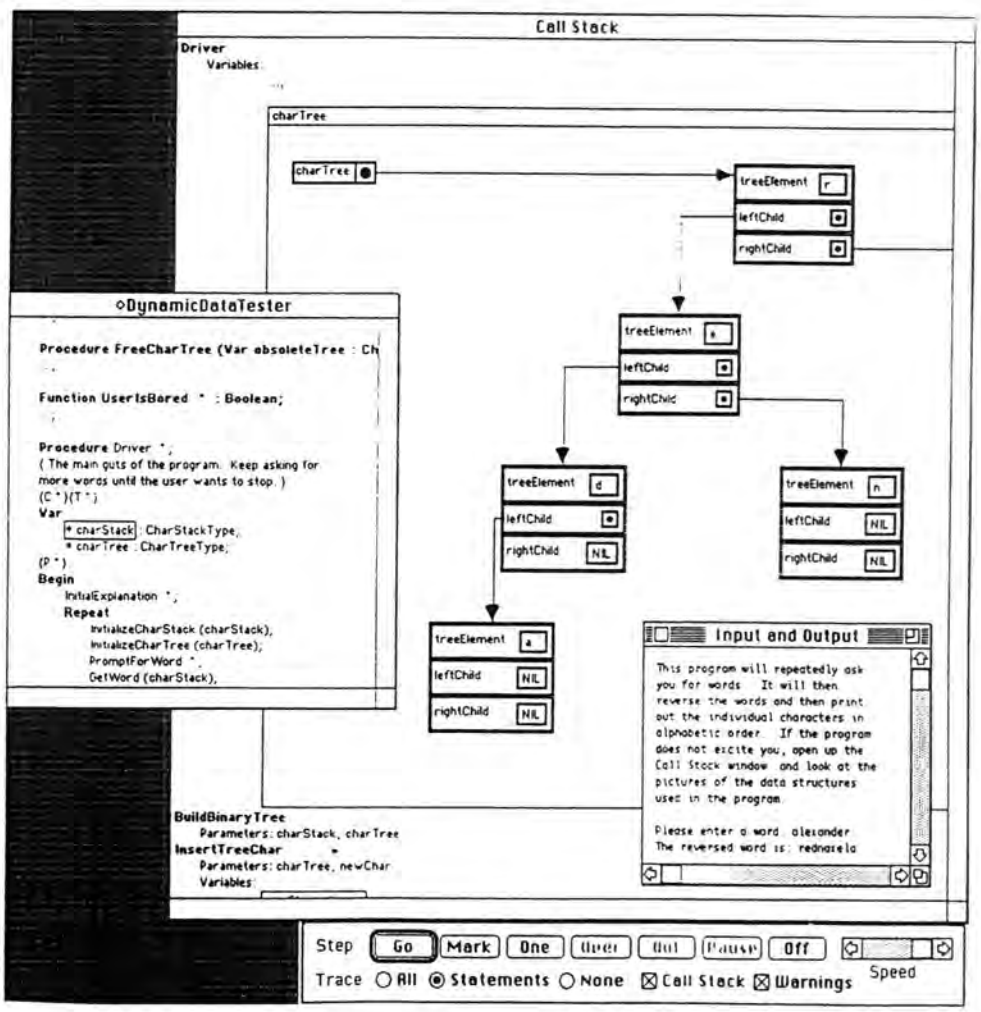

Figure 2-5 A snapshot of Pascal Genie running a program

F Pascal Genie

Pascal Genie [Chan91] is a system designed to create graphical displays of program data structures. It provides displays for the simple data types as well as the composite data types (records, arrays and pointers). Figure 2-5 shows a snapshot of the Pascal Genie running a program. The source code window on the left shows the currently executing line highlighted with several function bodies elided. The large window in the background is the call stack showing all of the data on the stack. Some variables are elided completely, some are shown by their name only, and the variable **charTree** is shown fully expanded with an automatically-generated binary tree showing the data. The program's input and output appear in the window near the bottom right and the execution control panel appears at the bottom.

G SHriMP Views

SHriMP, which stands for *S*imple *Hi*erar*chi*cal *M*ulti-*P*erspective, is a visualisation technique introduced by Storey and Müller [Stor95]. In the paper, they describe a technique for visualising software structures which are modelled as nested graphs, together with the use of fisheye views. Nested graphs are used for visualising the structure and organisation of a program, whereas the fisheye views emphasise detail of current interest within the context of the overall program structure. The fisheye view algorithm works by selectively enlarging sets of nodes within an area of interest while simultaneously shrinking the rest of the graph. The authors argue that when visualising a large amount of information, it is important to be able to create different views of the information where each one provides a different perspective. They believe that this can be achieved by SHriMP which provides a mechanism to create views that can show multiple perspectives concurrently.

H The McCabe Tool Set

The McCabe Tools[1] include tools for software and design validation, code comprehension and tools for producing measurements and metrics for the software systems. The focal point of the McCabe Toolset is the BattleMap Analysis Tool[2] (BAT) which provides a description of the analysis of the structure of a program and the flow of control (control flow graphs) within its corresponding parts. Figure 2-6 shows a screen shot from the McCabe Tools.

A BattleMap shows the calling relationships between all of its modules. Other toolsets which can be invoked from BAT including tools which produce various complexity metrics, provide analysis of the dynamic behaviour of code in a testing environment and tools for aiding the understanding of the software's internal architectures.

[1] The McCabe Tools is a registered trademark of McCabe Associates.
[2] BattleMap Analysis Tool is a registered trademark of McCabe Associates.

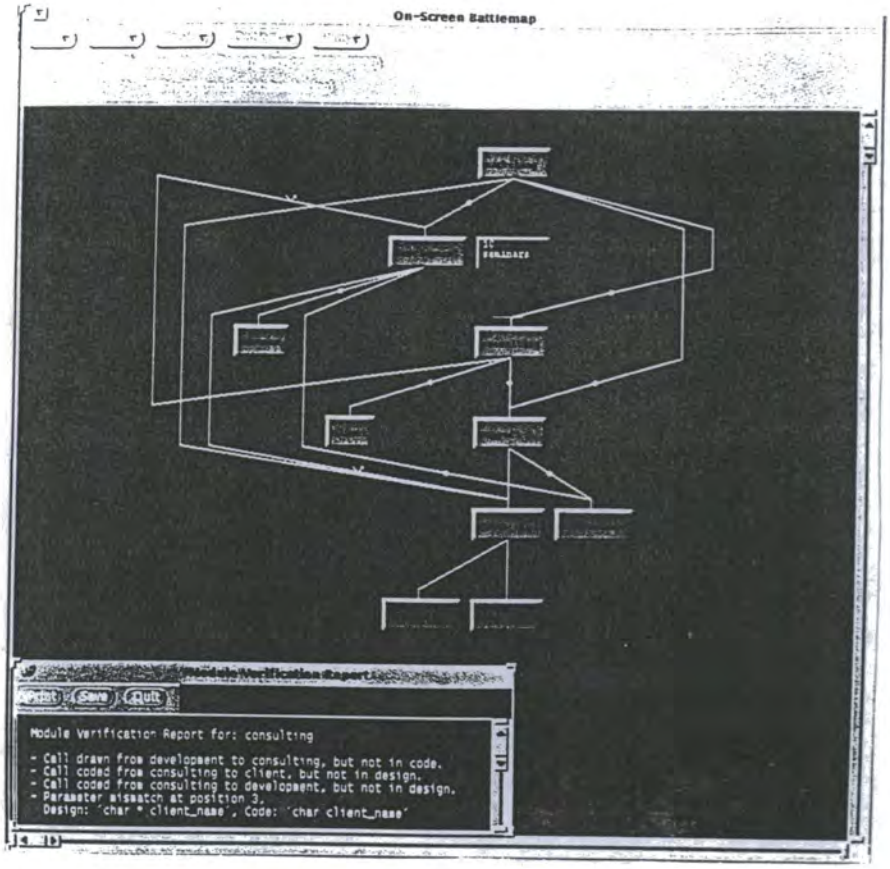

Figure 2-6 A screen showing the running of the McCabe tools

I Logiscope

Logiscope[3] is a code visualisation tool. It is a complete CASE tool which can be used from the creation of the source code to the end of the life cycle of a system. It can perform static analysis and limited dynamic analysis of programs. Special provision is provided for the mappings from the call graphs and the control flow graphs to the exact locations of the definitions of modules and functions, and to the declaration parts of data. Logiscope provides not only the usual analysis of programs but it also provides suggestions on the structures of the modules and the component parts (functions). If a relative threshold of a particular metric has been crossed, Logiscope will suggest and display a list of components which require restructuring, subdividing or rewriting.

J SNiFF+

SNiFF+[4] is another CASE tool designed for the development of C and C++ software systems. The main objective is to create an environment which makes it possible to edit and browse through large software systems textually and graphically. A running version of SNiFF+ consists of two operating system processes: the information extractor and the programming environment. The information extractor is used to extract information about definitions and declarations from the source code; the programming environment consists of a number of tools that are organised around a kernel consisting of a symbol table and a project manager. Among the many functions provided by the information

[3] Logiscope is released by Verilog.
[4] SNiFF+ is released by TakeFive Software.

extractor, there is a Class Browser and a Hierarchy Browser. The Class Browser can be used to browse through locally defined and inherited elements of a class whereas the Hierarchy Browser displays the inheritance hierarchy.

K Code Measurement Tool and Code Monitor

Code Measurement Tool[5] or CMT is a tool which collects information from the beginning of a software project and builds up a project portfolio for that particular software. Information such as the output from different metrics, the lists of changes made through various releases of the software and the development costs is stored in a knowledge base. The 'quality' of the different releases of a software can then be compared using the outputs from different metrics. CMT can also extract the information it requires from the knowledge base to produce some measures on the development and maintenance costs using different models, such as the COCOMO model [Boeh81]. A more ambitious goal of CMT is to 'train' CMT to 'learn' the history and information available in the knowledge base using a neural network. This is based on the idea that if any recurring patterns or trends can be detected, then predictions on the costs and quality of the software which is under development can be made. Code Monitor is the front end of CMT. It has a window interface which allows a user to pick up various aspects of information about the software at various levels of abstraction.

2.4 Summary

Most of the theories and models of Program Comprehension discussed in section 2.2 are inferred from the results obtained from observational studies, where typically, programmers are given a task to complete within a time limit. In some studies, the programmers were tested against their understanding at the end of the task whereas in other cases, they were encouraged to think out loud so that their thoughts could be recorded. Despite the diversities on the theories and models of Program Comprehension, they all possess a set of similar concepts:

- Program Comprehension is an assimilation process. A better understanding of a program can be built from a knowledge base which consists of a varieties of knowledge.
- The process of Program Comprehension is complicated and the behavioural patterns of the maintainers correspond to the type of maintenance activities they are engaged in.
- For large software applications, there will be a need to modify/augment the strategies to suit particular needs.

[5] Code Measurement Tool is developed by British Telecommunications.

- Maintainers should record their understanding of the programs for the benefit of future maintainers/developers.

The advancing power of computers have made it possible to manipulate larger and larger amounts of information but humans are cognitively ill-suited for understanding the resulting complexity. All information is readily available but users are unable to efficiently access individual items or maintain a global context of how the information fits together.

Although visualisation is often associated with the colourful representations of exotic scientific phenomena that frequently appear on the covers of magazines, it is important to recognise that visualisation can be usefully applied to the most prosaic data. The goal of visualisation is to represent data in ways that make them perceptible, and thus able to engage the human sensory systems.

The central problem to be addressed is what can be done when there is just too much information to deal with. With some collections of information the traditional node-link graphical structure can be used, but for modern real-world problems, which require users to understand large collections of information, solutions must be found for managing the large amounts of complex information. This problem can be decomposed into three subproblems:

- how to make a meaningful visualisation of a single object
- how to make a meaningful visualisation of a collection of objects
- how to allow the users to control the selection of the visualisation efficiently

In the same vein, programs are complex and abstract objects which include many components with many different attributes that are interrelated in complicated ways. Maintainers may find it difficult to understand and navigate through these complex interrelationships among different parts.

One way to tackle this problem is to decompose the program into smaller components so as to scale down the complexity to a manageable limit. Ideally, these components should group related concepts and functions together. These components can then be analysed in turn and a deeper understanding of a program can be built upon successive examinations. The understanding is then gradually assimilated in the mental model which resides in the mind of the maintainer.

It can be argued that a maintainer does not need to have a thorough understanding of the program structure before commencing a modification [Litt86, Shne79]. He only has to concentrate on the areas where modifications are to be made and other areas which will be affected by these modifications. Nevertheless, even when the program has been decomposed into smaller components, the resulting textual representation may not always reveal the interrelations straight away especially when the

important partitions and relations such as program architecture, are scattered in large amounts of local information [Leto86b].

Visualisation of programs can be an important step towards the right direction. The ultimate goal of Program Visualisation is to help maintainers form clear and correct mental images of a program's structure and functions. Graphical representations are useful in that they are easy to understand and to manipulate. These representations can convey the abstract links and structures of the source code in a relatively simple form. The information is presented in a form such that there is little room for misleading interpretations which means the level of confusion and ambiguity caused by communication can be minimised.

From the survey of Program Visualisation systems in section 2.3.5, it can be seen that the first major effort in building software visualisation packages was concentrated on exposing the inner workings of commonly used algorithms in the software systems. Packages such as Sorting out Sorting and BALSA are of highly historical importance. Both of the packages made use of visual cues so that the essence of the algorithms could be captured into visual forms.

After the success of Sorting out Sorting, the work of visualisation was expanded and extended to the form of data visualisation. BALSA, Pascal Genie and SNiFF+ are among the ones which support the visualisation of program data structures. Programmers have been using simple debuggers, and sometimes debuggers with visual aid, to keep track of the various states of simple data structures. Obviously these data visualisation packages suit the purpose of a debugger perfectly but they may be perceived as far more sophisticated to be used as debuggers. Take SNiFF+ for example, it is a complete CASE tool designed for the development of Object-oriented software systems.

The work of software visualisation also branched into code visualisation at around the same time. Static analysis tools for different languages have been built and most of the output for these analysis tools is displayed graphically. There are a number of program relations which can be extracted from a program. The function calls and the control flow relationships are the most frequently used.

Most of the code visualisation tools only provide a simple view of the software system with the rest of the program information presented as text. However, some researchers have begun to explore the possibilities of combining and linking simple relations together in the same environment. Systems such as VIFOR, CARE, the McCabe tool set and Logiscope are examples of software packages which support multiple views of source code. However, they are not based on any complete analysis of the relationships between the elements of programming languages. They represent some useful relationships derived in an ad-hoc way but they do not show any of the attributes associated with the program constructs and relationships.

Study has shown that maintainers often want more information than is currently available on the display but they are not sure what exactly would be most helpful [Shne86]. The ability to provide different viewpoints on a same object, whether its a file, a function or a variable, is important because it can provide various levels of detail about the object at different stages. A visualisation system which can integrate and support a variety of program relationships is therefore much desired.

Early work on building the software maintenance tools was based on the use of simple relations of function calls and control flow, such as the work carried out by Foster [Fost87] and Fletton [Flet88]. As programs grow in size and complexity, the gap between the types of information required by the maintainers and the amount of information which can be provided by the maintenance tools widens. It is shown in Table 1 (Chapter Four) that function calls and control flow are not the only relationships present in a program. By allowing the other program relationships to be brought into the scene, maintainers will be able to get access to information in a wider spectrum and in a more consistent way.

Chapter Three

A Framework for Evaluation

3.1 Introduction

This chapter describes a framework for the evaluation of the Integrated Approach outlined in Chapter Four, the implementation outlined in Chapter Five and the Case Studies outlined in Chapter Six. The first part of the chapter explores the use of research methods such as Surveys, Formal Experiments and Case Studies. The second part of this chapter describes a set of objective criteria for the evaluation. The set of criteria is divided into two branches. The first branch is intended to capture the processes of the various comprehension theories such as the top-down, bottom-up and a mixture of both approaches. The other branch addresses the cognitive issues of a maintainer while he browses and navigates the visualisation of the program structures.

3.2 Research Methods

In order to evaluate a piece of research, a new technique or technology, the impact on the related processes and the environment that it is intended for operating in must be thoroughly investigated before it can be put into practice. There are three commonly used evaluation methods: Surveys, Formal Experiments and Case Studies [Kitc95, Pfle94]. Surveys are usually conducted *after* the application of particular techniques or technologies which span across a number of projects and organisations, whereas the purpose of Formal Experiments and Case Studies is to assess the use of the technique or technology *before* it is put into practice. Formal Experiments are based on scientific investigations which aim to provide an understanding of the processes and to expose any underlying assumption that the research, technique or technology is based on. Case Studies, on the other hand, can provide powerful and informative insights but they are less rigorous than Formal Experiments.

The choice of selecting the appropriate evaluation method depends largely on the scale and the nature of the research, technique or technology concerned. The technique of Surveys is often used when the

investigation is spanned across a large number of projects or organisations. Surveys attempt to observe and systematically characterise the techniques or technologies used over a number of projects.

Formal Experiments are sometimes difficult to conduct when the degree of control is limited. In addition, they require considerate effort in the planning, preparation and replication of experiments. The sample and the design of experiments must be carefully chosen in order to minimise the effect of confounding factors. The cost of setting up Formal Experiments are generally higher than that of Case Studies [Pfle94]. An appropriate degree of replication of experiments is required in order to attain reliable results.

Case Studies are different from Formal Experiments in several ways. They are easier to plan and organise than Formal Experiments. This implies that they cannot achieve the scientific rigor of Formal Experiments and are cheaper to set-up. Case Studies are often associated with a particular situation or organisation. The results obtained are context dependent and thus are more difficult to generalise. Nevertheless, they can provide sufficient information which can be used to assess the suitability of the use of a technique or technology in a particular situation or environment.

The differences among the three research methods are important because the conclusions they yield at the end may be different for each case [Kitc95]. The results obtained from each of these methods must be evaluated against a set of objective measures in order to increase the creditability of the conclusions derived.

The technique of Case Studies is chosen in this thesis to demonstrate the major ideas of this research. The success of this research is measured against a set of objective criteria described in the following section. They are used to evaluate against the Integrated Approach to Program Comprehension outlined in Chapter Four, the implementation outlined in Chapter Five and the case studies outlined in Chapter Six.

3.3 Cognitive Design Elements for Software Exploration Tools

The Integrated Approach to comprehension will be evaluated against a hierarchy of cognitive design elements proposed by Storey *et al.* [Stor97a]. The authors describe a hierarchy of cognitive issues which can be used to guide the design of software exploration and comprehension tools. The design elements are organised into two branches: Improve Program Comprehension and Reduce the Maintainer's Cognitive Overhead. Figure 3-1 shows the hierarchy of cognitive design elements.

This hierarchy has two main branches. Under the branch Improve Program Comprehension, the intention is to capture the essential processes of the various comprehension strategies. This includes the cognitive design elements from E1 to E7. Under the branch Reduce the maintainer's cognitive overhead, it addresses the cognitive issues of a maintainer while he browses and navigates the visualisation of the program structures. This includes the cognitive design elements from E8 to E15.

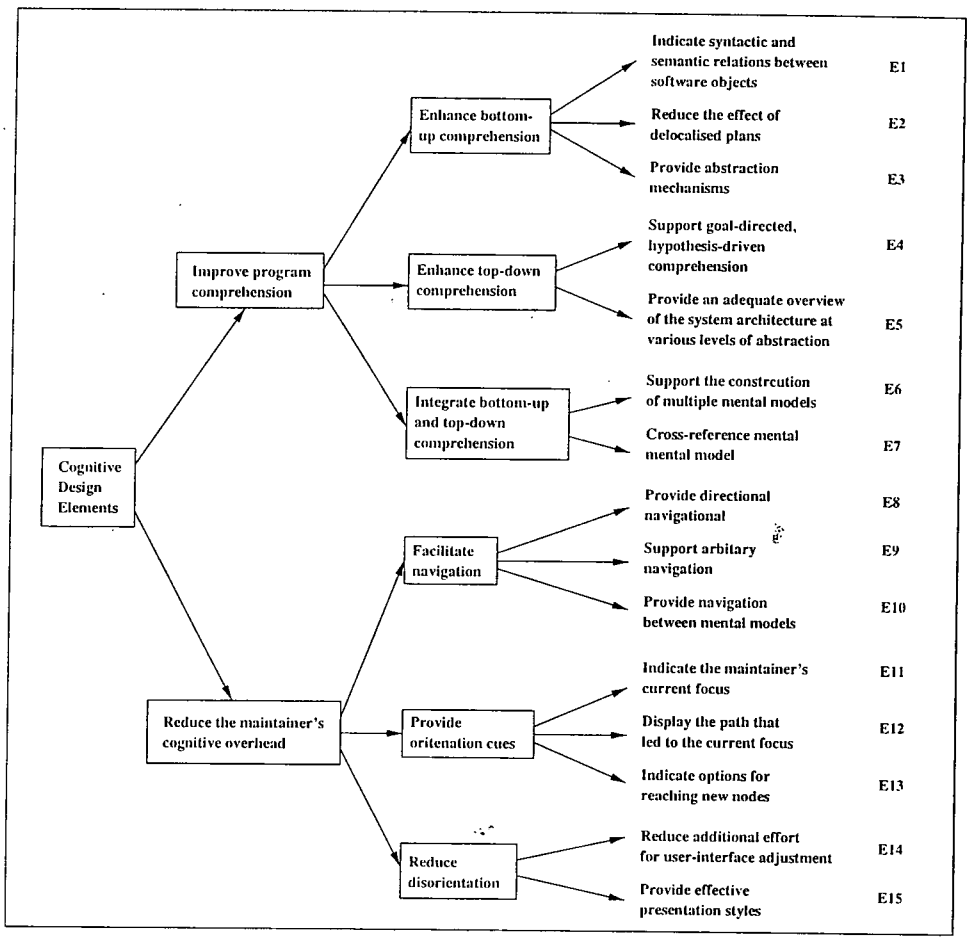

Figure 3-1 Cognitive design elements for software exploration

The hierarchy of the cognitive design issues is derived through an examination of the cognitive models of Program Comprehension. The following sections describe each of the cognitive issues in detail.

3.3.1 Improve Program Comprehension

It is argued that the comprehension model employed by a maintainer is dependent on a variety of issues governed by the experience of the maintainer and the type of maintenance activity he is engaged in [Mayr94, Mayr95]. It would be advantageous if a range of models are supported. The following is a list of cognitive design elements which are extracted from various comprehension models discussed in this paper [Stor97a].

I Enhance Bottom-up Comprehension

Storey *et al.* argue that a bottom-up comprehension involves reading program statements and chunking them into higher level abstractions. This is repeated until an overall understanding of a program is attained. This can be achieved by:

- identifying program units, such as variables, statements and functions, and the relationships between them
- browsing code in delocalised plans
- by building abstractions from lower level units

The following sections discuss each of the activities in detail.

E1 Indicate Syntactic and Semantic Relations between Software Objects

The authors suggest that the syntactic and semantic relationships are essential during a bottom-up comprehension. The syntactic relation can be derived from the source code by systematically identifying a set of program units. The semantic relation can be attained by an analysis of the relationships between these program units.

E2 Reduce the Effect of Delocalised Plans

A delocalised plan is a result of the fragmentation of source code related to a particular algorithm or a program plan. The authors argue that it can be cumbersome when reading fragments of code belonging to a delocalised plan. This activity may involve frequent switching between files which can lead to a feeling of disorientation. Techniques such as program slicing can be applied to identify the fragments of code which belong to a delocalised plan.

E3 Provide Abstraction Mechanisms

The authors believe that the process of building hierarchical abstractions from the low level program units and their relationships is the most difficult part during a bottom-up comprehension. A maintainer should be equipped with the capability to create higher levels of abstraction by systematically aggregating low level program units into higher level abstractions.

II Enhance Top-down Comprehension

The authors believe that a top-down comprehension requires application domain knowledge. A maintainer formulates hypotheses and examines the source code in a depth-first manner in order to verify their hypotheses. This can be achieved by:

- supporting the recording of hypotheses and linking them to relevant parts of the program, as well as supporting the refinement of hypotheses
- providing overviews of the program so that the maintainer can explore its structure in a top-down fashion

The following sections discuss each of the activities in detail.

E4 Support Goal-directed, Hypothesis-driven Comprehension

A maintainer should be equipped with the capability to create, record and relate the hypotheses which concern the properties of a program to relevant parts of a program. This information is valuable as it can be used to facilitate future maintenance.

E5 Provide an Adequate Overview of the System Architecture at Various Levels of Abstraction

Information regarding the software architecture should be provided at different levels of abstraction during the top-down comprehension so that the maintainer can systematically explore the program structures in a top-down fashion.

III Integrate Bottom-up and Top-down Approaches

The authors acknowledge that a maintainer will create and switch between various mental models during the course of comprehension [Mayr94, Mayr95]. They believe that relationships such as control flow, data flow and function abstractions are the keys to the creation of these mental models. These relationships are often illustrated by using graphical representations. The integration of the bottom-up and top-down approaches can be facilitated by supporting the construction and integration of various mental models (graphical representations).

E6 Support the Construction of Multiple Mental Models

The authors believe that the mental models created by one maintainer are likely to be different to the ones created by another maintainer. Support should be given for the construction of the mental models which represent various aspects of a program. The authors suggest that various mental models of a program may be represented by using both textual and graphical notations.

E7 Cross-reference Mental Model

The authors believe that a maintainer often switches from one mental to another during the course of comprehension [Mayr94, Mayr95]. This happens when a maintainer tries to cross-reference different mental models mentally. This activity can be facilitated by supporting the cross-referencing of the representations between various parts of the mental models (graphical representations).

3.3.2 Reduce the Maintainer's Cognitive Overhead

Storey *et al.* believe that when comprehending large software systems, the cognitive overheads imposed on a maintainer will increases rapidly. This problem can be alleviated by providing good navigation facilities, meaningful orientation cues and effective information presentations.

I Facilitate Navigation

When exploring large software systems, it is important that a maintainer is equipped with the facilities so that he can navigate through the vast amount of information with ease. The authors suggest that the navigation facilities should include mechanisms for browsing source code, program documentation, graphical views of program structures and documented mental models of the programs.

E8 Provide Directional Navigation

Directional navigation are the mechanisms for aiding the reading of source code and program documentation, the browsing of program relationships such as data flow and control flow and the traversing of program structures in a top-down fashion.

E9 Support Arbitrary Navigation

Arbitrary navigation should be supported when a maintainer navigates to locations that are not necessarily reachable by following direct links.

E10 Provide Navigation between Mental Models

The authors believe that to be able to navigate between the various mental models (graphical representations) smoothly is the key to a successful comprehension. They argue that this is a non-trivial problem as there may be one-to-many and many-to-one links from one model to another.

II Provide Orientation Cues

The authors suggest that orientation cues can be used to inform a maintainer of his whereabouts when exploring the program structures, how and why he is there and how to switch to a different focus when required.

E11 Indicate the Maintainer's Current Focus

During comprehension, a maintainer may need to access information relating to the many different program units. The maintainer may become 'lost' in that vast amount of information. The use of judicious orientation cues can be used to reinforce the maintainer's sense of focus and orientation.

E12 Display the Path that Led to the Current Focus

Recording why a maintainer is interested in a particular program unit may be very important. The reason for reading a piece of code may be the result of verifying a particular hypothesis or because the code must be modified in some way. The maintainer should be equipped with the facility which can display the sequence of actions and show how a particular decision is reached.

E13 Indicate Options for Reaching New Nodes

Support should be provided so that a maintainer is made aware of the facilities available for further exploration.

III Reduce Disorientation

When exploring a large information space, the problem of disorientation is a major issue. The authors suggest that disorientation can be alleviated by removing some of the unnecessary cognitive overheads resulting from poorly designed user interfaces, and by using specialised graphical views for presenting large amounts of information.

E14 Reduce Additional Effort for User-interface Adjustment

Extra effort should be made for the design of the user interfaces in order to reduce the cognitive overheads which can induced by switching between different mental models.

E15 Provide Effective Presentation Styles

For complex graphical representations, automatic layout algorithms are often used to display the representations in a more readable manner. Extra effort should be put into the layout of graphical representations and for the general presentation of information relating to various program units.

3.4 Summary

Although this hierarchy of cognitive design elements is orientated towards the design of software exploration tools, it is felt that the hierarchy is also suitable for the evaluation of this research.

It is decided that the cognitive issues from the first branch of the hierarchy (E1 to E7) are particularly applicable for the evaluation of the Integrated Approach and the rest of the cognitive issues from the

second branch (E8 to E15) are suitable for the evaluation of the prototype, PUI. The results of the Case Studies will also be evaluated against the hierarchy. The first branch addresses the theoretical issues of the comprehension theories which are closely related to the Integrated Approach, whereas the cognitive issues addressed in the second are more inclined to the evaluation of the interactions between the maintainer and the software exploration tool. An evaluation of the Integrated Approach, the prototype and the Case Studies will be presented in Chapter Seven.

Chapter Four

An Integrated Approach to Program Understanding

4.1 Introduction

This chapter introduces a framework and mechanism for the facilitation of the understanding of large software systems. In particular, it addresses the need for a more flexible approach to Program Comprehension and discusses the use of Program Relationships, rather than just those of function calls and control flow through carrying out a systematic analysis of Program Elements.

Maintainers are usually under pressure to accomplish maintenance tasks as quickly as possible. The problem for most maintainers is that they have to maintain unfamiliar code that has been modified and the accompanying documentation is usually out of date, inadequate, inconsistent or sometimes non-existent. More often than not, the source code may be the only information maintainers have got. The problem is how the maintainers find a systematic way to uncover this information.

4.2 Integrated Approach

The process of comprehension is a cognitive skill and therefore it is extremely difficult for machines to mimic human beings. It is widely acknowledged that a total automation of the comprehension process will not be feasible as human input and interpretation are vital to the process.

Studies have shown that experienced maintainers are better at using various comprehension strategies in order to direct their attention to areas which may contain crucial information about a program. A comparison can be drawn between master and novice chess players. Controlled psychological experiments have shown that chess masters are far more accurate than non-chess players at remembering chess board positions taken from real games, where the placement of pieces arose in

strategic play and represented meaningful tactical positions. These experiments have found that chess masters remember positions based on certain patterns, alignments and structures. Experience and knowledge accumulated over the years are the deciding factors in differentiating master chess players from novice chess players [Stor97b].

The memorisation of the arrangement of chess pieces is comparatively simple for the master chess players as there are plenty of visual cues. Maintainers, on the other hand, do not have as many visual cues available. The structure of a software system is arguably less defined and more abstract. Nevertheless, tools are available which can make the comprehension process a little simpler and smoother. The goal of software maintenance tools is to help the maintainers to form clear and correct mental images of the source code, and sometimes it is achieved with the help of software visualisation. Visualisation can provide alternative perspectives to textual information. Graphical representations are more compact than the textual representation and they resemble the mental models constructed by the maintainers. It is essential that maintainers are supplied with a range of visual cues (information with various degrees of details) in order to obtain better understanding of programs.

Each theory and model discussed in section 2.2 in Chapter 2 favours a different approach to Program Comprehension. Pennington's [Penn87] theory is a bottom up approach whereas Brooks [Broo83] and Littman *et al.* [Litt86] believe that comprehension should be performed in a top down fashion. Letovsky [Leto86a] and von Mayrhauser and Vans [Mayr94, Mayr95] argue that maintainers will use a mixture of both strategies depending on the cue of the additional information. The message is clear: there is no consensus on how maintainers understand programs and each of those theories can only model certain aspects of the maintainers' behaviour during comprehension. Further, the comprehension strategy used is also highly influenced both by the types and the goals of the maintenance activities that a maintainer is engaged in. Most of the maintenance tools are not powerful enough for use on a large scale as they only provide limited analysing power. What is needed is a software maintenance tool that can provide an environment which encompasses the essence of the different theories and models.

The Program Comprehension process can be roughly divided into two stages. Figure 4-1 shows a pictorial representation of this process. The first stage is information gathering. This is active when a maintainer tries to grasp an impression of the source code by glancing and wandering through the source code. It usually happens during the early stage of the comprehension process, though this activity can be repeated when the maintainer is in the latter stage of the process. The second stage is more directly geared towards specific problem solving. In this stage, the maintainer may actively reach out and look for cues and information regarding some program constructs such as a data type or a function. Often, the maintainer may need to explore new sections of code when he gets deeper and deeper into the area that he is analysing.

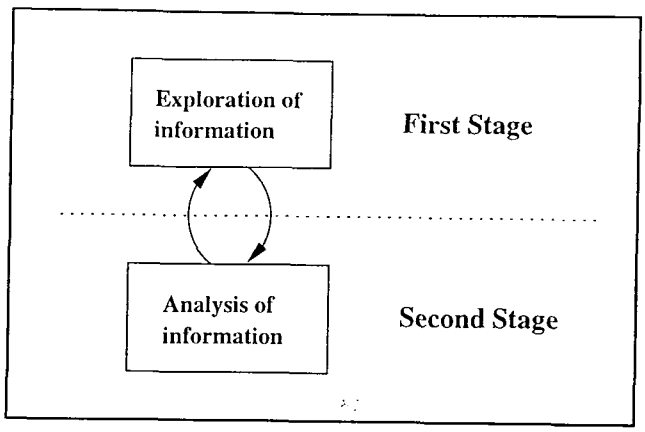

Figure 4-1 Two stages of the comprehension process

The two major aspects in the comprehension process are the exploration and the analysis of information. It can partly explain why most of the comprehension theories and models are inadequate in modelling the behavioural patterns of the maintainers. In order to capture both of the processes into one environment, a software maintenance tool needs to be flexible enough so that the maintainers can switch between the two processes when required. Moreover, the tool must provide a wide range of information about the source code to assist the maintainers in the analysis stage. This information should be managed and presented to the maintainers in a systematic and controllable way so that they will not be overloaded with too much information.

It is extremely difficult to contemplate exactly what kind of information a maintainer may need during the maintenance activities. The required information is largely dependent on the maintainer's experience, the types and the objectives of the maintenance tasks, as well as the Program Comprehension strategies used.

Maintenance activities can be broken down into four main categories [Lien78, Lien80]:

- **Perfective maintenance** involves implementing new functional or non-functional system requirements. These are generated by software customers as their organisation or business changes. Activities include understanding the system, diagnosing and defining requirements for improvements, developing preliminary and detailed perfective design, modifying program code, debugging and testing.

  For the Perfective maintenance, a maintainer needs to explore the relationships between the program code and the changes required as a result of the user requests

and/or business changes. For example, if the input data to a system is to be changed, a maintainer may have to look at the data definitions and structures used by the system, the variables that are associated with the data structures and the functions that are dealing with the input, output and manipulation of the data structures.

- **Corrective maintenance** involves the correction of processing, performance or implementation failures. It concerns bug fixing and correction of software errors. Activities include understanding the system, generating/evaluating hypotheses concerning problem, repairing code and testing.

  For the Corrective maintenance, a maintainer needs to understand, explore and assess the relationships between the program code and the software faults. He may have to examine the data flow relationships between variables, control flow relationships between statements and function call relationship between functions.

- **Adaptive maintenance** involves modifying the software in order to keep up with environmental changes. It may involve changes in hardware or data. It does not lead to changes in the system's functionality. Activities include understanding the system, defining adaptation requirements, developing preliminary and detailed adaptation design, modifying program code, debugging and testing.

  For the Adaptive maintenance, a maintainer needs to understand the impact of the program code regarding the environmental changes. In particular, special attention is required for dealing with the system interface and functions which utilise the built-in libraries provided by the hardware or the operating system.

- **Preventive maintenance** involves updating software in order to forestall future problems and to increase maintainability. Activities include understanding the system, defining lists of changes for improvement, modifying program code/system documentation, debugging and testing.

  For the Preventive maintenance, a maintainer needs to have knowledge about the program structures the and system architecture. It concerns updating documentation, adding comments and improving the modular structure of the system.

Information is required at different levels of abstraction ranging from high to low depending on the type of maintenance.

Another deciding factor influencing what kind of information is required for comprehension is the level of technical competence of the maintainers themselves. Experiments have shown that there are differences in how expert and novice programmers understand programs, and that both groups seek to look for different cues in the source code. This can be attributed to the different types of knowledge that a maintainer may possesses. The results of those experiments have shown that expert programmers often tend to conceptualise different areas of the source code and then map them to the application domain, whereas novice programmers tend to confine the comprehension process in the programming domain knowledge.

An obvious solution to get round this problem would be to develop specific tools which are geared towards the different types of software maintenance activities and for the different Program Comprehension theories and models. This solution is only feasible when it is certain that the type of the software maintenance activities is not to be changed regularly and that the comprehension process is carried out by following a particular theory.

Another way of tackling this problem involves explicitly exposing the interrelationships between the many program constructs within the source code. In theory, the source code itself should contain all the information a maintainer may need in order to obtain some degree of knowledge. Instead of anticipating and planning for the information that a maintainer may need, the attention is now focused on exposing the program relationships between the program constructs. The emphasis of the comprehension process is now on how the maintainers can make use of the information provided, rather than leaving them to chase for the elusive information themselves. This is the basis idea of the Integrated Approach.

The maintainers can make use of the information regarding the program constructs and relationships in order to expand or to refine their line of investigation as they see fit. This approach is realised by first identifying the program constructs and the interactions between them, and then setting up a framework to assist with the analysis of these program constructs and relationships. Relevant information about a particular program construct can be attained by examining related program constructs and program relationships.

The Integrated Approach does not impose any restriction on how the process of comprehension should be performed. On the contrary, it enables the utilisation of different comprehension theories and models. It is flexible and it allows comprehension to be conducted according to preferences of the maintainers. As described before, the use of a particular comprehension strategy alone may be insufficient. This approach allows the essence of the different strategies to be captured and performed

in a single environment. Maintainers can exploit the use of various strategies throughout the comprehension process as they examine the program constructs and relationships.

4.3 Program Elements and Program Relationships

Program Elements are program constructs used in a program. The grammar of a programming language governs the way these Program Elements are used. When assembled together, the Program Elements make up programs. The programs are in turn used as building blocks for larger software systems. This research is interested in the understanding of programs written in the C programming language [ANSI84, Kern78, Kern88]. Typically, a C program may include Program Elements such as identifiers, constants, variables, expressions, types, statements, functions and files. The inter-relationships which arise between these Program Elements are often simple and straightforward, but they can become complicated depending on how these Program Elements are used.

Various problems may arise over time as programs grow in size and complexity. Maintainers may find it difficult to understand and navigate through the complex interrelationships among the Program Elements. Nevertheless, these complex interrelationships and interdependencies can be untangled with ease if various Program Elements and Relationships are identified at an early stage. These Program Relationships may be used as a handle to tackle the task of comprehension. Most of the common problems found during comprehension are related to the confusion of different interrelationships. For example, variables which have different scopes and meanings but have the same name can sometimes cause havoc. By examining the Relationships between two Program Elements carefully, a more accurate picture about these elements can be established and it may lead to better understanding of the system as a whole.

A natural form of representing relationships is graphs. Examples such as call graphs and control flow graphs are frequently used to illustrate higher levels of abstraction of programs. At present, most of the effort has been concentrated in devising tools to support the analysis of mainly two Program Relationships:

- the calling relations which is between functions and functions
- the control flow relations which is between statements and statements

Disentangling the different relationships in a program efficiently is essential to the process of program understanding. The function calls and control flow relationships have gained a lot of attention because they are simple and intuitive. Undoubtedly, the analysis of these relationships can yield a substantial amount of information about the source code itself. For example, measurements such as the complexity of a piece of code can be obtained from analysing these two relationships. Nevertheless,

modern programming languages are not just built from the utilisation of the function calls and control flow relationships. Other Program Elements and Relationships present in the source code should arguably be of equal importance and they also hold important information about the source code. These other Program Elements and relations may have been overlooked as they are perceived as less informative. This information, however, holds the links which can bridge the gaps between the 'chunks' of knowledge acquired by just analysing the control flow graphs and the call graphs.

A far more informative overview of a program can be attained if various program relationships between program constructs are supported and brought into play. Table 1 shows the relationships which may be present in a C program. The table should be read from left to right, and from top to bottom. For instance, the relation between an **Argument** and an **Identifier** is *has an*, and it should be read as **Argument** *has an* **Identifier**. On the whole, the table possesses a high degree of symmetry with a few exceptions. The following is an explanation of the terminology used and a discussion of all the Program Relationships between pairs of Program Elements shown in Table 1.

4.3.1 Glossary

I The Program Elements

The main Program Elements in the C language are as follows.

Identifier is a name associated with **Constant**, **Variable**, **Argument**, **Function** and **File**.

Constant is a storage unit where data is stored and will remain unchanged throughout the execution of a program. It includes numeric constants, character constants, string constants and enumeration constants.

Variable is a storage unit where data is stored. It can be changed by other Program Elements during its lifetime.

Argument is the parameter (formal/actual) passed to a **Function**.

Expression is a symbolic representation of a mathematical or logical statement.

Primitive Type includes `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`, `unsigned`, `enum`, `pointer` (`*`) and `array` (`[]`).

Complex Type includes `struct` and `union`.

Statement is a coded instruction which the program can recognise and carry out.

Block includes a list of declarations followed by **Statements**.

Function is a sequence of **Statements** that are grouped together to perform certain tasks.

File includes a collection of declarations and/or definitions.

II The Program Relationships

The main Program Relationships in the C language are as follows. The relations referred to here are those given in Table I with appropriate tense changes. The relations can also be both active and passive.

To *Associate* a Program Element *with* another indicates that they are connected in some way.

To *Call* a Program Element indicates that the flow of control is passed from another Program Element onto that Program Element.

To *Coerce* a Program Element to another involves explicit/implicit type conversion.

To *Contain* a Program Element indicates that the element is part of the definition of another.

To *Declare* a Program Element indicates that it is introduced to the program for the first time in accordance with the rule of scope.

To *Define* a Program Element indicates that it has been assigned a value or a full definition.

To *Depend on* a Program Element indicates that the value one Program Element is directly linked to affected by that element.

To *Follow* a Program Element indicates the presence of ordering.

To *Have* a Program Element indicates that one Program Element must possess another to complete a definition.

To *Have I/O interface with* a Program Element indicates that one Program Element communicates with another by way of exchanging information.

To *Have the same interface as* indicates that one Program Element possesses the same parameter declaration as another.

To *Have the same name as* indicates that the names of two Program Elements are identical.

To *Have the same type as* indicates that the types of two Program Elements are identical.

To *Have the same value as* indicates that the values held by two Program Elements are identical.

To *Import* a Program Element by a file indicates that its declaration (and/or the definition) is copied and incorporated into that file.

To *Return* a Program Element indicates that a value and it's associated type is assigned to a memory location upon the completion of the instructions.

To *Use* a Program Element indicates that it is involved in the definition of another Program Element.

The type of a Program Element *is compatible with* another indicates that the types are interchangeable.

Table 1 Program Relationships between Program Elements

	Identifier	Constant	Variable
Identifier	has the same name as	is associated with	is associated with
Constant	has an [may have an]	has the same type as, has the same value as	has the same type as
Variable	has an	has the same type as, is dependent on	has the same type as, is dependent on
Argument	has an	is a	is a
Expression	uses	uses	uses
Primitive Type	is associated with	is associated with	is associated with
Complex Type	is associated with	is associated with	is associated with
Statement	uses	declares, uses	declares, defines, uses
Block	uses	uses	uses
Function	has an, uses	uses	uses
File	has an, uses	uses	uses

	Argument	Expression	Primitive Type
Identifier	is associated with	is used in	is associated with
Constant	is used as	is used in	is associated with
Variable	is declared as, is used as	is used in	has a
Argument	has the same type as	is an	has a
Expression	is used as	uses, is used in	has a
Primitive Type	is associated with	is associated with, is used in	is coerced to, is compatible with
Complex Type	is associated with	N/A	uses
Statement	declares	uses	has a, declares, uses
Block	uses	uses	uses
Function	uses	uses	uses, returns
File	uses	uses	uses

	Complex Type	Statement	Block
Identifier	Is associated with	is declared in, is used in	is used in
Constant	Is associated with	is declared in, is used in	is declared in, is used in
Variable	has a	is declared in, is defined in, is used in	is declared in, is defined in, is used in
Argument	has a	is declared in	is used in
Expression	has a	is used in	is used in
Primitive Type	is used in	is associated with, is declared in, is used in	is declared in, is used in
Complex Type	is compatible with	is declared in, is used in	is declared in, is used in
Statement	declares, uses	follows, is followed by	is used in
Block	uses	contains	follows, followed by
Function	uses, returns	contains	contains
File	uses	contains	contains

	Function	File
Identifier	is declared in, is used in	is declared in, is used in
Constant	is declared in, is used in	is declared in, is used in
Variable	is declared in, is defined in, is used in	is declared in, is defined in, is used in
Argument	is declared in, is used in	is declared in, is used in
Expression	is used in	is used in
Primitive Type	is declared in, is used in	is declared in, is used in
Complex Type	is declared in, is used in	is declared in, is used in
Statement	declares, defines, is used in	is used in
Block	is used in	is used in
Function	calls, is called by, has the same interface as	is declared in, is defined in, is used in
File	contains, uses	imports, is imported by, has I/O interface with

4.3.2 The Table of Program Relationships

I Identifier

An **Identifier** is used to give a Program Element a name.

An **Identifier** *has the same name as* {[another] **Identifier**}
An **Identifier** *is associated with* {**Constant, Variable, Argument, Primitive Type, Complex Type**}
An **Identifier** *is used in* {**Expression, Statement, Block, Function, File**}
An **Identifier** *is declared in* {**Statement, Function, File**}

II Constant

A **Constant** is a storage unit where data is stored and will remain unchanged throughout the execution of a program.

A **Constant** *has an [may have an]* {**Identifier**}
A **Constant** *has the same type as* {**Constant, Variable**}
A **Constant** *has the same value as* {[another] **Constant**}
A **Constant** *is used as* {**Argument**}
A **Constant** *is used in* {**Expression, Statement, Block, Function, File**}
A **Constant** *is associated with* {**Primitive Type, Complex Type**}
A **Constant** *is declared in* {**Statement, Block, Functions, File**}

By definition, a constant can be a numeric constant (**Primitive Type `int/float`**), character constant (**Primitive Type `char`**), string constant (**Primitive Type `array of char`**) and enumeration constant (**Primitive Type `int`**).

III Variable

A **Variable** is a storage unit where data is stored. The data which it holds can be changed during its lifetime.

A **Variable** *has an/a* {**Identifier, Primitive Type, Complex Type**}
A **Variable** *has the same type as* {**Constant**, [another] **Variable**}
A **Variable** *is dependent on* {**Constant, Variable**}
A **Variable** *is declared as* {**Argument**}
A **Variable** *is used as* {**Argument**}
A **Variable** *is used in* {**Expression, Statement, Block, Function, File**}
A **Variable** *is declared in* {**Statement, Block, Function, File**}
A **Variable** *is defined in* {**Statement, Block, Function, File**}

Example:
```
int   x, y, z;
x = y + 2 * z;
```

It can be deduced from the above example that **Variable `x`:**

- *has an* **Identifier**
- *has a* **Primitive Type `int`;**
- *has the same type as* the **Constant `2`**
- *has the same type as* **Variables `y`** and **`z`**
- *is dependent on* the **Constant `2`**, **Variables `y`** and **`z`**
- *is declared in* the **Statement `int x, y ,z;`**
- *is used in* the **Statement `int x, y ,z;`**
- *is used in* the **Expression `x = y + 2 * z`**
- *is defined in* the **Statement `x = y + 2 * z;`**
- *is used in* the **Statement `x = y + 2 * z;`**

IV Argument

An **Argument** is the parameter passed to a **Function**. The Argument can be either formal at the point of declaration or actual at the point of function call.

An **Argument** *has an/a* {**Identifier**, **Primitive Type**, **Complex Type**}
An **Argument** *is a/an* {**Constant**, **Variable**, **Expression**}
An **Argument** *has the same type as* {[another] **Argument**}
An **Argument** *is declared in* {**Statement**, **Function**, **File**}
An **Argument** *is defined in* {**Statement**, **Function**, **File**}
An **Argument** *is used in* {**Block**, **Function**, **File**}

Example:
```
printf("pi = %f\n", 22/7);
```

It can be deduced from the above example that the **Argument `22/7`**:

- *is an* **Expression**
- *has a* **Primitive Type `float`**
- *is used in* the **Statement `printf("pi = %f\n", 22/7);`**
- *is used in* the **Function `printf`**

Expressions can be used as actual arguments as illustrated in the above example.

V Expression

An **Expression** is a symbolic representation of a mathematical or logical statement.

An **Expression** *uses* {**Identifier**, **Constant**, **Variable**, [another] **Expression**}
An **Expression** *is used as* {**Argument**}
An **Expression** *is used in* {[another] **Expression**, **Statement**, **Block**, **Function**, **File**}
An **Expression** *has a* {**Primitive Type**, **Complex Type**}

As every variable and constant is associated with a **Type** whether it is **Primitive** or **Complex**, an expression which comprises constants, variables and operators should also have a **Type**.

Example:
```
int   x, y;
.............
....(x = y * 3) ....
....(x = y / 3) ....
```

It can be deduced from the above example that:

- the **Expression** `y * 3` *uses* a **Constant** `3`
- the **Expression** `y * 3` *uses* a **Variable** `y`
- the **Expression** `y * 3` *has a* **Primitive Type** `int`
- the **Expression** `y * 3` *is used in* the **Expression** `(x = y * 3)`
- the **Expression** `(x = y * 3)` *has a* **Primitive Type** `int`
- the **Expression** `y / 3` *uses* a **Constant** `3`
- the **Expression** `y / 3` *uses* a **Variable** `y`
- the **Expression** `y / 3` *is used in* the **Expression** `(x = y / 3)`
- the **Expression** `y / 3` *has a* **Primitive Type** `float`
- the **Expression** `(x = y / 3)` *has a* **Primitive Type** `int`

An **Expression** which has different **Primitive Types** for each operand will automatically converted the lower precision **Primitive Type** into a higher precision **Primitive Type**.

VI Primitive Type

A **Primitive Type** is a pre-defined type built into the programming language. It cannot be broken up further into smaller units.

A **Primitive Type** *is associated with* {**Identifier**, **Constant**, **Variable**, **Argument**, **Expression**, **Statement**}
A **Primitive Type** *is used in* {**Expression**, **Complex Type**, **Statement**, **Block**, **Function**, **File**}

A **Primitive Type** *is declared in* {**Statement**, **Block**, **Function**, **File**}

A **Primitive Type** *is coerced to* {[another] **Primitive Type**}

A **Primitive Type** *is compatible with* {[another] **Primitive Type**}

Examples:

```
char  *name;
char[50] address;
int   age;
```

It can be deduced from the above examples that:

- the **Primitive Type** `pointer to char` *is associated with* the **Identifier** `name`
- the **Primitive Type** `pointer to char` *is associated with* the **Variable** `name`
- the **Primitive Type** `pointer to char` *is declared in* the **Statement** `char *name;`
- the **Primitive Type** `void` *is associated with* the **Statement** `char *name;`
- the **Primitive Type** `array of char` *is associated with* the **Identifier** `address`
- the **Primitive Type** `array of char` *is associated with* the **Variable** `address`
- the **Primitive Type** `array of char` *is declared in* the **Statement** `char[50] address;`
- the **Primitive Type** `void` *is associated with* the **Statement** `char[50] address;`
- the **Primitive Type** `pointer to char` *is compatible with* the **Primitive Type** `array of char`
- the **Primitive Type** `int` *is associated with* the **Identifier** `age`
- the **Primitive Type** `int` *is associated with* the **Variable** `age`
- the **Primitive Type** `int` *is declared in* the **Statement** `int age;`
- the **Primitive Type** `void` *is associated with* the **Statement** `int age;`

In addition, in the case where the operator = is involved and the types on both sides are different, the type of the right operand is coerced to the type of the left operand which is the type of the result.

Example:

```
int   x, y;
.....
....(x = y / 3) ....
```

It can be deduced from the above example that:

- the **Primitive Type** `int` *is associated with* the **Variable** `y`
- the **Primitive Type** `int` *is associated with* the **Constant** `3`
- the **Primitive Type** `int` of the **Variable** `y` *is coerced to* the **Primitive Type** `float` before the arithmetic operation

- the **Primitive Type** `int` of the **Constant** `3` *is coerced to* the **Primitive Type** `float` before the arithmetic operation
- the **Primitive Type** `float` *is associated with* the **Expression** `y / 3`
- the **Primitive Type** `float` of the **Expression** `y / 3` *is coerced to* the **Primitive Type** `int` after the arithmetic operation
- the **Primitive Type** `int` *is associated with* the **Expression** `(x = y / 3)`

VII Complex Type

A **Complex Type** is a type built from **Primitive Type**.

A **Complex Type** *is associated with* {**Identifier**, **Constant**, **Variable**, **Argument**}
A **Complex Type** *uses* {**Primitive Type**}
A **Complex Type** *is compatible with* {[another] **Complex Type**}
A **Complex Type** *is declared in* {**Statement**, **Block**, **Function**, **File**}
A **Complex Type** *is used in* {**Statement**, **Block**, **Functions**, **File**}

By definition, `struct` and `union` are both a **Complex Type**. Structures and unions may consist of different **Primitive Types**. For example, the details of an employee may include a name and his age. It is possible to represent this information separately using two different data structures: a name can be represented using an array of characters and the age can be represented as an integer. However, it may become inconvenient if the details of more than one employee are to be stored. The use of the **Complex Type** `struct` would be a more sensible choice. The following example shows a data structure which can be used to represent the above information.

Example:

```
struct employees {
   char  name[29];
   int   age;
} employee;
```

It can be deduced from the above example that:

- the **Complex Type** `employees` *is associated with* the **Identifier** `employees`
- the **Complex Type** `employees` *is associated with* the **Variable** `employee`
- the **Complex Type** `employees` *uses* the **Primitive Type** `pointer to char`
- the **Complex Type** `employees` *uses* the **Primitive Type** `int`
- the **Complex Type** `employees` *is declared in* the above **Statement**

The Relationship between **Complex Type** and **Constant** *is associated with* but it is less commonly used. Nevertheless, it is possible to declare a **Complex Type Constant** in the same way as the **Primitive Type Constant**.

Example:

```
struct employees {
   char  name[29];
   int   age ;
};

const struct employees Chan = {"Pui-Shan Chan", 25};
```

The above construct is a constant declaration. It can be deduced from the above example that:

- the **Complex Type** `employees` *is associated with* the **Identifier** `employees`
- the **Complex Type** `employees` *uses* the **Primitive Type** `pointer to char`
- the **Complex Type** `employees` *uses* the **Primitive Type** `int`
- the **Complex Type** `employees` *is declared in* the first **Statement**
- the **Complex Type** `employees` *is associated with* the **Constant** `Chan`
- the **Complex Type** `employees` *is used in* the second **Statement**

In theory, the values stored in the fields `name` and `age` will not be changed during the lifetime of the **Constant** `Chan`.

VIII Statement

A **Statement** is a coded instruction which the program can recognise and carry out. In this thesis, **Statement** also includes the C pre-processor statements `#define` and `#include` on the assumption of simple use of the `#define` statements to define values.

A **Statement** *uses* {**Identifier**, **Constant**, **Variable**, **Expression**, **Primitive Type**, **Complex Type**}

A **Statement** *declares* {**Constant**, **Variable**, **Argument**, **Primitive Type**, **Complex Type**, **Function**}

A **Statement** *defines* {**Variable**, **Function**}

A **Statement** *has a* {**Primitive Type**}

A **Statement** *follows* {[another] **Statement**}

A **Statement** *is followed by* {[another] **Statement**}

A **Statement** *is used in* {**Block**, **Function**, **File**}

Example:

```
main () {
int   x, y, z;          [1]
   x = y = 2;           [2]
   z = 3 * (x / y);     [3]
   printf("z = %d\n", z);  [4]
}
```

It can be deduced from the above example that:

- **Statement** [1] *uses* the **Identifiers** `x`, `y` and `z`
- **Statement** [1] *declares* the **Variables** `x`, `y` and `z`
- **Statement** [1] *declares* the **Primitive Type** `int`
- **Statement** [1] *has a* **Primitive Type** `void`
- **Statement** [1] *is followed by* **Statement** [2]
- **Statement** [1] *is used in* the **Function** `main()`
- **Statement** [2] *uses* the **Identifiers** `x` and `y`
- **Statement** [2] *uses* the **Constant** `2`
- **Statement** [2] *defines* the **Variables** `x` and `y`
- **Statement** [2] *uses* the **Variables** `x` and `y`
- **Statement** [2] *uses* the **Expression** `y = 2`
- **Statement** [2] *uses* the **Expression** `x = y = 2`
- **Statement** [2] *has a* **Primitive Type** `void`
- **Statement** [2] *uses* the **Primitive Type** `int`
- **Statement** [2] *follows* **Statement** [1]
- **Statement** [2] *is followed by* **Statement** [3]
- **Statement** [2] *is used in* the **Function** `main()`
- **Statement** [3] *uses* the **Identifiers** `x`, `y` and `z`
- **Statement** [3] *uses* the **Constant** `3`
- **Statement** [3] *defines* the **Variable** `z`
- **Statement** [3] *uses* the **Variables** `x`, `y` and `z`
- **Statement** [3] *uses* the **Expression** `(x / y)`
- **Statement** [3] *uses* the **Expression** `3 * (x / y)`
- **Statement** [3] *uses* the **Primitive Types** `int` and `float`
- **Statement** [3] *has a* **Primitive Type** `void`
- **Statement** [3] *follows* **Statement** [2]
- **Statement** [3] *is followed by* **Statement** [4]
- **Statement** [3] *is used in* the **Function** `main()`
- **Statement** [4] *uses* the **Identifier** `z`
- **Statement** [4] *uses* the **Variable** `z`
- **Statement** [4] *has a* **Primitive Type** `void`
- **Statement** [4] *follows* **Statement** [3]
- **Statement** [4] *is used in* the **Function** `main()`
- **Statements** [1], [2], [3] and [4] *defines* the **Function** `main()`

IX Block

A **Block** includes a list of declarations followed by **Statements**.

A **Block** *uses* {**Identifier**, **Constant**, **Variable**, **Argument**, **Expression**, **Primitive Type**, **Complex Type**}

A **Block** *contains* {**Statement**}

A **Block** *follows* {[another] **Block**}

A **Block** *is followed by* {[another] **Block**}

A **Block** *is used in* {**Function**, **File**}

A **Block** contains both declarations and **Statements**. These declarations are nested within an enclosing **Function**.

Example:

```
main () {
const two = 2;
int X;
   scanf("%d", &X);
   if X <= 0 then {
      printf("X = %d\n", X);
   }
   else {
      int      z = 5;
      printf("X = %d\n", two * z);
   }
}
```

Here, the **Variable** `z` *is declared in* and *is defined in* the inner **Block** in the `else` part of the `if` **Statement**, which is used in the **Function** `main()`.

X Function

Function is a sequence of **Statements** that are grouped together to perform certain tasks.

A **Function** *has an* {**Identifier**}

A **Function** *uses* {**Identifier**, **Constant**, **Variable**, **Argument**, **Expression**, **Primitive Type**, **Complex Type**}

A **Function** *returns* {**Primitive Type**, **Complex Type**}

A **Function** *contains* {**Statement**, **Block**}

A **Function** *calls* {**Function**}

A **Function** *is called by*{**Function**}

A **Function** *has the same interface as* {[another] **Function**}

A **Function** *is declared in* {**File**}

A **Function** *is defined in* {**File**}

A **Function** *is used in* {**File**}

The most noticeable Program Relationships in this group is the *calls* and the *is called by* relationships. These are commonly used in the static analysis of programs.

XI File

File includes a collection of declarations and/or definitions.

A **File** *has an* {**Identifier**}

A **File** *uses* {**Identifier**, **Constant**, **Variable**, **Argument**, **Expression**, **Primitive Type**, **Complex Type**, **Function**}

A **File** *imports* {**File**}

A **File** *contains* {**Statement**, **Block**, **Function**}

A **File** *is imported by* {[another] **File**}

A **File** *has I/O interface with* {[another] **File**}

4.3.3 The Attributes

Apart from the Program Relationships which can be deduced between the pairs of Program Elements, attributes which are affiliated to the Program Elements and the Relationships can provide the extra information that a maintainer may need. These attributes are generally associated with the scope and the states of the Program Elements, and also measurements, which are usually in the form of software metrics.

I Scope

The Scope of an identifier is the region of the program over which occurrences of each can be matched with the defining declaration. In C, nested function declaration is not allowed. Any Program Element declared inside a function is only visible within that function by default. Program Elements which are declared in this fashion are of a local nature. Once the function is exited, these Program Elements cease to exist (with the exception of static variables which will be discussed in the next section). It is however, possible to declare a Program Element of a global nature. It means this Program Element has a scope that encompasses the entire file and thus can be used for communication between functions. It can be done by declaring the Program Elements outside the function definitions. Table 2 shows the attributes affiliated with each of the Program Elements from Table 1.

	Scope		Scope
Identifier	local, global	**Complex Type**	local, global
Constant	local, global	**Statement**	local
Variable	local, global	**Block**	local
Argument	formal, actual	**Function**	global
Expression	local	**File**	global
Primitive Type	local, global		

Table 2 Scope of Program Elements

II Storage Class

Besides a type, variables in the C programming language can be designated to have a particular storage class. It is used to determine how the compiler allocates memory to that variable. There are four storage classes, namely **auto**, **extern**, **static** and **register**.

Global variables in C are classified as static variables, meaning that they come to existence when the program is executed and it continues to exist until the program terminates. A static global variable cannot be accessed by functions in other files other than the one in which it is declared. Local variables are by default classified as auto variables. This is due to the fact that memory is allocated automatically to these variables when a function is executed and then deallocated when the function terminates. It is possible to declare local variables as **static**, however. If a static local variable is assigned a value the first time when a function is called, it will retain its value on subsequent calls of the function.

The **register** storage class can be specified only for local variables. Such a declaration will instructs the compiler to store the value of a local variable in a register. The **register** storage class can also be applied to a formal argument in a function. Since arguments are passed to functions through memory, the supplied argument value is loaded into a register when the function is executed.

The **extern** storage class does not create a variable, but it only informs the compiler of its existence. When a global extern declaration is made outside a function, it indicates that the variable referred to is declared in another file. In order words, global extern declaration enables global variables to be shared among several files.

A function can be declared as **static**. Such as function can be called by other functions within the same program file, but not by functions in other files. A function can also be declared as **extern**. It works the same way as an **extern** variable. The above discussion is summarised in Table 3.

	Storage Class
Variable	auto, static, register, extern
Function	static, extern

Table 3 Storage classes in C

4.4 A Framework for the Integrated Approach

As discussed before, more informative overviews of the programs can be attained if various Program Relationships between Program Elements are supported and brought into play. Most of the software maintenance tools discussed in section 2.3.5 in Chapter Two offer some degree of visualisation. For example, the relationships function calls and control flow are frequently illustrated in various graphical forms in those tools. However, the use of the graphical representations in some cases may be unhelpful due to their scale and complexity. The attention of the users is often drawn back to the source code as there is inadequate support for extracting information from these graphical representations. Most of the users may prefer to construct a mental model of their own whilst others may prefer to trace the relationships by drawing lines to link different areas of the source code.

The Program Elements and Relationships are the key to the Integrated Approach. The Program Elements are linked together governed by the grammar of a programming language. When combined together, they form various relationships. The Program Relationships between pairs of Program Elements represent various levels of abstraction of the source code. A higher level Program Relation can be refined to a lower level one during comprehension and a lower level Program Relation can be abstracted into a higher level one. For example, the relation *imports* between the Program Elements **File** and **File** is of a higher level of abstraction than the relation *follows* between the Program Elements **Statement** and **Statement**. It is argued that comprehension can be achieved by refining, expanding and analysing the Program Relationships between pairs of Program Elements.

The process of Program Comprehension can be facilitated by setting up a framework. Program Elements and Relationships discussed above are the basic ingredients in this framework. The other components in the framework include:

- context sensitive navigational aids
- information displays which include both textual and graphical information

The context sensitive navigational aids are the focal point in this framework. They provide a mechanism for easy access to the Program Elements and Relations shown in Table 1.

A natural way of representing relationships is in the form of graphs. The Program Relationships shown in Table 1 can be easily illustrated graphically with the respective pair of Program Elements. When the utilisation of graphical representations alone is insufficient, textual display can also be used to provide extra information.

4.4.1 Context Sensitive Navigational Aids

Programs are built from Program Elements which are held together via a network of Program Relationships. It is this connectivity which enables the realisation of the Integrated Approach. When one Program Element is under scrutiny during the comprehension process, it will inevitably pave the way to other related Program Elements and subsequently reveals the underlying Relationships between them. For example, when studying the Relationship **File** *imports* **Files**, a maintainer will be presented with other Relationships such as **File** *contains* **Functions**. If he chooses to explore this relation further, he will be presented with more Relationships such as **Function** *returns* **Primitive/Complex Type**, and **Function** *uses* **Variable**. Information regarding a Program Element is gathered by observing the interactions between the related Elements and analysing the Relationships. Under the Integrated Approach, the path of information gathering is not fixed and the maintainer is free to explore any of the Program Elements and Relationships that he chooses. It is flexible and it allows comprehension to be conducted according to preferences of the maintainer. The context sensitive navigational aids are designed to provide a mechanism to retrieve the relevant Elements and Relationships to the maintainer. Information can be attained by executing and switching between various comprehension theories and models. The following figures show a set of navigational aids when the appropriate Program Elements are selected.

Figure 4-2 shows a list of Program Relationships which may be of interest to the maintainer when he is inspecting the Program Element **File**. The Program Relationships represented by the navigational aids are:

- **File** *contains* **Function**; **Function** *is declared in/is defined in* **File**..... [more on functions]
- **File** *uses* **Constant**; **Constant** *is declared in* **File**..... [more on constants]
- **File** *uses* **Variable**; **Variable** *is declared in* **File** ..... [global variables]
- **File** *uses* **Type (Primitive/Complex)**; **Type** *is declared in* **File** ..... [more on types]
- **File** *contains* **`#define` Statement** ..... [more on #define]
- **File** *contains* **`#include` Statement**..... [more on #include]
- **File** *imports/is imported by/has I/O interface with* **File** ..... [more on system]

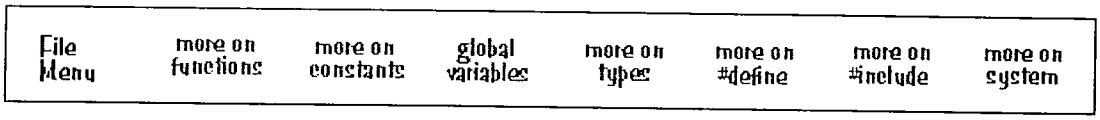

Figure 4-2 A set of navigational aids when the Program Element **File** is selected

Figure 4-3 shows a list of Program Relationships which may be of interest to the maintainer when he is inspecting the Program Element **Function**. The Program Relationships represented by the navigational aids are:

- **Function** *calls* **Function** ..... [call graph]
- **Function** *is called by* **Function** ..... [called by...]
- **Statement** [in the **Function**] *follows* **Statement** [in the **Function**]..... [control flow graph]
- **Function** *uses* **Constant**; **Constant** *is declared in* **Function** ..... [constants]
- **Function** *uses* **Variable**; **Variable** *is declared in* **Function**..... [local variables]
- **Function** *uses* **Type (Primitive/Complex)** ; **Type** *is declared in* **Function**... [types]
- **Function** *uses* **Argument; Function** *returns* **Primitive Type** ..... [parameters]
- **Function** *is declared in/is defined in /is used in* **File**..... [related files]

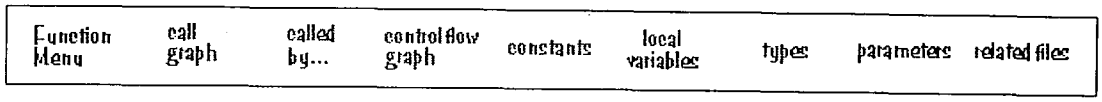

Figure 4-3 A set of navigational aids when the Program Element **Function** is selected

Figure 4-4 shows a list of Program Relationships which may be of interest to the maintainer when he is inspecting the Program Element **Variable**. The Program Relationships represented by the navigational aids are:

- **Variable** *is declared in* **Function** ..... [declared in...]
- **Variable** *is used as* **Argument**..... [as parameters...]
- **Variable** *has a* **Type (Primitive/Complex)**..... [variable type]

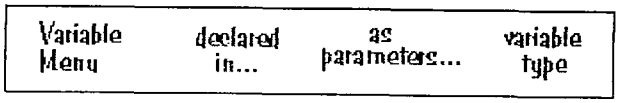

Figure 4-4 A set of navigational aids when the Program Element **Variable** is selected

Information regarding a Program Element can be gathered by observing the interactions between the related elements and analysing the Program Relationships. To limit the scope of the exploration and to isolate the investigation to just one component at a time may hinder the comprehension process.

Maintainers may have difficulties in combining pieces of disjointed information, especially when the number of components concerned increases. In an ideal situation, the process is continued without interruptions until sufficient information about the program is attained. Hence, Program Comprehension is both an assimilation and an opportunistic process. This non-deterministic nature is the justification why the incorporation of an element of flexibility in a software maintenance tool is important. Maintainers should be equipped with the ability to expand and refine the Program Relationships so that they can explore the different aspects of a program when required.

The context sensitive navigational aids are designed with this purpose in mind. They provide a mechanism for easy access to the Program Elements and Relationships discussed in Table 1. When a Program Element or a Relationship is encountered, a maintainer will be presented with its details. His attention will also be drawn to the other Program Elements and Relationships that are related to the Program Element or Relationship first encountered. The navigational aids resemble the context sensitive menu systems used in most modern day applications. For example, in a word processor, the menu changes when the cursor is placed upon an array of cells (tables) so that the extra features can be used to operate on these cells. The context sensitive navigational aids are in place to ensure that the process of comprehension can be continued without interruption. They are designed to provide a mechanism to retrieve the relevant Elements and Relationships. By explicitly exposing these relationships, maintainers can have access to a wider range of information with various degrees of granularity.

4.4.2 Information Display

In order to understand a piece of source code, a maintainer needs to acquire different levels of information at various stages. Both the textual and graphical representations have distinct advantages in depicting relationships at different levels. Textual representations are important because they record exactly how different Elements are related to each other whereas the graphical representations are a higher order abstraction of the Relationships described by the textual representations. In general, the textual representations offer a lower level of insight into the programs and they provide the facts about the programs. The graphical representations, on the other hand, offer a higher level overview. In addition, they have the added advantage of being easily rearranged and manipulated. Higher orders of abstraction can be obtained by reducing the complexity of the graphical representations. When engaged in maintenance activities, maintainers may require an overview at one stage and get right down to the statement level the next. The key to a useful software maintenance tool is to strike a balance between the utilisation of visualisation and the traditional text-based static analysis tools.

I Textual Display

Text windows are used for the display of source code and information regarding the Program Elements and Program Relationships in this framework. Experiments have shown that most

maintainers are often drawn back to the source code in order to infer or to verify their queries even when they have been presented with other alternative representations. More often than not, it requires manually tracing a Program Element, or a Program Relationship, in pages and pages of program listing. The following describes techniques which can be used to enhance the usefulness of textual representations.

A Search Engine

With the advance of CASE tools, a database of the Program Elements used in a program can be built with ease. Information for each Program Element is recorded and can be made available in the form of a searchable database. The criteria for a search machine may include the following:

- case sensitive search
- pattern matching search
- search patterns which form part of an identifier
- indicating a percentage of occurrences at file level, function level and statement level

The set of criteria helps to locate related information quickly and effectively.

B Homogeneous Information

Apart from helping to link different Program Elements together, the database has another application. Since it holds the locations and scope of all the Program Elements within a program, it can be used to locate a Program Element efficiently. These components can be linked together by way of hypertext links. In a hypertext system, all text documents are indexed and held together by hyperlinks.

Figure 4-5 shows how the hypertext links can be extended across a number of documents. The figure shows two screen shots with two different listings. The first one is the file **`convert.c`** and the second one is the file **`use.h`**. Both program files are part of a system named **`convert`**.

In the first diagram, it shows a function **`main()`** in file **`convert.c`** together with the local variable declarations. The data type **`UseData`** contains a hypertext link to its full definition in a file named **`use.h`**. A click on **`UseData`** will invoke the hypertext system to show its full definition in the file **`use.h`** as shown in the second diagram.

The hypertext links allow information to be accessed instantly, thus helping to save time and to cut down the possibility of human error.

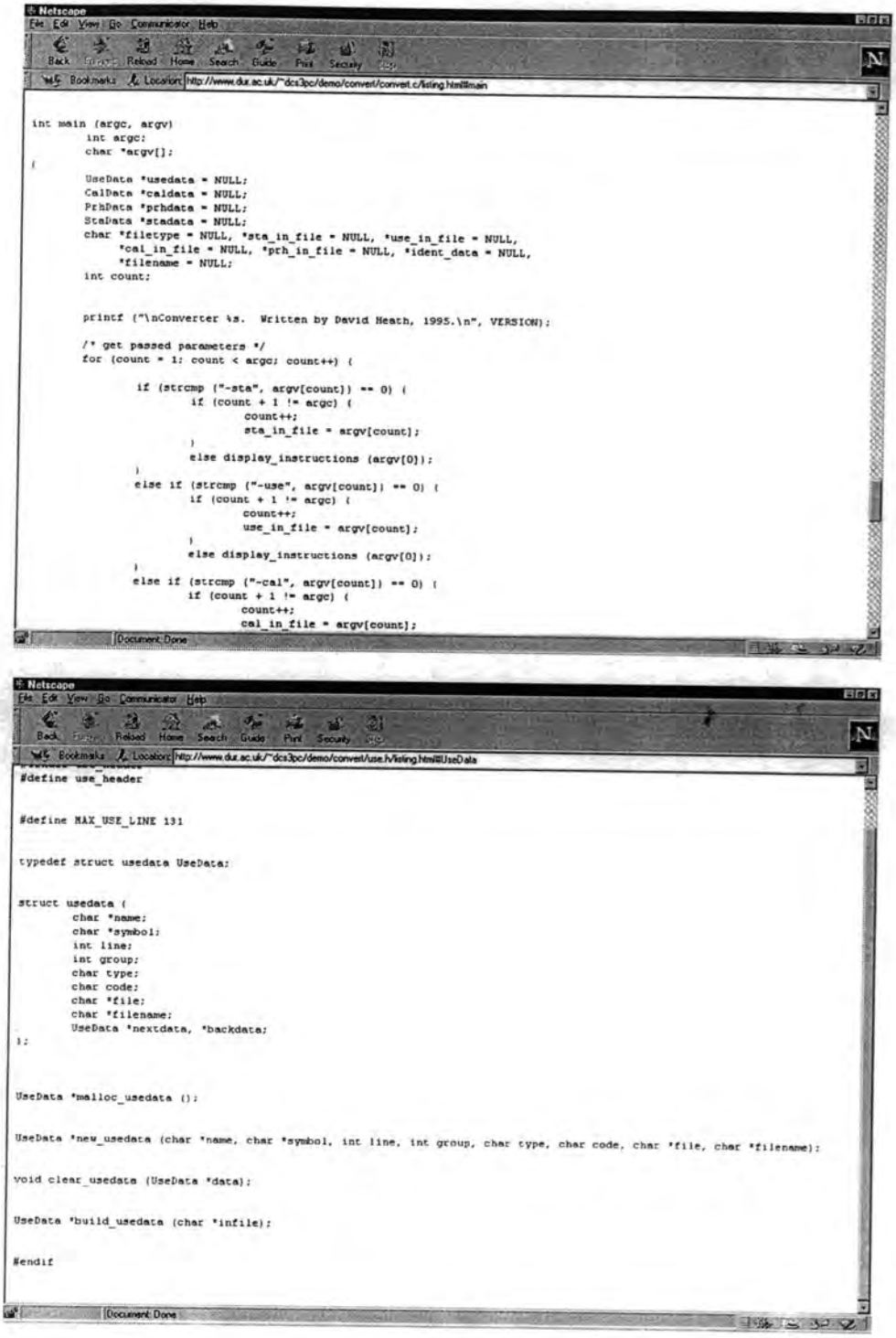

Figure 4-5 Screen shots showing the use of a hypertext link across a set of hypertext documents

C Heterogeneous Information

A program listing can also be annotated so that the Program Elements are linked to different areas of the graphical representations and vice versa. This is a natural extension of the hypertext links. The type of information which are held together via hypertext links need not be homogenous. Indeed, the essence of hypertext links are the ability to link heterogeneous information together.

Figure 4-6 shows how the hypertext links can be used to cross-reference information in various representations. In the first diagram, it shows a call graph for the function **`main()`** in the file **`sortline.c`**. A click on the node 'readlines' in the call graph will invoke the browser to show the definition of the function in a program listing as shown in the bottom diagram.

Both the textual and graphical representations can be made to contribute to the comprehension process so that information can be attained in a more effective and cohesive manner.

II Graphical Display

A natural way of representing relationships is in the form of graphs. The Program Relationships discussed in Table 1 can be illustrated graphically with the respective pair of Program Elements. The most frequently illustrated relationships are function calls and control flow. It is evident that there are still a number of relations which can be illustrated graphically as shown in Table 1. For example, the relationships such as file inclusion and type dependencies can be depicted graphically to give an overview of a program.

Visualisation is often associated with the problem of graph layout. It is widely recognised that the problem of finding a graph drawing algorithm which satisfies a set of criteria is NP-hard [Supo83, DiBa84] as the criteria are incompatible in nature. Nonetheless, algorithms can still be found for use in different situations but the problem may still persist as it is governed by physical constraints such as the size and resolution of a screen. For example, a graph cannot often be displayed in its entirety and has to be squeezed into a window with vertical and horizontal scroll bars as visual aids. Only a small portion of the entire graph can be studied at a time which makes it difficult to visualise the whole structure. On the other hand, to display a graph in its entirety may not help to yield much information about the underlying structure as it may be too complex to handle. What is needed is a systematic way of decomposing the graphical representations so that they become less complex and more manageable. A number of strategies which can be applied to these graphical representations are suggested in section 2.3.5 in Chapter Two. These include the use of layout, colour, graph simplication and grapph slicing techniques.

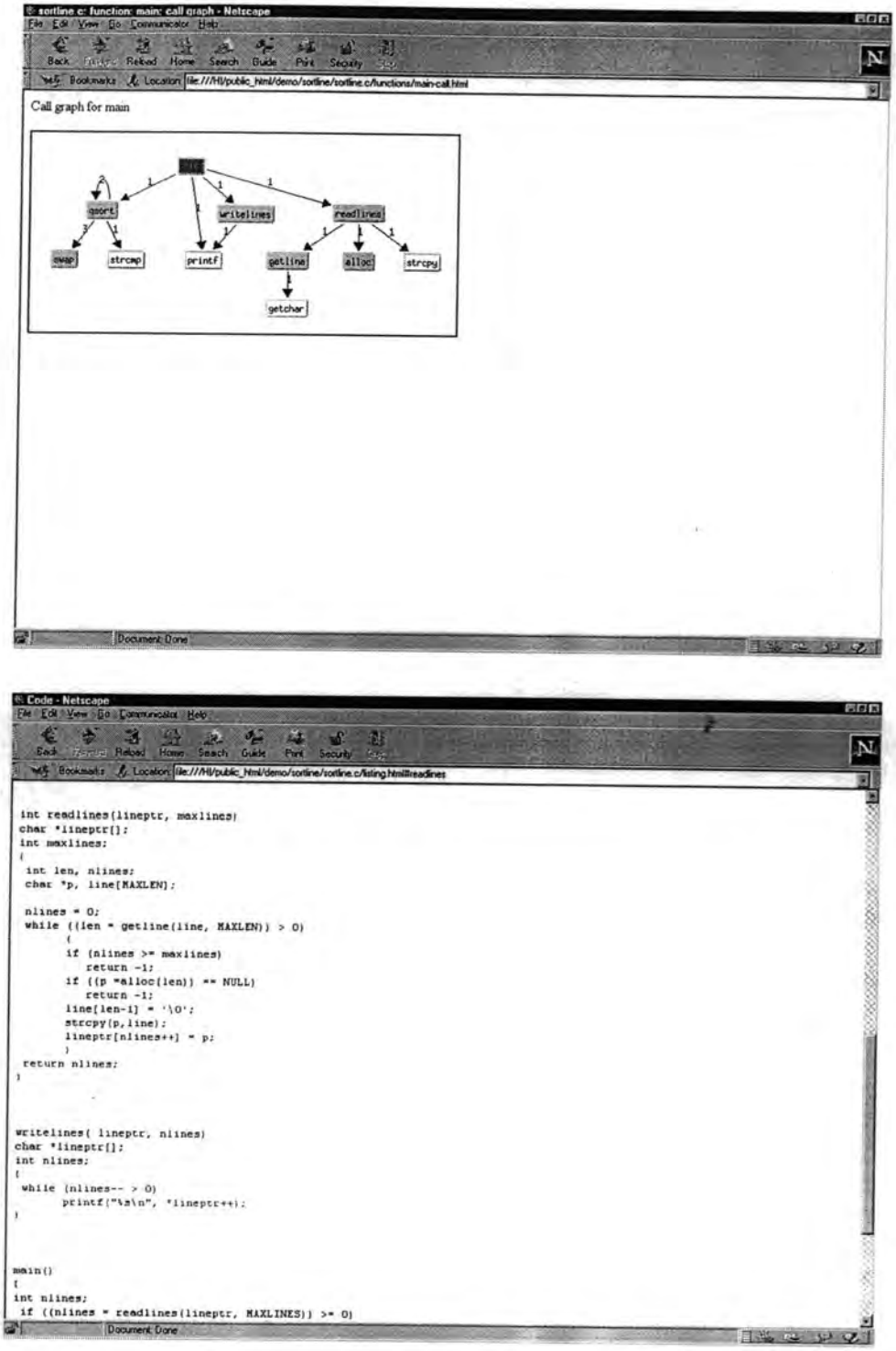

Figure 4-6 Screen shots showing the use of a hypertext link to cross-reference information

A Layout

Most of the graphical representations used in software maintenance tools are depicted in hierarchical fashion. It is rooted in the culture of Computer Science practice. In the C programming language, there is always only one starting point, i.e. the function **`main()`**, where all the rest of the program follows. Figure 4-7 shows the call graph of the function **`main()`** in a file named **`sortline.c`**. A complete listing can be found in Appendix A.

Apart from analysing the Relationship *calls* to extract information, the Relationship *follow* can also be a useful source of information. In most cases, the graphical representations of the control flow relationship often involve a larger number of nodes and arcs than that of the function call graphical representations, and hence the denser the graphical representations, the less readable these representations will be. Figure 4-8 shows a graphical representation for the control flow relationship. It shows the control flow graph for the function **`main()`** in the file **`sortline.c`**. This representation is a simplified version, which shows Program Elements such as the different types of statement and the identifiers of the functions. Statements which are included in the graphical notation are: **`for`** Statement, **`if else`** Statement, **`while`** Statement and **`switch break`** Statement. The arcs are labelled with the letters **`u`**, **`t`** and **`f`**, which represent the conditions needed in order to pass the control from one Program Element to the another. The letter **`u`** stands for Unconditional, the letter **`t`** stands for True and **`f`** for False. In addition, the positions of the function names in figure 4-8 indicate the sequence of function calls.

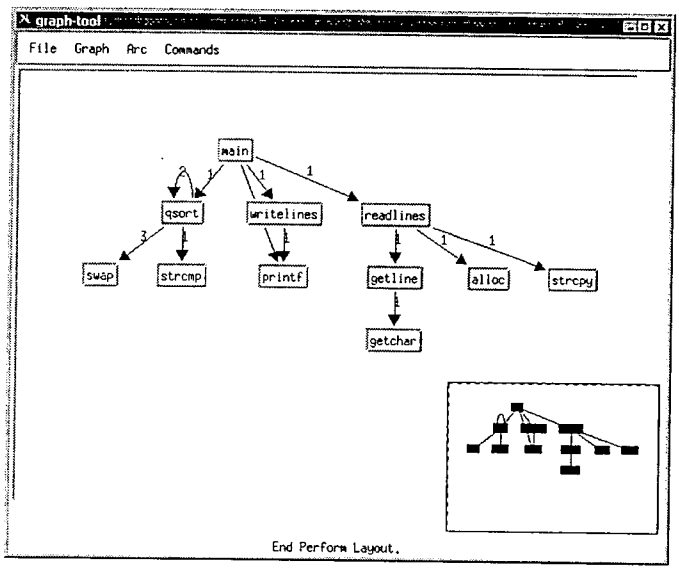

Figure 4-7 A call graph of the function **`main()`** in file **`sortline.c`**

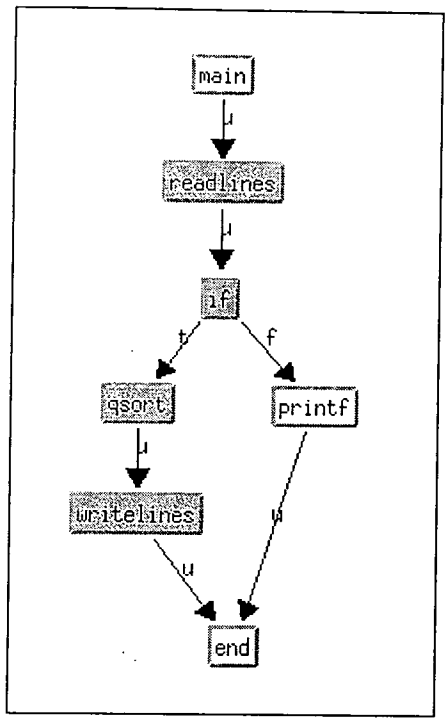

Figure 4-8 A simplified control flow graph of the function **`main()`** in the file **`sortline.c`**

Apart from the graphical representations for the Relationships *calls* and *follow*, there is another Relationship which can be illustrated graphically. This is shown in Figure 4-9.

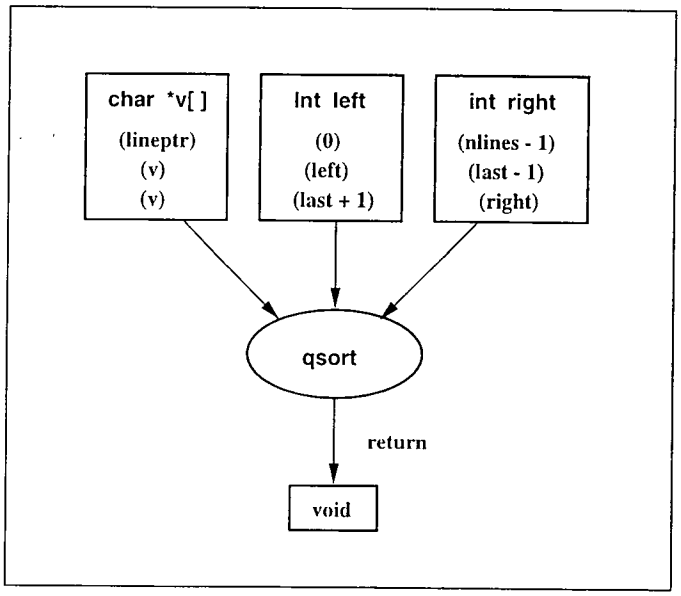

Figure 4-9 The function interface of the function **`qsort`** in the file **`sortline.c`**

The above figure is a graphical representation for function interface. The representation attempts to depict the following Relationships between the Program Elements:

- **Function** and **Argument**
- **Function** and the **Type** that is returned by it
- **Argument** and **Type**
- **Argument** and **Variable**
- **Variable** and **Type**

The rectangular boxes in the first row show the **Type** and names of the formal **Arguments** declared in a **Function**. The names in brackets represents the names of **Variables** which are the actual **Arguments** when the **Function** is called, the oval shape shows the names of the **Function** and the rectangular box in the third row shows the **Type** that is returned by that **Function**.

B Colour

As mentioned in section 2.3.5 in Chapter Two, colour can also be used to identify a program's hierarchical composition. The primary goal of the hierarchical layout is to try to reveal the ancestral relationship among nodes clearly and unambiguously. Perfect hierarchies rarely exist in programs because of features such as recursion. It may be difficult to locate the connecting nodes from a node under investigation and colour can be conveniently used to illustrate this connectivity. Figure 4-10 shows how colour can be used to locate all the connecting nodes from the node 'readlines'. The use of colour can also be used to highlight library function calls, external function calls and nodes with a high number of fan-in and fan-out.

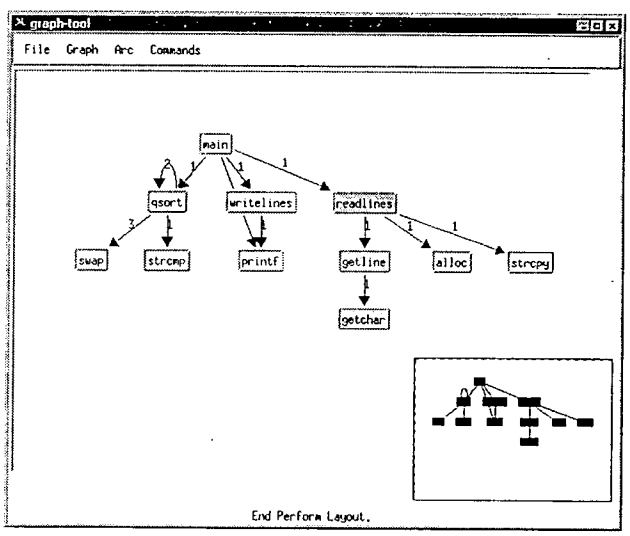

Figure 4-10 Nodes which are connected to 'readlines' are highlighted using colour

C Graph Simplification

Information clustering is the process whereby information is abstracted by removing nodes from the graphical representations. The information clustering principle can be used in a number of ways:

- to number arcs
- to isolate subgraphs
- to hide third party libraries
- to hide ANSI C standard libraries
- to hide external function calls to the application's libraries

Figures 4-11 illustrate how graph simplification can be applied to reduce the overall complexity of the graphical representations. The top diagram shows a graph call of a function named **`build_call`**. The bottom diagram shows the same graph call with the library functions removed. For the purpose of comparison, the relative positions of the remaining nodes in the bottom diagram are unchanged.

D Graph Slicing

Graph slicing is another way of reducing complexity. Contrary to the technique of graph simplification, the attention is given to a small number of nodes and their connecting nodes. By concealing the rest of the nodes present in the graphical representation, a small section of the representation can be studied with more care. The slicing principle can be used in a number of ways:

- to investigate the characteristics of function calls
- to investigate the characteristics of library function calls
- to investigate the ripple effort after a modification

Figure 4-12 illustrate how graph slicing can be applied to extract a small portion of nodes from the graphical representations. It shows the portion of call graph after applying the graph slicing technique on the node 'build_sys_call'. The node 'build_sys_call' is selected from the top diagram in Figure 4-11.

This technique can be applied to any arbitrary nodes that are of interest to the maintainers. In addition, the depth of the sliced graphical representations can be controlled by an attribute which determines when the algorithm should terminate.

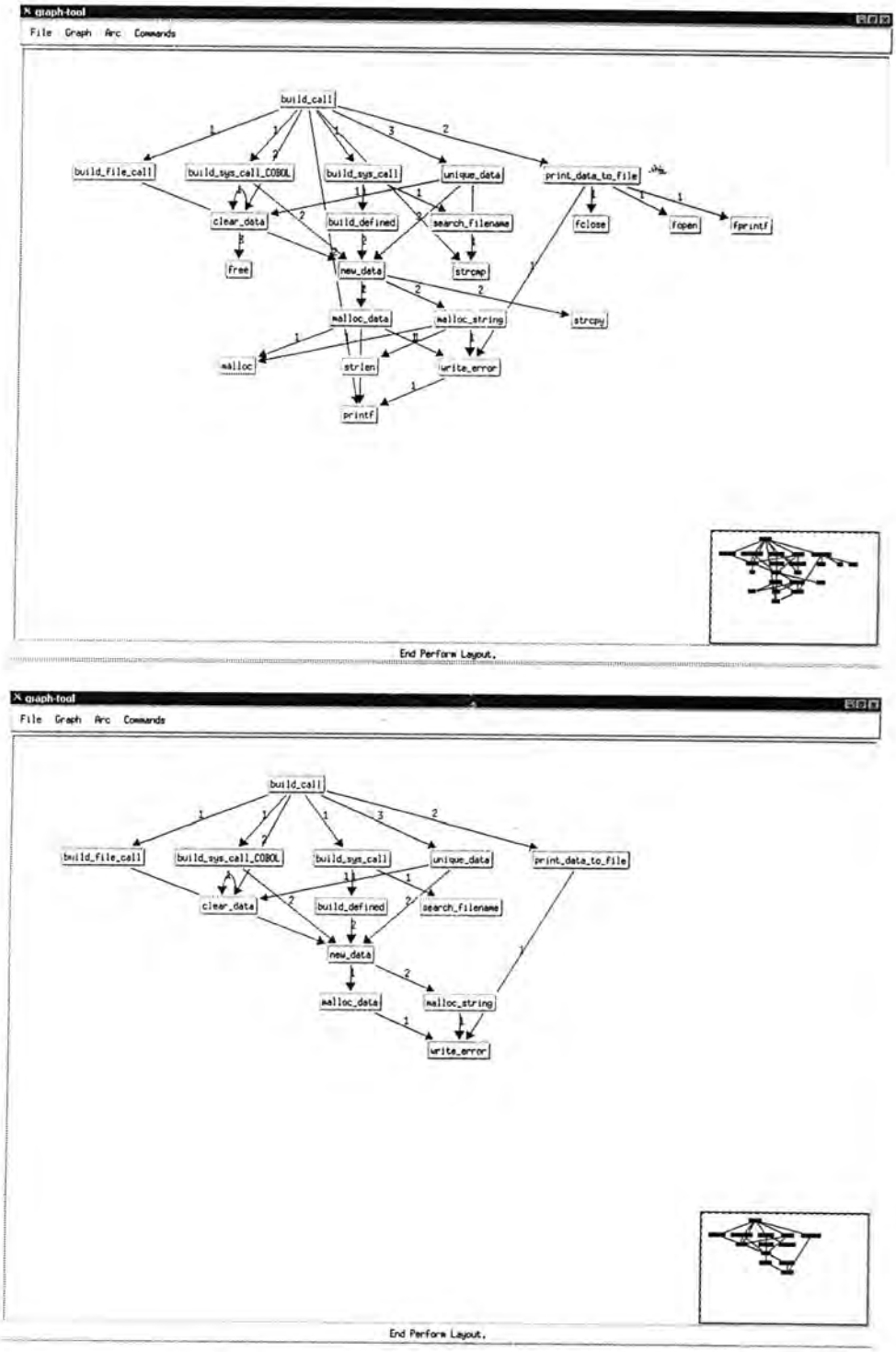

Figure 4-11 The use of clustering technique on the call graph of the function **build_call**

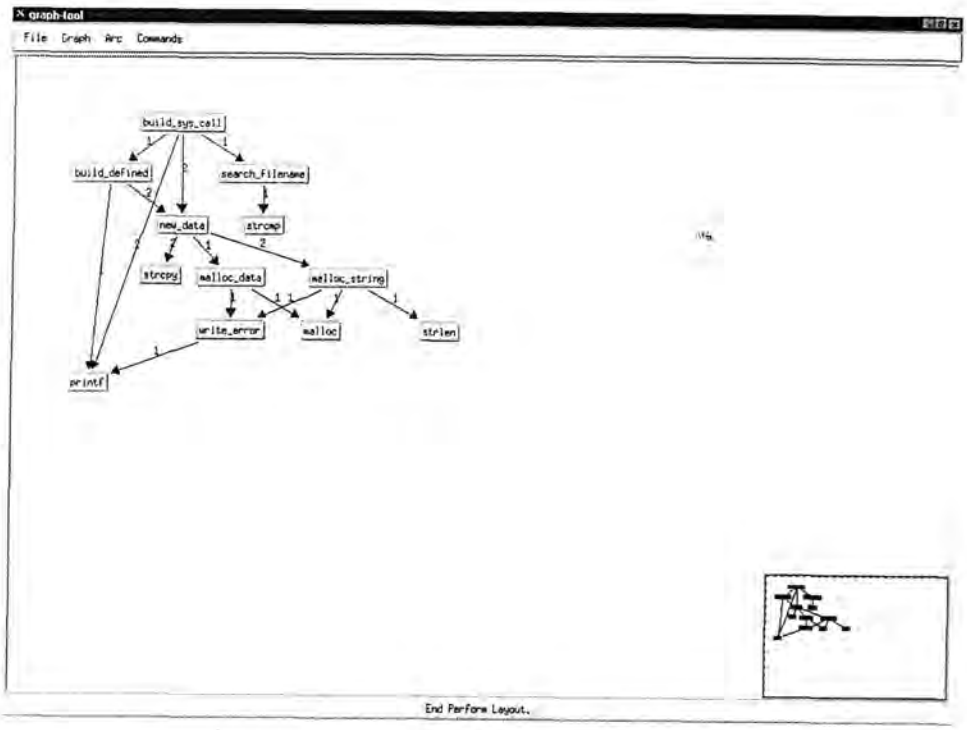

Figure 4-12 The portion of call graph containing the node 'build_sys_call' and its connecting nodes

4.5 Summary

From the overview of existing Program Comprehension theories discussed in Chapter Two, it is evident that there is no real consensus on how maintainers understand systems. Often, maintainers may employ various theories and use cues in either the source code or the system documentation as guidance. It is argued that when maintainers are engaged in the maintenance tasks, they may exploit the use of both the top down and the bottom up approaches when certain information comes to light [Chan97, Leto86a, Mayr94, Mayr95]. However, the use of existing software maintenance tools *alone* may not be sufficient to facilitate the comprehension process. Early work on building these maintenance tools was based on the use of simple relations of function calls and control flow, such as the work carried out by Foster [Fost87] and Fletton [Flet88]. It is unlikely that a single tool will be found which has the capability to assist all activities which are encompassed by the various cognition models. The development of a new Program Comprehension tool should take this into account and provide the flexibility that the maintainers may need so they will not be hindered by the limitations of the tools.

As programs grow in size and complexity, the gap between the types of information required by the maintainers and the amount of information which can be provided by the maintenance tools widens. It is shown in Table 1 that the Relationships *calls* and *follows* are not the only relationships present in a program. These two Program Relationships receive a lot of attention because of their significance in the way programming languages are used. A program consists of Program Elements which are interlinked via a network of Relationships. By allowing the other Program Relationships to be brought into the scene, maintainers will be able to get access to information in a wider spectrum and in a more consistent way.

The Integrated Approach proposed in this chapter is based on a matrix of Program Relationships between pairs of Program Elements. The Approach acknowledges that the process of comprehension is opportunistic. Information about the programs can be gathered by expanding, refining and analysing the Program Relationships. It is a flexible and it allows various comprehension theories and models to be performed in a single environment. The Program Elements and Relationships are supported by a set of context sensitive navigational aids whereby information is presented using both the textual and graphical representations.

Chapter Five

Implementation

5.1 Introduction

Static analysis tools are useful for extracting information from programs. Maintainers are more likely to be overloaded with information extracted from these analysis tools as programs grow in size. This chapter describes how the various Program Comprehension theories and models can be realised by a simple browsing tool PUI (*P*rogram *U*nderstanding *I*mplement), which allows maintainers to understand the Relationships between Program Elements. The tool is based on a matrix of Program Elements and Program Relationships discussed in Chapter Four which are designed to reflect the multi-dimensional nature of programs.

5.2 The Prototype

The main objective of the prototype, *P*rogram *U*nderstanding *I*mplement (PUI), is to facilitate the process of comprehension and it is based on a matrix of Relationships between pairs of Program Elements discussed in Chapter Four. The PUI tool offers support to the top-down, bottom-up and a mixture of both approaches by having a number of *implements* that probe the relationships between the elements.

Figure 5-1 shows an overview of the composition of PUI which is enclosed in the inner rectangle. CCG [Kinl95], which stands for Combined C Graph, is a static analysis tool and Graph Tool [Bodh95] is a graphical display tool. Both were developed in the Department of Computer Science in Durham. Perl is a programming language available in the UNIX, Windows95 and Windows NT operating systems. CGI, which stands for Common Gateway Interface, can take advantage of any resource available to the server computer to generate output and it can also accept input from the user. The main advantage of using CGI scripts is the ability to provide dynamic data and create dynamic

hypertext documents. HTML, which stands for HyperText Mark-up Language, is a standard set of instructions which can be recognised by most of the existing hypertext browsing tools.

The input to CCG is the C programs. They may be either ANSI [ANSI84] or Kernighan and Ritchie [Kern78, Kern88] C. The output of CCG is in a textual format. It is a CCG fact base which is a representation of C programs.

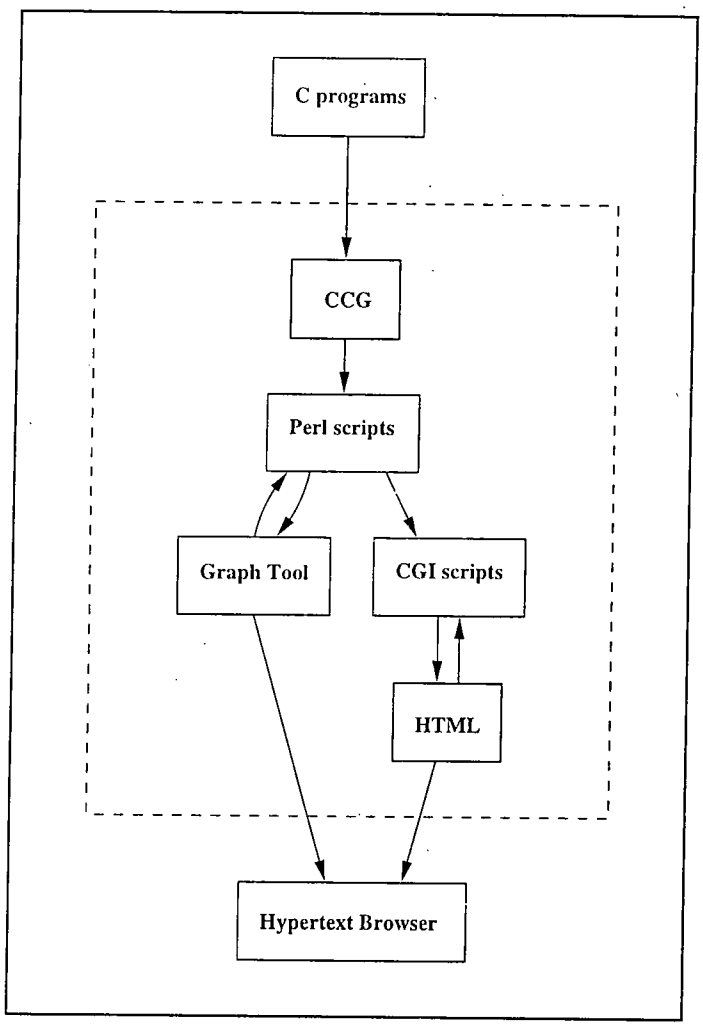

Figure 5-1 An overview of PUI together with the supporting tools

Perl is a language available in the UNIX, Windows95 and Windows NT operating systems. It has a rich reservoir of functions for handling textual information. The output from CCG is fed into the Perl scripts where information about the Program Elements and Program Relationships are extracted. Program Relationships which can be represented visually are then translated into a format which is

recognised by a graphical display tool named Graph Tool. The rest of the textual information relating to the Program Elements and Program Relations is fed into the CGI scripts.

The CGI scripts represent the *implements* of the PUI tool. Depending on the selections of Program Elements and Program Relationships, the CGI scripts deliver different types of information relating to them as shown in Table 1. Information, whether it is in textual form or in a graphical form, will be translated into HTML which can be observed using hypertext browsing tools such as Netscape Communicator[1] and Internet Explorer[2].

Figure 5-2 shows a typical screen of the PUI tool. The hypertext browser used in this research is Netscape Communicator.

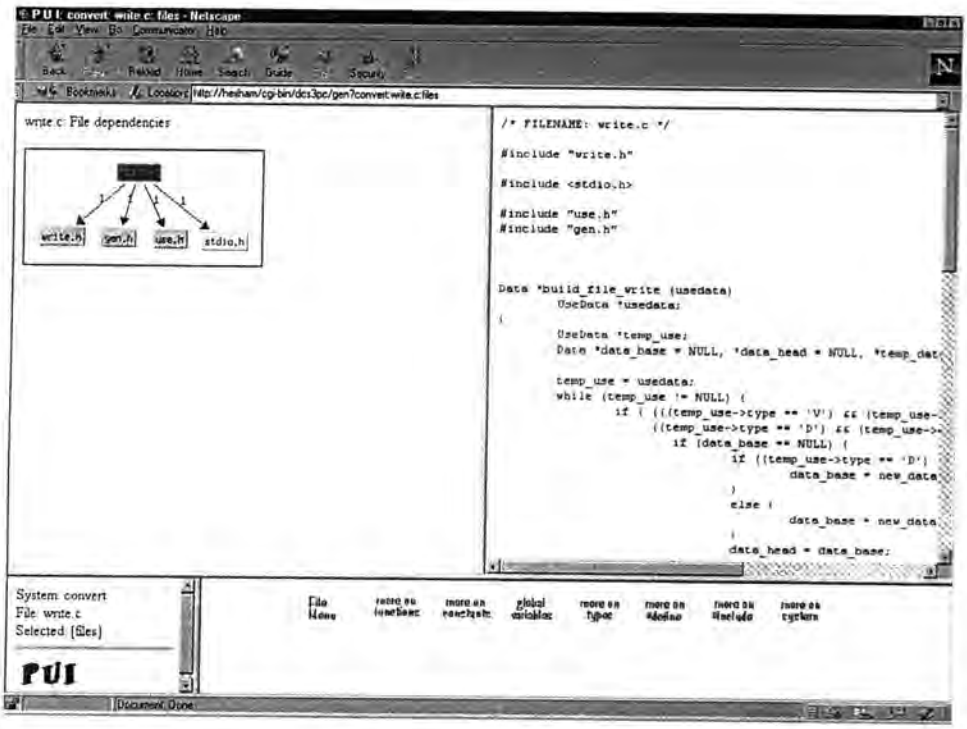

Figure 5-2 The PUI tool

All of the Program Elements within the PUI tool are cross-referenced which ensures the maintainers have easy and quick access to the information required.

[1] Netscape Communicator is a registered trademark of Netscape Communications Corporation.
[2] Internet Explorer is a registered trademark of Microsoft Corporation.

5.3 Tool Support

The format of the CCG fact base is not compatible with the input format for Graph Tool and thus relevant information must be extracted from the fact base and converted into a suitable format. Figure 5-3 shows a valid input for Graph Tool. It is a file dependency graph for a file named **write.c**. A small portion of the source code in the file **write.c** is shown in Figure 5-2.

```
( object ) 1 0 0 0 0 ( write.c ) ( _ ) ( _ ) object
( object ) 2 0 0 0 0 ( write.h ) ( _ ) ( _ ) object
( link ) 1 2 0 0 0 0 0 0 ( 1 ) ( directed ) ( LineSolid ) link
( object ) 3 0 0 0 0 ( gen.h ) ( _ ) ( _ ) object
( link ) 1 3 0 0 0 0 0 0 ( 1 ) ( directed ) ( LineSolid ) link
( object ) 4 0 0 0 0 ( use.h ) ( _ ) ( _ ) object
( link ) 1 4 0 0 0 0 0 0 ( 1 ) ( directed ) ( LineSolid ) link
( object ) 5 0 0 0 0 ( stdio.h ) ( _ ) ( _ ) object
( link ) 1 5 0 0 0 0 0 0 ( 1 ) ( directed ) ( LineSolid ) link
```

Figure 5-3 Input to Graph Tool

When used as an input to Graph Tool, the file in Figure 5-3 will produce a graph as shown in Figure 5-4.

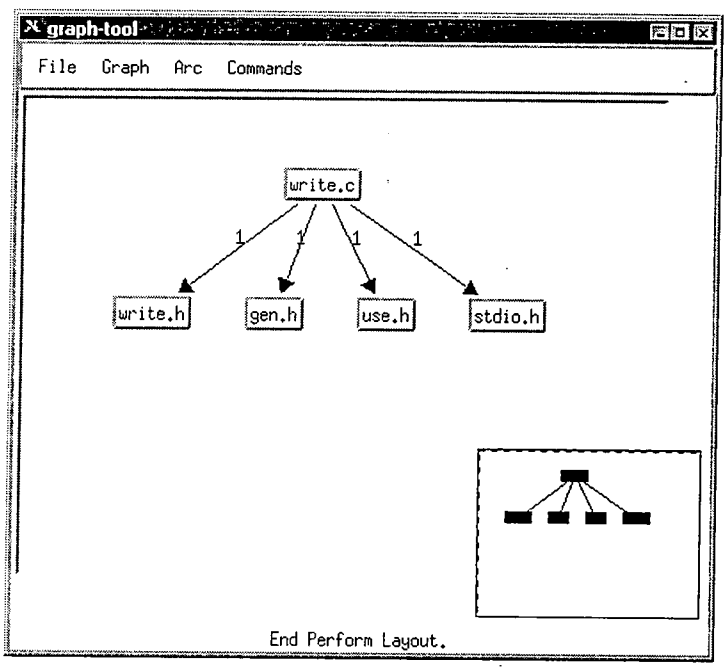

Figure 5-4 A snapshot of Graph Tool depicting a graph using the input from Figure 5-3

The numbers on the arcs represent the number of times a Program Relationship is used. The rectangle box on the bottom of the right hand side is a context map. The map is a miniature of the entire graph and it indicates the portion of the graph which is shown in the main window. The graphical representations shown in Figure 5-2 and Figure 5-4 are identical.

There are three main functions of the Perl scripts:

1 to extract the information relating to the relational aspects of the Program Elements
2 to translate this information into a format recognised by Graph Tool
3 to prepare the rest of the CCG fact base so that it is ready to be fed into the CGI scripts

Strategies on how to improve the layout of the graphical representations discussed in section 4.4.2 in Chapter Four are realised in the second function.

The main objectives of the CGI scripts are:

1 to provide a mechanism to probe the relationships between the Program Elements
2 to produce a set of hypertext documents using HTML

The first objective of the CGI scripts is to provide a mechanism to probe and to retrieve information relating to the Program Elements and relationships in a context sensitive manner. This is done in the form of the context sensitive navigational aids. Some of the context sensitive navigational aids are shown in the bottom half of the Figure 5-2. A full discussion of these navigational aids is presented in section 4.4.1 in Chapter Four. A demonstration of the use of these aids will be presented in Chapter Six.

As discussed earlier, the Program Elements are held together by different Program Relationships. It is difficult to try to find out the characteristics of a Program Element without stumbling on the related Program Elements and Relationships. Maintainers should be provided with some degree of support so that they are able to select and explore the many different Program Elements and Relationships when required. For example, the CGI scripts can help to find out the name of the file which contains a data type's declaration when first encountered or they can be used to find out a list of functions which use that data type. These CGI scripts are similar to the queries made in a relational database. When given the names of a pair of Program Elements, these scripts try to retrieve information relating from the matrix shown in Table 1.

5.4 A Brief Introduction to PUI

A natural way of linking heterogeneous information together is to place the information into a hypertext environment. Various information extracted from static analysis tools can be put together using hypertext links. These may include textual and graphical representations, software metrics and system documentation.

Figure 5-5 shows the start-up screen of the PUI tool. The tool indicates that there are three systems currently available for analysis. A system can be selected by clicking on its name.

After selecting a system, PUI will proceed to a screen similar to Figure 5-6. The user is then asked to select one of the following before entering into the main user area.

- Overview of the system
- User defined functions
- User defined types

The above selections represent different levels of abstraction. They are intended to be used as a guide to direct the user's attention to different areas of the source code initially.

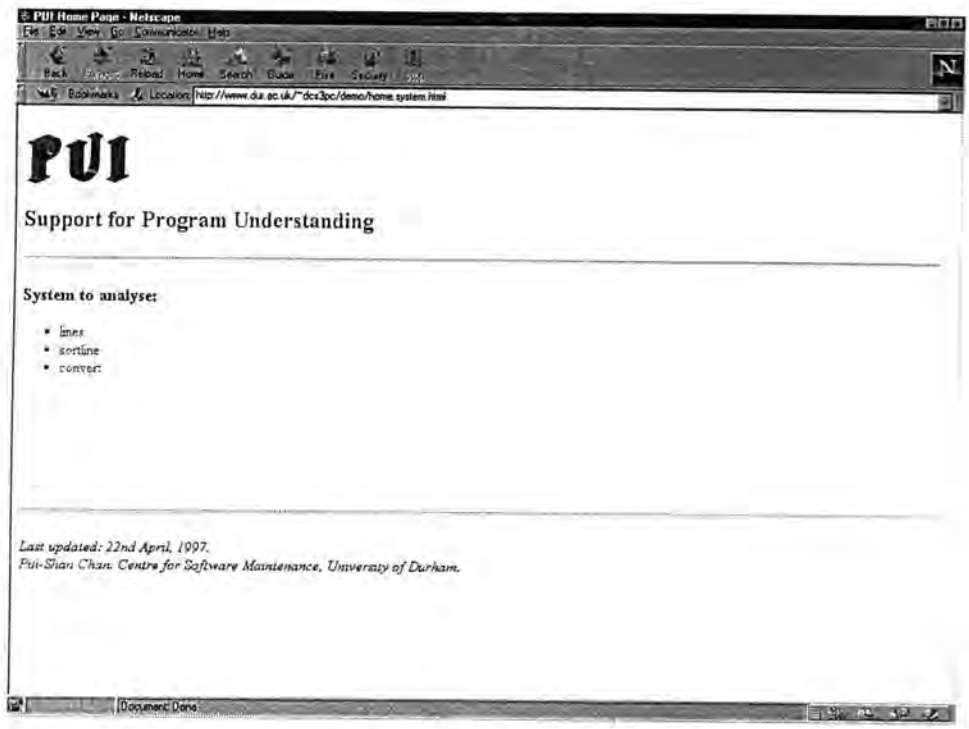

Figure 5-5 The start-up screen of PUI

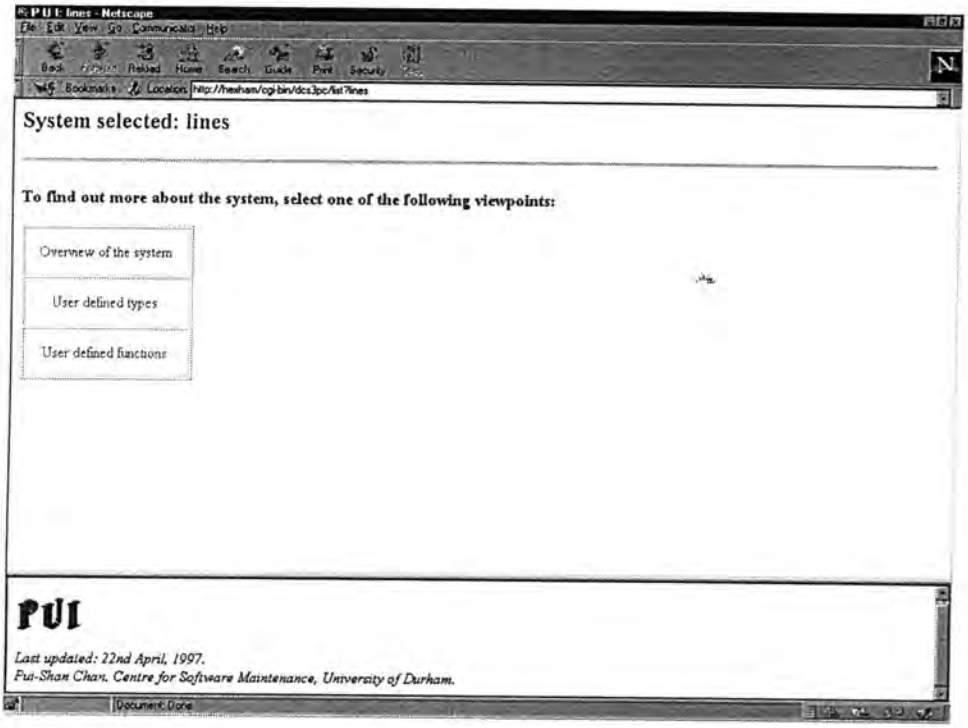

Figure 5-6 Screen showing the viewpoints

Figure 5-7 shows a typical screen of the rest of the PUI tool.

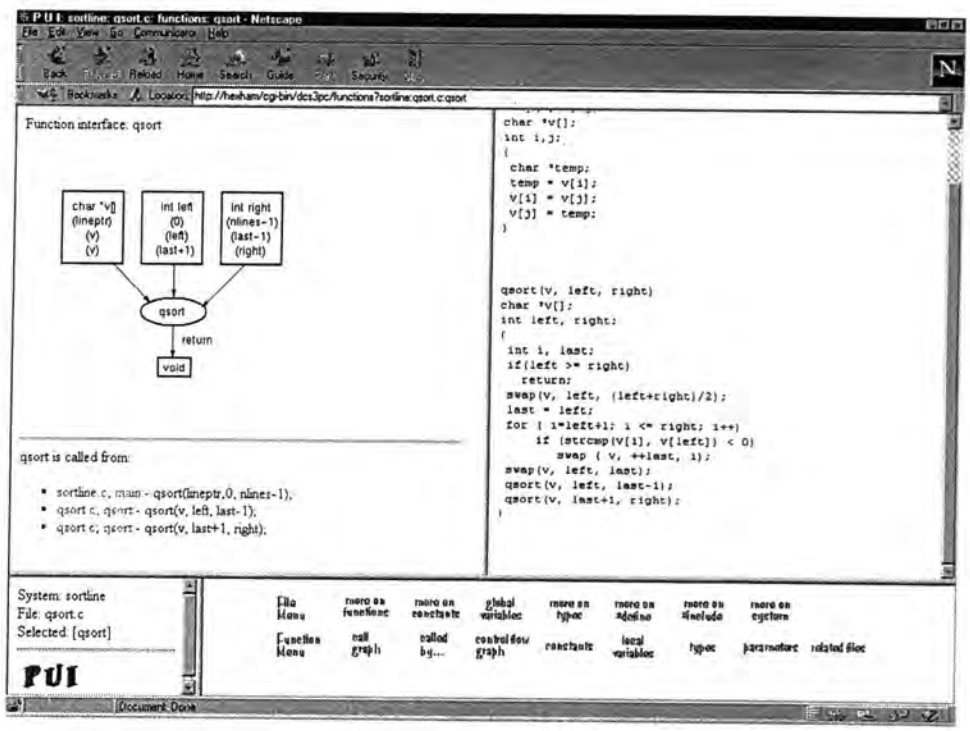

Figure 5-7 A typical screen of the PUI tool

From Figure 5-7, it is shown that the screen is divided into four frames. Starting clockwise from the top left hand corner, they are named:

- Information display
- Listing
- Control panel
- Status report

The frame 'Information display' resides in the top left hand corner. It shows information about the Program Elements and Relationships. This may includes a mixture of graphical and textual representations. The frame next to it is 'Listing'. It shows the source code listing of a program. The frame at the bottom left hand corner is 'Status report'. It records the Program Elements selected in the previous screen. The widest frame next to 'Status report' is 'Control panel'. It denotes the navigational aids designed to help the users to navigate through the hypertext documents. These aids will change according to the selected Program Elements and are based on the Relationships between Program Elements shown in Table 1.

Most of the graphical and textual representations shown in the frame 'Information display' contain hypertext links to other parts of the tool. The program shown in the frame 'Listing' is annotated with special HTML tags. A change of context in 'Information display' will cause the browser in 'Listing' to point to a different area in the program listing. Each of the key words inside the control panel represents an *implement* which retrieves information related to the Program Elements and the Program Relationships.

All of the screens in the PUI tool have a title. The title of the screen shown in Figure 5-7 reads:

P U I: sortline: qsort.c: functions: qsort - Netscape

It shows the path which leads to the current focus. P U I is the name of the prototype, **`sortline`** is the name of the system selected, **`qsort.c`** is the name of the file selected, functions is the last Program Element selected, **`qsort`** is the name of a function found in the file **`qsort.c`**, and finally, Netscape is the name of the hypertext browser.

5.5 Summary

The PUI tool presents the maintainers with a wide range of information and alternative perspectives. This is achieved by providing a mechanism to retrieve information that range from a large and crude representation to give an overview of the structure of a system, to a more fine and delicate representation. The Program Elements and Relationships are interlinked and carefully managed in the tool so information can be retrieved in a controlled and gradual manner.

The Program Relationships shown in Table 1 can be easily illustrated graphically with the respective pair of Program Elements. It is widely acknowledged that graphical representations can help maintainers to attain a better insight into the program structures. Textual information such as source code and system documentation also plays a key role in helping maintainers to form mental models of the software. Both the graphical and textual representations complement each other as the graphical representations are best suited for communicating abstract ideas and the textual representation for recording and presenting the facts behind the abstract ideas.

Chapter Six

Case Studies

6.1 Introduction

The Integrated Approach described in Chapter Four is realised in a prototype named PUI (*P*rogram *U*nderstanding *I*mplement) described in Chapter Five. This chapter demonstrates the principal use of the prototype by way of Case Studies. The Case Studies are based on two systems named **`sortline`** and **`convert`**. Demonstrations of how both the top-down and the bottom-up approaches to Program Comprehension can be utilised by using PUI will be presented in the following sections. The PUI tool is a simple browsing tool which allows maintainers to recover information as they browse through the various hypertext documents.

6.2 An Overview

6.2.1 A Generalisation of the Top-down and the Bottom-up Approaches

The following sections describe two general structures for the top-down and the bottom-up comprehension approaches.

I The Top-down Approach

In order to achieve a top-down comprehension, a maintainer needs have knowledge of the domain which is modelled by a software system and the environment which the system interacts with. Information such as the system architecture, file inclusion, function calls and data dependencies play an important part in the top-down comprehension.

Starting from the top level, a maintainer examines the system architecture to obtain an overview of the system that he is investigating.

The maintainer then examines the file inclusion relationship and identifies a set of files which may require further investigation.

The maintainer examines the function definitions within those files and identifies a set of functions, statements, data structures and/or variables which require investigation.

The maintainer formulates a set of hypotheses which are based on the type of maintenance activities he is engaged in. The source code is examined in a depth-first manner. This involves tracing function calls made within the set of functions, use of data structures and variables and the flow of control between statements and statements. This process is repeated until all the hypotheses are verified.

II The Bottom-up Approach

In order to achieve a bottom-up comprehension, a maintainer needs to have syntactic and semantic knowledge of the programming language that a software system is written in.

Starting from the source code level, a maintainer browses, locates and identifies a set of variables, data structures, statements, and/or functions which require investigation.

Related statements are then grouped together based on the maintainer's expectations. This helps the identification of design decisions behind the source code. They are generally in the form of program plans and beacons.

Information at the lower level is repeatedly abstracted into a higher level until the maintainer obtains sufficient information to build a mental model of the source code.

6.2.2 Structures of the Case Studies

The structures of Case Study One and Case Study Two are organised as follows. The Case Studies include two systems named **`sortline`** and **`convert`** which are written for different purposes. They are also different in size and complexity. Each Case Study will begin with a description of the contents of the programs concerned. This will be followed by a description of a scenario and a list of expected changes/results. Demonstrations of the use of the top-down and bottom-up approaches to Program Comprehension will be presented together will a summary for each approach at the end.

6.3 Case Study One

6.3.1 Content of Programs

The system **sortline** contains three program files:

- **sortline.c**
- **qsort.c**
- **qsort.h**

The source code for the system **sortline** is taken from the book *The C Programming Language* [Kerg88] from pages 108 to 110. It has been modified so that the original source code spans across three different program files named above. A complete listing can be found in Appendix A. The purpose of **sortline** is to read in a number of lines of text (maximum of ten lines), and to sort and print them out in alphabetical order.

6.3.2 Scenario Description

The purpose of this scenario is to modify the input to the system **sortline** so that it accepts only integer inputs. In addition, the modification should not change the order of the output, i.e., the numbers should be printed out in ascending order as intended in the original system.

The system **sortline** accepts character inputs at present. The source code contains a function named **alloc**, which emulates the C library function **malloc**. All the memory management and allocation in **sortline** is done via this function.

Demonstrations of how the top-down and the bottom-up approaches to Program Comprehension can be utilised using PUI will be presented in the following sections.

6.3.3 Expected Changes

The modification will involve changes in data structures and any function definition which uses the data types. A complete understanding of how the input data is stored and processed is essential before the commencement of any modification. The following shows the list of changes which are necessary for the modification.

I File sortline.c

The following statements which deal with dynamic memory allocation will be deleted:

```
#define MAXLEN   30 /* length of input line */ ..... [S1]
#define ALLOCSIZE 100 /* available space */ ..... [S2]
static char allocbuf[ALLOCSIZE]; ..... [S3]
```

```
static char *allocp = allocbuf;
```
[S4]

The function **alloc** will be removed. [S5]

The global data structure will change to:

```
int lineptr[MAXLINES];
```
[S6]

The parameter declaration of the function **getline** will change to:

```
int getline (s)
int *s;
```
[S7]

The definition of the function **getline** will change to:

```
int getline (s)
int *s;
{
   int c;
   c = scanf("%d", s);
   return c;
}
```
[S8]

The parameter declaration of the function **readlines** will change to:

```
int readlines(lineptr, maxlines)
int lineptr[];
int maxlines;
```
[S9]

The definition of the function **readlines** will change to:

```
int readlines(lineptr, maxlines)
int lineptr[];
int maxlines;
{
   int nlines, line;
   nlines = 0;
   while (getline(&line) > 0)
   {
      if (nlines >= maxlines)
         return -1;
      lineptr[nlines++] = line;
   }
   return nlines;
}
```
[S10]

The parameter declaration of the function **writelines** will change to:

```
writelines(lineptr, nlines)
int lineptr[];
int nlines;
```
[S11]

The definition of the function **writelines** will change to:

```
writelines( lineptr, nlines) ..... [S12]
int lineptr[];
int nlines;
{
   while (nlines-- > 0)
      printf("%d\n", *lineptr++);
}
```

II File qsort.h

The parameter declaration of the function **swap** will change to:

```
swap(v, i, j) ..... [S13]
int v[];
int I, j;
```

The parameter declaration of the function **qsort** will change to:

```
qsort(v, left, right) ..... [S14]
int v[];
int left, right;
```

III File qsort.c

The definition of the function **swap** will change to:

```
swap(v, i, j) ..... [S15]
int v[];
int I, j;
{
   int temp;
   temp = v[i];
   v[i] = v[j];
   v[j] = temp;
}
```

The definition of the function **qsort** will change to:

```
qsort(v, left, right) ..... [S16]
int v[];
int left, right;
{
   int i, last;
   if(left >= right)
      return;
   swap(v, left, (left+right)/2);
   last = left;
   for ( i=left+1; i <= right; i++)
      if (v[i] < v[left])
         swap ( v, ++last, i);
   swap(v, left, last);
   qsort(v, left, last-1);
   qsort(v, last+1, right);
}
```

6.3.4 Using a Top-down Approach

I Detailed Description

The following shows a demonstration on how PUI can help to carry out the modification by following a top-down approach.

On starting up the PUI tool, a user will be greeted by a screen as shown in Figure 5-5. Select the system **sortline** by clicking on its name.

The PUI tool will bring the user to the screen similar to Figure 5-6. Select "Overview of the System" to reveal the system architecture, together with a list of files which make up the system. The screen is shown in Figure 6-1.

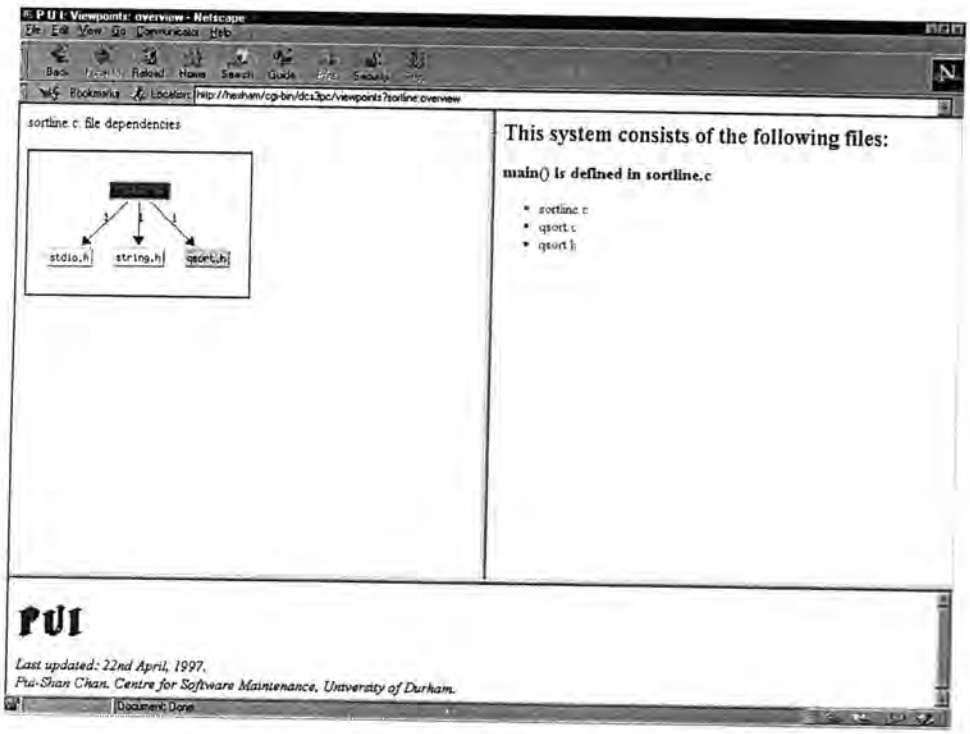

Figure 6-1 Screen showing the overview of the system **sortline**

This helps to give the user an initial impression of the system as a whole. The number of files which constitute a system denotes a simple complexity measure. The graphical representation in the frame 'Information display' on the left illustrates the file inclusion relationship.

The title of this screen is:

P U I: Viewpoints: overviews - Netscape

It reflects the selection of "Overview of the system".

The PUI tool has determined that function **`main()`** is defined in the file **`sortline.c`**. A click on "sortline.c" in the frame 'Listing' in Figure 6-1 brings the user to the screen shown in Figure 6-2.

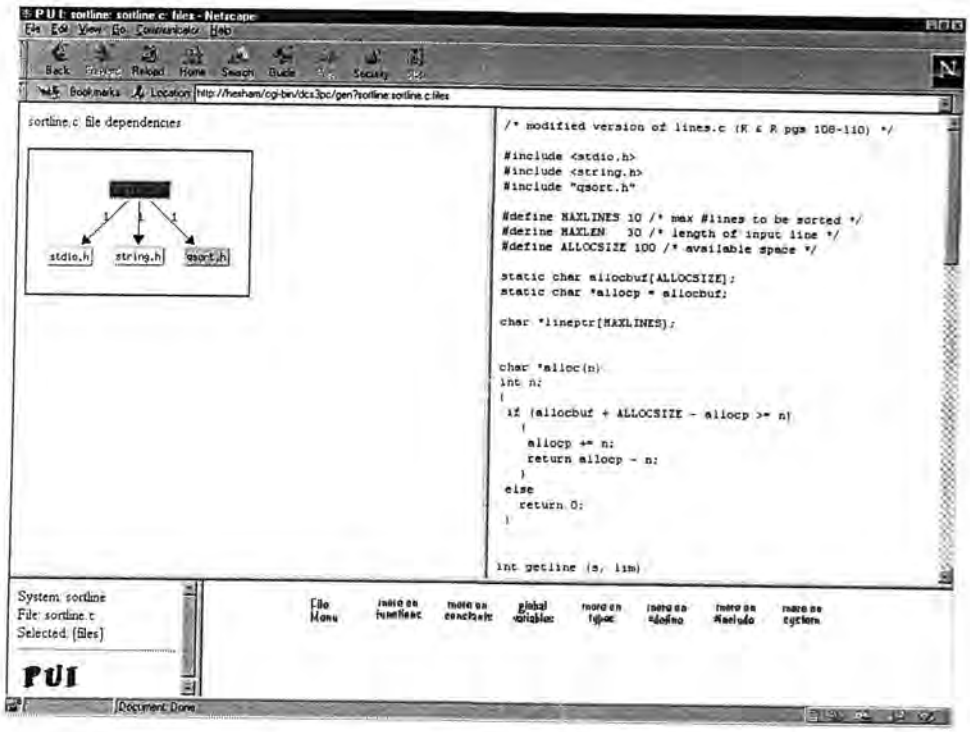

Figure 6-2 Screen showing information regarding the file **`sortline.c`**

The title of this screen has changed to:

P U I: sortline: sortline.c: files - Netscape

It reflects the path which leads to the current hypertext document. The system selected is **`sortline`**, the file selected is **`sortline.c`**, and the Program Element selected is **File**.

The scroll bar in the frame 'Listing' allows the user to browse through the source code and examine the structure of **`sortline.c`**. Note that the bottom half of the screen in Figure 6-2 has changed. The frame 'Status report' now shows the information on the selections so far: the system selected is **`sortline`**, the file selected is **`sortline.c`**, and the Program Element selected is **File**. The frame

'Control panel' has also changed. It shows a list of *implements* that are available for the Program Element **File**.

Return to Figure 6-1 by selecting the *Back* button in Netscape's own menu system in the second row. Select "qsort.h" to reveal more information about this file. Return to Figure 6-1 by selecting the *Back* button. Select "qsort.c" to reveal more information about this file. Return to Figure 6-1 by selecting the *Back* button. It is revealed that the files **`qsort.h`** and **`qsort.c`** contain the function declarations and definitions of **`swap`** and **`qsort`** respectively. It is deduced that the data structure related to the system input is declared in the file **`sortline.c`**.

Proceed to Figure 6-2 by selecting "sortline.c" in the frame 'Listing' shown in Figure 6-1. Retrieve the global variable declarations by selecting the *implement* "global variables" in the frame 'Control panel'. The screen is shown in Figure 6-3.

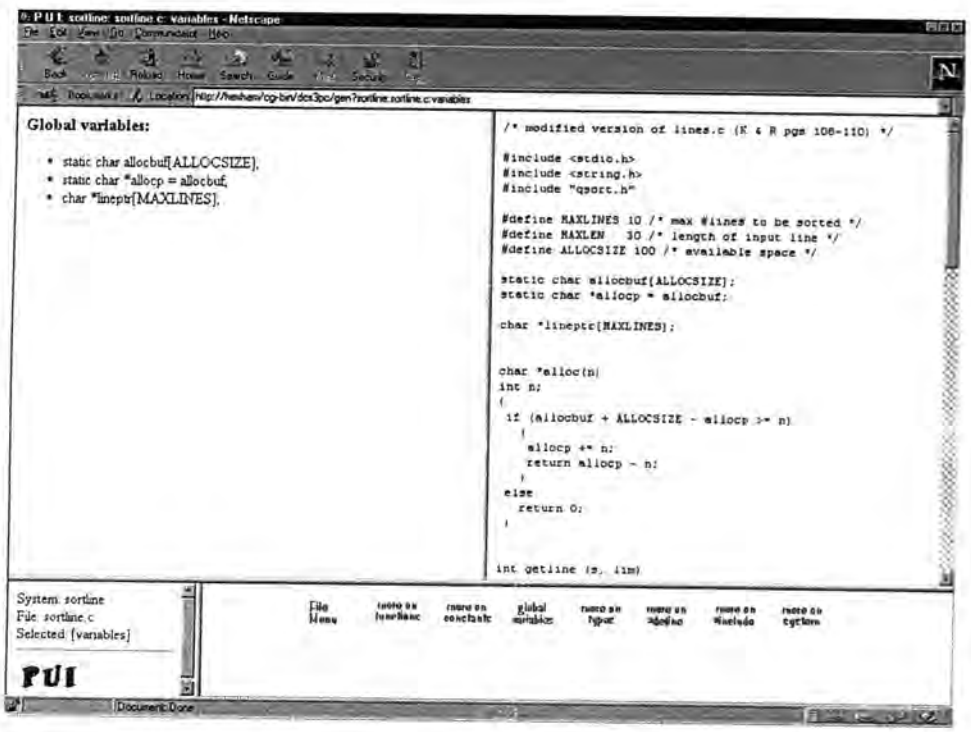

Figure 6-3 Screen showing the global data declarations in the file **`sortline.c`**

An examination of the global variable declarations in the frame 'Information display' leads to the deduction that:

```
char *lineptr[MAXLINES];
```

is the data structure that stores the input to the system **sortline**.

This data structure has to be changed in order to comply with the modification. It is changed to:

```
int lineptr[MAXLINES];
```

in the file **sortline.c**. Instead of declaring an array of pointers to strings, the declaration is changed to an array of integers. The change [S6] is complete.

Select the *implement* "more on function" in the frame 'Control panel' in Figure 6-3. It retrieves a list of functions which are defined in the file **sortline.c**. This is shown in Figure 6-4.

Note that the title of the screen has changed again. It reflects the change in the selection of the Program Element. Select "main" in the frame 'Information display' in Figure 6-4 to retrieve more information on the function. The result is shown in Figure 6-5. It shows the call graph of the function **main()**. The frame 'Listing' has positioned itself to reveal the definition of the function. The frame 'Control panel' in Figure 6-5 now reveals more *implements*.

The control flow graph of the function **main()** can be retrieved by selecting the *implement* "control flow graph" in the frame 'Control panel' in Figure 6-5. The screen is shown in Figure 6-6.

From the control flow graph, the sequence of function calls in the function **main()** is revealed. The first function call made within the function **main()** is **readlines**. The next function call is dependent on the state of the system **sortline**. The continued sequence can be either **qsort** and **writelines**, or **printf**. Each of these functions may have some impact on the global variable **lineptr** and will be examined in turn.

A closer examination of these function definitions reveals that these functions communicate by passing the variable **lineptr** as an actual argument. The parameter declarations of each of these functions must be modified accordingly as a result.

Select "readlines" in the graphical representation in Figure 6-6 to retrieve more information on the function. Select the *implement* "parameters" in the 'Control panel'. The screen is shown in Figure 6-7.

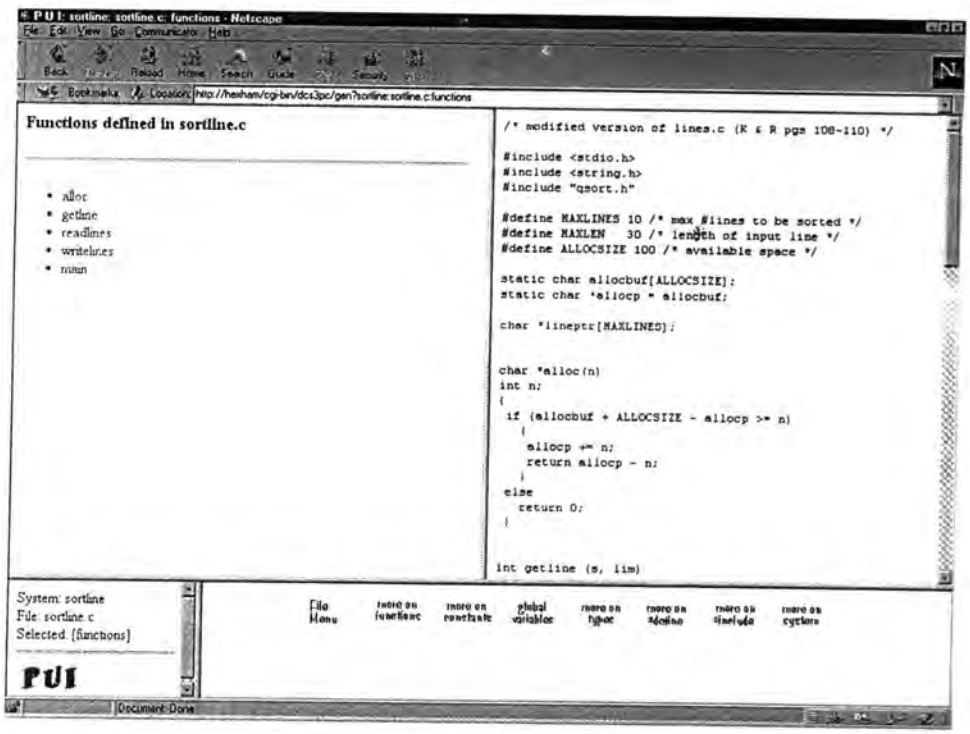

Figure 6-4 Screen showing the list of functions defined in the file **sortline.c**

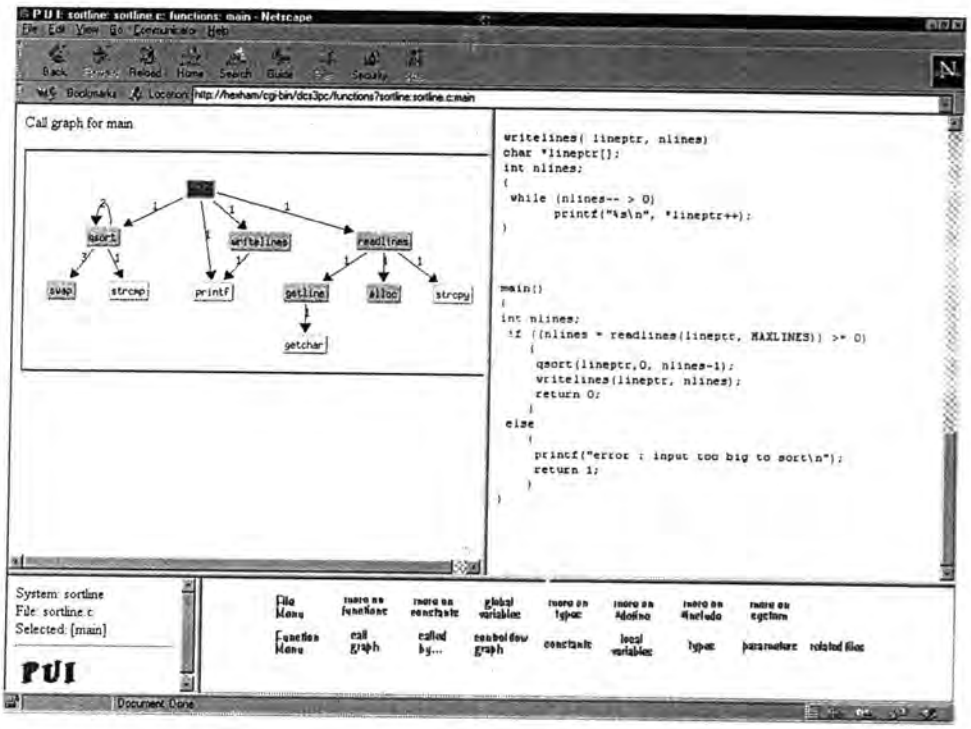

Figure 6-5 Screen showing information regarding the function **main()**

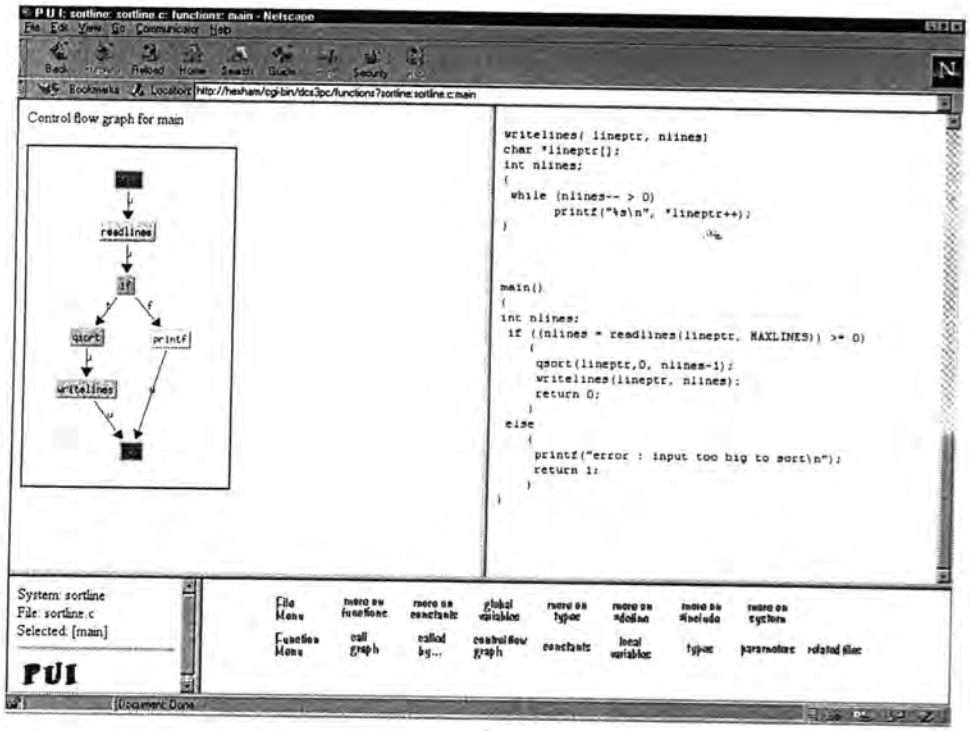

Figure 6-6 Screen showing the control flow graph of the function **`main()`**

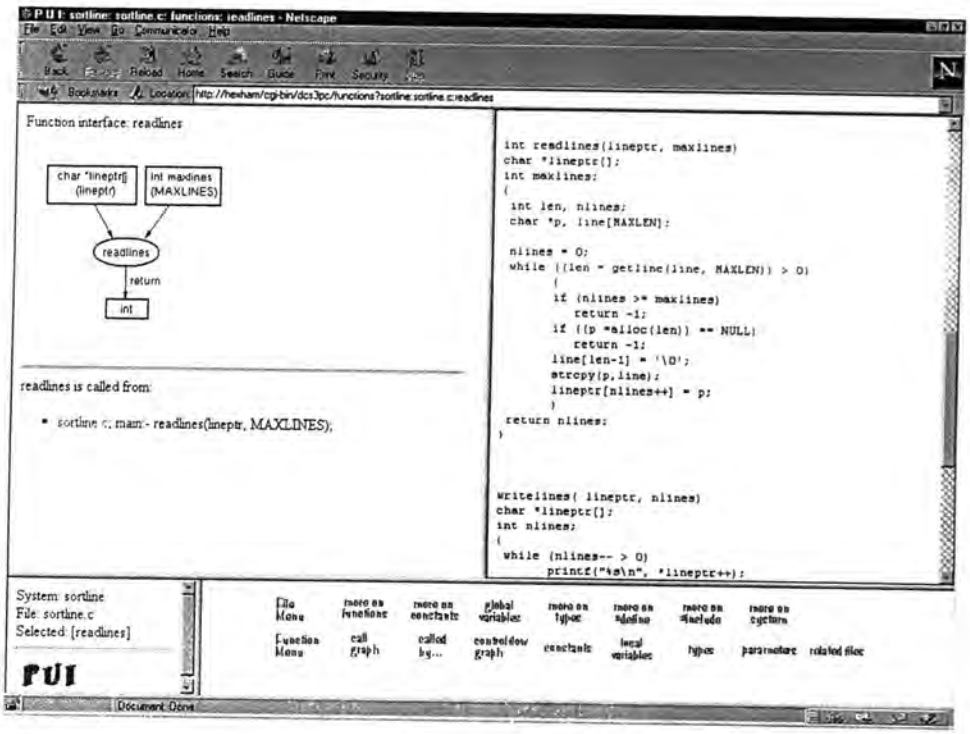

Figure 6-7 Screen showing the function interface of the function **`readlines`**

The parameter declaration of **lineptr** in the function **readlines** is changed to:

```
int lineptr[];
```

in the file **sortline.c** to reflect the change in the global data structure. The change [S9] is complete.

Select the *implement* "control flow graph" in the 'Control panel' in Figure 6-7. It reveals that the sequence of function calls made in the function **readlines** is **getline**, **alloc** and **strcpy**. The screen is shown in Figure 6-8. These functions will be examined in turn.

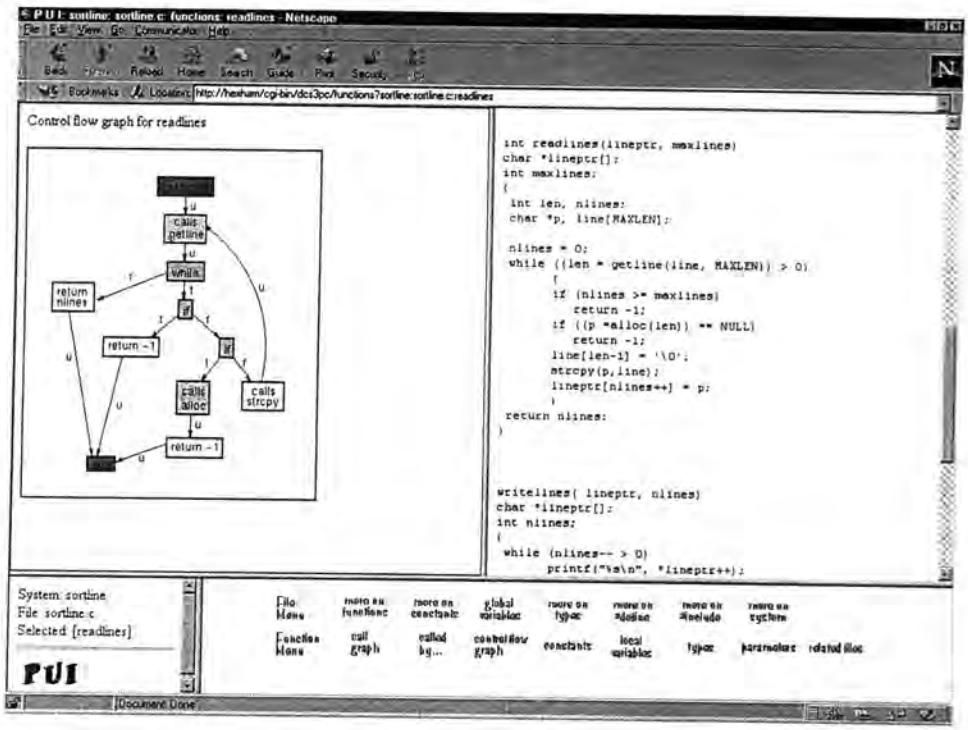

Figure 6-8 Screen showing the control flow graph of the function **readlines**

Select "getline" to retrieve more information on the function in the graphical representation in Figure 6-8. It reveals that a library function call to **getchar()** is made. It is determined that the input to **sortline** is stored in the argument **s** in the function **getline**. The argument **lim** is used for array bound check. To reflect the change in the global data structure, the type of the argument **s** is changed to:

```
int *s;
```

in the file **sortline.c**. The argument **lim** is now redundant and is eliminated as a result. The change [S7] is complete.

The function definition of the function **getline** is changed to:

```
int getline (s)
int *s;
{
   int i;
   i = scanf("%d", s);
   return i;
}
```

in the file **sortline.c**. The change [S8] is complete.

The next function to be examined is **alloc**. As explained in section 6.3.2, **alloc** is a function which emulates the C library function **malloc**. The system **sortline** now accepts integer inputs and therefore no dynamic memory allocation is required. The function **alloc** can be removed. The change [S5] is complete.

Variables which are accessed by the function **alloc** are limited to the argument **n**, the identifier **ALLOCSIZE**, and the global variables **allocbuf** and **allocp**. An examination of the identifier and the global variables reveals they are not used by any other function. Thus, the argument **n**, the identifier **ALLOCSIZE**, and the variables **allocbuf** and **allocp** can be removed from the file **sortline.c**. The changes [S3], [S4] and [S2] are complete.

The next function to be examined is **strcpy**. The function call to **strcpy**, which deals with string manipulations, in the function **readlines** can be eliminated to reflect the change in the global data structure.

All the functions that are referred to in the function **readlines** have been dealt with. The reference to the identifier **MAXLEN** in the function **readlines** is eliminated to reflect the change in the definition of the function **getline**. A closer examination reveals that the identifier **MAXLEN** is accessed only by the function **readlines**. Thus, the following statement:

```
#define MAXLEN   30 /* length of input line */
```

is removed from the file **sortline.c**. The change [S1] is complete.

The function definition of **readlines** is changed to:

```
int readlines(lineptr, maxlines)
int lineptr[];
```

```
int maxlines;
{
   int nlines, line;

   nlines = 0;
   while (getline(&line) > 0)
      {
         if (nlines >= maxlines)
            return -1;
         lineptr[nlines++] = line;
      }
   return nlines;
}
```

in the file **sortline.c**. The change [S10] is complete. The function **qsort** is to be examined next.

Select the *implement* "more on functions" in Figure 6-8 to retrieve a list of functions which are defined in the file **sortline.c**. The screen is shown in Figure 6-4. Select "main" in the frame 'Information display' to retrieve the screen shown in Figure 6-5. Select the *implement* "control flow graph" to retrieve the screen shown in Figure 6-6. Select "qsort" in the graphical representation in the frame 'Information display' to retrieve more information on the function. The screen is shown in Figure 6-9.

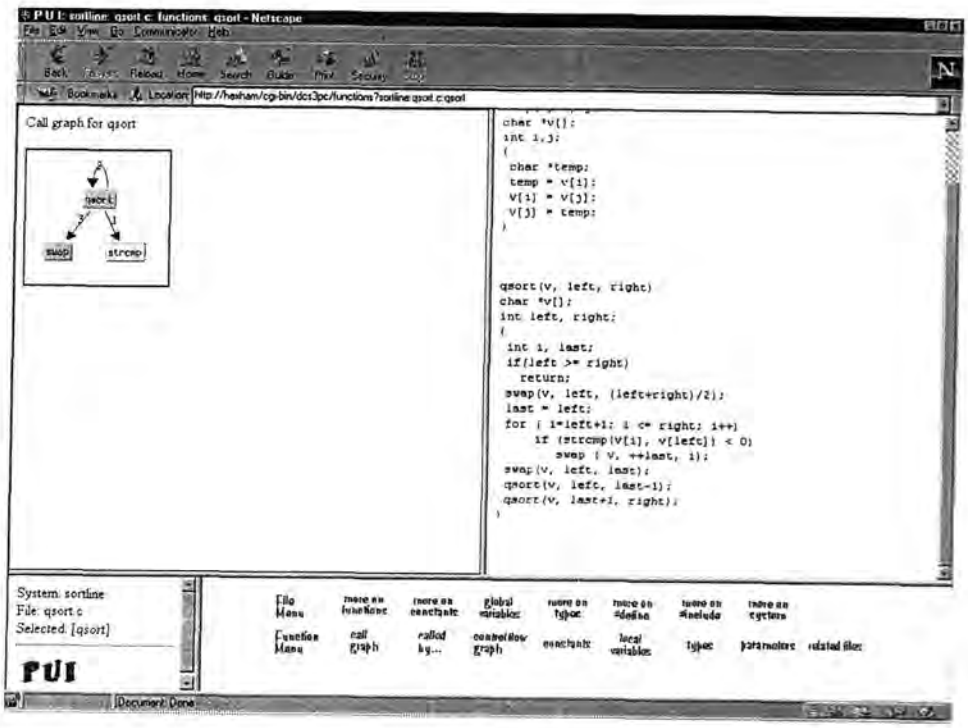

Figure 6-9 Screen showing information regarding the function **qsort**

The title of this screen is:

P U I: sortline: qsort.c: functions: qsort - Netscape

It shows the path which leads to the current hypertext document. The system selected is **sortline**, the file selected is **qsort.c**, and the Program Element selected is **Function**, and the name of the function selected is **qsort**.

The function declaration and definition of the function **qsort** are found in the files **qsort.h** and **qsort.c** respectively. Within the PUI tool, comprehension is not bounded by the physical locations of the various Program Elements. The title of the hypertext document and the frame 'Status report' are used to remind the user of the locations of the Program Elements last selected.

From the name of the function, it is conjectured that **qsort** performs some kind of sorting algorithm on a data structure. After inspecting the definition, it is determined that **qsort** is used to perform a quicksort algorithm on a data structure which at present is an array of pointers to strings.

The type of the formal argument **v** in the function **qsort** is found and changed to:

```
int v[];
```

in the file **qsort.h** to reflect the change in the global data structure. The change [S14] is complete.

Select the *implement* "control flow graph" in Figure 6-9. It reveals that the sequence of function calls made in the function **qsort**. The sequence is **swap**, **strcmp** and recursive calls to **qsort** itself. The function **swap** is to be examined next.

Select "swap" in the graphical representation to retrieve more information on the function. The result is shown in Figure 6-10.

The PUI tool has determined that no function call is made in the function **swap**. The function **qsort** passes its formal argument **v** to the function **swap** as its actual argument. The parameter definition of **v** in the function **swap** is found and changed to:

```
int v[];
```

in the file **qsort.h** to reflect the change in the global data structure. The change [S13] is complete.

The local variable **temp** in the function **swap** is defined to hold an array of characters. It is found and changed to:

```
int temp;
```

in the file **qsort.c** to reflect the change in the global data structure. The change [S15] is complete.

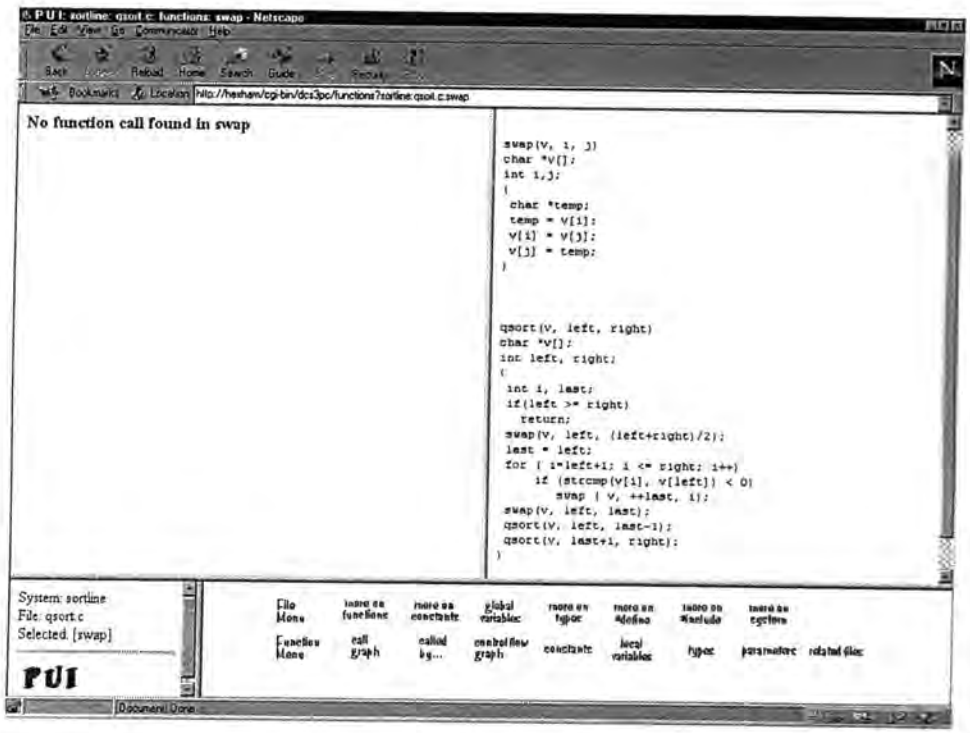

Figure 6-10 Screen showing information regarding the function **swap**

The library function **strcmp** is next to be examined. It is called within the function **qsort**. Select "qsort" in the frame 'Listing' in Figure 6-10 to reveal more information on the function. The call to **strcmp** is removed. The **if** statement in the function **qsort** is changed from:

```
if (strcmp(v[i], v[left]) < 0)
   swap ( v, ++last, i);
```

to

```
if (v[i] < v[left])
   swap(v, ++last, i);
```

in the file **qsort.c**. The change [S16] is complete.

From the control flow graph of the function **main()**, it reveals that one of the sequence of function calls is **readlines**, **qsort** and then **writelines**. The functions **readlines** and **qsort** have been dealt with. The function **writelines** is to be examined next.

Select the *implement* "called by..." in Figure 6-10 to reveal a list of functions which called the function **qsort**. It reveals that this function is called by the functions **main()** and **qsort**. Select "main" to reveal more information on the function. The screen is shown in Figure 6-5. Select "writelines" in the graphical representation to reveal more information on the function.

The type of the formal argument **lineptr** in the function **writelines** is changed to:

```
int lineptr[];.
```

in the file **sortline.c** to reflect the change in the global data structure. The change [S11] is complete.

The function call to **printf** is changed to:

```
printf("%d\n", *lineptr++);
```

in the file **sortline.c** to reflect the change in the global data structure. The change [S12] is complete.

From the control flow graph of the function **main()**, it reveals that the other sequence of function calls in **main()** is **readlines** and **printf**. The last function to be examined is the function **printf**. An examination of the function call reveals that no further change is needed.

The modification is complete. The input to the system **sortline** has been changed from a character-based input to an integer-based input. The output of the system **sortline** produces a set of numbers which are printed in ascending order. The revised program files can be found in Appendix B.

II Summary

The following is a summary of a list of tasks performed during the top-down comprehension.

Locate the source files for the system **sortline**. Examine the architecture of the system **sortline**.

Examine the relationship file inclusion to get a feel of the complexity of the system.

Locate the file which has the definition of the function **main()**. The file is **sortline.c**.

Examine the global variable and type declarations in the file **sortline.c**. The global variable and type declarations are changed. ..... [S6]

The functions **readlines**, **qsort**, **writelines** and **printf** are called within the function **main()**.

The parameter declaration of the function **readlines** is found and changed in the file **sortline.c**..... [S9]

The functions **getline**, **alloc** and **strcpy** are called within the function **readlines**.

The parameter declaration of the function **getline** is found and changed in the file **sortline.c**. ..... [S7]

The definition of the function **getline** is changed in the file **sortline.c**. ..... [S8]

The function **alloc** in the file **sortline.c** is removed after the change in the global data structure. ..... [S5]

The following statements are removed from the file **sortline.c** as the variables are only used in the function **alloc**.

```
static char allocbuf[ALLOCSIZE];
static char *allocp = allocbuf;
#define ALLOCSIZE 100 /* available space */
```
[S3] [S4] [S2]

A statement is removed from the file **sortline.c** as the identifier **MAXLEN** is only used in the function **readlines**. ..... [S1]

The function call to **strcpy** in the function **readlines** is removed. The definition of the function **readlines** is changed in the file **sortline.c**. ..... [S10]

The parameter declaration of the function **qsort** is found and changed in the file **qsort.h**..... [S14]

The function **qsort**, **swap** and **strcmp** are called within the function **qsort**.

The parameter declaration of the function **swap** is found and changed in the file **qsort.h**. ..... [S13]

The definition of the function **swap** is changed in the file **qsort.c**. ..... [S15]

The function call to **strcmp** in the function **qsort** is removed. The definition of the function **qsort** is changed in the file **qsort.c**. .....[S16]

The parameter declaration of the function **writelines** is found and changed in the file **sortline.c**. .....[S11]

The definition of the function **writelines** is changed in the file **sortline.c**. .....[S12]

The final function call made in the function **main()** is to the function **printf**. No change is needed for this.

The modification is complete.

6.3.5 Using a Bottom-up Approach

I Detailed Description

The following shows a demonstration on how PUI can help to carry out the modification by following a bottom-up approach.

On starting up the PUI tool, a user will be greeted by a screen as shown in Figure 6-1. Select the system **sortline** by selecting its name.

The PUI tool will bring the user the screen similar to Figure 6-2. Select "User defined functions" to reveal the list of functions defined in each of the program files. The screen is shown in Figure 6-11.

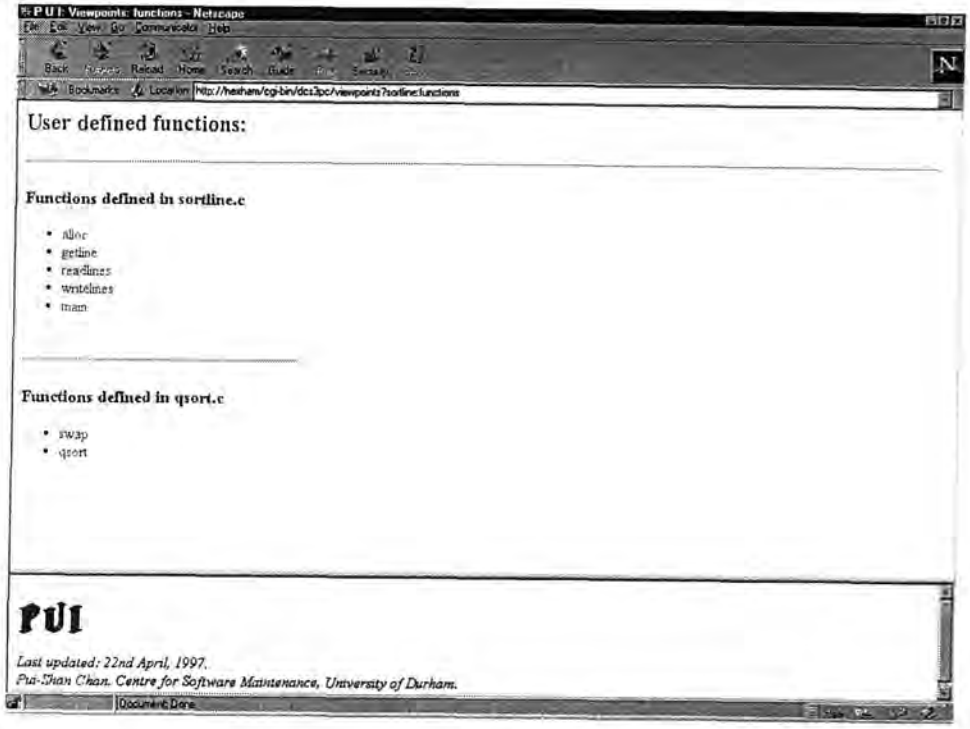

Figure 6-11 Screen showing the list of functions defined in each of the files in the system **sortline**

This helps to give the user an initial impression of the system at the function level. It is revealed that the file **sortline.c** has five function definitions including the function **main()**, and the file **qsort.c** has two function definitions.

Select "sortline.c" in Figure 6-11 to retrieve more information on the file. This brings the user to the screen shown in Figure 6-2. The files **qsort.h** and **qsort.c** can also be accessed in Figure 6-2.

An examination of the files **sortline.c**, **qsort.h** and **qsort.c** reveals that the data structure which holds the input to the system is declared in the file **sortline.c**.

Select "sortline.c" in Figure 6-11 to retrieve more information on the file. Select the *implement* "global variables" in the frame 'Control panel' to retrieve the global variable declarations in the file **sortline.c**. The screen is shown in Figure 6-3. An examination of the variable declarations leads to the deduction that:

```
char *lineptr[MAXLINES];
```

is the data structure that stores the input to the system **sortline**.

This data structure has to be changed in order to comply with the modification. It is changed to:

```
int lineptr[MAXLINES];
```

in the file **sortline.c**. Instead of declaring an array of pointers to strings, the declaration is changed to an array of integers. The change [S6] is complete.

Select "lineptr" in the frame 'Information display' shown in Figure 6-3 to retrieve more information on the variable. The screen is shown in Figure 6-12.

Select the *implement* "as parameters..." in the frame 'Control panel' in Figure 6-12. The result is shown in Figure 6-13. It reveals the type of the variable **lineptr**, and it shows that it is used as an actual argument in the functions **readlines**, **qsort** and **writelines**.

The type of the argument **lineptr** in the functions **readlines** and **writelines** is found and changed to:

```
int lineptr[];
```

in the file **sortline.c**. The changes [S9] and [S11] are complete.

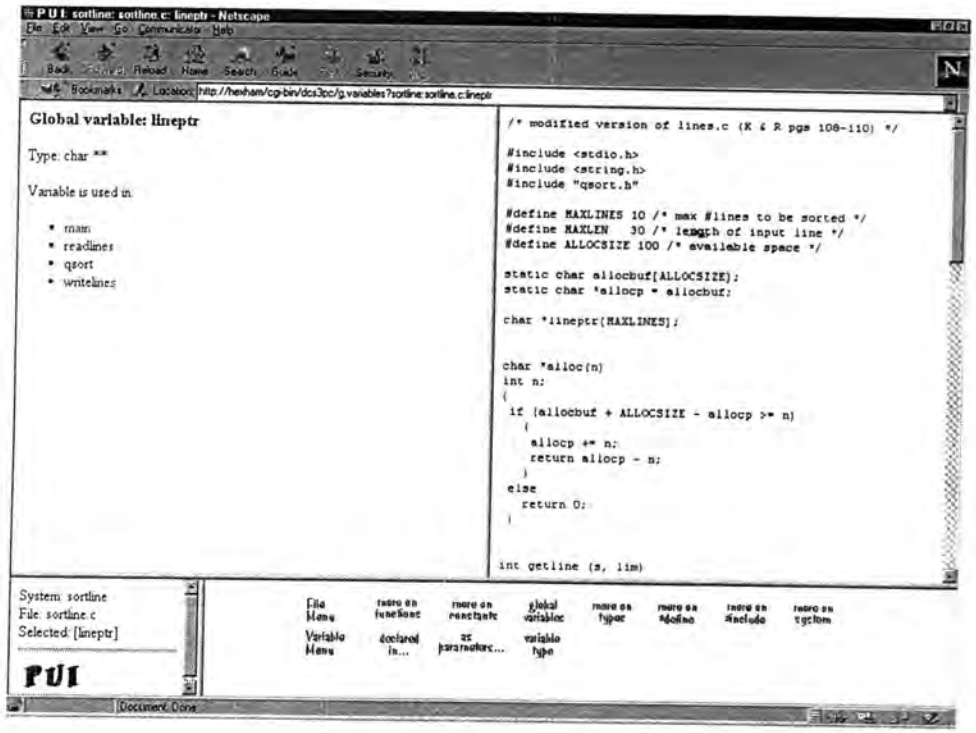

Figure 6-12 Screen showing information regarding the global variable **lineptr**

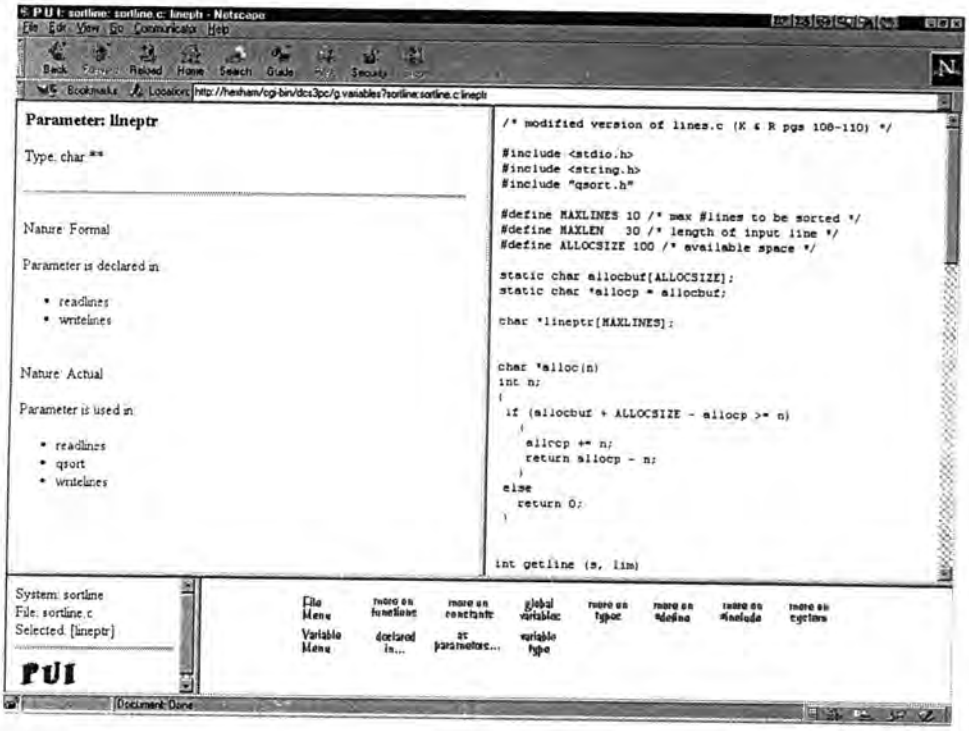

Figure 6-13 Screen showing that the global variable **lineptr** is used as an argument

The type of the argument in the function **qsort** is found and changed to:

```
int v[];
```

in the file **qsort.h** in order to reflect the change in the global data structure. The change [S14] is complete.

The next step is to examine the way the input to the program is handled.

From the search engine, a list of functions which has made function calls to library functions dealing with characters and strings are found. They are functions **getline**, **readlines** and **qsort**.

Return to Figure 6-11 by selecting the *implement* "File menu" in the frame 'Control panel' in Figure 6-13. Select "getline" in Figure 6-11 to retrieve more information on the function. The result is shown in Figure 6-14.

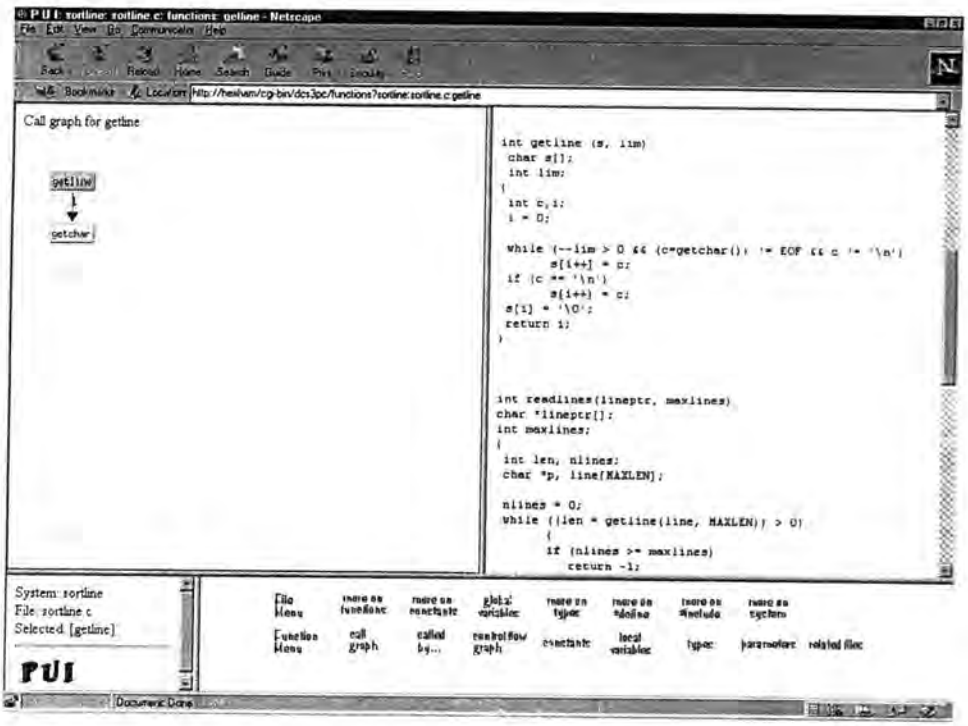

Figure 6-14 Screen showing information regarding the function **getline**

From the call graph of the function **getline**, it is confirmed that a library function call to **getchar()** is made. It is determined that the input to the system **sortline** is stored in the

argument **s**. The argument **lim** is used for array bound check. To reflect the change in the global data structure, the type of argument **s** is changed to:

```
int *s;
```

in the file **sortline.c**. The system **sortline** now accepts integer inputs and the argument **lim** is eliminated as a result. The change [S7] is complete.

The function definition of **getline** is changed to:

```
int getline (s)
int *s;
{
   int i;
   i = scanf("%d", s);
   return i;
}
```

in the file **sortline.c**. The change [S8] is complete.

Select the *implement* "called by..." in the frame 'Control panel' in Figure 6-14 to retrieve a list of functions which called the function **getline**. The screen is shown in Figure 6-15.

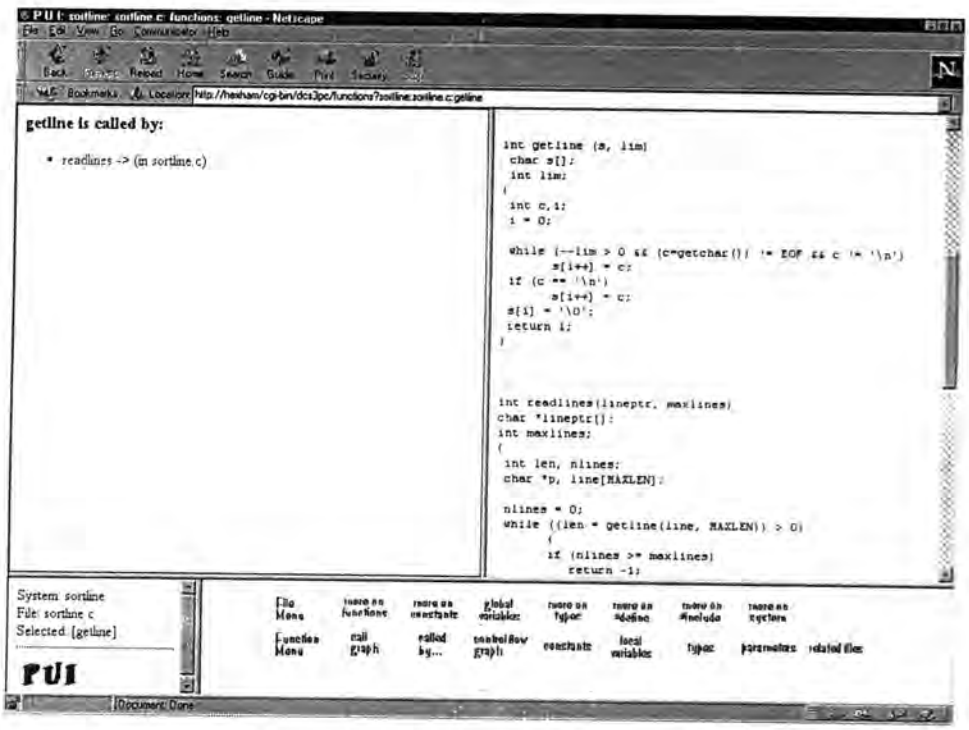

Figure 6-15 Screen showing the list of functions which called the function **getline**

It is revealed that **getline** is called by the function **readlines**. The function **readlines** is next to be examined.

Select "readlines" in the frame 'Information display' in Figure 6-15 to retrieve more information on the function. It is revealed that function calls to the functions **getline**, **alloc** and **strcpy** are made.

The call to the function **getline** is altered in the function **readlines** in order to reflect the change in its definition. The function **alloc** is to be examined next.

As mentioned in section 6.3.2, **alloc** is a function which emulates the system function **malloc** by providing its own memory management and allocation. The system sortline now accepts integer inputs and so the function **alloc** is redundant. Select the *implement* "called by..." in the frame 'Control panel' in Figure 6-15 to retrieve a list of functions which called the function **alloc**. The screen is shown in Figure 6-16. It is revealed that the function **alloc** is only called by the function **readlines**. The function **alloc** is removed from the file **sortline.c**. The change [S5] is complete.

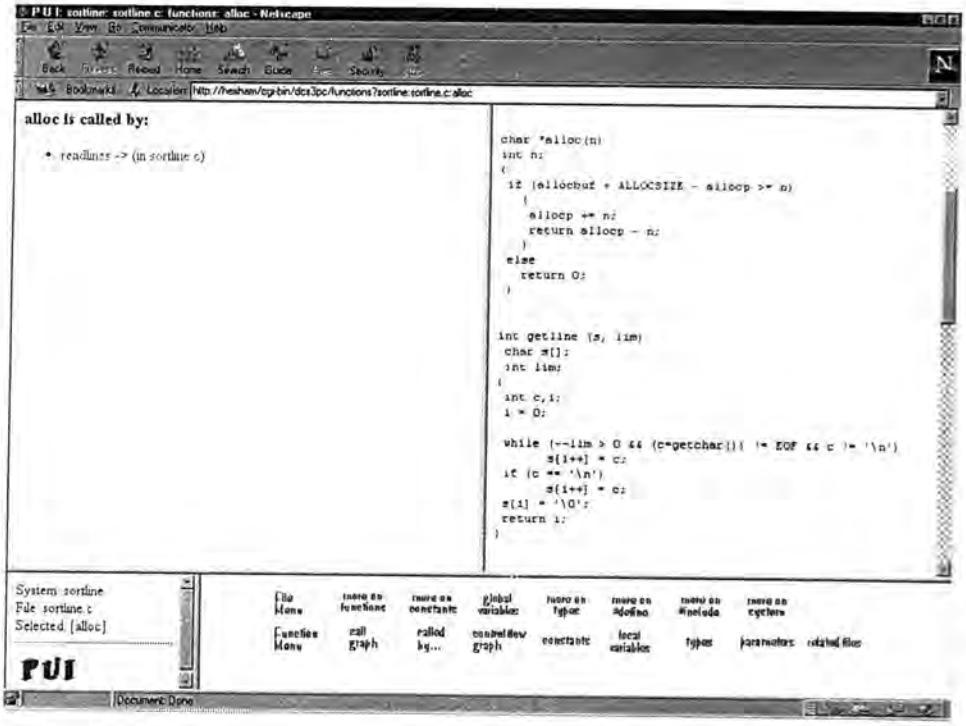

Figure 6-16 Screen showing the list of functions which called the function **alloc**

Variables which are accessed in the function **alloc** are limited to the argument **n**, the identifier **ALLOCSIZE** and the global variables **allocbuf** and **allocp**. An examination of the identifier and variables reveals that they are not used by any other function. Thus, the argument **n** and the following statements:

```
static char allocbuf[ALLOCSIZE];
static char *allocp = allocbuf;
```

are be removed from the file **sortline.c**. The changes [S3] and [S4] are complete.

The variable **allocbuf** refers to an identifier **ALLOCSIZE**, which is only accessed by **allocbuf**. The following statement:

```
#define ALLOCSIZE 100 /* available space */
```

is removed from the file **sortline.c**. The change [S2] is complete.

Select "readlines" in the frame 'Information Listing' in Figure 6-16 to reveal more information on the function. Select the *implement* "control flow graph" to retrieve the screen shown in Figure 6-8. It reveals that a call to the library function **strcpy** is made in the function **readlines**. The type of the local variable **line** in the function **readlines** is changed from:

```
char line[MAXLEN];        to      int line;
```

in the file **sortline.c** to reflect the change in the global data structure.

The identifier **MAXLEN** is not used by any other function and the following statement is removed:

```
#define MAXLEN   30   /* length of input line */
```

in the file **sortline.c**. The change [S1] is complete.

The definition of **readlines** is now changed to:

```
int readlines(lineptr, maxlines)
int lineptr[];
int maxlines;
{
   int nlines, line;

   nlines = 0;
   while (getline(&line) > 0)
      {
         if (nlines >= maxlines)
            return -1;
```

```
            lineptr[nlines++] = line;
        }
    return nlines;
}
```

in the file **sortline.c**. The change [S10] is complete.

From the search engine, a list of functions which has made function calls to library functions dealing with characters and strings are found. They are functions **getline**, **readlines** and **qsort**. The function **qsort** is to be examined next.

Select "qsort" in the frame 'Listing' shown in Figure 6-8 to reveal more information on the function. The result is shown in Figure 6-9.

An examination of the call graph of the function **qsort** reveals that function calls to the functions **swap**, **strcmp** and recursive calls to **qsort** itself are made. The function **swap** will be examined next.

The function **qsort** passes its formal argument **v** to the function **swap** as its actual argument. The argument definition of **v** in the function **swap** is found and changed to:

```
int v[];
```

in the file **qsort.h**. The change [S13] is complete.

The local variable **temp** in the function **swap** is defined to hold an array of characters. It is changed to:

```
int temp;
```

in the file **qsort.h** to reflect the change in the global data structure. The change [S15] is complete.

The library function call to the function **strcmp** in the function **qsort** is removed. The **if** statement in the function **qsort** is changed to:

```
if (v[i] < v[left])
```

in the file **qsort.c**. The change [S16] is complete.

Select the *implement* "call by..." shown in Figure 6-9 to retrieve a list of function which called the function **qsort**. The result is shown in Figure 6-17. It reveals that **qsort** is called by the function **main()** and is recursively called by itself.

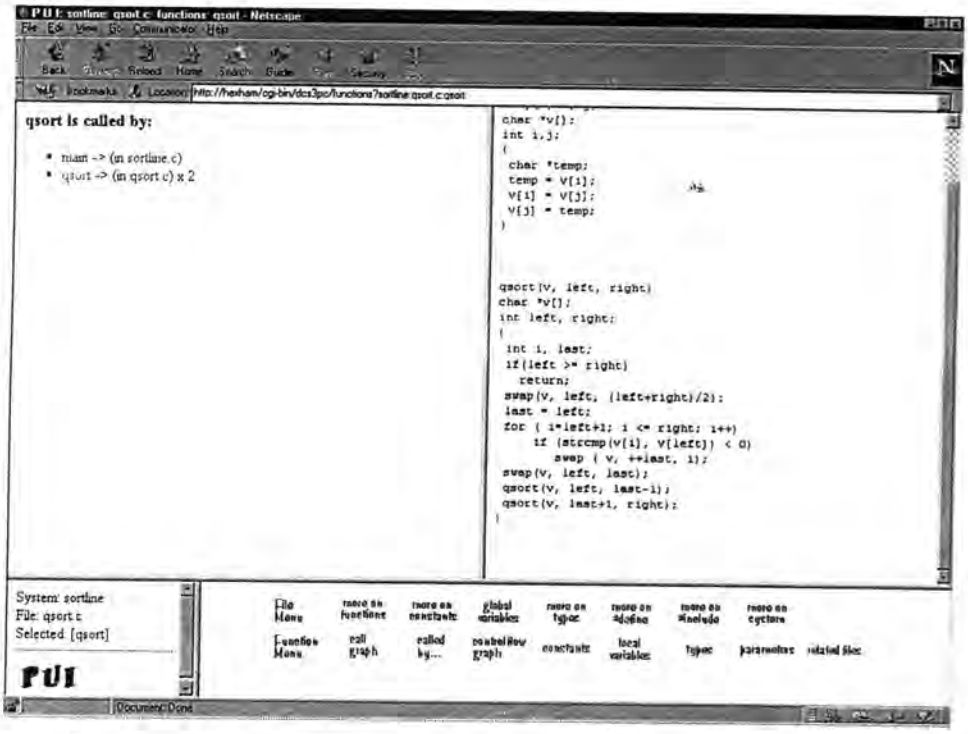

Figure 6-17 Screen showing the list of functions which called the function **qsort**

Select "main" in the frame 'Information display' in Figure 6-17 to retrieve the information on the function. The screen is shown in Figure 6-5. From the call graph of the function **main()**, it is revealed that it has made direct function calls to the functions **qsort**, **readlines**, **writelines** and **printf**. All the functions except the functions **writelines** and **printf** have been examined. The function **writelines** is to be examined next.

Select "writelines" in the graphical representation in Figure 6-5 to reveal more information on the function. The function **writelines** made one function call to **printf**. This function call is changed to:

```
printf("%d\n", *lineptr++);
```

in the file **sortline.c** in order to reflect the change in the global data structure. The change [S12] is complete.

The function **printf** which is called by the function **main()** prints an error message. It has no impact on the global data structure and is left unchanged.

The modification is complete. The input to the system **sortline** has been changed from a character-based input to an integer-based input. The output of the system **sortline** produces a set of numbers which are printed in ascending order. The revised program files can be found in Appendix B.

II Summary

The following is a summary of a list of tasks performed during the bottom-up comprehension.

Examine the architecture of the system **sortline**.

Examine file inclusion to get a feel of the complexity of the system.

Locate the file which has the definition of the function **main()**. The file is **sortline.c**.

Examine the global variable and type declarations in the file **sortline.c**. The global variable and type declaration are changed. .....[S6]

The global variable is used in the functions **readlines**, **writelines** and **qsort**.

The parameter declaration of the function **readlines** is found and changed in the file **sortline.c**.....[S9]

The parameter declaration of the function **writelines** is found and changed in the file **sortline.c**. .....[S11]

The parameter declaration of the function **qsort** is found and changed in the file **qsort.h**.....[S14]

The functions **getline**, **readlines** and **qsort** have made function calls to the library functions which deal with characters and strings.

The parameter declaration of the function **getline** is found and changed in the file **sortline.c**. .....[S7]

The definition of the function **getline** is changed in the file **sortline.c**. .....[S8]

The functions **alloc** and **strcpy** are called within the function **readlines**.

The functions **alloc** is removed from the file **sortline.c** as it is no longer required after the change in the global data structure. .....[S5]

The following statements are removed from the file **sortline.c** as the variables are only used in the function **alloc**.

```
static char allocbuf[ALLOCSIZE]; .....[S3]
static char *allocp = allocbuf; .....[S4]
#define ALLOCSIZE 100 /* available space */ .....[S2]
```

A statement is removed from the file **sortline.c** as the identifier **MAXLEN** is only used in the function **readlines**. .....[S1]

The function **strcpy** deals only with strings and therefore the function is removed. The definition of the function **readlines** is changed in the file **sortline.c**. .....[S10]

The functions **qsort**, **swap** and **strcmp** are called within the function **qsort**.

The parameter declaration of the function **swap** is found and changed in the file **qsort.h**. .....[S13]

The definition of the function **swap** is changed in the file **qsort.c**. .....[S15]

The function **strcmp** deals only with strings and therefore the function call is removed. The definition of the function **qsort** is changed in the file **qsort.c**. .....[S16]

The definition of the function **writelines** is changed in the file **sortline.c**. .....[S12]

The final function call made in the function **main()** is the function **printf**. No change is needed for this function.

The modification is complete.

6.4 Case Study Two

6.4.1 Content of Programs

The system **convert** contains twenty five program files:

- `convert.c`
- `cal.c`, `cal.h`
- `call.c`, `call.h`
- `ds.c`, `ds.h`
- `gen.c`, `gen.h`
- `mod.c`, `mod.h`
- `param.c`, `param.h`
- `prh.c`, `prh.h`
- `read.c`, `read.h`
- `send.c`, `send.h`
- `sta.c`, `sta.h`
- `use.c`, `use.h`
- `write.c`, `write.h`

The source code is developed by an in-house team from the System Application Integration Unit in the Network Integration Centre, British Telecommunications. The system **convert** is part of an existing software maintenance tool used within the department.

The purpose of this system is to convert data obtained from an analysis tool into a suitable format for the input to a front-end user interface. This is a stand-alone system with specific input and output formats.

6.4.2 Scenario Description

The purpose of this scenario is to find out the names and the format of the input data files to the system **convert**.

6.4.3 Expected Results

By way of executing the system, it is found that four different data files are required as input to the system **convert**. Figure 6-18 shows a default screen when no parameter is supplied to the system.

```
Converter v1.0f.  Written by David Heath, 1995.

Unusable number of parameters.
The option '-filename' must be given.
Usage :    convert [options]

The [options] are:
    -sta <file>
    -use <file>
    -cal <file>
    -prh <file>
    -ident <system name>
    -type <C or COBOL>
    -filename <filename>

The <files> are Xray output files to use to convert the Xray
output data to Infoflow input files.
```

Figure 6-18 The default screen when no parameter is supplied to the system **convert**

The parameters **-sta <file>**, **-use <file>**, **-cal <file>** and **-prh <file>** indicate that system **convert** takes the respective files as its raw input. The keywords **Xray** and **Infoflow** are also noted.

As this is a pure comprehension exercise, no modification is required. The following shows the names and the formats for each of the input files.

I File Format One

A default filename will be **xray.STA**. ..... [C1]

Each line will have the following format:

```
a thirty-character string ..... [C2]
a thirty-character string
a thirty-character string
a two-digit integer
a two-digit integer
a nine-character string
```

II File Format Two

A default filename will be **xray.USE**. ..... [C3]

Each line will have the following format:

```
a thirty-character string ..... [C4]
a thirty-character string
```

```
a six-digit integer
a five-digit integer
a single character
a single character
a thirty-character string
a thirty-character string
```

III File Format Three

A default filename will be **xray.CAL**. ..... [C5]

Each line will have the following format:

```
a thirty-character string..... [C6]
a thirty-character string
a single character
a thirty-character string
a thirty-character string
a six-digit integer
a three-digit integer
a thirty-character string
a five-digit integer
```

IV File Format Four

A default filename will be **xray.PRH**. ..... [C7]

Each line will have the following format:

```
a thirty-character string..... [C8]
a six-digit integer
a thirty-character string
a single character
a thirty-character string
```

6.4.4 Using a Top-down Approach

I Detailed Description

The following shows a demonstration on how PUI can help to carry out the comprehension by following a down-top approach.

On starting up the PUI tool, a user will be greeted by a screen as shown in Figure 6-1. Select the system **convert** by selecting its name.

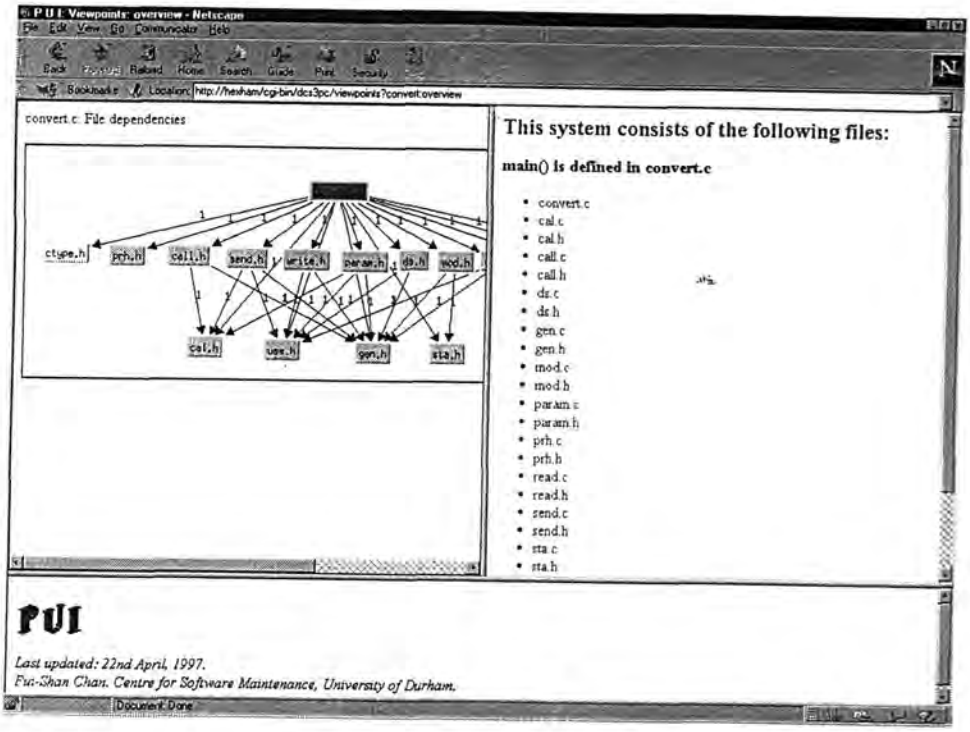

Figure 6-19 Screen showing the overview of the system **convert**

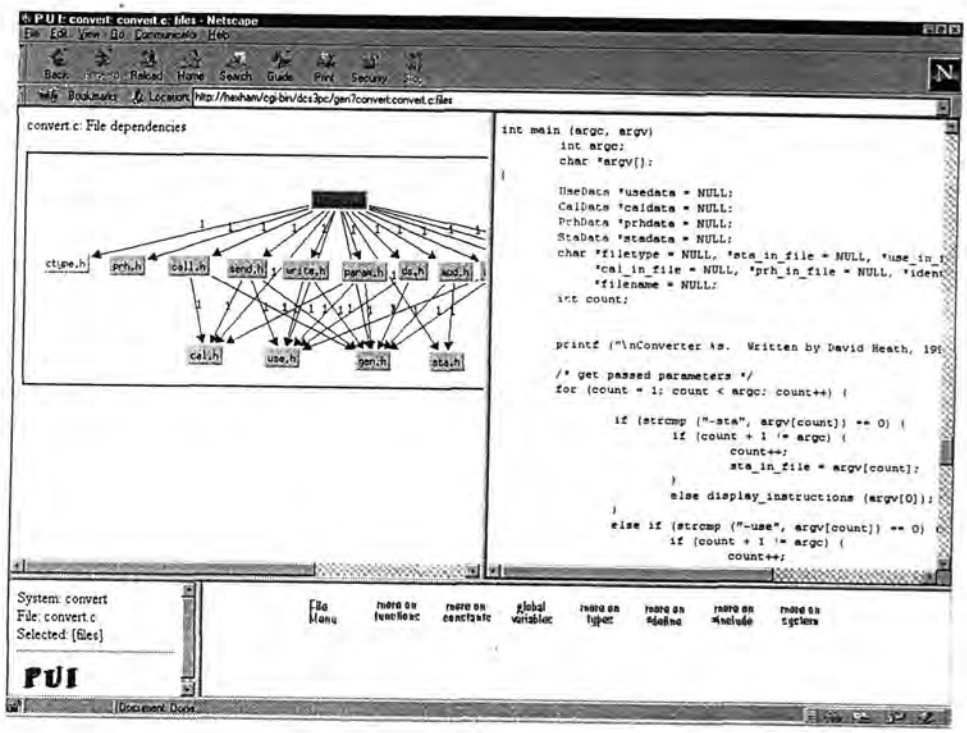

Figure 6-20 Screen showing the information regarding function **main()**

The PUI tool will bring the user to the screen similar to Figure 6-2. Select "Overview of the System" to reveal the system architecture, together with a list of files which make up the system. The screen is shown in Figure 6-19.

The first task of this comprehension is to find out the filenames of the input data files. The system has determined that the function **`main()`** is defined in the file **`convert.c`**.

Select "convert.c" in the frame 'Listing' in Figure 6-19 to retrieve more information on the file. The result is shown in Figure 6-20.

Select the *implement* "more on #define" in the frame 'Control panel' in Figure 6-20 to inspect the **`#define`** statements. The result is shown in Figure 6-21.

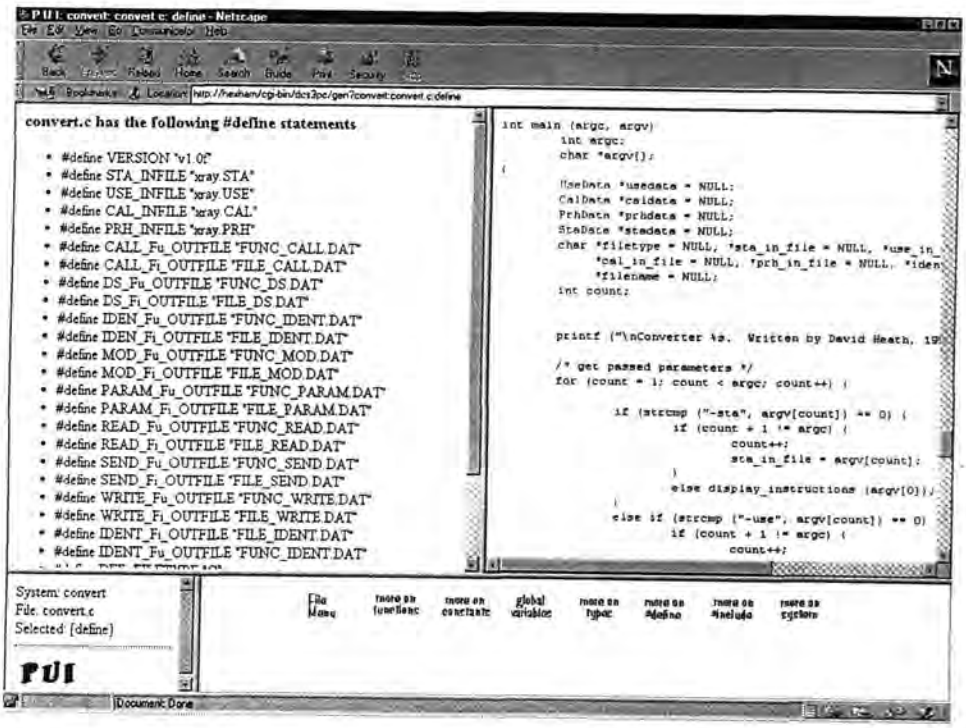

Figure 6-21 Screen showing the **`#define`** statements in the file **`convert.c`**

The identifiers used in the **`#define`** statements are mostly self-explanatory. There are predominately two groups of names which contain the phase INFILE and the phase OUTFILE. The ones containing the phase INFILE are:

```
#define STA_INFILE "xray.STA"
#define USE_INFILE "xray.USE"
#define CAL_INFILE "xray.CAL"
```

```
#define PRH_INFILE "xray.PRH"
```

These names correspond to the list of parameters supplied to the system as shown in Figure 6-18. It is conjectured that these identifiers hold the default input filenames to the system.

An examination of the source code reveals that these identifiers are used in the function **main()**. Select the *Back* button in Netscape's own menu system in the second row to return to Figure 6-20.

Select the *implement* "local variables" in the frame 'Control panel' in Figure 6-20 to reveal the local variables declarations in the function **main()**. The screen is shown in Figure 6-22. It is conjectured that the variables **sta_in_file**, **use_in_file**, **cal_in_file** and **prh_in_file** are used to hold the input filenames supplied in the command line.

The variable declarations in the function **main()** show that each of the local variables mentioned above is initialised to hold the value **NULL**.

The following statements show how information is extracted from the prompt supplied in the command line.

```
if (strcmp ("-sta", argv[count]) == 0) {
    if (count + 1 != argc) {
        count++;
        sta_in_file = argv[count];
    }
    else display_instructions (argv[0]);
}
else if (strcmp ("-use", argv[count]) == 0) {
    if (count + 1 != argc) {
        count++;
        use_in_file = argv[count];
    }
    else display_instructions (argv[0]);
}
else if (strcmp ("-cal", argv[count]) == 0) {
    if (count + 1 != argc) {
        count++;
        cal_in_file = argv[count];
    }
    else display_instructions (argv[0]);
    }
else if (strcmp ("-prh", argv[count]) == 0) {
    if (count + 1 != argc) {
        count++;
        prh_in_file = argv[count];
    }
    else display_instructions (argv[0]);
}
```

If no filename has been supplied, each of the local variables is then assigned a default value as shown in the following statements:

```
if (sta_in_file == NULL) sta_in_file = STA_INFILE;
if (use_in_file == NULL) use_in_file = USE_INFILE;
```

```
if (cal_in_file == NULL) cal_in_file = CAL_INFILE;
if (prh_in_file == NULL) prh_in_file = PRH_INFILE;
```

The default filenames for the four different files can be inferred from the **#define** statements above. The default filenames for the input files are: **xray.STA**, **xray.USE**, **xray.CAL**, **xray.PRH**. The results [C1], [C3], [C5] and [C7] are confirmed.

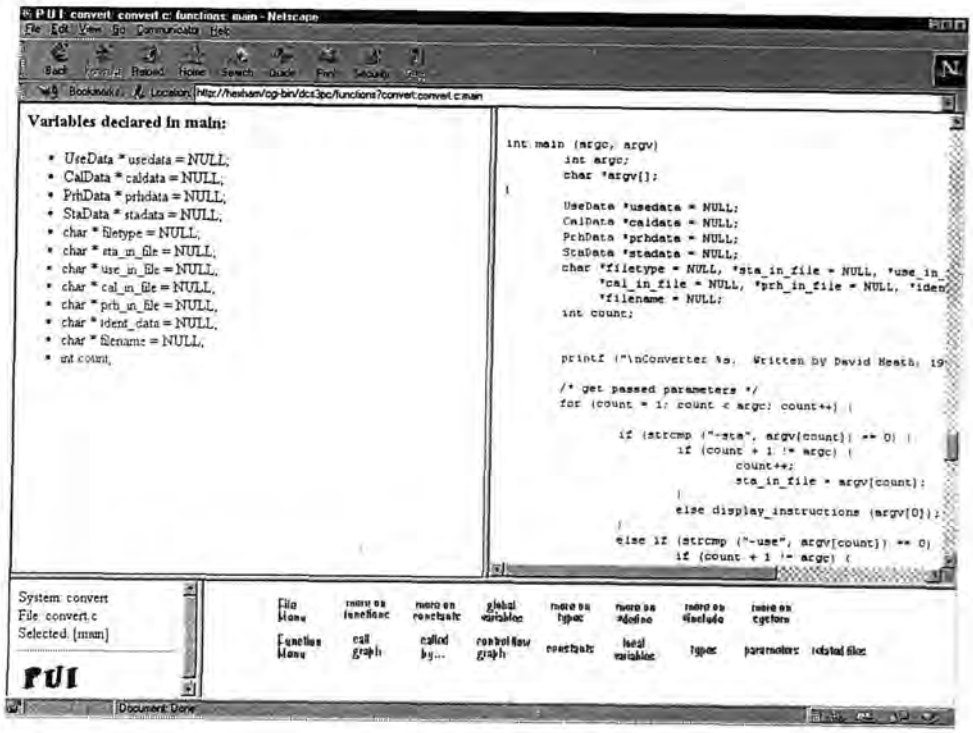

Figure 6-22 Screen showing the local variable declarations in the function **main()**

The next task of the comprehension is to investigate the format of these data files. Each of the variables **sta_in_file**, **use_in_file**, **cal_in_file** and **prh_in_file** are to be examined in turn and the variable **sta_in_file** is the first one to be examined.

Select "sta_in_file" in the frame 'Information display' in Figure 6-22 to retrieve more information on the variable. The result is shown in Figure 6-23.

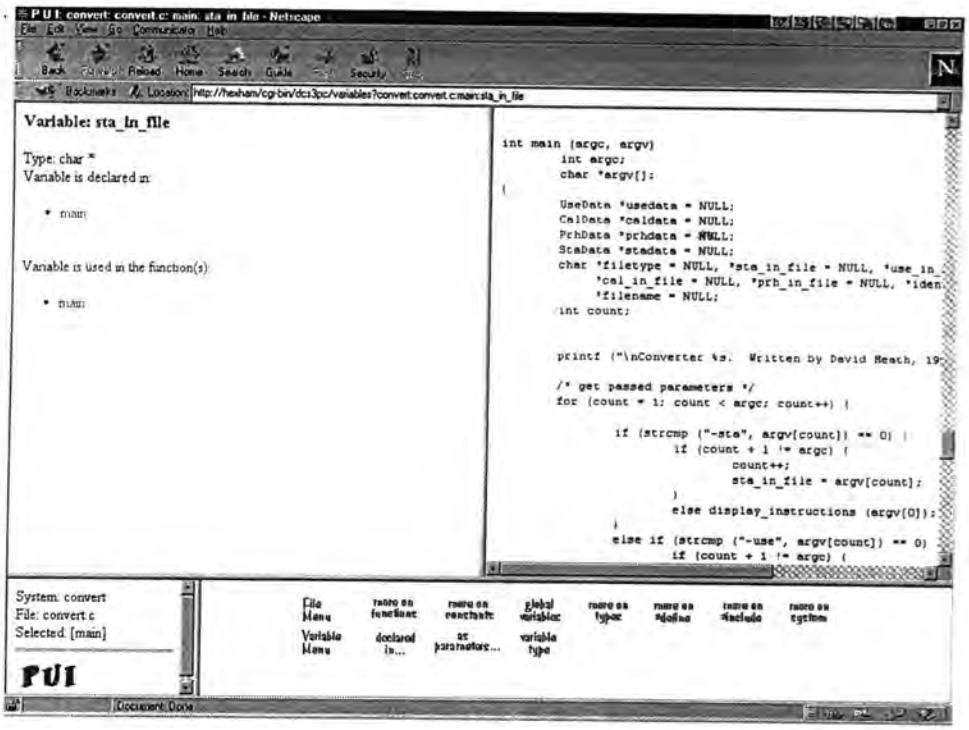

Figure 6-23 Screen showing information regarding the variable **sta_in_file**

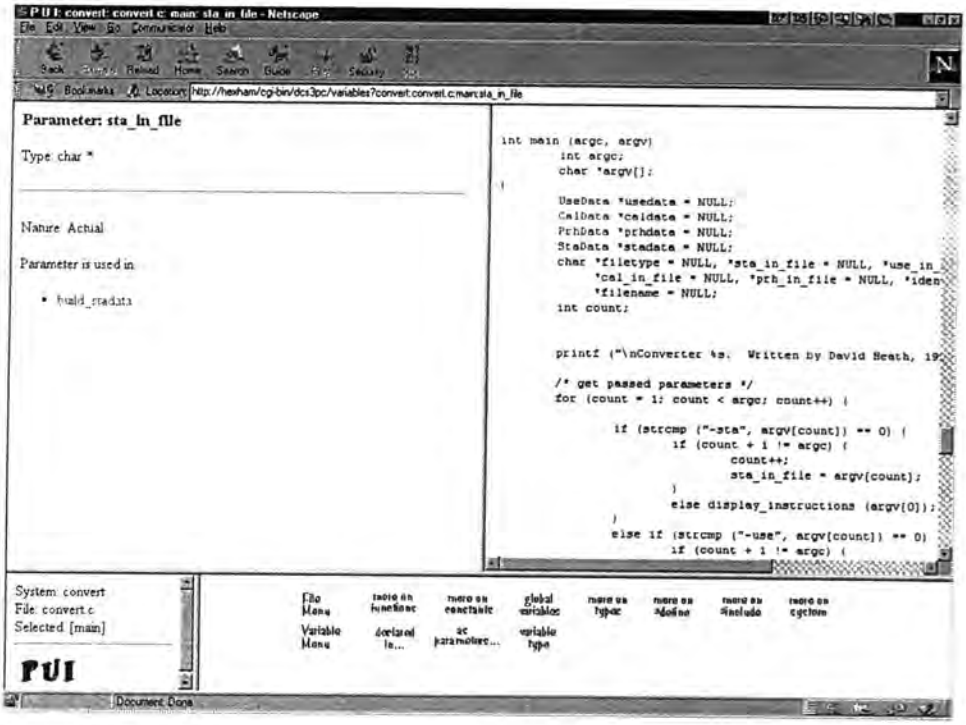

Figure 6-24 Screen showing that the variable **sta_in_file** is used as an argument

The PUI tool has determined that the variable **sta_in_file** is of the type **char ***, and it is declared and used only in the function **main()**.

Select the *implement* "as parameters..." in the frame 'Control panel' in Figure 6-23 to retrieve a list of functions which use this variable as the actual argument. The screen is shown in Figure 6-24.

It is revealed that **sta_in_file** is used as an actual argument in a function named **build_stadata**. Select "build_stadata" in the frame 'Information display' in Figure 6-24 to retrieve more information on the function. The screen is shown in Figure 6-25.

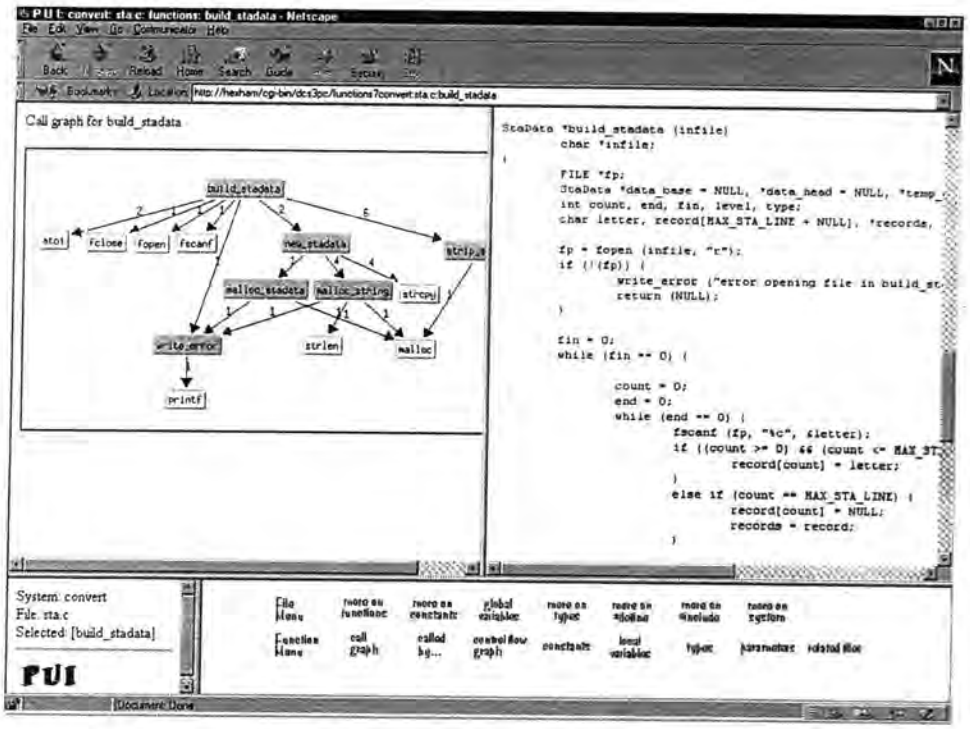

Figure 6-25 Screen showing information regarding the function **build_stadata**

From the call graph of the function **build_stadata**, it is determined that library functions which deal with file input and output are called within this function. These library functions include **fopen**, **fscanf** and **fclose**. An examination of the variable declarations in the function **build_stadata** has found the following statement:

```
FILE *fp;
```

This confirms that **build_stadata** indeed performs some operations on file input and output.

Select the *implement* "parameter" in the frame 'Control panel' in Figure 6-25 to retrieve more information on the use of arguments in function **build_stadata**. The screen is shown in Figure 6-26.

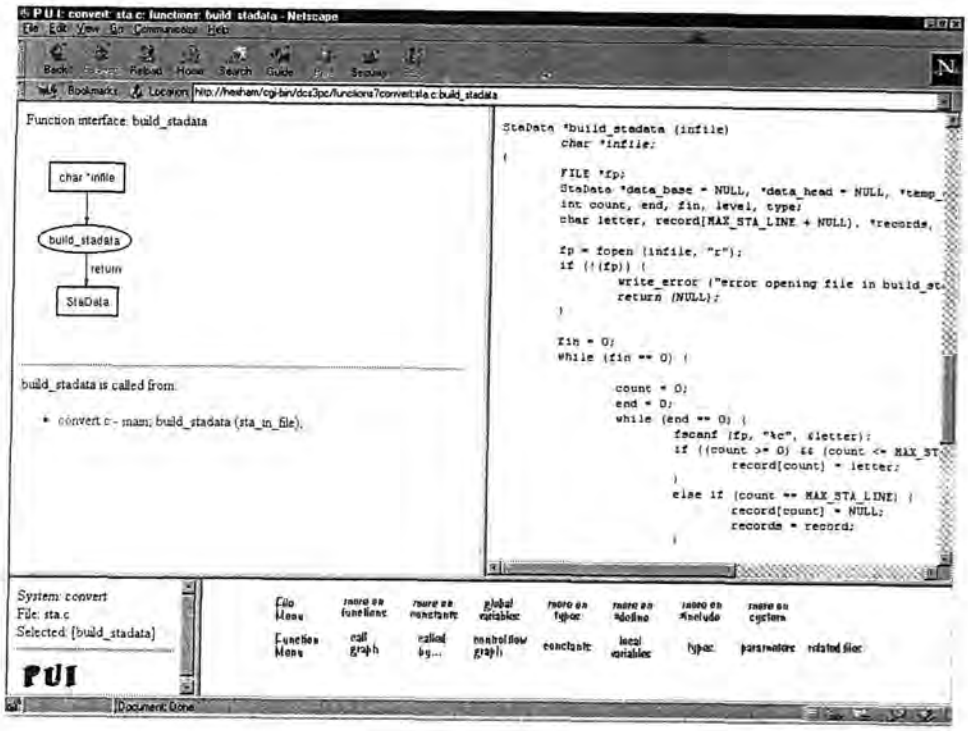

Figure 6-26 Screen showing information regarding the use of argument in the function **build_stadata**

The variable **sta_in_file** which holds the default filename **xray.STA** is used as the actual argument in place of the formal argument **infile** in the function **build_stadata**. The following statement shows that the file being opened is the default file, unless another filename is supplied in the command line.

```
fp = fopen (infile, "r");
```

After opening a text file, the function **build_stadata** is instructed to read in the text from the file on a character by character basis until it reaches the end of a line. The line of text is then stored in an array. The following statements record these instructions.

```
while (end == 0) {
   fscanf (fp, "%c", &letter);
   if ((count >= 0) && (count <= MAX_STA_LINE - 1)) {
      record[count] = letter;
   }
   else if (count == MAX_STA_LINE) {
      record[count] = NULL;
```

```
            records = record;
        }

        if (letter == '\n') {
            end = 1;
            if (count < MAX_STA_LINE) fin = 1;
        }
        else if (feof(fp)) {
            end = 1;
            fin = 1;
        }

        count++;
    }
```

After finishing reading the text, a series of function calls to the function **strip_string** are made:

```
sysname = strip_string (records, 0, 29);
filename = strip_string (records, 30, 59);
name = strip_string (records, 60, 89);
level = atoi (strip_string (records, 90, 91));
type = atoi (strip_string (records, 91, 92));
total = strip_string (records, 92, 100);
```

The above instructions are repeated until the function **build_stadata** reaches the end of the file. Select "strip_string" in the frame 'Listing' in Figure 6-26 to retrieve more information on the function. An examination of the function definition of **strip_string** reveals that this function dynamically allocates memory space for arrays. The following shows the signature of the function **strip_string**.

```
char *strip_string(string, start, end)
    char *string;
    int start;
    int end;
```

It is deduced that the numeric parameters used in the function calls to **strip_string** in the function **build_stadata** are the positions of characters within an array. The purpose of **strip_string** is to extract characters within these positions and to create and copy those characters into another array.

Return to Figure 6-26 by selecting the *Back* button in the second row of the menu system. Select the *implement* "local variables" in the frame 'Control panel' in Figure 6-26 to reveal more information on the local variables in the function **build_stadata**. The screen is shown in Figure 6-27. The types of the variables **sysname**, **filename**, **name**, **level**, **type** and **total** are noted.

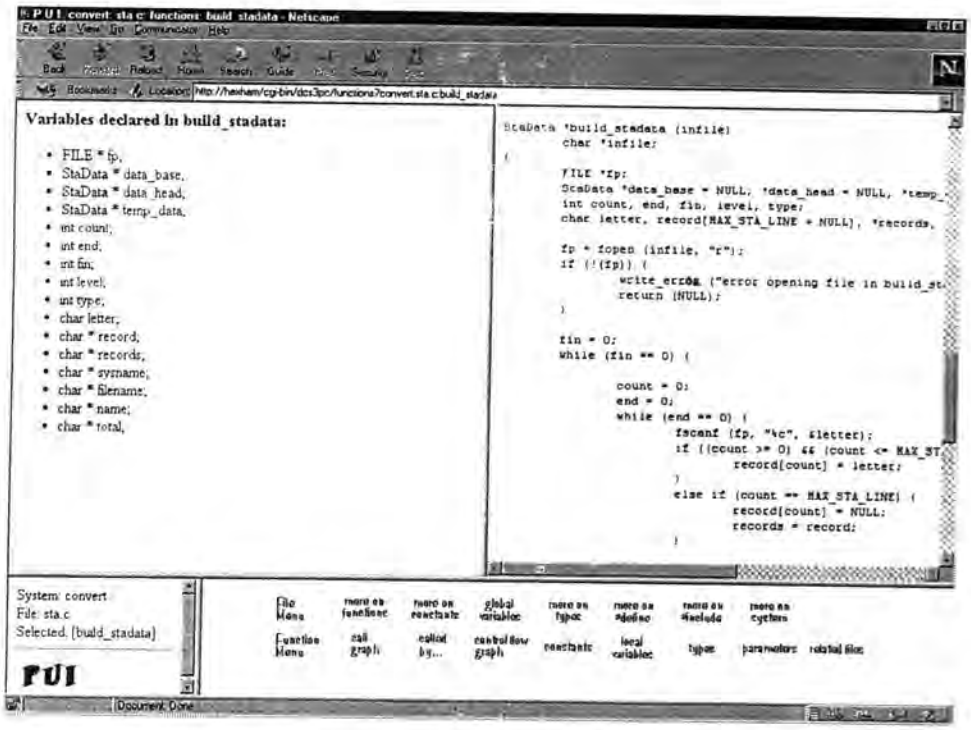

Figure 6-27 Screen showing the local variable declarations in the function **build_stadata**

The function **build_stadata** has also made direct function calls to the user defined functions **new_stadata** and **write_error**. An examination of the function definitions reveals that they do not interfere the input file in any way.

The comprehension process has concluded that the input file format for one of the data files is as follows:

```
a thirty-character string
a thirty-character string
a thirty-character string
a two-digit integer
a two-digit integer
a nine-character string
```

This file will have the default name **xray.STA**. Each line of the file has six fields, and the strings and integers must be of the exact length specified. Spaces must be used to fill the gaps whenever a string or an integer is shorter than that specified. The result [C2] is confirmed.

It has been deduced that the variables **sta_in_file**, **use_in_file**, **cal_in_file** and **prh_in_file** are used to hold the input filenames supplied in the command line. The variable **use_in_file** is to be examined next.

The comprehension is repeated as in the case of the variable **sta_in_file**.

The system has determined that the variable **use_in_file** is of the type **char *.** It is declared and used in the function **main()**, and is used as an actual argument in the function **build_usedata**.

From the call graph of **build_usedata**, it is determined that library functions which deal with file input and output are called within this function. These include the functions **fopen**, **fgets**, **fscanf** and **fclose**. An examination of the variable declarations in the function **build_usedata** has found the following statement:

```
FILE *fp;
```

This confirms that this function indeed performs some operations on file input and output.

The variable **use_in_file**, which holds the default filename **xray.USE**, is used as the actual argument in place of the formal parameter **infile** in the function **build_usedata**.

The structure of the function **build_usedata** is very similar to the function **build_stadata**. After opening a text file, the function **build_usedata** is instructed to read in the text on a line by line basis, then to store the characters in an array. After finishing reading in the text, a series of function calls to the function **strip_string** are made:

```
name = strip_string (records, 0, 29);
symbol = strip_string (records, 30, 59);
line = atoi (strip_string (records, 60, 65));
group = atoi (strip_string (records, 65, 69));
type = record[69];
code = record[70];
file = strip_string (records, 71, 100);
filename = strip_string (records, 101, 130);
```

The above instructions are repeated until the function **build_usedata** reaches the end of the file. The types of the variables **name**, **symbol**, **line**, **group**, **type**, **code**, **file** and **filename** are noted.

The function **build_usedata** has also made direct function calls to the user defined functions **new_usedata**, **removecr** and **write_error**. An examination of the function definitions reveals that they do not interfere the input file in any way.

The comprehension process has concluded that the format for one of the input files is as follows:

```
a thirty-character string
a thirty-character string
a six-digit integer
a five-digit integer
a single character
a single character
a thirty-character string
a thirty-character string
```

This file will have the default name **xray.USE**. Each line of the file has eight fields, and the strings and integers must be of the exact length specified. Spaces must be used to fill the gaps whenever a string or an integer is shorter than that specified. The result [C4] is confirmed.

Following similar steps outlined above, it is deduced that the format for the file with a default filename **xray.CAL** is as follows:

```
a thirty-character string
a thirty-character string
a single character
a thirty-character string
a thirty-character string
a six-digit integer
a three-digit integer
a thirty-character string
a five-digit integer
```

Each line of the file has nine fields, and that the strings and integers must be of the exact length specified. Spaces must be used to fill the gaps whenever a string or an integer is shorter than that specified. The result [C6] is confirmed.

The format for the last input file with a default filename **xray.PRH** is as follows:

```
a thirty-character string
a six-digit integer
a thirty-character string
a single character
a thirty-character string
```

Each line of the file has five fields, and the strings and integers must be of the exact length specified. Spaces must be used to fill the gaps whenever a string or an integer is shorter than that specified. The result [C8] is confirmed.

The investigation is complete. The formats for the four different files have been recovered and the names for each of the input file have been identified.

II Summary

The following is a summary of a list of tasks performed during the top-down comprehension.

Locate the source files for the system **convert**. Examine the architecture of the system **convert**.

Examine file inclusion to get a feel of the complexity of the system.

Locate the file which has the definition of the function **main()**. The file is **convert.c**.

Examine the **#define** statements. ..... [C1, C3, C5, C7]

Examine the use of the variables **sta_in_file**, **use_in_file**, **cal_in_file** and **prh_in_file**. Each variable is examined in turn.

The variable **sta_in_file** is used as an actual argument in the function **build_stadata**. The function declaration is found in the file **sta.h** and the function definition is found in the file **sta.c**.

A function call to function **strip_string** which is responsible for extracting characters from a source file is found in the function **build_stadata**. The positions and the length of the characters are noted. .....[C2]

The variable **use_in_file** is used as an actual argument in the function **build_usedata**. The function declaration is found in the file **use.h** and the function definition is found in the file **use.c**.

A function call to function **strip_string** which is responsible for extracting characters from a source file is found in the function **build_usedata**. The positions and the length of the characters are noted. .....[C4]

The variable **cal_in_file** is used as an actual argument in the function **build_caldata**. The function declaration is found in the file **cal.h** and the function definition is found in the file **cal.c**.

A function call to function **strip_string** which is responsible for extracting characters from a source file is found in the function **build_caldata**. The positions and the length of the characters are noted. .....[C6]

The variable **prh_in_file** is used as an actual argument in the function **build_prhdata**. The function declaration is found in the file **prh.h** and the function definition is found in the file **prh.c**.

A function call to function **strip_string** which is responsible for extracting characters from a source file is found in the function **build_prhdata**. The positions and the length of the characters are noted. .....[C8]

The investigation is complete.

6.4.5 Using a Bottom-up Approach

I Detailed Description

The following shows a demonstration on how PUI can help to carry out the comprehension by following a bottom-up approach.

On starting up the PUI tool, a user will be greeted by a screen as shown in Figure 6-1. Select the system **convert** by selecting its name.

The PUI tool will take the user to the screen similar to Figure 6-2. Select "User defined functions" to reveal the list of functions defined in each of the program files. The screen is shown in Figure 6-28.

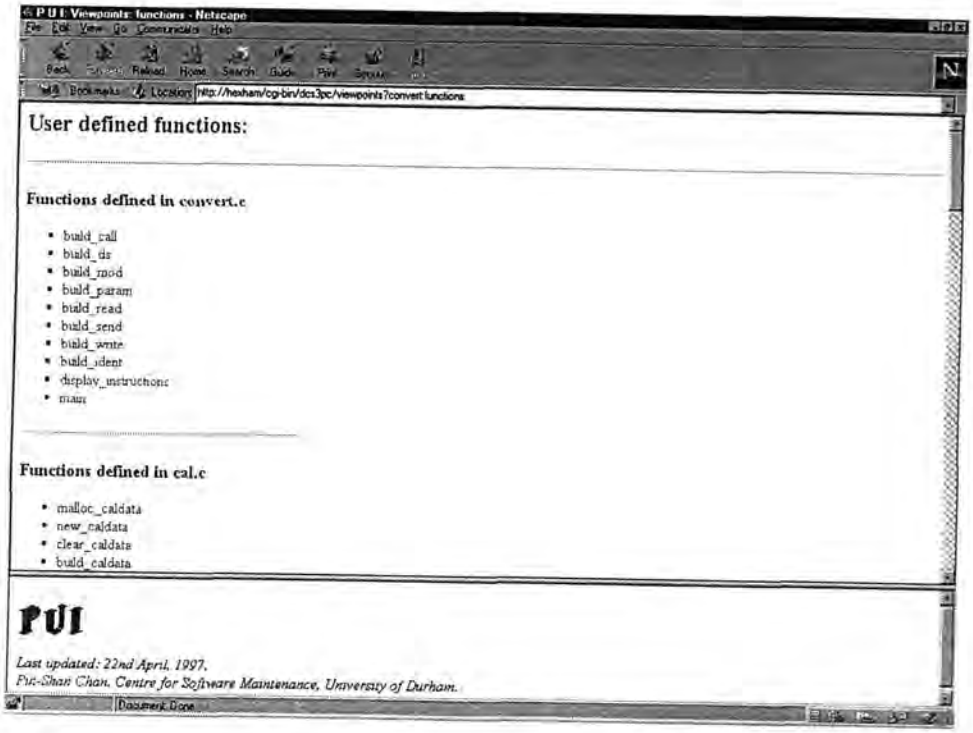

Figure 6-28 Screen showing the list of functions defined in each of the files in the system **convert**

This helps to give the user an initial impression of the system at the function level. It is revealed that the function **main()** is defined in the file **convert.c**. The names of the rest of other user defined functions are mostly self-explanatory.

From the result of a search, it is found that the library functions **fopen** and **fclose** are called by the functions **build_caldata**, **build_prhdata**, **build_stadata** and **build_usedata** respectively. The function **build_caldata** is the first to be examined.

Select "build_caldata" in the screen shown in Figure 6-28. An examination of the local variables declared in this function has found the following variable declaration:

```
FILE *fp;
```

This confirms that this function indeed performs some operations on file input and output.

After opening a text file, the function **build_caldata** is instructed to read the text in a character by character basis until the end of a line, and to store the characters in an array. The following statements record these instructions.

```
while (end == 0) {
      fscanf (fp, "%c", &letter);
   if ( (count >= 0) && (count <= MAX_CAL_LINE - 1) ) {
      record[count] = letter;
   }
   else if (count == MAX_CAL_LINE) {
      record[count] = NULL;
      records = record;
   }
   else if (count > MAX_CAL_LINE) {
      while (letter != '\n') fscanf (fp, "%c", &letter);
   }

   if (letter == '\n') {
      end = 1;
      if (count < MAX_CAL_LINE) fin = 1;
      else records[count] = NULL;
   }
   else if (feof(fp)) {
      end = 1;
      fin = 1;
   }

   count++;
}
```

After finishing reading the text, a series of function calls to the function **strip_string** are made:

```
name = strip_string (records, 0, 29);
section = strip_string (records, 30, 59);
type = records[60];
file = strip_string (records, 61, 90);
sect_name = strip_string (records, 91, 120);
line = atoi (strip_string (records, 121, 126));
argument = atoi (strip_string (records, 126, 128));
variable = strip_string (records, 128, 157);
group = atoi (strip_string (records, 158, 162));
```

The above instructions are repeated until the function **build_caldata** reaches the end of the file. An examination of the function definition of **strip_string** reveals that this function dynamically

allocates memory space for arrays with variable lengths. The following shows the signature of the function **strip_string**.

```
char *strip_string(string, start, end)
   char *string;
   int start;
   int end;
```

It is deduced that the numeric arguments used in the function calls to **strip_string** in the function **build_caldata** are the positions of characters within an array. The purpose of **strip_string** is to extract characters within these positions and to create and copy those characters into another array.

Select the *implement* "local variables" in the frame 'Control panel' to reveal the local variables declared in the function **build_caldata**. The screen is shown in Figure 6-29.

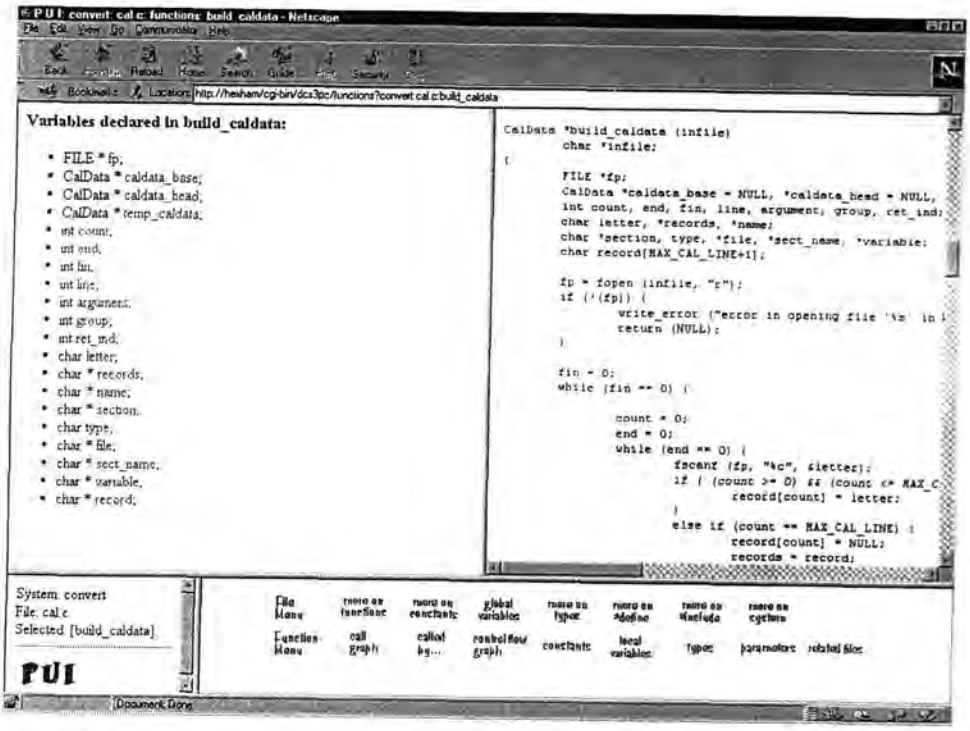

Figure 6-29 Screen showing the local variable declarations in the function **build_caldata**

The types of the variables **name**, **section**, **type**, **file**, **sect_name**, **line**, **argument**, **variable** and **group** are noted.

The function **build_caldata** has also direct made function calls to the user defined functions **new_caldata** and **write_error**. An examination of the function definitions reveals that they do not interfere the input file in any way.

The comprehension process has concluded that the input file format is as follows:

```
a thirty-character string
a thirty-character string
a single character
a thirty-character string
a thirty-character string
a six-digit integer
a three-digit integer
a thirty-character string
a five-digit integer
```

The result [C6] is confirmed. The next task is to find out the name of this input file.

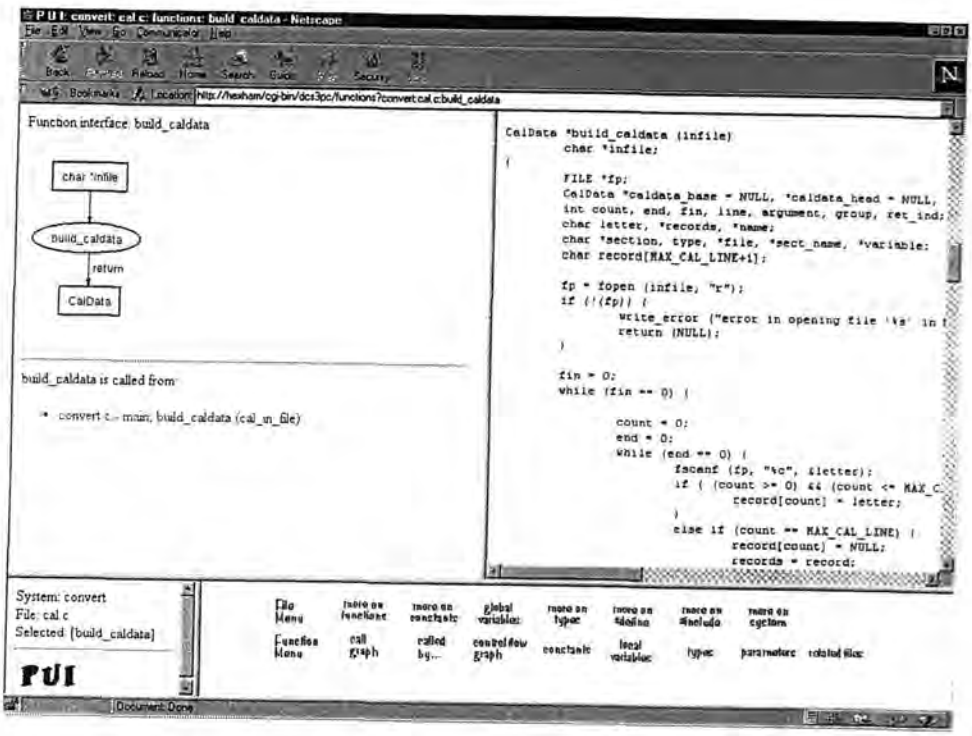

Figure 6-30 Screen showing information regarding the use of argument in the function **build_caldata**

The following statement from the function **build_caldata** suggests that the name of the file being opened is held in the variable **infile**.

```
fp = fopen (infile, "r");
```

The variable **infile** is a formal parameter belonging to the function **build_caldata**. Select the *implement* "parameter" in the frame 'Control panel' to reveal more information. This is shown in Figure 6-30.

It is revealed that the type of the argument **infile** is **char *.** The actual argument supplied to the function **build_caldata** is the variable **cal_in_file**, and the function **build_caldata** is called by the function **main()**.

From Figure 6-18, it is deduced that the system **convert** uses **-sta <file>**, **-use <file>**, **-cal <file>** and **-prh <file>** as arguments, and each **<file>** holds the name of an input file.

Select "main" in the frame 'Information display' in Figure 6-30 to retrieve more information on the function. From the function definition of **main()**, the following statements are observed:

```
if (sta_in_file == NULL) sta_in_file = STA_INFILE;
if (use_in_file == NULL) use_in_file = USE_INFILE;
if (cal_in_file == NULL) cal_in_file = CAL_INFILE;
if (prh_in_file == NULL) prh_in_file = PRH_INFILE;
```

The identifiers **STA_INFILE**, **USE_INFILE**, **CAL_INFILE** and **PRH_INFILE** are defined in the **#define** statements at the beginning of the file **convert.c**.

Select the *implement* "more on #define" to retrieve a list of **#define** statements in the file **convert.c**. The screen is shown in Figure 6-21. The identifiers used in the **#define** statements are mostly self-explanatory. There are predominately two groups of names which contain the phase INFILE and the phase OUTFILE. The ones containing the phase INFILE are:

```
#define STA_INFILE "xray.STA"
#define USE_INFILE "xray.USE"
#define CAL_INFILE "xray.CAL"
#define PRH_INFILE "xray.PRH"
```

The variable **cal_in_file**, which holds the default filename **xray.CAL** can be overwritten when a filename is supplied in the command line.

It is thus concluded that one of the input files will have the default name **xray.CAL** and the format outlined above. Each line of the file has nine fields, and the strings and integers must be of the exact length specified. Spaces must be used to fill the gaps whenever a string or an integer is shorter than that specified. The result [C5] is confirmed.

From the result of a search on the library functions, it is found that the library functions **fopen** and **fclose** are called in the user defined functions **build_caldata**, **build_prhdata**, **build_stadata** and **build_usedata** respectively. The function **build_prhdata** is the next to be examined.

The comprehension is repeated as in the case of the function **build_caldata**.

Select the *implement* "File menu" in Figure 6-21 to proceed to the screen as shown in Figure 6-28. Select "build_prhdata" in the frame 'Information display' to retrieve more information on the function. An examination of the local variables declared in this function has found the following statement:

```
FILE *fp;
```

This confirms that this function indeed performs some operations on file input and output.

After opening a text file, the function **build_prhdata** is instructed to read in a string of characters and then to store them in an array. The length of the string is dependent on the identifier **MAX_USE_LINE**, which holds the value **131**. This identifier is declared in the file **use.h**. The following statement record these instructions.

```
fgets (record, MAX_USE_LINE + 1, fp)
```

After finishing reading in the text, a series of function calls to the function **strip_string** are made:

```
caller = strip_string (ch, 0, 29);
line = atoi (strip_string (ch, 30, 35));
called = strip_string (ch, 35, 64);
type = ch[65];
file = strip_string (ch, 66, 95);
```

The types of the variables **caller**, **line**, **called**, **type** and **file** are noted.

The above instructions are repeated until the function **build_prhdata** reaches the end of the file. The function **build_prhdata** has also made direct function calls to the user defined functions **new_prhdata** and **write_error**. An examination of these function definitions reveals that they do not interfere the input file in any way.

The comprehension process has concluded that the input file format is as follows:

```
a thirty-character string
a six-digit integer
```

```
a thirty-character string
a single character
a thirty-character string
```

The result [C8] is confirmed. The next task is to find out the name of this input file.

The structure of the function **build_prhdata** is very similar to the structure of **build_caldata**. An examination of the actual argument of the function **build_prhdata** has led to the variable **prh_in_file**. This variable is declared in the function **main()**.

Further analysis has shown that the variable **prh_in_file** holds a default value **xray.PRH**. This will be the default filename for the above file input format. The default filename can be overwritten if a filename is supplied in the command line. Each line of this file has five fields, and the strings and integers must be of the exact length specified. Spaces must be provided to fill the gaps whenever a string or an integer is shorter than that specified. The result [C7] is confirmed.

From the result of the search, it was found that the library functions **fopen** and **fclose** are called by the functions **build_caldata**, **build_prhdata**, **build_stadata**, **build_usedata**. The remaining functions to be examined are the functions **build_stadata** and **build_usedata**.

Following similar steps outlined above, it is deduced that the format for the file with a default filename **xray.STA** is as follows:

```
a thirty-character string
a thirty-character string
a thirty-character string
a two-digit integer
a two-digit integer
a thirty-character string
```

The default filename can be overwritten if a filename is supplied in the command line. Each line of this file has six fields, and the strings and integers must be of the exact length specified. Spaces must be provided to fill the gaps whenever a string or an integer is shorter than that specified. The results [C2] and [C1] are confirmed.

The format for the last input file with a default filename **xray.USE** is as follows:

```
a thirty-character string
a thirty-character string
a six-digit integer
a five-digit integer
a single character
a single character
a thirty-character string
```

a thirty-character string

The default filename can be overwritten if a filename is supplied in the command line. Each line of this file has eight fields, and the strings and integers must be of the exact length specified. Spaces must be provided to fill the gaps whenever a string or an integer is shorter than that specified. The results [C4] and [C3] are confirmed.

The comprehension is complete. The formats for the four different files have been recovered and the names for each of the input file have been identified.

II Summary

The following is a summary of a list of tasks performed during the bottom-up comprehension.

Locate the source files for the system **convert**. Examine the architecture of the system **convert**. Examine file inclusion to get a feel of the complexity of the system.

The functions **build_caldata**, **build_prhdata**, **build_stadata** and **build_usedata** have made function calls to library functions which deals with file input and output. Each of the functions is examined in turn.

The variable **cal_in_file** is used as an actual argument in the function **build_caldata**. The function declaration is found in the file **cal.h** and the function definition is found in the file **cal.c**.

A function call to function **strip_string** which is responsible for extracting characters from a source file is found in the function **build_caldata**. The positions and the length of the characters are noted. .....[C6]

The variable **cal_in_file** is dependent on **CAL_INFILE** which holds the value **xray.CAL**. This is the default filename for the format found in [C6]. .....[C5]

The variable **prh_in_file** is used as an actual argument in the function **build_prhdata**. The function declaration is found in the file **prh.h** and the function definition is found in the file **prh.c**.

A function call to function **strip_string** which is responsible for extracting characters from a source file is found in the function **build_prhdata**. The positions and the length of the characters are noted. .....[C8]

The variable **prh_in_file** is dependent on **PRH_INFILE** which holds the value **xray.PRH**. This is the default filename for the format found in [C8]. .....[C7]

The variable **sta_in_file** is used as an actual argument in the function **build_stadata**. The function declaration is found in the file **sta.h** and the function definition is found in the file **sta.c**.

A function call to function **strip_string** which is responsible for extracting characters from a source file is found in the function **build_stadata**. The positions and the length of the characters are noted. .....[C2]

The variable **sta_in_file** is dependent on **STA_INFILE** which holds the value **xray.STA**. This is the default filename for the format found in [C2]. .....[C1]

The variable **use_in_file** is used as an actual argument in the function **build_usedata**. The function declaration is found in the file **use.h** and the function definition is found in the file **use.c**.

A function call to function **strip_string** which is responsible for extracting characters from a source file is found in the function **build_usedata**. The positions and the length of the characters are noted. .....[C4]

The variable **use_in_file** is dependent on **USE_INFILE** which holds the value **xray.USE**. This is the default filename for the format found in [C4]. .....[C3]

The investigation is complete.

6.5 Discussion

This chapter describes the feasibility of the Integrated Approach by way of Case Studies. The process of comprehension has been conducted in both a top-down and a bottom-up fashion. The sections 6.3.3 and 6.4.3 contain two lists of tasks (goals) which have to be completed for each of the Case Studies. The order of these tasks are determined by sequentially browsing through the source code.

The use of both of the approaches has been proven successful in completing the modifications and investigation using the PUI tool.

In Case Study One, the sequence of the tasks completed under the top-down approach is: [S6, S9, S7, S8, S5, S3, S4, S2, S1, S10, S14, S13, S15, S16, S11, S12]. The sequence of tasks completed under the bottom-up approach is: [S6, S9, S11, S14, S7, S8, S5, S3, S4, S2, S1, S10, S13, S15, S16, S12].

In Cast Study Two, the sequence of the tasks completed under the top-down approach is: [C1, C3, C5, C7, C2, C4, C6, C8]. The sequence of tasks completed under the bottom-up approach is: [C6, C5, C8, C7, C2, C1, C4, C3].

The use of the prototype PUI has demonstrated that the Integrated Approach is flexible enough to support comprehension in either direction. More importantly, it has also demonstrated that the user can engage in the top-down and/or the bottom-up approaches at any stage during the comprehension process.

Chapter Seven

Evaluation

7.1 Introduction

This chapter presents an evaluation of the work undertaken. It is evaluated against the existing Program Comprehension theories and models, the prototype implementation and the results of the Case Studies. They are evaluated against a hierarchy of cognitive issues raised in Chapter Three. This is followed by a discussion on the requirements for automation.

7.2 Evaluation of the Integrated Approach

7.2.1 Theories of Program Comprehension Revisited

Each theory and model discussed in section 2.2 in Chapter Two favours a different approach to Program Comprehension. Pennington's theory is a bottom-up approach [Penn87] whereas Brooks's approach is performed in a top-down fashion [Broo83]. On the other hand, von Mayrhauser and Vans [Mayr94, Mayr95] and Letovsky [Leto86a] reason that maintainers use a mixture of both strategies depending on the cue of the additional information. Others such as Soloway and Ehrlich [Solo84, Solo86], Shneiderman and Mayer [Shne79] and Littman *et al.* [Litt86] advocate that Program Comprehension is based on a knowledge base and it is a process of assimilation. The message is clear: there is no consensus on how maintainers understand programs and each of those theories can only model certain aspects of the maintainers' behaviour during comprehension. The comprehension strategies used by a maintainer are also highly dependent on both the types and the objectives of the maintenance activity he is engaged in.

Software engineering activities are a cognitive skill and it is subjected to the limitation of human brains, i.e., we are only able to study/memorise a limited amount of information at a time. A common approach to tackle this problem would be to decompose a large program systematically into 'chunks'

or the respective smaller counterparts. Shneiderman [Shne80] conjectures that the information chunking process is used in understanding programs. His views are echoed by Burnstein and Roberson, who believe that comprehension of a program begins by first processing the individual statements and grouping them together into cohesive units called chunks which are components of a mental model [Burn97].

Littman *et al.* propose two strategies which can be used in Program Comprehension: the systematic strategy and the as-needed strategy [Litt86]. Both strategies arise primarily from different goals. The former is used when the intention is to understand global program behaviour; the latter is used to minimise the comprehension effort. For a large system which involves several hundred thousands of lines of code, the as-needed strategy seems to be the only solution. However, as Littman *et al.* have pointed out, employing the as-needed strategy *alone* may not be sufficient. It only allows a weak mental model to be constructed and it may lead the maintainer to an inaccurate comprehension because he may not be aware of the interconnections between particular software components. It is therefore necessary to augment the as-needed strategy so that additional information can be acquired.

Letovsky [Leto86a] argues that the comprehension process is a mixture of top-down and bottom-up strategies. Maintainers may switch and exploit the two strategies when certain information becomes available. Once the basic goals and functionalities have been recognised, the immediate representations of the source code are later used as a basis for a more detailed study. A mental model is then constructed to store these abstractions (goals and operations). Shneiderman [Shne80] suggests that programmers do not store 'raw information' (the syntactic knowledge) in a mental model but rather, they will abstract the information and store it into an internal semantic structure. This knowledge can later be translated into different representations.

Soloway and Ehrlich [Solo84] believe that program plans play an important role in the comprehension process. They detected that experts have strong expectations about what programs should look like and these expectations would lead them to look for certain operations and structures in the program. However, this process may be complicated by delocalised plans [Leto86b], where statements within a plan are scattered throughout the whole of a program. Letovsky and Soloway [Leto86b] believe that delocalised plans are more liable to misinterpretation and it is a fundamental problem because maintainers have a tendency to make plausible but incorrect assumptions based on local information.

Program plans are related to another branch of research: beacons. Brooks [Broo83] first introduced the notion of beacons. Beacons are important in Brooks's theory because they form the mappings between the hypotheses of the maintainers and the actual program text. They represent key features which a maintainer may look for when he encounters information like the name of a program or the

name of a variable. Wiedenbeck [Wied86, Wied91] has extended Brooks's theory on beacons. She refers to program plans as stereotyped program fragments. They represent features of a program which strongly points to a function's functionality and such are the features that maintainers are generally looking for.

Pennington [Penn87] argues that the comprehension process is predominantly performed in a bottom-up process. When programmers are asked to study a piece of source code for the first time, it is strongly suggested that the procedural (control flow) relation dominates the programmers' mental representation of the source code. The results suggest the importance of the text structure knowledge in the comprehension process.

von Mayrhauser and Vans [Mayr94, Mayr95] express a similar view to Letovsky [Leto86a]. They advocate that the comprehension process is performed using a mixture of top-down and bottom-up approaches. The four major components in this metamodel are: the top-down model, situation model, program model and a knowledge base. The authors argue that maintainers seldom perform comprehension in a single direction, i.e. in either a pure top-down or bottom-up fashion. Any one of the four submodels may become active at any time during the comprehension process. The choice of the use of these submodels is largely dependent on the cues available to and the preferences of the maintainers.

From the overview of these Program Comprehension theories, it is evident that there is no real consensus on how maintainers understand software systems. Each theory and models discussed above favours a different approach to Program Comprehension. These theories attempt to model certain aspects of the maintainers' behaviour during comprehension. Chan [Chan97] and von Mayrhauser and Vans [Mayr94, Mayr95] believe that the disparities in the comprehension strategies used are largely dependent on the personal and circumstantial factors. Factors such as the level of technical competence of the maintainers, the size and complexity of the piece of software, and the types and goals of the maintenance activities can influence the process of comprehension.

All of these strategies embody a common characteristic: they seek to model a continuous and non-linear process within a set of parameters whereby knowledge is assimilated incrementally. Some have expressed the concern that the sole use of any of the theories and models may be insufficient on a larger scale. They may have to be augmented with other techniques when required.

7.2.2 Integrated Approach Revisited

More often than not, maintainers employ various strategies and use cues from either the source code or the system documentation as guidance. It is argued that when maintainers are engaged in the maintenance tasks, they are likely to exploit the use of both the top-down and the bottom-up

approaches when new information is encountered [Chan97, Leto86a, Mayr94, Mayr95]. An approach which is flexible enough to support the use of different comprehension strategies, as well as having the capability to cope with the different behavioural patterns of the maintainers, will be more applicable.

The Integrated Approach involves explicitly exposing the interrelationships between the many Program Elements within the source code. It is extremely difficult to contemplate exactly what kind of information a maintainer may need during the maintenance tasks. Instead of anticipating, planning and providing the information that a maintainer may need, the attention is now focused on exposing the Program Relationships between pairs of Program Elements. This approach is realised by first identifying the interactions between the Program Elements and then setting up a framework to assist the analysis of those Elements and Relationships involved.

The Integrated Approach does not impose any restrictions on how the process of comprehension should be performed. On the contrary, a maintainer is free to explore the Program Elements and Relationships that he chooses and hence enables the utilisation of different comprehension strategies and models. As discussed in section 7.2.1, the use of a particular comprehension strategy *alone* may be insufficient. This approach allows the essence of the different strategies to be captured and performed in a single environment. Maintainers can exploit the use of various strategies throughout the comprehension process as they examine the Program Elements and relationships. Under the Integrated Approach, maintainers are encouraged to make use of the information available, rather than being put in a position to ponder on how to chase for the elusive piece of information. They can make use of the Program Elements and Relationships in order to expand or to refine their line of investigation as they see fit. Relevant information about a particular Program Element is attained by examining other related Elements and Relationships.

7.2.3 Cognitive Design Elements

The Integrated Approach to comprehension and results from the Case Studies are evaluated against a hierarchy of cognitive design elements proposed by Storey *et al.* [Stor97a]. The framework is discussed in Chapter Three. The following are the issues raised under the section Improve Program Comprehension. This includes the cognitive design elements from E1 to E7.

I Enhance Bottom-up Comprehension

E1 Indicate syntactic and semantic relations between software objects

Storey *et al.* suggest that the syntactic and semantic relationships are essential during a bottom-up comprehension. The syntactical relationships between the program units are governed by the grammar of a programming language. The analysis of the semantic relationships between the program units would require data-flow or functional knowledge of a program.

The Program Elements described in Chapter Four represent the basic units that are present in a program. Under the Integrated Approach, relationships such as control flow, function calls and data dependencies have been identified so that maintainers can have easy access to the semantic relationships. The table of Program Relationships is shown in Table 1 in section 4.3 in Chapter Four.

E2 Reduce the effect of delocalised plans

Program plans are program fragments which represent stereotypical action sequences in a program. The recognition of plans may be complicated by delocalised plans, where statements within a plan are scattered throughout the whole of a program. The technique of program slicing is often employed to retrieve the program plans [Weis82, Weis84, Weis86].

The Integrated Approach does not support program slicing directly, and hence no program slice will be produced. Relationships such as the declaration and the use of variables have been identified which in turns offers limited program slicing power. The analysis of variable dependencies can be achieved by examining these relationships. For example, Figure 6-12 shows the declaration of the variable **`lineptr`** and where it is used within a program and Figure 6-13 shows how the same variable is used as an argument.

E3 Provide abstraction mechanisms

The Program Relationships between pairs of Program Elements represent various levels of abstraction of the source code. For example, the relation *imports* between the Program Elements **File** and **File** is of a higher level of abstraction than the relation *follows* between the Program Elements **Statement** and **Statement**. Under the Integrated Approach, a lower level Program Relationship can be abstracted into a higher level one progressively by selecting the appropriate Program Relationships. For example, Figure 5-6 shows a screen offering a selection of three different levels of abstraction.

II Enhance Top-down Comprehension

E4 Support goal-directed, hypothesis-driven comprehension

Under the top-down approach, comprehension is conducted by systematically establishing a mapping between the source code and the corresponding application domain. A maintainer begins with an initial hypothesis about the functionality of a program which is generated from documentation or from sources such as filenames.

Under the Integrated Approach, maintainers can verify their hypothesis by investigating the interaction between the Program Elements and examining the Program Relationships using the context sensitive navigational aids. For example, the recognition of the default input filenames to the

system **convert** is driven by the hypothesis that the names of identifiers and variables declared in the programs reflect their purposes.

The Integrated Approach, however, does not support the documentation of these hypothesis and the linking of them to specific parts of the source code.

E5 Provide an adequate overview of the system architecture at various levels of abstraction

In order to understand a piece of source code, a maintainer needs to acquire different levels of information about the source code at various stages. The Program Relationships shown in Table 1, which can be found in section 4.3 in Chapter Four, encompass various levels of abstraction. These Program Relationships can be organised systematically in the order of abstraction levels. Maintainers are empowered with the capability to access information at different levels of abstraction during comprehension under the Integrated Approach.

III Integrate Bottom-up and Top-down Approaches

E6 Support the construction of multiple mental models

Both the textual and graphical representations play an equally important role during comprehension. Under the Integrated Approach, information regarding the Program Elements and Program Relationships are shown in both textual and graphical forms. The Program Relationships are illustrated using graphical representations wherever possible. All of them are augmented with textual information extracted by the static analysis tool. For example, Figure 6-1 shows a graphical representations of the relationship *imports* between the Program Elements **File** and **File** in the frame 'Information display'. This information is reinforced in the frame 'Listing'.

E7 Cross-reference mental model

The Integrated Approach comprises several components: the Program Elements, the Program Relationships, graphical and textual displays. These components are held together by the context sensitive navigational aids, which link the corresponding graphical and textual representations for each of the Program Elements and Program Relationships. A discussion on the navigational aids can be found in section 4.4.1 in Chapter Four.

7.3 Evaluation of the Implementation

The Integrated Approach is realised in a prototype named PUI. In essence, it is a framework where graphical and textual representations are brought together using the technique of cross-referencing, and driven by the Relationships between Program Elements. The main objective of the

implementation is to demonstrate that pure top-down and bottom-up comprehension, and combinations of both approaches can be supported and utilised in a single environment.

7.3.1 Using the Web as the Underlying Structure

Other components which are present in PUI include a static analyser, a database containing facts about a program, a textual display tool and a graphical display tool. These components are brought together under a uniform user interface using World Wide Web technologies. Most of the web browsers have the capability to display many different types of information including textual, graphical, audio and visual information. In this case, the web browsers provide an ideal vehicle for the realisation of the Integrated Approach. All of the information can now be captured in the same environment which means that the notations, layout and representations are consistent throughout.

The idea of utilising the technologies of the World Wide Web is supported by Tilley and Smith [Thür95, Till97]. They believe that the web is a convenient infrastructure for Re-engineering. They argue that it is logical to exploit a technology which is widely available, at low cost and can be employed with little effort.

Web browsers such as Netscape Communicator and Internet Explorer are widely used and they provide simple and easy to use graphical user interfaces. In addition, the browsers have the added advantage of having a cross-platform interface which means that PUI can be used in a number of platforms such as PC, Macintosh and UNIX workstations.

7.3.2 Cognitive Design Elements

The implementation and results from the Case Studies are evaluated against a hierarchy of cognitive design elements proposed by Storey *et al.* [Stor97a]. The framework is discussed in Chapter Three. The following is a list of issues raised under the section Reduce the maintainer's cognitive overheads. This includes the cognitive design elements from E8 to E15.

I Facilitate Navigation

E8 Provide directional navigation

In the prototype, textual and graphical representations are placed in a windowing interface equipped with vertical and horizontal scroll bars. These representations are transformed into hypertext documents which contain 'anchors' or hyperlinks. They are always shown as highlighted text or coloured graphic designs. These hyperlinks act as the glue which holds the many hypertext documents together. For example, Figure 6-19 shows an overview of the system **`convert`**. Both the coloured nodes in the graphical representation in the frame 'Information display' on the left, and the list of

filenames in the frame 'Listing' on the right contain the hyperlinks to their corresponding counterparts.

E9 Support arbitrary navigation

The Integrated Approach encourages the users to explore the programs by repeating the process of selecting and examining the Program Elements and Program Relationships. The context sensitive navigational aids provide the mechanism which helps the users to achieve this goal. It is flexible and it allows comprehension to be conducted in a way preferred by the maintainers. Users can switch instantly from one model of comprehension and engage in another by using the navigational aids. A discussion on the navigational aids can be found in section 4.4.1 in Chapter Four.

E10 Provide navigation between mental models

Under the Integrated Approach, the mental models of the maintainers are represented by a mixture of textual and graphical displays. The graphical representations are annotated so that nodes in a graph are linked to the corresponding piece of textual information such as the source code and output from a static analysis tool. The source code is also annotated so that multiple instances of Program Elements which are scattered throughout the software system can be located quickly and effectively.

II Provide Orientation Cues

E11 Indicate the maintainer's current focus

Disorientation is a common symptom as far as using the World Wide Web is concerned. The prototype has a special provision in the form of Status Report, which serves the purpose of informing the users of their current focus. It shows the name of the system that a user is analysing and the names of the Program Elements currently selected.

For example, the frame 'Status report' shown in Figure 6-2 reads:

```
System: sortline
File: sortline.c
Selected: [files]
```

It shows that the system selected is named sortline, and the Program Element selected is the file **sortline.c**. The frame 'Status report' changes according to the choice of the selected Program Elements. The frame 'Status report' shown in Figure 6-3 has changed to the following when another Program Element **Variable** is selected:

```
System: sortline
File: sortline.c
Selected: [variables]
```

E12 Display the path that led to the current focus

Special provision is also provided for displaying the path which leads to the current focus of the maintainers. The path is displayed as the title of a hypertext document. The order of the sequence of selections is recorded by the web browser which can be displayed at any time. In the hypertext browser Netscape, this is achieved by selecting *Go* in the first row of the browser's menu system.

E13 Indicate options for reaching new nodes

Programs are built from Program Elements which are held together via a network of Program Relationships. It is this connectivity which the context sensitive navigational aids are based upon. These aids have two purposes: to retrieve the relevant information relating to the selected Program Elements and Relationships, and to provide the options for reaching other types of information by presenting the user with a list of related Elements and Relationships. A discussion on the navigational aids can be found in section 4.4.1 in Chapter Four.

III Reduce Disorientation

E14 Reduce additional effort for user-interface adjustment

Special consideration has been made during the design of the interface of the prototype to ensure cognitive overheads are kept to a minimum. The notations, layout and formats of the graphical representations and the navigational aids are consistent throughout the prototype. The hyperlinks are always shown as highlighted text or coloured graphic designs.

E15 Provide effective presentation styles

The presentation and the relative positioning of the textual and graphical windows are consistent throughout the prototype. This reduces the possibility of unpleasant surprises when retrieving hypertext documents. Most of the hypertext documents within the prototype have a fixed format where applicable, i.e., they are all divided into four different frames: Information display, Listing, Status report and Control panel. A typical screen of the prototype is shown in Figure 5-7.

7.4 Requirements for Automation

The prototype described in Chapter Five is a realisation of the Integrated Approach outlined in Chapter Four. The main objective of the Integrated Approach is to facilitate the process of comprehension and it is based on a matrix of relations between pairs of Program Elements shown in Table 1. The prototype consists of five parts:

- CCG, a static analysis tool
- Graph Tool, a graphical display application

- Perl scripts
- CGI scripts
- a set of hypertext (HTML) documents

Essentially, the output from CCG is fed into the Perl scripts where information about the Program Elements and Program Relationships are extracted. Program Relationships which can be represented visually are then translated into a format which is recognised by Graph Tool. The rest of the textual information is fed into the CGI scripts. The CGI scripts represent the *implements* of the prototype which deliver context sensitive information depending on the selections of the Program Elements and Program Relations. The output, whether it is textual or graphical, is translated into HTML which can be viewed using a web browser.

The objective of the following discussion is to examine the state of the prototype and to investigate the effectiveness of the implementation in terms of the success of automation, the integration of tools support, and proposed solutions to the problem of graph layout.

7.4.1 Automation

The main objectives of the Perl scripts are:

- to extract the information relating to the relational aspects of the Program Elements
- to translate this information into a format recognised by Graph Tool
- to prepare the rest of the CCG fact base so that it is ready to be fed into the CGI scripts

The first objective is to extract information from the static analysis tool. The information is then held in a database which is created and maintained by the Perl scripts. The Perl scripts support the extraction of the following Program Elements and Program Relationships:

- **Constant** *has an* **Identifier**
- **Constant** *is used as* **Argument**
- **Constant** *has* **Primitive/Complex Type**
- **Constant** *is declared in* **Function**
- **Constant** *is used in* **Function**
- **Constant** *is declared in* **File**
- **Constant** *is used in* **File**
- **Variable** *has an* **Identifier**
- **Variable** *is used as* **Argument**
- **Variable** *has* **Primitive/Complex Type**
- **Variable** *is declared in* **Function**
- **Variable** *is used in* **Function**
- **Variable** *is declared in* **File**
- **Variable** *is used in* **File**
- **Argument** *has an* **Identifier**
- **Argument** *is defined as* **Variable**
- **Argument** *has a* **Primitive/Complex Type**

- **Argument** *is used in* **Function**
- **Primitive/Complex Type** *is associated with* **Identifier**
- **Primitive/Complex Type** *is associated with* **Constant**
- **Primitive/Complex Type** *is associated with* **Variable**
- **Primitive/Complex Type** *is associated with* **Argument**
- **Primitive/Complex Type** *is declared in* **Statement**
- **Primitive/Complex Type** *is declared in* **Function**
- **Primitive/Complex Type** *is declared in* **File**
- **Statement** *declares* **Constant**
- **Statement** *declares* **Variable**
- **Statement** *declares* **Function**
- **Statement** *declares* **Primitive/ Complex Type**
- **Statement** *defines* **Function**
- **Statement** *follows* **Statement**
- **Statement** *is followed by* **Statement**
- **Function** *has an* **Identifier**
- **Function** *uses* **Variable**
- **Function** *uses* **Argument**
- **Function** *returns* **Type**
- **Function** *contains* **Statement**
- **Function** *calls* **Function**
- **Function** *is called by* **Function**
- **Function** *is used in* **File**
- **File** *has an* **Identifier**
- **File** *contains* **Function**
- **File** *uses* **Constants**
- **File** *uses* **Variables**
- **File** *uses* **Argument**
- **File** *uses* **Primitive/Complex Type**
- **File** *contains* **Statement**
- **File** *imports* **File**

These Program Relationships are chosen because they represent a small cross-section of the level of abstraction generally found in a C program. They are used to demonstrate the principle of the Integrated Approach in the different scenarios in the Case Studies described in Chapter Six.

The second objective of the Perl scripts is to translate the relational information into a format suitable for a graphical display tool. Relationships such as file dependencies, function calls, control flow and function interface are illustrated graphically. All of the graphical representations are laid out automatically with the exception of the function interface, which is drawn semi-automatically. One of the file inputs for a graphical display tool is shown in Figure 5-3 in Chapter Five.

The third objective of the Perl scripts is to prepare the rest of the CCG fact base so that it is ready to be input into the CGI scripts. Information which is related to the Program Relationships listed above have been filtered out from the CCG fact base, and then redirected into various text files.

The main objectives of the CGI scripts are:

- to provide a mechanism to probe the relationships between the Program Elements

- to produce a set of hypertext documents using HTML

The first objective of the CGI scripts is to provide a mechanism to probe and to retrieve information relating to the Program Elements and relationships in a context sensitive manner. This is done in the form of the context sensitive navigational aids discussed in section 4.4.1 in Chapter Four.

The second objective of the CGI scripts is to produce a set of hypertext documents by reading information from a set of text files which have been previously processed by the Perl scripts. This process is still largely semi-automatic with a large proportion of the hypertext documents being created manually.

7.4.2 Tool Support

Two of the components of PUI are CCG, a static analysis tool, and Graph Tool, a graphical display tool. Both are complete and stand-alone applications which can be used in their own right. CCG has a command-line interface and is largely run in the background. Graph Tool, on the other hand, has a graphical user interface and it forms an integral part of the prototype. At present, most of the graphical representations shown in the prototype are screen shots taken from Graph Tool. It means that the graphical representations in the prototype are static in nature and direct manipulation to these representations are prohibited. Similar graphical representations from other graphical display applications have been tested and used in the prototype. It is found that the simplicity and flexibility of the input format of Graph Tool would give an advantage over the others. A logical extension of Graph Tool which can be fully integrated into a web browser is yet to be developed.

Another important feature in the prototype is the textual information. It is displayed using a text window with vertical and horizontal scroll bars as visual aids. These text windows, however, do not support any textual manipulation. Text is displayed 'as is' and cannot be altered unless it is done via the CGI scripts. The prototype itself does not support any other text processing tool.

7.4.3 Graph Layout

It is recognised that the problem of finding any drawing algorithm which satisfies the aesthetic features and semantic constraints of a graph is NP-hard [Supo83, DiBa94]. The objective of this research focuses on providing support which can help to alleviate the problem by implementing a number of techniques suggested in section 2.3.5 in Chapter Two. Support has been provided for graph simplification and graph slicing. Colour has also been used for highlighting nodes in the graphical representations.

7.5 Discussion

The Integrated Approach embraces the idea that the process of comprehension is opportunistic and it provides a means for the fusion of the various comprehension strategies. The way maintainers conduct this process is influenced by the objectives of the maintenance activities they are engaged in and governed by their personal preferences. The Integrated Approach acknowledges that any one of the strategies may become active at any time and hence the need for a more flexible approach towards comprehension. Under this approach, maintainers have the option of selecting and executing the various strategies as they see fit. Pure top-down and pure bottom-up comprehension can also be achieved as demonstrated in the different scenarios in the case studies in Chapter Six.

The concept of information management is not new. It is about setting a proper framework to organise and retrieve relevant information. The PUI tool allows the maintainers to find out the information they require speedily, therefore reducing the time spent in studying the source code. Most of the output from existing software analysis tools is quite simple. In some cases, a large amount of information has either been filtered out, or simply lost due to successive transformations. The PUI tool enables maintainers to acquire better overviews of the programs since information is introduced gradually. The amount of information available to the maintainers will be limited to manageable chunks at any stage so they can easily integrate the information together without feeling confused.

Chapter Eight

Conclusion

8.1 Introduction

This chapter presents a summary of this research and evaluates the success of the research against the criteria defined in section 1.5 in Chapter One. An indication on the directions for further work on this research is also presented.

8.2 Summary of Research

Program Comprehension plays a critical part in all aspects in Software Engineering, especially in software maintenance. Activities such as Reverse Engineering and Reuse require the same amount of skill and attention as Testing and Software enhancement. A good understanding of the source code is required before the commencement of any of these activities. For a maintainer, the primary desire is the ability to decipher the source code accurately, quickly and efficiently. Studies have shown that maintainers spend a considerable amount of time studying programs, especially when engaged in maintenance activities. This figure can be as high as three-and-a-half times as long as they studied the documentation [Litt86].

Maintainers are often under pressure to accomplish the maintenance activities within a fixed time frame and the sheer complexity of the programs makes the tasks seem formidable. In the absence of a complete and consistent documentation, the source code may be the only information available to the maintainers. As a result, there is a strong desire for strategies and techniques which can be utilised to facilitate the comprehension process. The problem is how the maintainers find a systematic way to uncover this information.

There are a number of theories and models of Program Comprehension advocated by psychologists who are interested in studying the behaviour of programmers. Most of the work has been carried out

by observational studies, where typically, programmers are given a task to complete within a time limit. They were tested against their understanding, while the others were encouraged to think out loud so that their thoughts could be recorded. Some of the results show that the approach to Program Comprehension is performed in a top-down fashion whereas others suggest a bottom-up approach. However, the authors Chan [Chan97], Letovsky [Leto86a] and von Mayrhauser [Mayr94, Mayr95] suggest that an opportunistic approach which combines both the top-down and the bottom-up approaches would be a more robust model.

This research proposes an alternative approach to Program Comprehension. It acknowledges that the process of comprehension is opportunistic, and that the current comprehension theories are inadequate in addressing this. There is a need for a more flexible approach towards comprehension, and the Integrated Approach proposed provides a way for the utilisation of the various comprehension theories under a single environment. It recognises that any one of the comprehension theories may become active during comprehension and maintainers have the option of selecting and executing the comprehension strategies as they prefer.

The Integrated Approach to Program Comprehension aims to provide a solution to the problem of information overloading. Information is systematically categorised into different levels of abstraction under the Integrated Approach. Relevant information about a particular Program Element can be uncovered by analysing the Program Relationships and other related Elements. This approach does not impose any restrictions on how the comprehension should be performed, instead it enables the utilisation of different comprehension strategies and models. It is flexible and it allows comprehension to be conducted according to preferences of the maintainers. It is argued that the use of any one of the theories and models discussed in Chapter Two alone may be insufficient. This approach allows the essence of the different theories captured and performed in a single environment, and thus facilitating the comprehension process in a more effective manner.

Static analysis tools are useful in extracting information from programs. Maintainers are more likely to be overloaded with information extracted from these analysis tools as programs grow in size. It is widely acknowledged that graphical representations can help maintainers to gain a much better insight into the program structures. These graphical representations are frequently used as aids to comprehend programs. Most of the software maintenance tools discussed in Chapter Two offer some degree of visualisation which is based on the simple relationships of function calls and control flow. However, these graphical representations may not be very helpful due to their scale and complexity. The attention of the users are often drawn back to the source code as there is inadequate support for extracting information from the complex graphical representations.

This research addresses more relationships than just those of function calls and control flow through carrying out a systematic analysis of Program Elements and their Relationships. Study has shown that maintainers often want more information than is currently available on the display but they are not sure what exactly would be most helpful. The ability to provide alternative prospective on a same element, whether its a file, a function or a variable, is important because it can provide information with different granularity.

This research describes how the various strategies can be realised by a simple browsing tool, PUI (*P*rogram *U*nderstanding *I*mplement), which allows maintainers to understand the Relationships between Program Elements. The prototype is based on a matrix of Program Relationships designed to reflect the multi-dimensional nature of programs. This work is centred on the C programming language. The programs may be either ANSI [ANSI84] or Kernighan and Ritchie [Kern78, Kern88] C.

8.3 Evaluation of Research

The research is evaluated against a list of criteria defined in section 1.5 in Chapter One.

8.3.1 Criteria for Success

A In order to facilitate the process of Program Comprehension, a maintainer needs to have access to different kinds of information concerning a piece of source code. This can be in textual and/or graphical forms. Hence:

- maintainers should have easy and quick access to information at different levels of abstraction during various stages of comprehension
- support should be provided for maintainers with various degrees of experience and abilities
- support should be provided for the different types of maintenance activities that they may engage in

B There are a number of theories and models of Program Comprehension. Some researchers argue that it is done in a top-down fashion, whereas others advocate that it should be conducted in a bottom-up manner. There is no real consensus on how maintainers should perform comprehension. Moreover, most maintainers may prefer to employ the use of a mixture of strategies when the situation arises. Hence:

- any alternative approach to Program Comprehension proposed should address the need for a more flexible approach

C The feasibility of the Integrated Approach proposed needs to be examined. Hence:

- it needs to be demonstrated that it is feasible to realise the Integrated Approach in a physical form which can be executed with minimal difficulty

D The size of a software system should not be a hindrance to the process of Program Comprehension. Much research effort has been devoted to the development of techniques which support understanding-in-the-small. Hence:

- the Integrated Approach should be equipped with the capability to support understanding-in-the-large

In the context of this thesis, the term understanding-in-the-small is used to refer to the set of activities that are associated with the understanding of small programs which are relatively simple. The term understanding-in-the-large refers to the understanding of larger programs which contain more complex program relations.

E The usability and practicality of the Integrated Approach and of the implementation needs to be examined. Hence:

- both the Integrated Approach and the implementation should be measured against a set of criteria, which should lead to an objective evaluation

8.3.2 Evaluation

A The Integrated Approach to Program Comprehension is based on a matrix of Program Relationships between Program Elements shown in Table 1. These Program Relationships are derived for the C programming language constructs. Each of these Program Relationships represents a different level of abstraction of the programs ranging from high to low. They are organised systematically and maintainers are provided with support which gives them easy and quick access to the information that they require. This is achieved by way of the context sensitive navigational aids which are discussed in section 4.4.1 in Chapter Four.

Studies have shown that expert and novice programmers perceive programs differently, which lead to the conclusion that both parties use different strategies during Program Comprehension. Expert programmers tend to look for cues which are at a higher level of abstraction whereas the

novice programmers tend to adhere to the source code and extract information from that representation. Under the Integrated Approach, the Program Relationships represent different levels of abstraction of the source code. support should be provided for maintainers with various degrees of experience and abilities.

It is extremely difficult to contemplate exactly what kind of information a maintainer may need during the maintenance tasks. The required information is largely dependent on the maintainer's experience, the Program Comprehension strategies used, as well as the types of the maintenance activities they are engaged in. Under the Integrated Approach, information related to the source code is systematically broken down into various Program Relationship which represent different levels of abstraction. Maintainers can examine information relevant to their tasks by selecting and analysing the appropriate Program Relationships.

B An alternative approach to Program Comprehension is proposed in section 4.4 in Chapter Four. The Integrated Approach acknowledges that the process of comprehension is opportunistic, and that the current comprehension theories are inadequate in addressing this. The Integrated Approach recognises that during comprehension, any one of the theories may become active and it provides a way for the utilisation of the various comprehension theories. Under this approach, maintainers are free to select and execute the various comprehension theories as they see fit.

C The Integrated Approach is realised in a simple browsing tool named PUI, together with the help of supporting tools such as CCG, a software analysis tool, Graph Tool, a graphical display application and Netscape, a hypertext browser. It has demonstrated that the idea of analysing Program Elements and Program Relations as an alternative approach to Program Comprehension is feasible. The algorithms used to process the output obtained from CCG are efficient. Little training is required in order to run the PUI tool.

D Two software systems have been used as Case Studies. The size of one of the systems named **`convert`** is much larger than the other one named **`sortline`**. The system **`convert`** contains twenty five program files with more than three thousand and five hundred lines of code. Although it is only a medium-sized software system, it is argued that the Integrated Approach can accommodate systems which are significantly larger.

The Integrated Approach organises and presents information in a systematic way. All the Program Elements within the PUI tool are cross-referenced and thus the process of comprehension is not bounded by the physical locations of the various Program Elements. With the help of context sensitive navigational aids, relevant information regarding a Program Element is only a mouse-click away. In addition, the size of the program files which the PUI tool can deal

with is dependent on the analysis tool, CCG. At present, CCG is able to model programs of any size [Kinl95].

E A framework of evaluation for the Integrated Approach, the implementation and the results of the Case Studies is presented in Chapter Three and reported in Chapter Seven. A detailed analysis on the usability and practicality of the prototype is also presented in section 7.4 in Chapter Seven.

8.4 Future Work

All the theories and models of Program Comprehension discussed in section 2.2 in Chapter Two share the same theme: they attempt to identify unique features from the comprehension process, and place them in a model which serves to define the process in some way. The theories and models are valuable as they have established a basic framework where research effort can be focused. They categorise the comprehension process into top-down and bottom-up approaches. Research is needed to investigate and establish a general process model for each of the two approaches so that they can be compared and illustrated how the Integrated Approach fit in.

The Program Elements and Relationships are the key to the Integrated Approach. The Elements and Relationships are based on the C programming constructs which means they are strictly on a lexical and syntactic level. Semantic relationships can be introduced in order to enrich the Integrated Approach as both the semantic and syntactic relationships play an equally important role in the process of comprehension [Shne79].

The Integrated Approach is orientated towards the C programming language. Work can be done to extend this approach to other higher level programming languages such as Pascal, C++ and Java.

The PUI tool is a simple browsing tool which takes advantage of the web document design technologies. One of the shortcomings of PUI is that direct modifications cannot be made in real time. The maintainers may encounter situations where they would like to record their understanding during the comprehension process or to modify the source code when errors are found. A text editor and a compiler may incorporate into PUI so that the maintainers are equipped with the ability to edit text files and recompile the source code when required. Where appropriate, an area can be set aside for the maintainer to record information about a Program Element or a Program Relation. This information can be stored and then retrieved accordingly when the program component is encountered.

The *implements* (written in CGI scripts) are in the form of context sensitive navigational aids. They are essential in the process of recovering information about programs. Nevertheless, the attributes scope and storage classes, which are affiliated both to the Program Elements and the relations

discussed in section 4.3.3 can provide the extra information that the maintainers may need. Carefully selected attributes can be incorporated into the Integrated Approach. Simple measurements of the source code, Program Elements and Program Relationships, which are usually in the form of software metrics, can also be included to provide a base for comparison between pieces of source code.

The present graphical representation used in PUI is limited to that of two-dimensions. Three-dimensional visualisation techniques can be used to enhance the power of visualisation [Greg94, Riba94, Walk93, Youn96, Youn97]. This may include the use of animation and Virtual Reality.

Appendix A

The following is a listing of the system **sortline** used in Case Study One before the modifications. It consists of three files.

File: sortline.c

```
/* modified version of lines.c (K & R pgs 108-110) */


#include <stdio.h>
#include <string.h>
#include "qsort.h"

#define MAXLINES 10        /* max #lines tc be sorted */
#define MAXLEN   30        /* length of input line */
#define ALLOCSIZE 100      /* available space */

static char allocbuf[ALLOCSIZE];
static char *allocp = allocbuf;

char *lineptr[MAXLINES];


char *alloc(n)
int n;
{
   if (allocbuf + ALLOCSIZE - allocp >= n)
      {
         allocp += n;
         return allocp - n;
      }
   else
      return 0;
}

int getline (s, lim)
char s[];
int lim;
{
   int c,i;
   i = 0;

   while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
      s[i++] = c;

   if (c == '\n')
      s[i++] = c;

   s[i] = '\0';
   return i;
}

int readlines(lineptr, maxlines)
char *lineptr[];
int maxlines;
{
   int len, nlines;
   char *p, line[MAXLEN];
```

```
    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
    {
        if (nlines >= maxlines)
            return -1;

        if ((p =alloc(len)) == NULL)
            return -1;

        line[len-1] = '\0';
        strcpy(p,line);
        lineptr[nlines++] = p;
    }
    return nlines;
}

writelines(lineptr, nlines)
char *lineptr[];
int nlines;
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}


main()
{
int nlines;

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0)
    {
        qsort(lineptr,0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
    }
    else
    {
        printf("error : input too big to sort\n");
        return 1;
    }
}
```

File: qsort.c

```
/* filename: qsort.c */

swap(v, i, j)
char *v[];
int i,j;
{
    char *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

qsort(v, left, right)
char *v[];
int left, right;
{
    int i, last;

    if(left >= right)
        return;

    swap(v, left, (left+right)/2);
    last = left;

    for ( i=left+1; i <= right; i++)
        if (strcmp(v[i], v[left]) < 0)
            swap ( v, ++last, i);

    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

File: qsort.h

```
/* filename: qsort.h */

#ifndef qsort_header
#define qsort_header

void swap(char *v[], int i, int j);

void qsort(char *v[], int left, int right);

#endif
```

Appendix B

The following is a listing of the system **sortline** used in Case Study One after the modifications. It consists of three files.

File: sortline.c

```
/* modified version of lines.c (K & R pgs 108-110) */
/* modified to accept integer as input */

#include <stdio.h>
#include <string.h>

#define MAXLINES 10 /* max #lines to be sorted */

int lineptr[MAXLINES];


int getline (s)
int *s;
{
    int c;

    c = scanf("%d", s);
    return c;
}

int readlines(lineptr, maxlines)
int lineptr[];
int maxlines;
{
    int nlines, line;

    nlines = 0;
    while (getline(&line) > 0)
    {
        if (nlines >= maxlines)
            return -1;

        lineptr[nlines++] = line;
    }
    return nlines;
}

writelines(lineptr, nlines)
int lineptr[];
int nlines;
{
    while (nlines-- > 0)
        printf("%d\n", *lineptr++);
}


main()
{
    int nlines;

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0)
    {
        qsort(lineptr,0, nlines-1);
```

```
        writelines(lineptr, nlines);
        return 0;
    }
    else
    {
        printf("error : input too big to sort\n");
        return 1;
    }
}
```

File: qsort.c

```
/* filename: qsort.c */

swap(v, i, j)
int v[];
int i,j;
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}


qsort(v, left, right)
int v[];
int left, right;
{
 int i, last;

 if(left >= right)
   return;

 swap(v, left, (left+right)/2);
 last = left;

 for ( i=left+1; i <= right; i++)
     if (v[i] < v[left])
        swap ( v, ++last, i);

 swap(v, left, last);
 qsort(v, left, last-1);
 qsort(v, last+1, right);
}
```

File: qsort.h

```
/* filename: qsort.h */


#ifndef qsort_header
#define qsort_header

void swap(int v[], int i, int j);

void qsort(int v[], int left, int right);


#endif
```

References

[Alkh92] **Alkhatib, G.** *The Maintenance Problem of Application Software: An Empirical Analysis.* Journal of Software Maintenance: Research and Practice. June 1992. Vol. 4, No. 2, pages 83-104.

[ANSI83] **ANSI/IEEE.** *Software Engineering Standards.* Wiley-Interscience. 1983.

[ANSI84] **ANSI/IEEE.** *Software Engineering Standards.* Wiley-Interscience. 1984.

[Arga90] **Argawal, H., and Horgan, J.R.** *Dynamic Program Slicing.* Proceedings ACM SIGNPLAN '90 Conference on Programming Language Design and Implementation. 1990. ACM Press. Pages 246-256.

[Baec81] **Baecker, R.M.** *Sorting Out Sorting.* Narrated colour videotape, 30 minutes, presented at ACM SIGGRAPH '81 and excerpted in ACM SIGGRAPH Video Review. No. 7, 1983.

[Baec90] **Baecker, R.M. and Marcus, A.** *Human Factors and Typography for More Readable Programs.* Addison-Wesley, Reading, Massachusetts. 1990.

[Basi82] **Basili, V.R., and Mills, H.D.** *Understanding and Documenting Programs.* IEEE Transactions on Software Engineering. March 1982. Vol. SE-8, No. 3, pages 270-283.

[Bati85] **Batini, C., Furlani, L., and Nardelli, E.** *What is a good diagram? A pragmatic approach.* Proceedings of the 4th International Conference on Entity Relationship Approach. Chicago, IL. 1985.

[Bigg93] **Biggerstaff, T.J., Mitbander, B.G., and Webster, D.** *The Concept Assignment Problem in Program Understanding.* Proceedings of The Working Conference on Reverse Engineering. May 21-23, 1993. Baltimore, Maryland. IEEE Computer Society Press. Pages 27-43.

[Bigg94] **Biggerstaff, T.J., Mitbander, B.G., and Webster, D.E.** *Program Understanding and the Concept Assignment Problem.* Communications of the ACM. May 1994. Vol. 37, No. 5, pages 72-83.

[Bodh95] **Bodhuin, T.** *An Interaction Paradigm for Impact Analysis.* PhD. Thesis. Department of Computer Science, University of Durham. 1995..

[Boeh81] **Boehm, B.W.** *Software Engineering Economics.* Prentice-Hall. 1981.

[Boeh86] **Boehm, B.W.** *A spiral model of software development and enhancement.* ACM SIGSOFT Software Engineering Notes. April 1986. Vol. 11, No. 4, pages 22-42.

[Boeh88] **Boehm, B.W.** *A Spiral model of software development and enhancement.* IEEE Computer. May 1988. Vol. 21, No. 5, pages 61-72.

[Booc91] **Booch, G.** *Object-oriented Design with Applications.* Benjamin/Cummings. 1991

[Broo75] **Brooks, F.P.** *The Mythical Man-month.* Addison-Wesley. 1975.

[Broo83] **Brooks, R.** *Towards a Theory of the Comprehension of Computer Programs.* International Journal of Man-Machine Studies. 1983. Vol. 18, No. 6, pages 543-554.

[Brow84] **Brown, M.H., and Sedgewick, R.** *A System for Algorithm Animation.* Proceedings of ACM SIGGRAPH '84. ACM Press. New York. Pages 177-186.

[Brow85] **Brown, M.H., and Sedgewick, R.** *Techniques for Algorithm Animation.* IEEE Software. 1985. Vol. 2, No. 1, pages 28-39.

[Burd96] **Burd, E.L., Chan, P.S., Duncan, I.M.M., Munro, M., and Young, P.** *Improving Visual Representation of Code.* Computer Science Technical Report 10/96. Department of Computer Science, University of Durham. 1996.

[Burn97] **Burnstein, I., and Robserson, K.** *Automated Chunking to Support Program Comprehension.* Proceedings of the IEEE 5th International Workshop on Program Comprehension. May 28-30, 1997. Dearborn, Michigan. IEEE Computer Society Press. Pages 40-49.

[Carp80] **Carpano, M-J.** *Automatic Display of Hierarchized Graphs for Computer-Aided Decision Analysis.* IEEE Transactions on Systems, Man, and Cybernetics. November 1980. Vol. SMC-10, No. 11, pages 705-715.

[Chan97] **Chan, P.S., and Munro, M.** *PUI: A Tool to Support Program Understanding.* Proceedings of the IEEE 5th International Workshop on Program Comprehension. May 28-30, 1997. Dearborn, Michigan. IEEE Computer Society Press. Pages 192-198.

[Chan91] **Chandhok, R., Garlan, D., Meter, G., Miller, P., and Pane, J.** *Pascal Genie.* Available from Chariot Software Group, San Diego, California. 1991.

[CSM] **Centre for Software Maintenance.** University of Durham.

[Dekl92] **Dekleva, S.M.** *Software Maintenance: 1990 Status.* Journal of Software Maintenance: Research and Practice. December 1992. Vol. 4, No. 4, pages 233-247.

[DiBa94] **Di Battista, G., Eades, P., Tammassia, R., and Tollis, I. G.** *Algorithms for Drawing Graphs: an Annotated Bibliography.* June 1994. This document can be obtained via anonymous ftp from wilma.cs.brown.edu.

[Fair85] **Fairley, R.** *Software Engineering Concepts.* McGraw-Hill, New York. 1985.

[Fitt79] **Fitter, M., and Green, T.R.G.** *When do diagrams make good computer languages?* International Journal of Man-Machine Studies. 1979. Vol. 11, pages 235-361.

[Flet88] **Fletton, N.T., and Munro, M.** *Redocumenting Software Systems using Hypertext Technology.* IEEE International Conference on Software Maintenance, Phoenix, Arizona. 1988. Pages 54 - 59.

[Fost87] **Foster, J.R., and Munro, M.** *A Documentation Method Based on Cross-Referencing.* IEEE International Conference on Software Maintenance, Austin, Texas. 1987. Pages 181-185.

[Gans88] **Gansner, E.R., North, S.C., and Vo K.P.** *DAG - A program that Draws Directed Graphs.* Software Practice and Experience. November 1988. Vol. 18, No. 11, pages 1047-1062.

[Gans93] **Gansner, E.R., Koutsofios, E., North, S.C., and Vo K.P.** *A Technique for Drawing Directed Graph.* IEEE Transactions on Software Engineering. March 1993. Vol. 19, No. 3, pages 214-230.

[Greg94] **Gregson, R.D.** *Virtual Reality and Program Comprehension: Application Using Spreadsheet Visualisation.* MSc. Thesis. Department of Computer Science, University of Durham. 1994.

[Harm97] **Harman, M., and Danicic, S.** *Amorphous Program Slicing.* Proceedings of the IEEE 5th International Workshop on Program Comprehension. May 28-30, 1997. Dearborn, Michigan. IEEE Computer Society Press. Pages 70-79.

[Jack85] **Jackson, M.A.** *Principles of Program Design.* Academic Press. 1985.

[Kern78] **Kernighan, B.W., and Ritchie, D.M.** *The C Programming Language.* McGraw-Hall. First Edition. 1978.

[Kern88] **Kernighan, B.W., and Ritchie, D.M.** *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey. Second Edition. 1988.

[Kinl95] **Kinloch, D.** *A combined Representation for the Maintenance of C Programs.* PhD. Thesis. Department of Computer Science, University of Durham. 1995.

[Kitc95] **Kitchenham, B., Pickard, L., and Pfleeger, S.L.** *Case Studies for Method and Tool Evaluation.* IEEE Software. July 1995. Vol. 12, No. 4, pages 52-62.

[Kore88] **Korel, B., and Laski, J.** *Dynamic Program Slicing.* Information Processing Letters. October 1988. Vol. 29, No. 3, pages 155-163.

[Kore97] **Korel, B., and Rilling, J.** *Dynamic Program Slicing in Understanding of Program Execution.* Proceedings of the IEEE 5th International Workshop on Program Comprehension. May 28-30, 1997. Dearborn, Michigan. IEEE Computer Society Press. Pages 80-89.

[Lait95] **Laitinen, K.** *Natural naming in software development and maintenance.* PhD. Thesis. University of Oulu. 1995.

[Ledg75] **Ledgard, H.F.** *Programming Proverbs.* Hayden, Rochell Park. New Jersey. 1975.

[Leto86a] **Letovsky, S.** *Cognitive Processes in Program Comprehension.* Empirical Studies of Programmers. Albex, Norwood NJ. 1986. Pages 58-79.

[Leto86b] **Letovsky, S., and Soloway, E.** *Delocalized Plans and Program Comprehension.* IEEE Software. May 1986. Vol. 19, No. 3, pages 41-48.

[Lien78] **Lientz, B., Swanson, E.B., and Tompkins, G.E.** *Characteristics of Application Software Maintenance.* Communications of the ACM. June 1978. Vol. 21, No. 6, pages 466-471.

[Lien80] **Lientz, B., and Swanson, E.B.** *Software Maintenance Management.* Addison-Wesley. 1980.

[Lien81] **Lientz, B. Swanson, B.E.** *Problems in application software maintenance.* Communications of the ACM. November 1981. Vol. 24, No. 11, pages 763-769.

[Lino93] **Linos, P.K., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P. and Tulula, P.** *Facilitating the Comprehension of C Programs: An Experimental Study.* Proceedings of the IEEE 3rd International Workshop on Program Comprehension. November 14-15, 1993. Washington, D.C. IEEE Computer Society Press. Pages 55-63.

[Lino94] **Linos, P.K., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P. and Tulula, P.** *Visualizing Program Dependencies: An Experimental Study.* Software Practice and Experience. April 1994. Vol. 24, No. 4, pages 387-403.

[Litt86] **Littman, D.C., Pinto, J., Letovsky, S., and Soloway, E.** *Mental Models and Software Maintenance.* Empirical Studies of Programmers. Albex, Norwood NJ. 1986. Pages 80-98.

[Luci96] **De Lucia, A., Fasolino, A. R., and Munro, M.** *Understanding Function Behaviors through Program Slicing.* Proceedings of the IEEE 4th International Workshop on Program Comprehension. March 29-31, 1996. Berlin, Germany. IEEE Computer Society Press. Pages 9-18.

[Mayr94] **von Mayrhauser, A., and Vans, A. M.** *Dynamic Code Cognitive Behaviors For Large Scale Code.* Proceedings of the IEEE 3rd International Workshop on Program Comprehension. Washington, D.C. November 14-15, 1994. IEEE Computer Society Press. Pages 74-81.

[Mayr95] **von Mayrhauser, A., and Vans, A. M.** *Program Comprehension During Software Maintenance and Evolution.* IEEE Computer. August 1995. Vol. 28, No. 8, pages 44 - 55.

[Mess91] **Messinger, E.B., Rowe, L.A., and Henry, R.R.** *A Divide-and-Conquer Algorithm for the Automatic Layout of Large Directed Graphs.* IEEE Transactions on Systems, Man and Cybernetics. January/February 1991. Vol. SMC-21, No. 1, pages 1-11.

[Miar83] **Miara, R.J., Musselman, J.A., Navano, J.A. and Shneiderman, B.** *Program Indentation and Comprehensibility.* Communications of the ACM. November 1983. Vol. 26, No. 11, pages 861-867.

[Myer90] **Myers, B.A.** *Taxonomies of Visual Programming and Program Visualisation.* Journal of Visual Languages and Computing. March 1990. Vol. 1, No. 1, pages 97-123.

[Oman90a] **Oman, P.** *Maintenance Tools.* IEEE Software. May 1990. Vol. 23, No. 3, pages 59-65.

[Oman90b] **Oman, P.W. and Cook, C.R.** *The Book Paradigm for Improved Maintenance.* IEEE Software. January, 1990. Vol. 7, No. 1, pages 39-45.

[Parn72] **Parnas, D.L.** *On the criteria to be used in decomposing systems into modules.* Communications of the ACM. December 1972. Vol. 15, No. 12, pages 1053-1058.

[Penn87] **Pennington, N.** *Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs.* Cognitive Psychology. July 1987. Vol. 19, No. 3, pages 295-341.

[Pfle94] **Pfleeger, S.L.** *Design and Analysis in Software Engineering. Part 1: The Language of Case Studies and Formal Experiments.* Software Engineering Notes. October 1994. Vol. 19, No. 4, pages 16-20.

[Pres92] **Pressman, R.S.** *Software Engineering: A Practitioner's Approach.* Third Edition. McGraw-Hill. 1992.

[Pric93] **Price, B.A., Baecher, R.M., and Small, I.S.** *A Principled Taxonomy of Software Visualizaiton.* Journal of Visual Languages and Computing. September 1993. Vol. 4, No. 3, pages 211-266.

[Rajl90] **Rajlich, V., Damaskinos, N., Linos, P., Khorshid, W.** *VIFOR: A Tool for Software Maintenance.* Software Practice and Experience. January 1990. Vol. 20, No. 1, pages 66-77.

[Rajl96] **Rajlich, V., and Adnapally, S.R.** *VIFOR 2: A Tool For Browsing and Documentation.* Proceedings of the IEEE International Conference on Software Maintenance, Monterey, California. November 4-8, 1996. IEEE Computer Society Press. Pages 296 - 300.

[Rein81] **Reingold, E.M., and Tilford, J.S.** *Tidier drawings of trees.* IEEE Transactions on Software Engineering. March 1981. Vol. SE-7, No. 3, pages 223-228.

[Riba94] **Ribarsky, W., Bolter, J., Op den Bosch, A., and can Teylingen, R.** *Visualization and Analysis Using Virtual Reality.* IEEE Computer Graphics and Applications. January 1994. Vol. 14, No. 1, pages 10-12.

[Robs91] **Robson, D.J., Bennett, K.H., Cornelius, B.J., and Munro, M.** *Approaches to Program Comprehension.* Journal of Systems and Software. February 1991. Vol. 14, No. 2, pages 79-84.

[Roma93] **Roman G-C., and Cox, K.C.** *A Taxonomy of Program Visualization Systems.* IEEE Computer. December 1993. Vol. 26, No. 12, pages 11-24.

[Royc70] **Royce, W.W.** *Managing the development of large software systems: concepts and techniques.* Proceedings IEEE WESCON. 1970. Pages 1-9.

[Ryde79] **Ryder, B.G.** *Constructing the Call Graph of a Program.* IEEE Transactions on Software Engineering. March 1979. Vol. SE-5, No. 3, pages 216-225.

[Schn87] **Schneidewind, N.F.** *The State of Software Maintenance.* IEEE Transactions on Software Engineering. March 1987. Vol. SE-13, No. 3, pages 303-310.

[Shne79] **Shneiderman, B., and Mayer, R.** *Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results.* International Journal of Computer and Information Sciences. 1979. Vol. 8, No. 3, pages 219-238.

[Shne80] **Shneiderman, B.** *Software Psychology.* Cambridge MA: Winthrop Publishers Inc. 1980.

[Shne86] **Shneiderman, B., Shafer, P., Simon, R., and Weldon, L.** *Display Strategies for Program Browsing: Concepts and Experiment.* IEEE Software. May 1986. Vol. 19, No. 3, pages 7-15.

[Shu88] **Shu, N.C.** *Visual Programming.* Van Nostrand Reinhold, New York. 1988.

[Solo84] **Soloway, E., and Ehrlich, K.** *Empirical Studies of Programming Knowledge.* IEEE Transactions on Software Engineering. September 1984. Vol. SE-10, No. 5, pages 595-609.

[Solo86] **Soloway, E.** *Learning to Program = Learning to Construct Mechanism and Explanations.* Communication of the ACM. September 1986. Vol. 29, No. 9, pages 850-858.

[Somm96] **Sommerville, I.** *Software Engineering.* Fifth Edition. Addison-Wesley. 1996.

[Stan84] **Standish, T.A.** *An Essay on Software Reuse.* IEEE Transactions on Software Engineering. September 1984. Vol. SE-10, No. 5, pages 494-497.

[Stas92] **Stasko, J.T., and Patterson, C.** *Understanding and characterizing software visualization systems.* Proceedings of the IEEE 1992 Workshop on Visual Languages. Seattle, Washington. 1992. IEEE Computer Society Press. Pages 3-10.

[Stor95] **Storey, M-A.D., and Müller, H.A.** *Manipulating and Documenting Software Structures Using SHriMP Views.* Proceedings of the IEEE 1995 International Conference on Software Maintenance. October 17-20, 1995. Opio (Nice), France. IEEE Computer Society Press. Pages 275-284.

[Stor97a] **Storey, M-A.D., Rracchia, F.D., and Müller, H.A.** *Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualisation.* Proceedings of the IEEE 5th International Workshop on Program Comprehension. May 28-30, 1997. Dearborn, Michigan. IEEE Computer Society Press. Pages 17-28.

[Stor97b] **Stork, D.G.** *The End of an Era, The Beginning of Another? HAL, Deep Blue and Kasparov.* 1997. This document can be obtained from this URL http://www.chess.ibm.com

[Sugi81] **Sugiyama, K., Tagawa, S., and Toda, M.** *Methods for Visual Understanding of Hierarchical System Structures.* IEEE Transactions on Systems, Man and Cybernetics. February 1981. Vol. SMC-11, No. 2, pages 109-125.

[Supo83] **Supowit, K.J., and Reingold, E.M.** *The complexity of drawing trees nicely.* Acta Informationca. 1983. Vol. 18, No. 4, pages 377-392.

[Tama88] **Tamassia, R., Di Battista, G., and Batini, C.** *Automatic graph drawing and readability of diagrams.* IEEE Transactions on Systems, Man and Cybernetics. February 1988. Vol. SMC-18, No. 2, pages 61-79.

[Teas94] **Teasley, B.E.** *The effects of naming style and expertise on program comprehension.* International Journal of Human-Computer Studies. 1994. Vol. 40, pages 757-770.

[Thür95] **Thüring, M., Hannemann, J., and Haake, J.M.** *Hypermedia and Cognition: Designing for Comprehension.* Communications of the ACM. August 1995. Vol. 38, No. 8, pages 57-66.

[Till97] **Tilley, S.R., and Smith, D.B.** *On Using the Web as Infrastructure for Reengineering.* Proceedings of the IEEE 5th International Workshop on Program Comprehension. May 28-30, 1997. Dearborn, Michigan. IEEE Computer Society Press. Pages 170-173.

[Venk91] **Venkatesh, G.A.** *The semantic approach to program slicing.* ACM SIGNPLAN Notices. June 1996. Vol. 26, No. 6, pages 107-119.

[Vlie93] **van Vliet, J.C.** *Software Engineering: principles and practice.* Wiley. 1993.

[Walk90] **Walker, J.Q.** *A Node-positioning Algorithm for General Trees.* Software Practice and Experience. July 1990. Vol. 20, No. 7, pages 685-705.

[Walk93] **Walker, G.R., Rea, P.A. Whalley, S., Hinds, M., and Kings, N.J.** *Visualisation of telecommunications network data.* British Telecom Technology Journal. October 1993. Vol. 11, No. 4, pages 54-63.

[Warf77] **Warfield, J.N.** *Crossing Theory and Hierarchy Mapping.* IEEE Transactions on Systems, Man, and Cybernetics. July 1977. Vol. SMC-7, No. 7, pages 505-523.

[Weis82] **Weiser, M.** *Programmers use Slices when Debugging.* Communications of the ACM. July 1982. Vol. 25, No. 7, pages 446-452.

[Weis84] **Weiser, M.** *Program Slicing.* IEEE Transactions on Software Engineering. July 1984. Vol. SE-10, No. 4, pages 352-357.

[Weis86] **Weiser, M., and Lyle, J.** *Experiments on Slicing-Based Debugging Aids.* Empirical Studies of Programmers. Albex, Norwood NJ. 1986. Pages 187-197.

[Weth79] **Wetherell, C., and Shannon, A.** *Tidy Drawings of Trees.* IEEE Transactions on Software Engineering. September 1979. Vol. SE-5, No. 5, pages514-520.

[Wied86] **Wiedenbeck, S.** *Processes in Computer Program Comprehension.* Empirical Studies of Programmers. Albex, Norwood NJ. 1986. Pages 48-57.

[Wied91] **Wiedenbeck, S.** *The Initial Stage of Program Comprehension.* International Journal of Man-Machine Studies. October 1991. Vol. 35, No. 4, pages 517-540.

[Wild91] **Wilde, N., and Huitt, R.** *A Reusable Toolset for Software Dependency Analysis.* Journal of Systems and Software. February 1991. Vol. 14, No. 2, pages 97-102.

[Wirt71] **Wirth, N.** *Program development by Stepwise Refinement.* Communications of the ACM. April 1971. Vol. 14, No. 4, pages 221-227.

[Youn96] **Young, P.** *Three Dimensional Information Visualisation.* Technical Report 12/96. Centre for Software Maintenance, University of Durham. March 1996.

[Youn97] **Young, P., and Munro, M.** *A New View of Call Graphs for Visualising Code Structure.* Technical Report 03/97. Centre for Software Maintenance, University of Durham. April 1997.

[Youn93] **Younger, E.J., and Bennett, K.H.** *Model-Based Tools to Record Program Understanding.* Proceedings of the IEEE 2nd International Workshop on Program Comprehension. July 8-9, 1993. Capri, Italy. IEEE Computer Society Press. Pages 87-95.