AN APPROACH TO SOFTWARE MAINTENANCE SUPPORT USING A SYNTACTIC SOURCE CODE ANALYSER DATA BASE

This thesis is presented in a partial fulfillment of the requirements for the degree of Master of Arts in Computer Science at Massey University.

PETER VIVIAN PARKIN

1987

ABSTRACT

In this thesis, the development of a software maintenance tool called a syntactic source code analyser (SSCA) is summarised. An SSCA supports other maintenance tools which interact with source code by creating a data base of source information which has links to a formatted version of program source code. The particular SSCA presented handles programs written in a version of COBOL.

Before developing a SSCA system, aspects of software maintenance need to be considered. Hence, the scope, definitions and problems of maintenance activities are briefly reviewed and maintenance support through environments, software metrics, and specific tools and techniques examined. A complete maintenance support environment for an application is found to overlap considerably with the application documentation system and shares some tools with development environments. Program source code is also identified as the fundamental documentation of an application and interaction with this source code is a requirement of many maintenance support tools.

ACKNOWLEDGEMENTS

I wish to record my gratitude to Professor Graham Tate for his guidance and supervision of this thesis.

Also, I would like to thank :

June Verner for her interest and support in this research;

My flatmates for encouraging my endevours;

and Massey University for providing the necessary facilities required for this thesis.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1. INTRODUCTION.

This thesis is concerned with software maintenance and tools and techniques for the support of software maintenance. This chapter briefly outlines the areas covered by later chapters and their sections.

Software maintenance is an expensive area of the system life cycle consuming an estimated 32% of system costs [MCK84]. Although maintenance is now beginning to be recognised as important, the amount of direct maintenance research which has been carried out is limited. Exactly what constitutes a maintenance task is still not completely defined especially the demarcation between maintenance and redevelopment. The emphasis of Chapter 2 is on defining and describing various aspects of maintenance (particularly aspects which are considered problematic), examining the relationship between development and maintenance and attempting to identify general principles for the modification of software.

In Section 2.1 a broad definition of maintenance is given and discussed. Reasons for maintenance are also examined in this section. The reasons suggest that maintenance is fundamental to most computer systems.

Although it could be supported by general tools, like fourth generation languages (4GLs), maintenance will certainly not disappear in the future [TAT85].

A task which has been identified as maintenance can be further classified using a number of categorisation schemes. These schemes, and some of the benefits and dangers in using them, are investigated in Section 2.2.

Source code produced in maintenance costs between 10 and 100 times more than in development [CON84]. High code production costs and maintenance backlogs of up to 2.5 years [TIN84] suggest that particular problems occur in maintenance which hamper increases in productivity. Several surveys of DP managers and/or programmers [CHA85] [LIE78] [REU81] have been carried out in an attempt to identify maintenance problems. Results from these surveys and suggestions from other researchers are discussed in Section 2.3.

Section 2.4 helps to further define maintenance in terms of its place within the system life cycle. In this section, the steps or actions associated with any software modification task (i.e. maintenance task) are also identified. The definition of aspects of maintenance is completed in Section 2.5 with an examination of direct influences on the process of software modification and a description of the phenomena known as "ripple effect" and "structural decay".

Section 2.6 and part of Section 2.5 are devoted to discussing principles for achieving successful maintenance. Difficulties with identifying such principles are illustrated through the design and implementation of modifications to a particular COBOL program (the program is given in Appendix 1).

Having defined maintenance and its problems in Chapter 2, tools and techniques to support various aspects of maintenance are presented in Chapter 3. Static complexity metrics (usually applied to individual programs) have been suggested as measures of the difficulty in understanding source code in maintenance and producing debugged source code in development. These metrics are directly applicable in maintenance as the code exists whereas for most development operations they must be estimated. The metrics range from simple counts of language tokens in a program through measures requiring the application of complex algorithms for their calculation. Section 3.1 reviews and compares many proposed complexity metrics.

Various documentation is used by managers, users and maintainers to aid understanding of an application system. As well as using documents, maintenance is concerned with keeping documents up-to-date and consistent. Program source code itself is a form of documentation. Several systems or environments have been proposed for general documentation support (these are summarised in Appendix 2). Aspects of documentation support relevant to maintenance, including document categorisation, are discussed in Section 3.2.

Software tools can automate or, at least, support many maintenance related tasks including reformatting, control and data flow analysis, restructuring and dynamic analysis of programs. Such tools are often useful both in development and maintenance (e.g. RXVP [EBE80] and SADAT [VOG80]). A number of tools are briefly summarised in Appendix 2. In Section 3.3, these tools are classified and general maintenance support through tools is examined.

Syntactic analysis of a program's source code is a feature of many tools. Frequently, tool functions make use of a pool of syntactic information gathered earlier. For example, the control and data tracing features of MAP [WAR82] and program instrumentation for dynamic analysis in RXVP [EBE80]. Syntactic analysis and the production of a syntactic data base are tasks worth isolating in single purpose software tool. In Section 3.4, the idea of a program analysis system composed of a variety of tools, most of which make use of syntactic data base information, is explored. The logical contents of such a data base are also identified in this section.

Chapter 4 summarises the implementation of a Syntactic Source Code Analyser (SSCA) and it's database (SSCA DB) for a version of COBOL. Such a system is a first step toward a maintenance support system based on static analysis. Availability of a SSCA DB should encourage development of more advanced COBOL analysis tools and provide a measure of integration between these tools.

COBOL was chosen as the language to analyse because of the large number of commercial programs written in it (approximately 80% according to Al-Jarrah and Torsun [TOR79]). However, the proposed revised X3.23-Sept. 1981 COBOL language definition [COB81] defines a large and complex language composed of a nucleus and eleven functional modules. For a prototype SSCA, it was considered desirable to reduce this standard by removing many special purpose modules and simplifying some language features. The reduction process is outlined in Section 4.1 and Appendix 3A. Appendix 3B contains the reduced COBOL language definition.

Part of developing a SSCA system involves selection of an appropriate Data Base Management System (DBMS) for the SSCA DB and detailed data design for the SSCA DB. Section 3.4 has already presented a logical view of what should be in this database. In Section 4.2 this view is elaborated for application to COBOL. The new data model is then used for the selection of a DBMS (the INGRES relational system was chosen) and, finally, an implementation data model is prepared.

Section 4.3 describes the methods employed to build a syntactic analyser and formatter for COBOL programs. The implementation was carried out using a number of construction devices available on a VAX 11/750 running ULTRIX-32. ULTRIX-32 is a trademark of the Digital Equipment Corporation. The construction tools included C (a general purpose programming language), AWK (a pattern matching language), LEX (a lexical analysis preprocessor for C), YACC (a grammar parsing preprocessor for C) and EQUEL (a C/INGRES interaction language). Extensive use was also made of the technique for transferring data between executing processes known as piping.

Chapter 5 presents conclusions from the research carried out in this thesis. The conclusions cover areas such as maintenance in general, maintenance support through software tools and evaluation of the SSCA development presented in Chapter 4.

CHAPTER 2. AN OVERVIEW OF SOFTWARE MAINTENANCE.

2.1. A MAINTENANCE DEFINITION AND REASONS FOR MAINTENANCE.

In this section the generally accepted broad definition of maintenance is presented and problems with this definition examined. Why maintenance is an essential function for the continued operation of any software system is also summarised.

2.1.1. A GENERAL DEFINITION OF MAINTENANCE.

The definition of maintenance used by most authors in this field and used in this work is :

> "that collection of activities that relate to correcting, adapting or perfecting software in production use"
>
> [ARN82]

In his paper Arnold points out that :

1. Software is a broad term and includes program code and related design information, as well as documentation.

2. "Correcting" is the removal of functional errors (i.e. resolving differences between the specification and implementation).

3. "Adapting" is the alteration of an application in response to changes in the hardware, software or business environment (this does not include addition of new functions).

4. "Perfecting" is modification to improve performance, efficiency or in some cases maintainability. Enhancements (addition of new functions), if classified as maintenance, are usually regarded as perfective maintenance.

Some writers in this field use other names for the activities. For example, to Vessey and Webber [VES83] perfecting is increasing productivity and Chapin [CHA81] lists the three activities as counteracting bugs, adding functions and modifying (& deleting) functions. However, in the main, there appears to be general acceptance of Arnold's definition.

2.1.2. THE REASONS FOR MAINTENANCE.

Riggs [RIG69] defines maintenance in terms of who "commissions" the process. His definition is that maintenance "is the activity associated with keeping operational computer systems continuously in tune with the requirements of users, DP operations, associated clerical functions and external demands by such entities as governments". This definition implies that the underlying reason for most maintenance is some kind of change in the environment outside the computer system which impacts on the system.

Lyons [LYO81] states that the function of software (and hence software systems) is to enable certain decision processes of an enterprise to be computerised. This suggests that, after a system's development process has been carried out and a new system effectively delivered to users, the new system carries out or supports various business functions as they currently exist in the enterprise. This would be the case if the development was considered successful. As few programs, at least intentionally, exhibit self modifying characteristics in current software technologies, a system is a snap-shot of a soon-to-be-outdated approach to business activities. The approach to activities usually changes to some extent over time and the main way a software system may evolve to meet these needs is through maintenance. This view of maintenance and software has also been expressed by Lehman (in [ELS82]) and others.

Thus although software does not rust or rot ([HAR83], [LYO81]), it is a mistake to consider software as stable ([PUN75], [MUN78]) because it tends to deteriorate in usefulness over time. Unfortunately, the nature and amount of deterioration is often impossible to forecast causing many maintenance management and scheduling problems [PUN75].

2.1.3. PROBLEMS WITH THE GENERAL MAINTENANCE DEFINITION.

The greatest problem with Arnold's maintenance definition occurs when major or costly changes are required in a software system. It is difficult to distinguish between new development that impinges on a system and maintenance of the system. The problem is further aggravated when a system is close to requiring a complete rewrite (i.e. it gets too far out of phase with the real world it models to be further modified). A complete rewrite has always been considered as development not maintenance.

Boehm [BOE81], under certain circumstances, regards redesign of less than 50% of existing code as maintenance and more than 50% development. Bell [BEL84] requires maintenance tasks to have at least 25% of the programmer's time spent on the understanding of the existing system. It is suspected that data processing management make the distinction between these tasks on the basis of the cost of making changes. If the cost is too great for the maintenance budget the task will be considered to be development.

No formal rules on where maintenance stops and development begins will be presented in this thesis. But as a guide to be used in later chapters the following is given. Large modification tasks involving existing software can be usually be divided into those which involve changing how the present code works and those which are mainly concerned with adding new features and extending the system (see next section). For major modifications to how software works (e.g. rewriting) Boehm's requirement for deciding whether a task is maintenance or not seems reasonable. For major additions to a system making the new development / maintenance decision should involve consideration of whether the logical function (defined in development) carried out by the system will be significantly altered when the modifications are made. One of Boehm' other task classification statements indirectly uses this criterion on which to base a maintenance definition. This statement defines activities involving little existing system redesign but development of sizable (greater than 20% existing code) extra modules as development.

2.2. MAINTENANCE CLASSIFICATION.

The manner in which the maintenance function is categorised in an organisation may affect the management control, resource allocation and effectiveness of the overall maintenance task. A number of ways of classifying types of maintenance are looked at in this section.

There are many ways to divide individual maintenance tasks into groups. Given Arnold's definition (see previous section), an obvious scheme is to group maintenance into corrective, adaptive and perfective tasks. According to Lientz [LIE83] this rudimentary classification has been used by some maintainers and found beneficial for cost estimation in a situation where maintenance costs are charged back to user organisations.

It seems anomalous that perfective maintenance includes enhancement maintenance as well as performance improvements. Performance improvements change how the processing is done inside the system hopefully without affecting what it does; whereas enhancement maintenance effectively adds to the functional requirements of the original system specification. Thus perfective maintenance can be subdivided into enhancement and true perfective maintenance. A survey by Lientz and Swanson [LIE80] gave the following estimates of the proportion of maintenance effort devoted to different categories of maintenance :

Maintenance Category	Proportion of Effort
Corrective (Emergency fixes & routine debugging)	22 %
Adaptive (from changes of data/hardware/software)	24 %
Perfective (improving efficiency or documentation)	10 %
Enhancement (new feature addition)	42 %

Richardson [RIC83], who uses the above categorisation, states that for correction and adaptation a maintainer needs a thorough understanding of the software modules being modified whereas perfective and enhancement tasks require greater knowledge of module interaction to re-evaluate the system structure. Also, all maintainers need a comprehensive understanding of the change request.

A slightly more extensive classification than those given so far is presented by Reutter [REU81]. Categories of emergency repair, corrective coding, upgrades, growth, support, changes in conditions and enhancements are suggested.

From the table above it can be seen that corrective maintenance is a relatively small portion of all maintenance tasks. Corrective tasks have also been seen by many authors as a special group as they tend to require immediate action by maintainers and stop gap measures are more likely to be employed to resolve any problems found. Vessey and Weber [VES83] found that in three organisations 90% of programs undergo less than three corrective repairs during their lifetime and the majority undergo none. Hutchinson [HUT86] handles repair work by dividing maintenance into error correction and system extension (which includes adaptation, perfection and expansion). Zvegintzov [ZVE82a] ignores error correction completely in his categories.

Marsh [MAR83a] includes two classifications in his paper. He divides tasks and requests into mandatory or discretionary divisions and also by where the maintenance requests originate (from users, from a failure or problem, and other). The results of his calculations over a total of 19 person years of software support are tabulated below.

Category	Percentage of requests	Percentage of effort
Mandatory	36.6	22.9
Discretionary	63.4	77.1

Request Category	Percentage of requests	Percentage of effort
User	43.4	66.9
Failure or Problem	35.8	20.0
Other	20.8	13.1

An alternative, or supplement, to dividing maintenance tasks by the type of request is to categorise the tasks by which software modules are to be maintained. Lyons [LYO81] states that an 80/20 rule exists for maintenance, "20% of the programs (in a system) cause 80% of the problems and corresponding costs". This suggests that a useful basis for a classification is the previous maintenance history of a module or program. This classification is already used, as managers usually attempt to allocate the same maintenance staff to the same group of software. The amount of information available for such a classification increases over time and if a modification task similar to a previously executed one (similarities determined by some other categorisation) is to be carried out, the history of the former is an extremely good indicator of costs and problems likely to be found in the latter.

Many problems in maintenance (see next section) stem from a lack of standard procedures for almost everything from accepting a change request to documenting a change. Classification of maintenance tasks is desirable for activities such as cost estimation, scheduling, and resource allocation but classifications shouldn't affect standards. For instance, all software modification tasks should receive appropriate retesting and redocumentation to assure a consistent quality within a particular system. Isolation of corrective work from system extension should not be complete. If time restrictions apply to a repair then post modification installation testing will be required.

However, given the restricted applicability of any maintenance categorisation, the test of a classification is its usefulness. At the moment so little maintenance information is available for managers or programmers to base decisions on, that virtually any classification is useful. But different divisions lend themselves to different uses. A mandatory / discretionary division is helpful for managers scheduling tasks, but irrelevant to maintenance programmers performing the source modifications. As more becomes known about the maintenance function, it is likely that more detailed, accurate and organisation-dependent classifications will be derived and several will be used in conjunction at any one site. On-site personnel will need to tune these categorisations which suggests analysis of a pool of detailed maintenance histories and constant updating of this "maintenance data base".

2.3. GENERAL PROBLEMS OF MAINTENANCE.

Problems within maintenance have been defined, recorded, and surveyed by a multitude of maintenance researchers in the past 15 years. In this section an attempt is made to categorise and briefly describe some of the problems uncovered by these researchers.

Maintenance problems seem to fall into two groups; problems within the overall environment which affect maintenance and problems which are intrinsic to the maintenance task.

2.3.1. FACTORS WITHIN THE OVERALL ENVIRONMENT.

There are two main groups of factors affecting maintenance within the environment; factors related to how the software maintenance activity is embedded in an organisation and factors which represent deficiencies in other areas which impinge on maintenance.

1) Maintenance in the organisation.

a) Management - Many authors ([COO78], [LIN73], [LIU76], [MAR83b], [PAR85b], [REU81], [RIC83], [TIN83]) identify as a major problem the lack of techniques and structures to effectively manage either programmers working on individual maintenance tasks or the scheduling of maintenance requests. Others ([LIE78], [MAR83a]) report that such activities are known to be difficult to manage. Inadequate management usually exacerbates other problem areas. For example, lack of defined maintenance standards reduces uniformity in change request handling and redocumentation [NAR84].

b) Organisational structure and personnel factors - Maintenance usually perceived to be a never-ending boring chore by maintainers [LIU76]. Work in this area often goes unrewarded and unrecognised and programmers are assigned to maintenance if they don't meet the requirements to be allocated to a development project (see [CHA81]). This, combined with other factors (such as no formal training in maintenance techniques [PAR85a], [TIN83] and no maintenance career structure), causes low morale and lack of professionalism among maintainers. From a corporate resource perspective, there is very little protection of the human investment in maintenance [MAR83b]. Chapin [CHA85] found by survey that personnel factors accounted for 19% of what could be termed the "maintenance problem".

2) Lack of support from other areas.

a) Development - Maintenance uses the products of development. These products include the application system as well as various levels of documentation (e.g. system, module, user and operator documentation). Short cuts in development can add dramatically to the quantity and cost of maintenance required on a system [PAR85b]. This can be as direct as leaving bugs in code, but is usually caused by poor or missing documentation coupled with source code which is difficult to understand, modify and test ([PUN75]). Inadequate documentation on its own is considered a major problem ([CHA85], [LIE83], [LIU76]).

b) Users - The main problems which users give maintainers stem from the users' lack of understanding of the application system being maintained. The problems usually surface in the form of large numbers of unreasonable or unrecognisable change requests requiring action by maintenance personnel. Lack of user knowledge may be symptomatic of problems from other areas (e.g. minimal user training during development or after maintenance). Conflicting proportions of the overall "maintenance problem" are attributed to lack of user knowledge in the literature, Lientz [LIE81] 59% and Chapin [CHA85] 5% (although Chapin doesn't regard things like difficulty in communicating with users as a subproblem of user knowledge).

2.3.2. FACTORS INTRINSIC TO THE MAINTENANCE TASK.

These problems arise from the fact that almost any software is difficult to maintain. Marselos [MAR83b] states that maintainers must constantly use creative energy to understand, or get around the constraints of, the system being maintained.

Intrinsic problems have been recorded under headings such as program understandability, maintenance task difficulty, program modifiability, program testability (from Harrison's subfactors in maintenance [HAR83]), and software complexity. It is arguable that most of the content of these problems is traceable back to other factors (e.g. difficulties with understanding programs because of poor documentation, a development oversight). However, given that even experts cannot agree on what makes a program hard to understand [HAR84], in some sense "good" [OGD72] or complex [WEI74], it is clear that there are implicit difficulties in maintaining software which, although they can be reduced, will not disappear with support from other areas. Possible reductions in maintenance problems may be traded-off in other areas (e.g. development costs).

An idea of the size of these intrinsic problems can be obtained from Chapin's calculation [CHA85] that software characteristics account for 48% of the overall maintenance problem. Intrinsic problems are looked at in more detail in Section 2.5.

2.3.3. CONCLUSIONS ON SOFTWARE MAINTENANCE PROBLEMS.

Maintenance is a multifaceted function and the fact that problems encountered in it are at least partially traceable to many other areas attest to how interrelated maintenance is with application users, application development and indirectly with the whole corporate structure. No one tool or technique is going to resolve all problems. Maintenance issues must be considered in many other organisational areas rather than just the section which is responsible for maintenance if problems are to be solved or alleviated.

2.4. MAINTENANCE LIFE CYCLES AND STEPS.

This section links the development and maintenance phases of an application system. A functional division of the maintenance task is also presented.

2.4.1. A GENERAL MAINTENANCE LIFE CYCLE.

2.4.1.1. THE SYSTEM LIFE CYCLE.

To get the maintenance function in perspective it is necessary to identify where it fits into the broad set of growth stages of application systems. These growth stages have come to be known as the "System Life Cycle Model"; although Zvegintzov [ZVE82a] argues convincingly that the model is neither cyclic nor portrays the system's life.

The system life cycle, as presented by Boehm [BOE84], has the major stages of :

1) Feasibility
2) Plans and Requirements
3) Product Design
4) Programming
5) Integration and Test
6) Maintenance and
7) Phase-out

The naming conventions and groupings of system life cycle steps vary among authors in this field. For instance, Boehm (above) has seven phases, Teague and Pidgeon [TEA85] eight, Zvegintzov [ZVE82a] six and Powers, Adams and Mills [POW84] five. However, there does appear to be general consensus about what functions are contained in the overall development stage of an application system and the order in which they occur. Post development activities are usually lumped into a phase called "Operation and Maintenance" or something equivalent. Some authors have extracted the endpoint of a system's life out of this last phase. Boehm calls the endpoint "Phase-out" and Powers, Adams and Mills "System Obsolescence". This is an attempt to recognise that special techniques are required to :

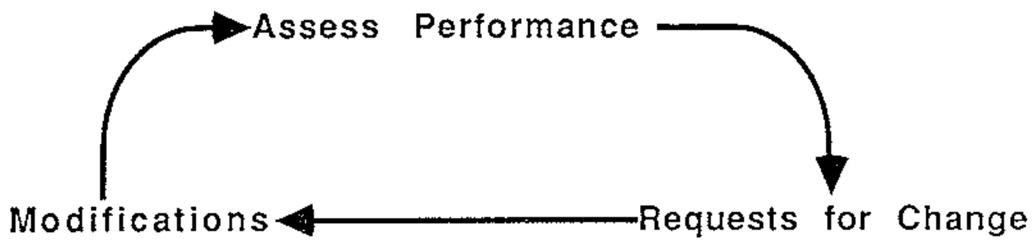
a) identify the system's end of life and

b) cope with the endpoint (usually by system replacement).

2.4.1.2. THE MAINTENANCE LIFE CYCLE.

Zvegintzov [ZVE82a] presents a model of the activities within the Operation and Maintenance phase of the system life cycle. This model is illustrated in Figure 2.4. The model identifies the fact that, as application system operation continues, constituents (users, managers etc.) generate requests for changes on the basis of system performance. These change requests prompt maintenance personnel to develop and implement modifications to the system. The modified system then becomes operational and subjected to more performance assessment by constituents. The model shows that this

<u>A Model of Operational and Maintenance Activities</u>



(From [ZVE82a])

Figure 2.4

phase in the system's life is truly cyclic.

A more elaborate breakdown of operation and maintenance is given by Taute [TAU83] :

1) Request a change

2) Estimate effort required for the change
3) Schedule the change
4) Program the modifications
5) Test to verify that the change performs as expected
6) Document the modifications
7) Release the modified system replacing the existing one and
8) Operate the system

This model is also cyclic with one step leading into the next and system operation prompting change requests.

2.4.2. THE SOFTWARE MODIFICATION TASK.

This subsection looks in more detail at the task of actually maintaining a program or system. In terms of the 8-activity software maintenance life cycle of Taute [TAU83] elaborated in the previous subsection, the content of the steps to be carried out by the maintenance programmer (usually on his own) is considered. These are primarily the phases of programming, testing and documenting and to a lesser extent training and implementation.

The programmer's task contains the actual work of maintenance and is the most resource consuming part of the maintenance cycle. All other activities (e.g.

maintenance estimation, scheduling and, say, the quality assurance checks of Centre's alternative maintenance cycle [CEN82]), although important, merely support the programmer's task. This subsection is an attempt to define what a maintainer does. This will form a basis for the development of tools and techniques to directly aid maintenance work.

2.4.2.1. SOFTWARE MODIFICATION STEPS.

Dividing the maintainer's activity into small definable portions is an attempt to specify a set of steps which can be applied to making almost any type of change to any system, subsystem or program written in virtually any programming language. Solution steps are, of necessity, going to be generalised and highly abstracted.

Relatively few authors have attempted to define the steps a maintainer would reasonably adhere to when modifying a program. Of the ones that do, Zvegintzov [ZVE82a], Harrison [HAR83], Liu [LIU76], Yau and Collofello [YAU80] give the most insight to the task.

Zvegintzov isolates 6 steps of the problem he calls "How to make a change" :

1) Understand the request
2) Transform the request to a change; the change

is the modification goal

3) Specify the change : choose cut-line and patch

4) Develop the patch

5) Test and

6) Install.

Cut-line refers to existing procedures or code which need to be changed or removed whereas the patch is the group of new procedures. For the patch procedures, the specifications of what each one should do are defined in step 3 so that patch generation (step 4) may proceed using the same techniques as the development cycle's phases of detailed design and implementation. Zvegintzov states that the major intellectual challenge of maintenance programming is the choice of the cut-line and the aim is to minimise the impact of the cut-line on the existing system as well as to attempt to isolate the patch from the variability in the existing system (i.e. to try to produce free standing code as patches where possible). The use this goal, although reasonably successful in many cases, does not always generate desirable consequences for maintenance in general. Maintenance goals and implementation methods are examined later (in Sections 2.5.3.3 and 2.6).

Harrison identifies a very similar group of steps to that of Zvegintzov. The only real difference is the insertion of an additional step between 2 and 3 above for "understanding the existing software". Yau and Collofello also have an understanding step but steps 3 and 4 are worded "generate particular maintenance proposal" and "account for ripple effect". This subdivides the combined specification and patch development task into portions pertaining to the local module and then to the rest of the system.

Liu looks specifically at modifying a piece of software and identifies the steps of the "maintenance function" as:

1) Understand the capacity, function or logic of the existing system
2) Develop new logic for the new request or additional feature and
3) Incorporate the new logic into the existing system.

Liu admits that the new logic may not always be able to be melded into the existing system, conflicting situations can be created. It is implied that if such conflicts arise either the new logic should be modified (without affecting what it does) or the existing system "redeveloped" with the new request or feature as part of the specifications. The goal of a maintainer, according to Liu, is to make certain the new logic correctly fulfills the original request and, as well, to make sure that unmodified portions of the existing system are not inadvertently disturbed.

From comparing both these authors' steps and from a number of breakdowns of the overall maintenance process by other authors ([ELS82], [CON84], [CEN82] and [TAU83]) a group of required actions within what can be called the "software modification task" can be determined :

1) Understand the request and transform it into some sort of modification or new feature
2) Understand the existing software
3) Identify a framework for modifying the existing software into what is required
4) Implement the modifications
5) Test the new software
6) Redocument and possibly retrain users and
7) Install the new software.

There are good reasons for personnel other than maintenance programmers carrying out, at least partially, the actions of request handling, testing, documenting, training and installation. Not the least of these reasons is to reduce the maintainer's workload. Most organisations currently have a three to four year backlog of maintenance requests [MAR83c]. A maintainer must, however, either directly use the results of, carry out, or supply some input to, all of these activities.

2.5. SOFTWARE MODIFICATION.

The cost of maintenance (discussed in section 2.1), the difficulty in defining the end-points of maintenance (see section 2.2) and the general problems associated with maintaining software systems (reviewed in section 2.3) lead us to the inevitable conclusion that modifying software is an extremely complex and usually much underrated activity. If the task was relatively simple it would not require the management procedures of request verification, cost estimation and scheduling or the post source modification activities of thorough testing, redocumentation and possibly user retraining. Most researchers in this area have also reached this conclusion. The abundance of recent articles on the subject, the number of maintenance or maintenance applicable tools becoming available (e.g. the SADAT and RXVP test systems reported by Geiger [EBE80], the source restructuring tool SUPERSTRUCTURE and path analyser SCAN/370 both mentioned by Richardson and Hodil [RIC84]) and statements that merely managing system support is fraught with problems ([MAR83a], [CHA85]), all attest to the underlying difficulty of modifying a software system. This section explores specific problems within software modification and some guidelines for maintenance programmers are presented.

2.5.1. SOFTWARE MODIFICATION INFLUENCES.

Some factors which directly influence software modification are:

a) Costs and Programmer Productivity - Virtually all business activities have minimisation of cost software as a basic objective and modification is no exception. Maintenance as a whole is viewed as an area in which savings can be made because it uses a lot of resources with little apparent effect, as maintenance is never ending. McKee [MCK84] calculates that maintenance is 32.1% of an application system's total hardware / software cost and Boehm [BOE76] estimates 40%. Both authors conclude that the portion of cost attributable to maintenance is increasing.

   Computer programmers usually perform both development and maintenance functions. Using a very simple model of programmer activity with parameters based on survey results and assuming a programmer maintains all code he writes, McNeile [MCN84] calculates that after 1.6 years only 20% of a programmer's time can be spent doing development tasks. Because of the high cost of maintenance and its stifling effect on new software production there is often a to minimal allocation of resources to maintenance projects. As a result, an environment develops which favours stop-gap measures which themselves usually, in the long run, increase the overall maintenance cost.

b) Personnel Factors - These factors have already been mentioned as an organisation problem in Section 2.3.1. Two specific aspects affecting software modification are maintainer availability (part of the corporation's resources allocated to maintenance) and maintainer quality (e.g. lack of specific maintenance training, see Section 2.3.1).

c) Maintainer Understanding - Arguably the most critical factor in the successful modification of a piece of software is how much the maintainer knows about both the particular software involved and also the system of which it forms a part. Marselos [MAR83b] and Basili and Mills [BAS82a] regard understanding as crucial to maintenance. The latter suggest that improving documentation is a major way of increasing understanding. Bell [BEL84] goes so far as to define maintenance partly in terms of existing system understanding (see Section 2.1.3). Techniques and aids for understanding software will be looked at in later chapters. Maintainer understanding is influenced by factors from a) and b) above but is strongly affected by software factors particularly aspects of documentation and maintainability, which are examined next.

d) Software Factors - Details of particular software factors which appear central to maintenance are investigated in the three subsections below. Specific software tools and support methods will be identified later in Section 3.2.

2.5.1.1. THE INFLUENCE OF DOCUMENTATION.

Again this has been cited as a problem in Section 2.3.1. Poor documentation of development or previous maintenance increases the time taken, and therefore the cost, for a maintenance programmer to understand a module or system. Documentation which is to be used by maintainers must be structured and organised for understanding and ease of update. As indicated above, a maintenance programmer needs information about the main system as well as about individual programs. Guimaraes [GUI83] reports that the kind of system information found most useful by maintenance programmers is a "system flowchart" containing structural and architectural information about both programs and data files. He also ranks the value of narrative descriptions of the function of programs and modules very highly. Schneider [SCH83], when producing a format for a formal document library, places both program definition data and program/data file interaction information under the heading of "program documentation". Guimaraes's method of having one or a group of "system chart(s)" should insure more consistence and clarity among this system documentation than a piece-wise approach. Unfortunately, many computing professionals still tend to regard maintainer documentation as just embedded comments in source listings.

However, program commenting is still important. The source text is the final authority on what is executed. Elshoff and Marcotty [ELS82] see embedded program documentation as enhancing the program's readability and they present an internal comment structure for programs. They also suggest that future modification difficulties can be reduced by commenting discoveries about a program as they are found. This suggestion is expanded upon by Basili and Mills [BAS82a]. Munson [MUN78] states, "even if this (program) documentation is good when delivered, invariably it will quickly fall out-of-date as the product is modified". Munson lists criteria for documenting source code listings which include accuracy requirements, purpose, error recovery procedures and, also, a modification history.

2.5.1.2. THE INFLUENCE OF MAINTAINABILITY.

Maintainability encompasses a large number of software characteristics including clarity, structure and flexibility. There are two major modification-related reasons for using development and maintenance techniques which enhance or support such characteristics :

1) To aid a maintainer in understanding a system or module.
2) To directly aid the process of extending or adjusting a system or module.

Many software development techniques which purport to support both the objectives above have been grouped under the banner of "structured programming". Munson [MUN78] identifies some of these techniques as well as apparent extensions into procedural abstraction and "data hiding".

Even when program source code is extremely complex, understanding is enhanced by readable source code. Elshoff and Marcotty [ELS82] define many "hints" for readability including making loops obvious, having explicit loop terminators and removing label variables used only for blocking. Ideas for directly supporting module enhancements are put forward by Punter [PUN75] and Hutchinson [HUT86]. Many of Punter's techniques are mostly applicable to relatively low-level assembler languages (e.g. using indirect addressing so that data areas and modules are more movable) but suggestions for program independence and open-ended design, where possible, are also presented. Hutchinson gives more general ideas with particular reference to data structures (thus program design) so that record formats and individual data fields can be changed (enlarged) without impacting on software not using the new or modified data elements.

2.5.1.3. THE INFLUENCE OF TESTABILITY.

Ideally, software should be coded clearly enough so that a program can be "seen" to work [PUN75]. Unfortunately, in any nontrivial example, testing is required. Testing is the last check that a maintainer has understood the existing software sufficiently enough to be able to alter its function in accordance with maintenance request without introducing side effects. Taute [TAU83] states "the quality of testing is a function of both the thoroughness of the test plan and the quality of the test data". Both Taute and Liu

[LIU76] identify the establishment a strict testing procedure as being essential for maintenance quality assurance (QA). Liu also presents a priority for tests after maintenance. He determines that the unmodified portions of a system should be tested before the modified portions. However, the likelihood of semantic errors in the modified software would suggest that the test ordering should be the other way round. Boehm [BOE73] found by survey that after altering only 10 statements in a program the chances of a successful first run are, at best, 50%. Hence, testing the modifications for errors before the rest of the system should reduce the amount of retesting necessary.

After maintenance, the entire application system needs to be throughly tested with a variety of data. Often testing is shortened so that only portions of the main system that a maintenance programmer "thinks" could be affected are checked. The reasons for this occurring may be because of cost, or that the maintainer just doesn't know enough about the system as a whole to test all aspects of it. As Deutsch [DEU81] points out, manual testing itself is an error-prone process and automated testing or error-checking is more reliable. If an automated software test driver system is used this facilitates the retention of test cases (and correct test results for automatic comparison) so that "regression testing" (as suggested by Panzl [PAN78]) can be carried out. Thus, automated testing, as well as enhancing reliability, may reduce the amount of detail about non-modified parts of the application system that a maintainer must learn. This test case data documents the application for maintenance.

In development, automatic test drivers and testing in general should strive for complete test coverage of software. Huang [HUA78] states that the test requirements for a program :

1) Each statement in the program should be executed at least once.
2) Each edge in the flowchart should be traversed more than once.

For most software these requirements are virtually impossible to attain (see Boehm's software reliability: technical problems [BOE73]). In maintenance, such requirements may be able to be realised in some cases because only statements and flow paths which have been modified in some way (e.g. through association with modified data items) need to be checked. The problem with this approach lies in reliably determining how much of the system to test. It is not just a case of directly comparing the old and new sources because of inexplicit links between source statements, data items and control flow. What is really needed to attempt to satisfy these test requirements is some kind of computerised ripple effect analyser (see Section 2.5.2.2. for a discussion of ripple effect).

2.5.2. MAINTENANCE QUALITY.

The quality of the work done in maintenance as perceived by maintainers is dependent on :

1) The role the maintainers see themselves fulfilling and

2) The verification and validation procedures adhered to during the maintenance process.

Aspects of the general maintenance role are examined in Section 2.6. Procedures for maintenance QA include not only testing after the implementation of changes (see previous subsection) but also the processes of maintenance preparation, data/file conversion, documentation update and system acceptance QA outlined by Center [CEN82]. These QA activities are also useful as milestones which provide management information for the scheduling of maintenance personnel. QA routines can usually be placed between phases of some stepwise division of the modification task (Center gives an example of this) producing an enhanced modification cycle.

2.5.2.1. QUALITY ASSURANCE.

For any one maintenance project the quality of the maintenance work depends primarily on the personnel involved. However, an attempt at providing some consistent degree of quality assurance in maintenance must identify a framework which promotes good practices among maintainers and helps to insure that no necessary activities are overlooked. Connell and Brice [CON84] recommend that maintenance quality controls consist of workable mechanisms for measurement, evaluation and feedback. Taute [TAU83] states that a quality assurance program should address four areas :

1) Phased approach
2) Procedure flows
3) Maintenance guidelines and
4) Implementation

The phased approach (such as adherence to enhanced modification cycle looked at above) provides a number of benefits, one of which is standardisation of maintenance procedures. Procedure flows document the system being maintained at a relatively high level (e.g. data flow diagrams and program specifications) and guidelines are meant to be suggestions (probably based on previous experience) to aid good judgement when making maintenance decisions. Implementation relates to how such cycles and guidelines are introduced and made workable in a maintenance department. Thus both software testing and documentation feature heavily in quality assurance measures.

Any maintenance support tools will directly or indirectly enhance quality so consideration of current quality problems is necessary to aid tool design. There seems little argument that quality problems are not major within maintenance. Boehm [BOE73] regards a lack of reliability certification as a major software problem. He states that every new release of OS/360 contains roughly 1,000 new software errors. Ogdin [OGD72] states that for typical software systems the reliability histories show that failure rates peak after maintenance. In addition, there is an overall rising failure rate throughout a system's life as successive maintenance tasks are carried out. Lyon's [LYO81] rule that 20% of programs cause 80% of the maintenance costs and Vessey and Weber's [VES83] supporting figures showing that at one site, 9% of programs had 47% of the repair maintenance tasks performed on them, suggest that, in many instances, maintenance spawns yet more maintenance.

2.5.2.2. MODIFICATION PHENOMENA.

Two phenomena have been isolated through experience in maintenance projects. From a quality standpoint, these effects highlight specific problems with current methods for designing and implementing software modifications. In the next subsection reasons for such phenomena occurring are discussed and possible solution schemes are considered.

RIPPLE EFFECT - "In software, the effect of a modification may not be local to the modification, but may also affect other parts of the program. There is a ripple effect from the location of the modification to the other parts of the program that are affected by the modification" [YAU80]

The above quote from Yau and Collofello defines ripple effect. These authors regard ripple effect as intrinsic to the modification task and provide a phase in their maintenance steps to account for it (see Section 2.4.2.1.). Discovering what ripple effects a particular change creates is complex and, like complete testing, complete ripple effect analysis usually cannot be performed because of maintainer time restrictions [CON84]. Boehm

[BOE73] quotes that, of all errors detected over a 3 year interval in one particular application, 19% were caused directly by unexpected side effects to changes (i.e. ripple effect).

STRUCTURAL DECAY - Maintenance work often melds old and new source code in an ad hoc manner which compromises any original structure and results in an unstructured mass of code. Connell and Brice [CON84] state that the problem is compounded by many small patches applied throughout the life of a system. Individually each patch is relatively insignificant structurally, but together they form a threat to even the most structured, modular, top-down original code.

The kinds of structure within existing software which are in jeopardy from maintenance are not only the hierarchical control structures of individual programs developed in some 'structured' manner but general design principles on which the system itself was founded.

Structural decay is a major side effect problem in maintenance. Its affects can be seen in an application system's operational and maintenance history. Lientz and Swanson [LIE81] concluded from survey evidence that the older a system is the more perceived maintenance problems it usually has. Ogdin [ODG72] suggests that system failure rates increase throughout a system's life (see previous subsection). This decrease in reliability may be partially due to introduced ripple effects but a lack of structure will make any modification and testing problems worse. Psychological complexity of software is increased by structural decay because structured code is easier to understand than unstructured code. Curtis [CUR79] found (from tests done with programmers attempting to comprehend relatively small FORTRAN programs) that although "naturally" structured code could be maintained more accurately, the time taken to perform modifications did not vary much with increased program structuredness. Regardless of whether these results translate to most maintenance projects, higher source code complexity must result in increased costs for any subsequent maintenance tasks (even if it is only as a result of the time taken to hunt down introduced errors).

At some stage it is cheaper to scrap an old system and build a new one rather than attempt any further large maintenance projects. In many situations the scrapping of the old system may be almost entirely due to its lack of structural integrity. A short-term alternative to scrapping a system is to restructure or recode some existing programs. As Tate and Hayward [TAT85] state

"Restructuring is always difficult and painful, whether in the economy, a business, a building, or in an information system".

Restructuring or recoding on any major scale is not really feasible unless it is automated or at least computer-assisted. Richardson and Hodil [RIC84] state that such tools are now becoming available although the need for these tools to be reasonably intelligent may reduce their general applicability.

Increasing the operational life of an application system can be achieved by minimising the maintenance-induced deterioration in system structure, complexity and reliability (in terms of introduced defects) [CON84]. Problems with attempting to reduce structural decay in maintenance are considered in Section 2.5.3.

2.5.3. THE IMPLEMENTATION OF SOFTWARE MODIFICATIONS.

2.5.3.1. OMISSIONS WHEN IMPLEMENTING CHANGES.

The problems of ripple effect and structural decay suggest the manner in which the first 3 actions of software modification (presented in Section 2.4.2.1) are often carried out. What seems to happen is that maintainers frequently attempt changes with minimal knowledge of the existing software. First, the modification request or new feature is divided into individual changes to software modules. This is a reasonable approach but to carry it out successfully requires an understanding of the overall system structure. Most of the reasons why this general knowledge is usually lacking have already been discussed in Sections 2.3.1 and 2.5.1. Minimal knowledge creates a tendency for modules which produce a "physical" input or output, such as a report or screen interaction pertaining to the new feature to be identified for modification whereas more obscure modules such as transaction monitors and other "collectors" of internal information are overlooked. When this modification technique is combined with incomplete testing ripple effect problems ensue.

If the original lack of software understanding culminating in ripple effect is also carried on within the task of implementing individual module changes, structural decay of module code can also result. As described in the next subsection, with minimal knowledge of the structure of a program, it is usually possible to patch a program at the top level (particularly when implementing additional functions or enhancements). The patching is successful, in so far that the modified program works without side effects, but the top level patch usually adds at least one additional control structure (or nesting level) to the program unnecessarily increasing complexity for later maintenance tasks.

2.5.3.2. A MODIFICATION EXAMPLE.

By way of an illustration of the dilemma facing maintenance programmers when attempting to implement changes the following program modification example is presented.

Jackson [JAC75] outlines a problem requiring a programmed solution. The problem involves a file of sorted card images; each card carries a branch-number and a card-type indicator, together with some other information. What needs to be done is to purge from the card image file (infile) records which are in some sense erroneous; producing two output files : a file of good data (outfile) and a error listing (errorfile). No detailed error diagnosis is required. Initially, in a "good" set of records for a branch there are exactly two branch records and the card types are numeric and not equal. The original card file records have been sorted into ascending order by card-type within branch-number.

An Initial Solution

Jackson [JAC75] gives a solution developed from the data structures involved. An alternative solution is given in the PURGE program in Appendix 1. Figures A1.1 and A1.2 of the same appendix contain program and data structures for PURGE. The PURGE program is loosely based on Jackson's ideas but is formed around the notion that in order to evaluate whether a set of records for a branch is "good" or not, three records (which are taken in sequence from the infile) must be looked at. If the first two records have the same branch number (and the third a different number) and valid, non-equal card-types, then the first two records form a good set. Given this basis of looking at a three record window on infile, starting and ending special cases can be derived as well as procedures for handling and recovering from groups of error records. The main set test is contained in paragraph Main-loop in a sequence of 4 IF-ELSE statements and outputting of the good set is done via a call to paragraph Print-good-set. In coding the solution the definition of a good set of records needs a little clarification. It was assumed that if an error card exists for any one branch, then all of that branch's cards are in error and can be outputted into the errorfile (see paragraph Error-branch-loop).

The First Modification

A reasonable modification to the above problem and solution concerns a minor redefinition of what constitutes a good set of branch records. We now stipulate that in some instances the two branch cards are allowed to have identical non-numeric card types. This may occur only when the card type is the special character group "MOD1".

Programs PURGE-A1 and PURGE-A2 in Appendix 1 give two solutions to the new problem. In program PURGE-A1 the new conditions have been carefully grafted onto the existing PURGE program's structure. The numeric card type test has had an extra AND clause added to the condition, as has the non-equality of card types test.

It is significant that the number of nested IF-ELSE statements in Main-loop has not been increased, although the complexity of individual conditions has. Given the modified problem to begin with, and applying the same notions and ideas for solution (as outlined above in the initial solution), it could reasonably be expected that the program produced would be very close to PURGE-A1. In program PURGE-A2 a simpler method of melding in the extra condition tests has been carried out. The modifier has attempted to maintain some structural integrity in the program, but has opted for adding a complete test for the new circumstances over the top of the other tests in Main-loop. Doing this successfully requires a lesser knowledge of the existing program than needed for PURGE-A1. Paragraphs Print-good-set and Get-2-cards need to be understood as the code to call to handle a good set of branch cards. However, the maintainer has kept away from the more complex and error-prone activities associated with adding extra condition clauses.

The Second Modification

This modification to the original PURGE program also involves redefining what a good set of branch cards is. An additional valid set now contains exactly one card which has the character card type "MOD2".

Programs PURGE-B1, PURGE-B2 and PURGE-B3 of Appendix 1 present possible programmed solutions to the new problem. PURGE-B1 gives a solution in similar spirit to the previous modification's PURGE-A1 solution. The careful combining of new and old tests in the paragraph Main-loop has been executed. The new problem requires an extra IF-ELSE statement to determine which kind of good branch set has been discovered (one with 1 card or one with 2 cards). In this solution, the extra condition clauses for both card type tests are rather obscure. The clauses test that the card set being checked has, in fact, 2 cards, regardless of whether one or both cards have type "MOD2". The precise coding for these tests is quite difficult and required several attempts. This shows the high potential for errors in the PURGE-B1 solution.

The PURGE-B2 solution is a kind of optimised way of adding an extra test for the new "MOD2" situation. Instead of putting the test at the beginning of paragraph Main-loop (which is possible and in some ways structurally superior but requires many more statements) it has been noted that if only one card exists for a branch then paragraph Branches-not-equal will be entered. A simple test and action has been added to this paragraph to see if the one card represents an error or not. The recovery actions to get into a state for testing the next card set are identical regardless of the outcome of the test.

The idea behind the PURGE-B3 solution is that the basic structure of the original PURGE solution is fine for testing for good branch sets which contain exactly two cards but seems inappropriate to be pushed into handling good sets containing one card. What has been done is to check a card just before it is written to the errorfile in paragraph Print-error-heading-card. If the card represents a good card set (with only one card for the branch) it is written to outfile instead of the errorfile. This solution is similar in idea to post-processing the errorfile to remove "MOD2" instances and sorting the selected cards into outfile. The other related possibility is to preprocess "MOD2" instances out of infile and again resort to sorting these cards back into outfile after PURGE has been executed. PURGE-B3 isn't quite just a melded post-processing function as only the first card of error-card sets is processed. This is all that's needed to be checked.

2.5.3.3. METHODS OF IMPLEMENTING SOFTWARE MODIFICATIONS.

We can use the modification examples given in the previous subsection to try and come up with some kinds of rules-of-thumb for reducing structural decay. Below we discuss what's wrong with various modification solutions.

In the first modification, the PURGE-A2 solution is slightly more complex than it really needs to be, the main problem being that a special case, the MOD1 situation, has been created where it need not be (at least in terms of the positioning of the special case test). PURGE-A1 will be easier to understand than PURGE-A2 when subsequent maintenance is done. The difference is almost imperceptible now, but after many similarly implemented patches PURGE-A2 is likely be a mass of special cases with many structural problems usually related to overlaps between the special cases themselves. Thus, a rule-of-thumb may be to try and meld the changes and existing program as much as possible. In the PURGE-A1 example, structural decay is reduced (compared to PURGE-A2). This is at the expense of the speed of designing the changes and, to some extent, the chances of making mistakes which may mean introduced errors.

In attempting to judge which second modification solution is in some sense "best", it seems that PURGE-B2 and PURGE-B3 are both more neat and more structurally sound (at least, in terms of complexity) than PURGE-B1. Although the original ideas behind testing a three card set at a time combine reasonably well with checking for the second modification, in the actual implementation of the PURGE solution extra tests for one card sets are difficult to mix into the existing condition structure. Both PURGE-B2 and PURGE-B3 represent structurally superior solutions with PURGE-B2 slightly preferred because it is less error-prone and less complex. There are reservations about the lack of explicit documentation of the new modification tests and their positioning in the code. Perhaps an external set of processing and sorting routines (outlined previously) would be the best alternative regarding the isolation of specific functions, although the increased amount of source code required seems extreme. Also adding external routines linked around the existing system may be just reducing program structural decay by increasing the decay in the functional arrangement of programs within the system.

It is reasonable to ask what kind of programmed solution would have been coded if the original PURGE function included the MOD2 situation. This solution (so long as it "fits in" with the design principles of the rest of the application system) should be the best structured. The design of PURGE-B2 is a possible solution to this new problem but PURGE-B1 doesn't seem appropriate. As illustrated by the example, a good rule-of-thumb to reduce structural decay in most cases is :

> "Maintain software so that the new code ends up looking something like what would have been produced if the existing system plus changes were designed from scratch".

The main problem with this rule is that it's going to take maintainers longer to produce well-structured patches and most of the time will be spent understanding the existing system. They will not only have to understand the design ideas behind the programs being changed but specific philosophies adhered to in application system development (e.g. the criteria that were used when the system was decomposed into functions or subsystems will be relevant in many cases). Much of this development documentation is not currently collected for maintainers. The rule above is supported by Parikh's statement [PAR85b] that development methodologies must offer exact guidelines and methods for maintenance.

The view of software put forward in the modification examples and the rule of thumb described are primarily at the program or module level. It is even more difficult and application-specific to attempt to give maintenance design decisions a system structural perspective. However, as a system is a combination of programs and modules, program-oriented rules and tools will still be very helpful in application maintenance as a whole.

2.6. THE ROLE AND GOALS OF MAINTENANCE.

Basili and Mills [BAS82a] specify the role of good maintenance as keeping the requirements, specifications, design and code documents up-to-date but this is not always done in practice. The overall goal is to "successfully" implement changes to an existing application system. At a minimum, the adjustments must fulfill the request which prompted them if they are to be successful. However, the long term definition of "successful" varies with factors like the expected remaining operational life of the application (or parts of the application system) and expected number of subsequent maintenance tasks. In the short term, constraints of time and money are likely to define perceived success.

In most instances, maintenance is being carried out on systems which are expected to operate for the foreseeable future and thus (from Section 2.1.2.) will certainly need further maintenance. This means that documentation update and avoidance of structural decay and introduced errors must be strongly considered within maintenance tasks. In this case the objectives maintainers should attempt to meet should be a combination of Basili and Mills's documentation task, Liu's request solving without new errors (considered above and described in Section 2.4.2.1.) and support of structural integrity within the application system.

From the modification example (Section 2.5.3.2.) and the discussion above, even low-level program redesign decisions require a reasonable amount of information about the application's development and future. It would seem appropriate for some of this information to come from maintenance managers (or senior maintainers usually in charge of programmers) and they should ideally have quite an active role, particularly in maintenance design decisions. The subtle trade-offs between structural integrity, time taken and error-proneness (as presented in Section 2.5.3.3.) are heavily dependent on the application and are unlikely to be performed well by programmers seconded to application support only for the duration of a particular maintenance project.

Whoever carries out the design and implementation of modifications, it has been shown in this chapter that understanding of various aspects of the existing system is absolutely fundamental to these tasks and hence maintenance itself. Designing tools to support understanding, as well as maintenance aids in general, are looked at in subsequent chapters.

CHAPTER 3. GENERAL MAINTENANCE SUPPORT.

3.1. COMPLEXITY METRICS.

In this section a number of software complexity metrics are reviewed and their applicability to maintenance is discussed. If aspects of complexity are critical to the difficulties involved in producing or modifying programs and if a particular metric successfully measures such aspects and is readily calculatable, then this metric would be generally useful in software development or maintenance.

In maintenance, most authors agree that the best measures will record the difficulty of understanding a program or the speed / accuracy of implementing changes. These metrics could be used for performance prediction and resource allocation in maintenance tasks or, possibly, even as an indicator of the "health" of parts or the whole of an application system (as suggested by Bell [BEL84]). In development, some metrics are proposed as predictors of development time and / or cost (e.g. Halstead's E) and also as measures of program quality.

Static analysis of source code, largely for understanding, is a major process in maintenance (see Section 2.5.1.). With this in mind this examination of metrics addresses static measures which attempt to quantify psychological complexity in source code (rather than computational or some other complexity [CUR79]). As Feuer and Fowlkes [FEU79] point out, these restricted metrics cannot really be expected to completely predict performance. However, such measures should provide a ranking system for programs and indicate abnormally complex software [ELS84]. Static metrics should also identify critical programmer and program task factors which contribute to performance [SCH81]. This would lead to firmer ground for the establishment of guidelines for good programming practice.

3.1.1. STANDARD METRICS.

The metrics most often cited by authors in the complexity area are lines of code (LOC), McCabe's cyclomatic number (V(G)) and Halstead's software science measures. These metrics and some variations are reviewed in this subsection.

3.1.1.1. LINES OF CODE.

Complexity metrics are deemed necessary in development and maintenance largely because a program's LOC is not a reasonable indicator of the difficulty in coding, modifying or comprehending the nature of the software. After analysing 585 PL/1 procedures from a large program analysis system, Elshoff [ELS84] found that both the number of source lines and number of input lines (preprocessed source lines expanding INCLUDE and other statements) were poorly correlated to construction effort. Sheil [SHE81] states :

> "The most salient single fact about programming is that the difficulty of programming is a very nonlinear function of the size of the problem."

For the "size of the problem", one could equally read "size of the program". However, LOC still have a part to play in determining complexity. Survey results (reported by Zolnowski and Simmons [ZOL80]) show that 52.2 % of respondents rated number of statements as significant in complexity. Many metrics taken on their own are unbounded (e.g. V(G) and E) and for inter-program comparison a size factor is needed compute what is effectively a measure of complexity density. LOC is still used extensively to indicate the size of a program or system for costing and programmer productivity, especially in development [ALB83].

3.1.1.2. MCCABE'S CYCLOMATIC NUMBER.

McCabe [MCC76] proposes a complexity measure derived (at least in theory) from the control flow (CF) graph of a program. A CF graph is a digraph model of the CF between statements in a program. In a simple CF graph, program statements are represented as vertices (nodes) and control branches between statements as directed arcs (edges). Paige [PAI77] describes CF graphs in more detail, including techniques for analysis and reduction of graphs. McCabe defines a metric called the cyclomatic number V(G) of a graph G by :

$$V(G) = e - n + 2 \qquad \text{where } e = \text{number of edges}$$
$$n = \text{number of nodes}$$

Using graph theoretic arguments, McCabe equates V(G) to the maximum number of linearly independent circuits in a CF graph of a program. Hansen [HAN78] states that three variations of cyclomatic numbers exist; CYC-MAX, CYC-MID, and CYC-MIN. The difference between these measures lies in the definition of a statement branch. CYC-MIN counts all selection and iteration statements as single branches. CYC-MID has the same branch definition except that multiple selection statements (e.g. CASE) are counted as if they were sets of nested IF-THEN-ELSE statements. CYC-MAX uses the expanded multiple selection statement and counts every logical operator in every selection or iteration statement as an individual branch. McCabe originally defined V(G) as CYC-MAX but Hansen maintains that CYC-MIN is the best of the variations as without CASE or IF statement expansions it reflects more of a readability element of the source. The main difference between the above cyclomatic numbers is whether a branch is defined by a condition or decision (predicate), where a statement such as IF (C1 AND C2) identifies two conditions but only one compound decision. However, from an analysis of 89 units (2040 files) of C source code Crawford, McIntosh and Pregibon [CRA85] concluded that there is a very close relationship between conditions and decisions. Elshoff [ELS84] measured this relationship for a large number of PL/I procedures as having a correlation coefficient of 0.93. Hence, it appears that V(G) should be fairly robust regarding branch definition. This does not support Myer's [MYE77] idea of using the interval CYC-MID:CYC-MAX as a complexity measure.

Most authors regard V(G) as a very useful measure. It is easy to apply, language independent (unlike LOC), and has a simple interpretation. Zolnowski and Simmons [ZOL81] found, by survey, that V(G) was a "significant consensus variable" in complexity and Baker and Zweben [BAK80] conclude that V(G) is a reasonable measure of the CF complexity of software. Schneidewind [SCH79] links V(G) to debugging difficulty and gives support evidence, from four ALGOL programming projects, to the view that digraph properties, like V(G), can quantify program quality. The main criticisms about the foundations of V(G) are that it is too simply based [MCT80] [PRA84] [WOO79] and fails to measure interactions between branch statements [HAR81a] [HAR81b] [PIW82].

3.1.1.3. HALSTEAD'S SOFTWARE SCIENCE MEASURES.

In his book Halstead [HAL77] derives a number of metrics which he calls "Software Science Measures". These metrics attempt to quantify coding effort, program level, predicted program length, predicted program volume, implementation difficulty, coding time and language level. All the metrics are based on the following counts of information :

$n1$ = number of unique operators

$n2$ = number of unique operands

$N1$ = total occurrences of operators

$N2$ = total occurrences of operands

No general agreement exists for determining which tokens in a language are operators and which are operands [CRA85] and ambiguities exist in Halstead's definitions for FORTRAN [SHE83]. Many classification techniques ignore all comments and declaration statements. This is consistent with the original theory which was intended to analyse algorithms but, as pointed out by Shen, Conte and Dunsmore [SHE83] and others, declarations form a large part of a program's LOC and production effort in many languages. In the remaining program statements tokens are usually regarded as operators or operands. Most languages define a set of unary, binary etc. operators but in addition punctuation tokens may be consider operators [HAL77] [FEU79] [ELS84] or ignored [HAN78]. Crawford et al [CRA85] allow provision for multiple tokens to be grouped as one operator. GOTO tokens are also a contentious issue. Halstead [HAL77] proposes that each GOTO token is counted as a unique operator for each label it branches to, whereas Elshoff [ELS84] counts the number of unique labels pointed at by GOTOs as operators. All schemes regard intrinsic function references as operators but user-defined function calls may be operators [ELS84] [HAN78] or operands ([CRA85] in most cases). Shen et al [SHE83] state that such references could serve both purposes at the same time.

Hansen [HAN78] suggests that subscripting of an array is an implicit operation. Overall operands tend to be user-defined variables or literals.

The most commonly cited of Halstead's metrics is the effort measure E which is suggested as a predictor of development cost. Both E and V(G) have been shown empirically to be related to program construction time in development [SUN81]. In maintenance, the difficulty indicator D (which is the reciprocal of the program level L) is likely to be more helpful as it supposed to measure relative error-proneness and ease of understanding. Feuer and Fowlkes [FEU79] report that L, when adjusted for size, appears to be a fair estimate of maintenance performance among PL/I modules. Shen et al [SHE83] summarise Halstead's metrics and conclude that E and D seem to be useful and are supported by empirical data. Computational formulae for E and D are :

$$D = \frac{n1 \times N2}{2 \times n2} \qquad E = D \times (N1+N2) \times \log2(n1+n2)$$

Halstead also presented a simple measure for the length of a program. Elshoff [ELS78] found that the observed length is highly correlated to Halstead's calculated length (how long the program "should" theoretically be). This observed length metric is defined as :

length = N1 + N2

There has been some criticism of the basis of Halstead's metrics and the problems in applying them. Elshoff [ELS78] found E and D were not robust to variations in operator / operand definition. Also because the metrics are language sensitive, Jones [JON86a] reports they cannot be effective for higher level and special purpose programming languages.

3.1.2. TYPES OF METRICS.

Weissman [WEI74] divides complexity of programs into four sections :

1. Program Form
2. Control Flow
3. Data References
4. Control Flow / Data Flow Interaction

Zolnowski and Simons [ZOL80] give a similar division for a number of COBOL program characteristics which are to be measured. They exclude Weissman's Program Form category but include an Instruction Mix category for raw counts of language tokens. Harrison [HAR84] does the same type of thing when suggesting groups of complexity metrics. Some metrics appear to measure properties which transcend these categories. Which category a particular metric is placed in is sometimes arbitrary. In this section an attempt is made to categorised measures from Section 3.1.1 and others which have been proposed into Weissman's groupings plus an Instruction Mix group.

3.1.2.1. INSTRUCTION MIX METRICS.

These measures are raw counts of numbers, types and specific attributes of source instructions. Some program size measures are also included in this category. Instruction mix (IM) metrics are language dependent but can be combined to form more language independent measures (e.g. V(G)). Groups of IM metrics often form a basis for more complex metrics. This category of measures includes LOC (Section 3.1.1.1.), McCabe's n and e (Section 3.1.1.2.), and Halstead's n1 and N1 (Section 3.1.1.3.).

Clearly there are many possible counts of information which can be calculated from source code. Information from Zolnowski and Simmons's paper [ZOL80] indicates that the percentage of IF statements and the total numbers of verbs, outer IFs and breaks in CF in a program may be useful in determining complexity. Admittedly this is based on a very small sample of only 13 COBOL programs, but almost all programs which were rated above average complexity (measured using an index derived from 44 discriminating characteristics) had a greater than average appearance of the above features. McTap [MCT80] uses the average number of verbs per module and the ratios of IF and MOVE statements when forming his COBOL complexity metric. Crawford, McIntosh and Pregibon [CRA85] present data from an analysis of a large group of C programs to suggest DSL (delivered source lines, macros and INCLUDEs expanded), NCSL (noncommentary source lines), and FND (function definitions) as parameters for code fault and maintenance effort formulae. Crawford et al also show that DSL, NCSL and SC (the number of end-statement tokens, semicolons or periods depending on the language) are all highly correlated measures of program size.

Branching measures, other than those mentioned above, which have been used to form general metrics include the number of GOTOs [BER85b] and the ratio of PERFORM verbs to all verbs (for COBOL programs) [MCT80].

3.1.2.2. PROGRAM FORM METRICS.

Weissman [WEI74] defines program form (PF) in terms of presence of well-placed and meaningful comments, declaration placement, indenting and layout of the program listing and choice and use of variable names. PF properties are generally difficult to quantify and interpretation of measurements tends to be rather subjective. These types of metrics are sometimes referred to as readability or clarity measures. Crawford et al [CRA85] identify DOC (ratio of noncommentary to total source lines, NCSL / DSL) as a reasonable measure of commenting. Berry and Meekings [BER85b] use the percentages of comment lines, blank lines, and indentation spaces to all characters; along with the average numbers of non-blank characters per line, blank characters per line, length of modules and identifier length when calculating their style metric. Zolnowski and Simmons [ZOL80] found that programs described as complex almost always have a less than average percentage of comment lines.

Many PF measures can also be regarded as IM metrics. Zolnowski and Simmons [ZOL81] report that many survey respondents stated that a major dilemma in current metrics is that a particular factor is known to contribute to complexity but the exact degree of its effect is very difficult to ascertain. This is particularly true for PF metrics.

3.1.2.3. CONTROL FLOW METRICS.

The number of decisions made in a program, and their interrelationships, are generally considered significant factors in overall complexity [ZOL81]. CF metrics are often calculated using IM metrics which measure size and branching properties.

Many of the CF metrics can be derived from the CF graph of a program (mentioned in Section 3.1.1.2.). Schneider [SCH79] suggests that a CF graph's adjacency matrix is useful for determining test coverage and that reachability (defined as the average number of ways any node can be reached) is related to complexity.

In the CF metric category, V(G) is the most well known, but many others have been proposed. These include measures of knots [WOO79] and nesting level [HAR81a] [HAR81b] [PIW82] [PRA84]. The knot measure K is an attempt to combine measures of branching density and statement ordering to get a measure of the interwoveness of source code. K is the number of unavoidable intersections of arcs which record transfers of control among the list of source statements. Baker and Zweben [BAK80] identify that a problem with K is that arbitrary amounts of structured transfers of control (DO and WHILE loops) have the same complexity as straight line code.

Criticisms of V(G) include the contention that it does not adequately measure the interaction of branch statements. Piwowarski [PIW82] proposes a modification to a version of the cyclomatic number to cater for nesting. His metric is

$$N = \text{CYC-MIN} + \text{sum-over-i-of}(\ P(i)\ )$$

where P(i) is the nesting depth of the i-th predicate

Piwowarski's N and Harrison and Magel's metrics are defined on modified CF graphs. In these graphs selection and iteration branches are taken to be part of the previous statement. These newly formed predicate statements branch forward in the case of selections and backward in the case of iterations (at least for REPEAT-UNTIL loops). The nesting depth of a statement is defined as the number of predicate (branch statement) scopes which overlap or contain the statement.

Harrison and Magel [HAR81a] [HAR81b] describe two nesting measures which require the use of the modified program CF graph in their calculation. The first is called the Scope Number. The Scope Number is defined as the sum of the adjusted complexities of all nodes (statements) in the modified CF graph. The adjusted complexity of a non-predicate node is just the node's raw complexity (which was supposed to reflect the complexity of the individual statement's contents, a Halstead metric applied to one statement was originally suggested). To determine the adjusted complexity of a predicate node the greatest lower bound (glb) of the predicate must be found. The glb measures the extent of the predicate and all paths from the predicate must, perhaps after iteration, contain the glb. An example of a glb for an IF branch in some language could be FI, for instance. The adjusted predicate complexity is the sum of the raw complexities of all nodes on paths between the predicate and its glb (excluding the predicate and glb), plus the raw complexity of the predicate. For comparison with V(G) and ease of calculation, Harrison and Magel set all raw complexities of nodes to one. Using this definition of raw complexity, the Scope Ratio is defined as the quotient of the number of nodes divided by the Scope Number. The Scope Ratio ranges between one and zero. As the magnitude of the Scope Ratio decreases, the complexity of the program increases.

Prather [PRA84] describes an alternative nesting strategy which recursively assigns complexity numbers to CF structures (selections, iterations, sequences) and simple statements. Complexities of the generalised structures are defined as :

```
      comp(simple statement)  =  1
      comp(sequence(S1 , S2))  =  comp(S1) + comp(S2)
      comp(IF P THEN S1 ELSE S2)  =
                                 Wgt x max(comp(S1),comp(S2))
      comp(WHILE P DO S1)  =  Wgt x comp(S1)
             where comp stands for complexity,
                S1 and S2 are statements or structures,
                P is a condition
             and Wgt = 2 to the power of number of
              simple boolean conditions in expression P
```

An overall nesting metric is defined as the sum of the complexities of the outermost CF structures and statements. For unstructured programs, Prather defines the complexity of a GOTO statement in terms of the complexity of the structures and statements between the GOTO and its label. Calculating GOTO complexity is difficult and involves determining maximal 'spanned' subflowcharts. The complexity given to GOTOs by Prather's scheme seems to be more related to chastising GOTO use rather than measuring complexity.

To compare the nesting metrics above, consider the addition of a selection (without an ELSE part) or iteration control structure over the top of a section of existing code with S statements and P predicates.

This effect happens frequently in maintenance (see program PURGE-A2 in Appendix 1 and Section 2.5.3.2.). The new control structure's conditional expression will be assumed to be only one simple boolean condition. Piwowarski's N is increased by P + 1. Harrison and Magel's Scope Number (assuming the raw complexity of all nodes is one) is increased by S + 1 (for the selection) or S + 2 (for the iteration). If the existing code has Prather complexity C then the new code has Prather complexity 2 x C. When comparing the metrics, Prather's metric has increase well out of proportion to the others. As most metrics are usually measured against some program size metric to aid interpretation (for instance, consider the Scope Ratio), this means that Prather's metric will probably be difficult to interpret against conventional size measures.

In a program, selections indicate one condition evaluation and a possible execution of a group of statements, whereas iterations indicate an unknown number of evaluations and executions of a condition and statement group respectively. This suggests that iterations will usually contribute more to complexity and understanding difficulty than selections. This contention is also put forward by Waters [WAT79]. The

Scope Number and Ratio are the only nesting metrics, of the ones given above, which attempt to account for this difference and even then the variation in the complexity rating between a selection and an iteration is insignificant. All the CF nesting metrics are time consuming to calculate for large programs, particularly if programs are very unstructured.

3.1.2.4. DATA REFERENCE METRICS.

Metrics in this category quantify the data aspect of a program. They attempt to measure the numbers and types of variables as well as the scope and clustering of variable references. Some measures of the use of data structures and pointers could also fall into this group.

Data reference measures which have been proposed include Halstead's n2 and N2 (Section 3.1.1.3.), the average number of variables per module [MCT80], the percentage of symbolic constants [BER85b], and (in COBOL) the ratio of the number of unique CALLs (to other programs) to the total number of CALLs [ZOL80]. Elshoff [ELS84] indicates that data difficulty (defined as N2 / n2) is a reasonable parameter of program construction effort and, possibly understanding effort.

Using Halstead's view of labels and GOTO statements (Section 3.1.1.3.), labels can be regarded as operands and their measures described in this category. Redish and Smyth [RED86] use the numbers of label references and labels defined but not referenced, and the total number of labels (as well as many other measures) in their style analysis tools (Section 3.1.3 and 3.3).

A group of data use measures derived from the "span" of identifiers has been suggested. A span is the number of statements between two textual references to the same identifier (declaration and comment references are ignored). The ordering for statements is taken directly off the source code listing for span measurement. Feuer and Fowlkes [FEU79] define two metrics :

$$\text{Mean-Variable-Span} = \frac{\text{Last-reference} - \text{First}}{\text{Number-of-references} - 1}$$

$$\text{Program-Span} = \frac{\text{Sum-of(Mean-Variable-Spans)}}{\text{Number-of-variables}}$$

Elshoff [ELS76] argues that the number of identifiers in a program which have mean span greater than, say, 100 statements is an indication of the difficulty a maintainer will have in understanding identifier use in the program.

Most attempts to measure data references are aided by the inclusion of CF information and, thus, the metrics produced fall also into the next category.

3.1.2.5. CONTROL FLOW / DATA FLOW INTERACTION METRICS.

These metrics measure aspects of both control and data flow in their derivation. Control and data flows are closely related, although data/information flow can exist when no CF exists [HEN81]. The motivation behind these measures is primarily that, alone, the CF graph of a program is not sufficient for complexity measurement [TAI84]. Programs with the same CF graphs can differ significantly in where and how variables are defined and used.

Tai [TAI84] proposes a metric based on data flow possible around conditions in selection and iteration structures in a CF graph. The measure is applicable only to structured programs and is really only an alternative CF metric. It has been included in this category because it attempts to add a data perspective even though only "possible" flows are considered. Tai explains that a conditional statement identifies some amount of data being used and this data must have been defined (initialised) previously in a program. For a restricted CF graph (statement sequences and structural groupings having been replaced by blocks), Tai allocates definitions to blocks so as to maximise the number of "live" definitions at the bottom of the graph. This scheme assigns definitions to iterated blocks and in selections to either the selected or alternative block whichever is smallest in height. For each condition (use), the number of possible positions where the condition's variables could have been defined is determined. The number of definition-use (d-u) tuples for a condition records the number of definition blocks on paths to the condition. Tai's DU metric is the sum of all the d-u tuple totals. Two advantages of this metric are that it appears to be a closer measure of the actual number of program paths than V(G) and it is bounded. If a CF graph G has P selection / iteration constructs, then :

$$P \le DU(G) \le P \times (P + 3) / 2$$

The former advantage applies because DU is influenced more by iterations than selections. This desirable property does not hold for the nesting CF measures considered in Section 3.1.2.3.

Chapin [CHA79] identifies four roles for data variables used in functions. Roles are determined by considering what happens to particular data in a function. Hence, CF information is implicitly applied. Counts of the four types of variables in a function form basic interaction measures. These measures are defined as :

P. The number of variables used for processing (production of output).

M. The number of variables changed, created or modified in value or identity by the function.

C. The number of variables which control aspects of the processing in the function.

T. The number of variables which pass through the function unchanged.

Many combined control/data flow metrics do not attempt to account for complexity on a statement by statement basis. Often the approach is to use module (subprogram, subroutine or procedure) interaction and interconnection information to derive the metrics. This assumes other measures are available to determine complexity within a module. Many metrics of this type can be considered as extended CF measures.

Yau and Collofello [YAU80] suggest that maintainability would be reflected by a measure of a module's resistance to the impact of modifications in other modules in the same program. This interconnection metric is termed the module's logical stability (LS) and it's reciprocal is the potential logical ripple effect (LRE). Ripple effect was discussed previously in Section 2.5.2.2. The magnitude of the LRE for a module is determined by considering how interconnected a particular module is to other parts of a program through local and global data variables. For each variable definition i in module k, a set Z[ki] of modules which would be affected by a change in the definition of i (either because they directly import i or use interface variables which are influenced by i) is calculated. The logical complexity of a change in definition i of module k (LCM[ki]) is then defined as the sum of the complexities of all modules in Z[ki]. Yau and Collofello use McCabe's cyclomatic number V(G) as a measure of an individual module's internal complexity. The LRE of a module k is :

$$\text{LRE}[k] = \frac{\text{Sum-of}(\ \text{LCM}[ki]\ )}{\text{Number-of-variables-i-in-k}}$$

The LRE for the whole program is :

$$\text{LREP} = \frac{\text{Sum-of}(\ \text{LRE}[k]\ )}{\text{Number-of-modules-k-in-program}}$$

In the LREP formula an assumption is made that every module in a program has equal chances of being selected for modification. An alternative strategy could be to assume that some modules are more likely to need changes than others. The modification likelyhood estimator could be related to a module's LOC, V(G) or the previous maintenance history of modules in the program. Support for this latter scheme lies in the discussion at the end of Section 2.5.2.1.

Henry and Kafura [HEN81] present a number of measures based on information flow. The first is a measure of the complexity of a procedure (supposedly a C procedure in some module / program).

Procedural-complexity =

L x square-of(fan-in x fan-out)

where fan-in of a procedure A is the number of local data flows into A plus the number of data structures from which A retrieves data

fan-out of a procedure A is the number of local data flows from A plus the number of data structures which A updates

L is the length of A (LOC)

Henry and Kafura calculated this metric for procedures from the UNIX operating system and summed the complexities of procedures in each module to form a module complexity measure. When this module complexity measure was correlated against the percentages of procedures per module which required changes (i.e. maintenance) the resulting coefficient was 0.94. When this same scheme using a procedural complexity metric equal to the square-of(fan-in x fan-out) was evaluated the correlation coefficient increased to 0.98. Thus, Henry and Kafura suggest that Halstead's length [HAL77] or McCabe's V(G) would be a better measure of procedure length L than LOC. The procedural complexity measurement is supposed to identify procedures with heavy information traffic and possibly inadequate functionality or refinement.

The second metric that these authors define is the number of global information flows through a particular data structure in a module. This may be used to determine refinement within data structures. The basis of the metric lies in identifying how many of a module's procedures read, write or read-write a particular data structure. The data structure metric is defined as :

$$(r \times w) + (w \times r\text{-}w) + (r \times r\text{-}w) + (r\text{-}w \times (r\text{-}w - 1))$$

where r is the number of read procedures
w is the number of write procedures
r-w is the number of read-write procedures

Henry and Kafura also suggest ways of measuring the strength of information connections between any two modules. Extending and combining these module interface measures may lead to a system-wide coupling metric. In development, a design goal is often the minimisation of module coupling. A system metric, as derived above, may be useful in evaluating system design trade-offs.

3.1.3. COMPOSITE MEASURES OF COMPLEXITY.

To measure complexity most authors measure a number of program characteristics and meld them to form a metric. When this is done on a large scale with many information counts or, equivalently, many simple metrics being used, the resulting measure can be considered composite. Often the calculation of a composite metric applies a weighting scheme to the constituent simpler measures. In this subsection, a number of composite metrics are described.

Zolnowski and Simmons [ZOL80] propose an index of complexity derived from 44 program characteristics. The characteristics are mostly counts of program constructs which are believed to be linked with complexity.

Subsets of the characteristics have been described in Sections 3.1.2.1, 3.1.2.2, and 3.1.2.4. The metric is calculated by counting the number of characteristic totals for a program which are above average (or below if lack of a characteristic is thought to increase complexity) to form a score. Averages for characteristic totals are simple means from a "large" group of programs (Zolnowski and Simmons use 13 COBOL programs). All program scores are normalised on a scale of 0 to 10 to create an index of complexity for each program.

There are some problems with this approach to metric production. McTap [MCT80] notes that the method is founded on the statistical law of large numbers. But no attempt was made to insure independence or completeness among characteristics. To calculate this measure for one program, measures for a large group of similar programs (i.e. at least of same language) must be known.

McTap [MCT80] defines a similar metric to the index of complexity, but attempts to resolve some of the latter's problems. The COBOL metric S uses six features of a program (see Sections 3.1.2.1 and 3.1.2.4) and is independent of any group of programs. The metric for a program is a comparison of measurements of source features against a reference vector R which represents local standards for the features. Two other vectors which must be available to produce the measure are weight W and direction D. The weight vector measures the contribution to complexity of each feature. The direction vector records the direction of the contribution (-1 if more of the feature adds to complexity and +1 if it adds to simplicity). Calculation of a program's score S proceeds by identifying which feature measurements are greater than the standard for -1 features, or less than the standard for +1 features, and adding the weight from W to S for each such feature. The metric can be applied to programs or modules of programs. S for a program is not an arithmetic average of module scores because of the particular features used. McTap explains that the features selected for comparison don't need to be the six identified in his paper. But care must be taken in feature selection as measurements must be virtually free of any software size (e.g. LOC) considerations.

An alternative to adding weights directly onto measures (depending on characteristics/features being present or absent) is to multiply a measurement of a characteristic by a weight and add the product to the complexity metric. This latter scheme would appear to be more sensitive to absence or duplication of characteristics used in a metric but potentially it is more accurate. Several measures which use the latter weighting system (sometimes heavily modified) are described below.

Chapin [CHA79] suggests a module or program metric Q based on interaction measures of the module or program. The basic measures used are P, M, C, and T described in Section 3.1.2.5. In the calculation of Q for a module, a total weighted count W is assigned to the module. W is defined as :

$$W = P + 2M + 3C + T/2$$

where P, M, C and T are calculated on variables which are used, modified, occur in conditions or just exist in the module

A value E is also assigned to the module. E is a weighted count of the number of C (control) items which serve in exit tests (inside the current module) for iterations through sub-modules. Such control items may be initialised / modified inside sub-modules, the current module or other modules. A control item which is changed inside the loop (in sub-modules) contributes one to E; outside the loop (in other modules), two to E; and both inside and outside, three to E. Contributions from in the current module are zero. E represents a measure of the complexity of loops controlled by a module and gives a strong control flow flavour to Q. A repetition factor R is defined as :

$$R = \text{square-of}(E / 3)$$

Finally Q is described as :

$$Q = \text{square-root-of}(R \times W)$$

Chapin explains that Q rarely exceeds 11 and leaf modules rarely exceed 5 (low complexity is indicated by low Q number). Q can be calculated for a program by averaging Q for it's modules.

Berry and Meekings [BER85b] have suggested a measure of the "style" of a C program. Their style score is defined as the weighted sum of 11 program characteristics. The 11 characteristics and weights are :

mean module length (15)

mean identifier length (14)

% of comment lines (12)

% of indentation spaces to all chars (12)

% of blank lines (11)

mean nonblank chars per line (9)

mean spaces per line (8)

% of symbolic constants (8)

number of reserved words used (6)

number of #INCLUDE files used (5)

number of GOTOs used (-20)

The characteristics are those identified by Rees [REE82] as essential components of an elegantly written program. Many of the characteristics are program form metrics (Section 3.1.2.2.). Two ranges of values for each characteristic are required for style evaluation. The first range defines a region within which the contribution to the metric is nonzero. The second range is within the first and defines the region in which the contribution is maximum (i.e. in the case of mean module length, 15). Characteristic measurements inside the first region but not the second, contribute a linear proportion of the maximum depending on their distance from the second region. The resulting style metric ranges between 0 and 100. The greater the metric, the better styled the program. Harrison and Cook [HAR86] evaluated the style metric for modules comprising over 35,000 lines of C. Correlation of style against error frequency (computed as NCSL / number-of-errors) for modules gave a coefficient of only -0.052. However, Berry and Meekings stated that the metric was produced to promote discussion and even Harrison and Cook describe it as a promising beginning.

Redish and Smyth [RED86] report on two FORTRAN-77 style analysis tools, AUTOMARK and ASSESS. AUTOMARK is used to mark student programs against a model answer. Overall, AUTOMARK measures the divergence of weighted stylistic factors from a model. There are 30 factors used and they include many of the instruction mix and program form measures of Sections 3.1.2.1 and 3.1.2.2. The marking scheme used is similar to that used by Berry and Meekings above. However, instructors (who produced the model solution) may apply different weightings to the factors to get a measure specific to their assignment.

ASSESS is more general than AUTOMARK as it does not require a model answer. It is used to evaluate 10 factors (a subset of AUTOMARK's factors) of any program on a non-numeric scale ranging from low (bad) to high (good). The 10 factors are :

Comments in the initial block
Statement spacing
Size of comment blocks
Ratio of comments to statements
Spacing
Sum of weighted operator types by nesting depth
Sum of weighted operand types by nesting depth
Average range of a control structure

Average range of a block structure

A measure of parametrisation

ASSESS does not actually produce a metric but in certain cases does provide recommendations for changes to structures and layout of a program.

3.1.4. A DISCUSSION OF COMPLEXITY METRICS.

From the multitude of complexity metrics presented in this section, it seems reasonable to ask which one is most suitable given particular circumstances. A partial answer to this question is that some metrics are only applicable to some kinds of program (e.g. structured) and, unless definitions are translated, some languages.

Generally, a more complete answer to the question posed above is currently unknown and more research is needed in this field. One could argue that some metrics have a stronger theoretical base or are better validated by their proponents against, say, occurrences of errors in software than others. Validation is a particular problem. For large commercial systems, source code is frequently unavailable for analysis because of privacy and copyright constraints [HAR84]. Authors calculate their metrics for particular software they have access to (e.g. a program analysis system [ELS84] or a business DP application [FEU79]). But this software is usually unavailable to other authors for either checking/extending original results or validating new metrics against constant data. A method which has been suggested to get around this situation is the Reduced Form technique [HAR85]. Translating software into reduced form counts most program characteristics and insures almost total security including disabling reproduction of the program. Aliasing of variable names and removal of indentation, embedded spaces, blank lines and comments are included in the technique. Unfortunately, this means some metrics, among them readability measures, cannot be effectively used on reduced form programs (e.g. the Berry-Meekings style metric [BER85b]).

The ultimate success for any metric will not follow from the theories which spawned it or its validation. Success can be interpreted as how widely used and accepted a metric becomes with developers or maintainers. It may be that some metrics are consistent enough to base management decisions on only when applied to particular kinds of applications or DP organisational structures. The easiest way for any use to occur is for a metric or group of metrics to be automatically calculated. The best method of calculation is as a byproduct of compilation [RED86], but an alternative is to use specific static analysis tools [ZOL80]. A design decision that must be made for any software analysis tool which attempts to measure complexity is, what metrics to calculate. For the analysis tool presented in this thesis this decision is examined in Section 4.3.5.

Whichever automatic calculation method is used, attempts at linking metrics to quality measurement can lead to unusual repercussions [ELS84]. For instance, creating a new constant Y equal to an existing constant X and replacing one occurrence of X in a program statement by Y increases Halstead's n2 metric by one. This artificially reduces the difficulty measure D for the program. A way to avoid undesirable measurement consequences is to calculate a large variety of different metrics assessing different aspects of a program. Hopefully, inappropriate adjustments in source code to enhance one metric detract from others, assuming all the metrics produced are regarded equally. This is possibly why Redish and Smyth [RED86] can report reasonable success when assessing quality by mere static means.

Static measurement of program complexity seems to be a still evolving theory [SHE83]. Even if it wasn't, it is clear that there are factors which affect maintenance performance which are not accounted for by these measures. A factor which influences the complexity of a particular maintenance task is the interaction of the kind of maintenance requested with the structure of the existing software. This factor is reflected strongly in our example in Appendix 1 (the modifications are described and discussed in Sections 2.5.3.2 and 2.5.3.3). Some changes are easier to graft onto the existing system's structure than others regardless of how complicated the change request initially appears.

Another suggested performance factor is the interaction of the programmer with the software. Boehm [BOE73] cites that variations in programmer productivity (mainly in development) have been measured as anything from 5:1 to 26:1. However, experiments by Schneider, Sedmeyer and Kearney [SCH81] identify that there are, at least, two distinct programmer populations within a mixed group of DP professionals and Computer Science (CS) students. There are so called experts (characterised by five or more years in DP or having passed seven or more CS courses) and less experienced novices. Although the sample sizes and the number of comprehension questions asked were small, variations in source code comprehension appear to be slight within these two populations. Variations between the populations was large, some novices having up to three and a half times more difficulty understanding large programs than experts. Curtis, Sheppard, Borst and Love [CUR79] found in their experiments that E and V(G) were more highly related to performance with unstructured and uncommented code and performance of less experienced programmers. Weissman [WEI74] states that very novice students (less than a year's experience) make poor subjects for analysing understanding difficulties between different programs because they have more problems with language constructs than algorithms. Curtis et al [CUR79] suggest that experienced programmers conceptualise programs at levels other than operator/operand tokens and individual statements. They are also more likely to use in-house programming standards which are known to ease maintenance tasks [WOO79].

Some desirable qualities, other than performance prediction abilities, have been described for complexity metrics by various researchers. Metrics which are largely language independent and noncoercible (i.e. measure an appropriate underlying program property without influence from, say, the specific coding) are preferred by Feuer and Fowlkes [FEU79]. Hansen [HAN78] includes independence as a favourable criteria when using two or more metrics in conjunction and states that measures should show enhancement for, or at least not penalise, the use of "good" programming practices. Given comments that program complexity is a function of the language [WEI74] and suggestions that development productivity differences of 2:1 and up to 3.5:1 are possible by varying the choice of implementation language [BOE73], attainment of language independence may actually detract from performance prediction in metrics.

3.2. A DOCUMENTATION SUPPORT ENVIRONMENT.

In this section the various kinds of documentation are identified and aspects of documentation support considered. Brief summaries of some suggested documentation tools are given in Appendix 2.

According to Mathis [MAT86], an underlying goal of structured programming, programming methodologies and software engineering is to improve the understandability of programming by humans. The use of high level languages rather than machine code essentially aids human understanding. Hence, a program in a high level language documents an activity for both humans and machines with the emphasis on humans (admittedly for a very small subset of the human population). This view identifies source code as fundamental documentation of an application. An oversimplified description of maintenance could be the activities of retrieving, updating and testing documentation in it's widest sense. Development could be similarly described as the production and testing of documents relevant to an application.

From the above, documentation support is central to any development or maintenance environment. Thus, a general documentation system could be used to integrate activities and software tools within an environment, as all such activities must manipulate documents of some form. This idea has also been promoted, in essence, by Anderson [AND81], Saib [SAI83], Richmond [RIC85] and in most development methodologies.

3.2.1. DOCUMENT GROUPS.

There are man kinds of possible documentation for a computer system. Groupings of documents are often linked to development phases in the software life cycle [AND81] [BRI83] [HOR86].

Anderson [AND81] gives a hierarchical structure to development documents. Six levels of abstraction in development are recognised : System, Subsystem, Program, Module, Procedure and Statement. Design document groupings are the System Overview Specification (SOS), Subsystem Functional Specification (SFS), Subsystem Design Specification (SDS) and Program Design Specification (PDS). Implementation documents for programs, modules, procedures and statements (Program Source Specifications, PSSs) consist largely of source code. Anderson claims that after system installation, classical program implementation documents like flow charts and structure diagrams are redundant as the source code exists and is more accurate. However, graphical documents, like these charts and diagrams, visualise structures present in source code and more easily convey information to maintainers than alphanumerical material (Wagner [EBE80]). Accuracy of graphical documents can be achieved by generating them from source code as and when they are required. There are considerable space savings with this approach, especially if diagrams were to be machine stored for ease of manipulation.

A simpler structure for development documentation is presented by Brice and Connell [BRI83]. Three main groups of documents are described; the System Requirements Definition (SRD), the System Design Document (SDD) and internal design information within the Requirements Specification Package (RSP).

Not all documentation is created for technical personnel (developers, maintainers and operators). End-user documentation in the form of manuals and on-line "help" facilities is a very important area in any application. Tinnrello [TIN84] identifies that, for 4GLs at least, there is considerable merit in end-users writing their own documentation. In this way applications would be documented at the business level. A major problem with end-user documentation in relation to other application documents is that it almost never covers the entire system. In general, users only need to know about what is presented to them in reports and on screens. Users are usually blissfully unaware about the underlying system / program structures and man of the "housekeeping" routines.

Anderson [AND81] suggests that basic end-user documents should be held in the Human Interface descriptions of the SFSs. If an integrated documentation system is to be put in place then user documents must be linked to other application documentation and differences between user business terms and application construction terms resolved.

3.2.2. A DOCUMENTATION SCHEME.

A possible architecture for an integrated documentation system could mirror the Text Data Base referred to by Richmond [RIC85]. The "physical" level of documentation is implementation and operations data which includes source code; the "logical" level, the hierarchy of requirements and specification documents; and the "presentation" level, the user manuals and other user documents (see Figure 3.2). In an extensive application system there would probably be several overlapping "presentation" views and hence documents for users

An Application Documentation Scheme

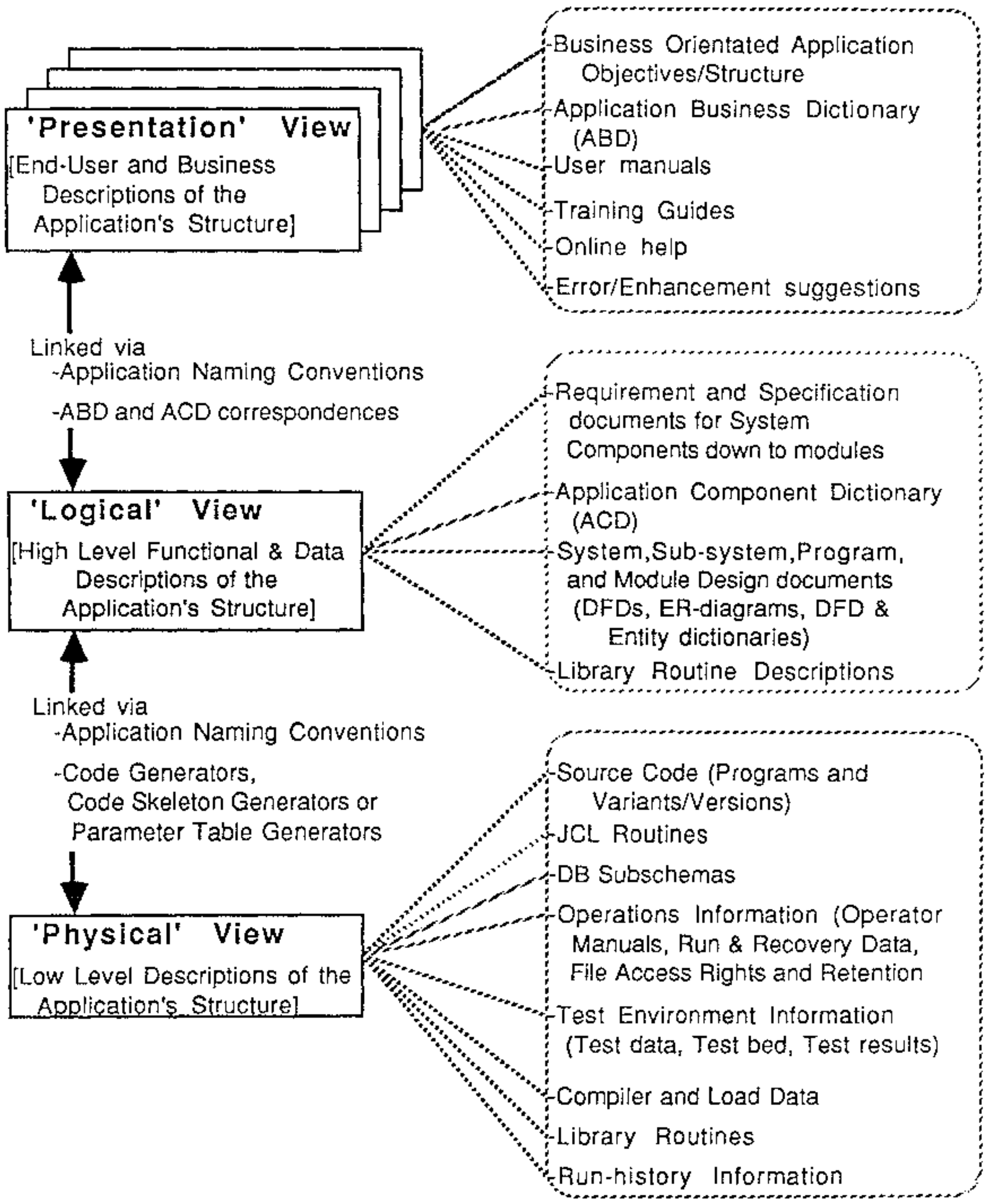

'Presentation' View
[End-User and Business Descriptions of the Application's Structure]
-Business Orientated Application Objectives/Structure
-Application Business Dictionary (ABD)
-User manuals
-Training Guides
-Online help
-Error/Enhancement suggestions
Linked via
-Application Naming Conventions
-ABD and ACD correspondences
'Logical' View
[High Level Functional & Data Descriptions of the Application's Structure]
-Requirement and Specification documents for System Components down to modules
-Application Component Dictionary (ACD)
-System,Sub-system,Program, and Module Design documents (DFDs, ER-diagrams, DFD & Entity dictionaries)
-Library Routine Descriptions
Linked via
-Application Naming Conventions
-Code Generators, Code Skeleton Generators or Parameter Table Generators
'Physical' View
[Low Level Descriptions of the Application's Structure]
-Source Code (Programs and Variants/Versions)
-JCL Routines
-DB Subschemas
-Operations Information (Operator Manuals, Run & Recovery Data, File Access Rights and Retention
-Test Environment Information (Test data, Test bed, Test results)
-Compiler and Load Data
-Library Routines
-Run-history Information

Figure 3.2

(depending on their department and position). Some document control between all three views can be attained simply by using appropriate naming conventions in the same way some compiler/loaders by default use suffixes to identify related source, binary and object files. However, as described in the subsection above, names of functions appearing in a user interface in an application may be unrelated to the names of programs being invoked. The link between "presentation" objects and application subsystems or programs can be enhanced by holding correspondences between terms in the Application Business Dictionary and members in the Component Dictionary. It is more reasonable to link end-user views to the specification view rather than directly to implementation because the specifications define the "logical" application structure and most development methodologies provide for user interaction within requirement / specification production.

Verifiably linking the specification view to the implementation has long been a perceived problem in development. Suggested methods usually involve formalising and formating requirement / specification documents (e.g. problem statements in PSL/PSA [TEI77]). From these documents it may be possible to generate application programs, but it is more likely that skeleton structures for programs would be produced, as in the proposed ADA environment [SAI83]. This would define a link back to the specification, although tools which check which data and INCLUDE files are used in a program against it's specification are advisable. More wide-ranging checks could involve which major code grouping names (i.e. section names in COBOL) and which global data structures are referenced and how they are used. The point is to make sure that major implementation objects are mentioned in the other related documentation. Verification works both ways, specification documents define code skeletons and the source code defines key words which must be referenced in specification documents.

An alternative method for linking specification and implementation is to place descriptive information (specification statements) within source code. This scheme is used in the DADA files of the CASE system [AME79]. Maintainers redocument as they change the source. The specifications can be placed in comments so as not to interfere with compilation and design documents can be extracted using a simple comment collecting tool. A data base documentation scheme, like the one suggested above, has several advantages over this alternative. Embedded specifications do not interfere with or add to the size of object code for compiled programming languages, but this is not the case with large JCL routines which are interpreted. Even when JCL is de-commented automatically before transfer from the documentation system to the executable system it is still easy to make minor patches to executable JCL to keep the application going without updating the documented JCL. Another problem with embedded documentation is that end-user manuals and on-line help remain effectively unlinked to design and implementation documents.

A goal of maintenance is to attempt to keep documentation consistent (See Section 2.6 and [BAS82a]). Figure 3.2 gives an idea of the diversity of documents which may be manipulated in a maintenance task and which an automated document system must support. Specific kinds of documents in a view will depend on the particular methodology used in application development. For instance, if a methodology producing Anderson's documentation groupings (see Section 3.2.1.) was used, SOS, SFS, SDS and PDS information would form the bulk of the "logical" level or view of application documentation. Within a view, documents may be arranged in hierarchies or possibly even "concept" trees (as in MIDOK [EBE80]). Some problems which occur in document support are described in the next subsection.

3.2.3. PROBLEMS WITH AUTOMATED SUPPORT.

From Section 3.2.1, it can be seen that documents used in analysis, design and implementation form a large portion of application documentation. Such development data is often designed to aid the decision processes within development and may need to be extremely extensive to describe the evolving system effectively to developers, managers and various groups of users. If held for maintenance, this data needs to be reduced in volume and tailored toward system understanding and update by maintainers. Another problem with these development documents is that they depend on the methodology used. A prime concern of documentation automation for maintenance is to standardise the access and update of documents. There is a need to identify a documentation standard within a business for all applications if maintenance and documentation is to be centralised.

Brice and Connell [BRI84] report that although automated data dictionaries and program documentors are commercially available, few link to other aspects of an organisation's functions. Important aspects which tend to be overlooked are an application's operation and the whole area of end-user involvement. A possible reason for omitting this data is that it needs to be accessed and sometimes updated by operators and users and much of it must be available when the application is running. But maintainers must also have access to this information. Most maintenance is carried out off-line, in the sense that it is separate from the operational application until re-installation. This identifies the problem of where to hold user and operations documentation and, if duplicates are required, what processes must exist to maintain consistency. Some users may also require access to the specification view of an application, particularly when producing error reports because they must know what functions the system was supposed to provide.

Although problems obviously exist in supplying access to documentation for the range of maintainers, managers, operators and users, this has not stopped centralised documentation systems being suggested. General editing and production facilities as well as comprehensive searching capabilities, like the proposed FORTUNE system [RIC85], would be required by such a system. The maintenance history of an application could be stored in the form of deltas of documents (see the

SAMOS tool, QUODOS [EBE80] and SCCS [ALL84]) within the documentation system. The complexity of application documentation suggests a type of expert system to manage searches and updates. This is the approach taken in the ME2 system [COL85a] and, for development documents only, the IEW/WS system [JON86b].

Some kind of central DBMS has been seen as a method of integrating many development tools ([HOR79] [HOR86] [RID81] [SAI83]). What has been indirectly suggested in this section is that this controller of development documentation and tools be extended to cater for maintenance activities and data. In the next section software maintenance tools will be discussed and categorised according to the documentation they are associated with.

3.3. MAINTENANCE SUPPORT TOOLS.

In this section, maintenance tools are categorised in a number of ways in order to discover what areas, tasks and documents each tool supports. Particular tools are briefly summarised in Appendix 2.

3.3.1. CLASSIFICATION ACCORDING TO ACTIVITY.

When examining software tools it is worth considering the general areas and activities which they support. Bell [BEL84] identifies three components in the maintenance environment : the programmers' environment, the managers' environment, and the institutional environment. Institutional aspects include the image of maintenance, training of personnel and career paths for maintainers. Many of these factors were identified in the problems of maintenance (Section 2.3.1.). In general, the institutional environment is not directly impacted by tools.

Management embraces planning, staffing, controlling, directing and organising activities across the whole application life cycle. Because it is affected by general management tasks like project reviews and personnel control, a maintenance management environment cannot be considered in isolation from the project or data processing management environment. Tools to exclusively aid managers include those that estimate effort, software "health", and resource requirements usually based on project history data and sets of software metrics [BEL84]. Some tools give general support to maintenance and transcend programmer and manager environments. Documentation systems such as MIDOK [EBE80] and SODOS [HOR86] are prime examples.

The managers' environment above is not to be confused with software management and control. Software management (including configuration control) is primarily used and directed by programmers in development and maintenance and is a component of the programmers' environment.

Bell [BEL84] states that a maintenance workbench forms the basis of the programmers' environment. Seven types of tools which should be present in this workbench are recognised : retrofitters (recoders), restructurers, static code analysers, interactive debuggers, test-data generators, automated documentors and specialist editors. Most tools which have been produced to aid specific maintenance tasks could be classified into one or more of the categories in the workbench. The proposed ME2 environment [COL85a] is an example of an intelligent maintenance workbench.

The tool categories of retrofitters and restructurers identify aspects of perfective maintenance which can be completely automated. These software aids have been described by Richardson and Hodil [RIC84] as code purifiers. Specific examples include the recoding portion of SUPERSTRUCTURE [RIC84] and Peat Marwick's Structured Retrofit [CAN86b]. Such tools do not just support maintenance, they actually carry out adjustments for compliance with some standard specification. Thus, none of the maintenance actions of Section 2.4.2.1. appear to be directly aided by these tools. Compilers, link loaders, code auditors and source formaters are similar tools. Boehm [BOE76] describes some of these software aids as testing and reliability tools. This whole group appears to be characterised by minimal interaction between maintainers and the particular tool and consequently the tool-user interface is virtually non-existent apart from small reporting facilities.

Boehm [BOE76] gives three main functions into which tools useful for maintenance could be categorised. These are understanding, modifying and testing/reliability. Boehm, using a wide view of tools which includes techniques and standards, maintains that testing and reliability tools are almost the same as those used in development. Modifying also has similar aids to programming in development. The understanding function, Bell's automated documentors and the maintenance redocumentation task (Section 2.4.2.1.) identify facets of the overall application documentation system. A possible structure for such a system has been discussed in Section 3.2.

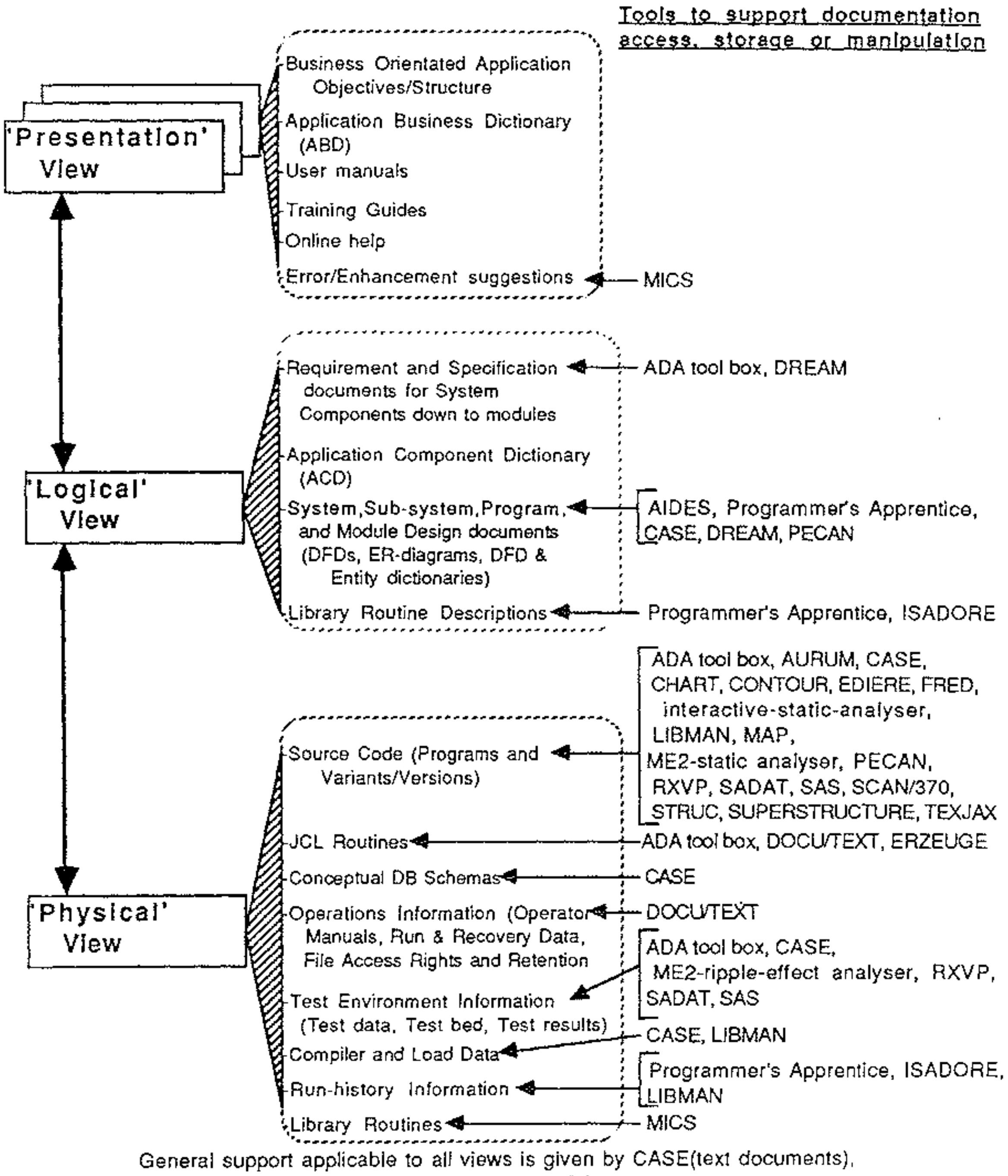
The understanding function is peculiar to, and very important in, maintenance (see Section 2.5.1.). As mentioned in the previous section, almost all documentation other than source code exists to aid someone's understanding. Although most documentation has a number of competing types of users, source code is different in that it is usually very formal and restrictive, largely to aid compiling. General document managing systems like MIDOK or SAMOS help the understanding of source code but this task is worthy of specific tool-based support. Letovsky and Soloway [LET86] identify that a maintainer requires a number of "views" in addition to the localised code view in order to carry out his job successfully. A local examination of source is often misleading. Alternative reporting or visualisation of the structure of source code for understanding is a method used in AURUM, Bell's static analyser, EDIERE (SAMOS), MAP, PECAN, RXVP, and SADAT.

Similar viewing methods for design information are possible in the SID and IEW/WS [JON86b] systems.

3.3.2. CLASSIFICATION ACCORDING TO DOCUMENTATION USED.

Figure 3.3 shows which kinds of documentation various tools summarised in Appendix 2 deal with. From this figure it can be seen that that bulk of software aids which have been reviewed are concerned with source code and that almost all support retrieval or modification of the physical structure of the system rather than the specification or business views. Houghton's paper [HOU83] generally supports the idea that the tools in Appendix 2 represent a reasonable cross-section of those currently available. For instance, of 362 software development tools, Houghton found that 67% catered for either source analysis and testing or software management, control and maintenance. The definition of software maintenance tools used by Houghton appears to be confined to those which directly support software modification through, say, version control. Although many individual programmer environment aids are available, few tools for integrating environments and few related groups of tools exist. Houghton found that only 3% of the tools

Application Documentation and Tools



Tools to support documentation access, storage or manipulation
'Presentation' View
Business Orientated Application Objectives/Structure
Application Business Dictionary (ABD)
User manuals
Training Guides
Online help
Error/Enhancement suggestions
MICS
'Logical' View
Requirement and Specification documents for System Components down to modules
ADA tool box, DREAM
Application Component Dictionary (ACD)
System, Sub-system, Program, and Module Design documents (DFDs, ER-diagrams, DFD & Entity dictionaries)
AIDES, Programmer's Apprentice, CASE, DREAM, PECAN
Library Routine Descriptions
Programmer's Apprentice, ISADORE
'Physical' View
Source Code (Programs and Variants/Versions)
ADA tool box, AURUM, CASE, CHART, CONTOUR, EDIERE, FRED, interactive-static-analyser, LIBMAN, MAP, ME2-static analyser, PECAN, RXVP, SADAT, SAS, SCAN/370, STRUC, SUPERSTRUCTURE, TEXJAX
JCL Routines
ADA tool box, DOCU/TEXT, ERZEUGE
Conceptual DB Schemas
CASE
Operations Information (Operator Manuals, Run & Recovery Data, File Access Rights and Retention
DOCU/TEXT
Test Environment Information (Test data, Test bed, Test results)
ADA tool box, CASE, ME2-ripple-effect analyser, RXVP, SADAT, SAS
Compiler and Load Data
CASE, LIBMAN
Run-history Information
Programmer's Apprentice, ISADORE, LIBMAN
Library Routines
MICS

General support applicable to all views is given by CASE(text documents), FORTUNE, ME2, MIDOK, QUODOS, SID and SODOS

Figure 3.3

surveyed had these features.

The preoccupation of current tools with the "nuts and bolts" of an application rather than higher abstract levels or integration is an indication that few firm ideas yet exist about what aids developers or maintainers should expect in an environment. Tool producers tend to develop "one off" and highly specialised products. Such tools often duplicate aspects of others. For instance : ERZEUGE has many JCL abbreviation features which may be present in an operating system; control flow analysis of FORTRAN programs is carried out independently in EDIERE, RXVP, and SADAT. The most extensive duplication among source code tools appears to be in the area of syntactic analysis. Syntactic analysis of source is an important property of many tools and it is examined in the next section.

An alternative view to the one expressed above is that because the whole software area lacks generally accepted standards for application analysis, design, implementation, testing, documentation and maintenance, many tool developers have done the only thing possible and concentrated on source code aids. As maintenance is often perceived as just hacking the source around to alter the application in some way, then isolated source code tools are likely to appear to give more direct gains in maintainer productivity than, say, a general documentation system which ensures that modified code cannot be installed without redocumentation of program or subsystem specifications.

Many standards did not exist when much of the software being used today was produced. In the late seventies, an average age of an installed system was estimated to be about 5 years [RIC84]. Given present software industry manpower shortages [STA84], it is thought that many systems are now more than 10 years old (10 years has been used as the time span to assess application maintenance costs in the system life cycle in several businesses and institutions [BOE76]).

Old software systems often lack both structure and documentation and source code tools could certainly be useful. A complete rewrite of all existing software is unrealistic. Cost information from Cane [CAN86b] and Richardson and Hodil [RIC84] indicate it is probably cheaper to both retrofit and redocument existing code using tools than rewrite even if the new development uses 4GLs for productivity gains [FOR85]. If, after future maintenance costs are considered, overall costing suggests large scale redevelopment would eventually pay off, there is often no-one to carry out this redevelopment. Vacancies now exist for 50-100,000 programmers in the software industry and the situation is worsening [STA84]. From the above data, the potential market for maintenance tools would appear to be substantially larger than that for development tools in terms of the amount of software involved. However, much of the code being maintained was produced using old and adhoc methods.

Appendix 2 identifies many tools and automated features which would be desirable in application development and maintenance. To support the entire application life cycle through tools what is required is an approach which integrates both development and maintenance methodologies and environments. Most of the tools available address only isolated features of an overall approach and even then there is often a lack of functionality and singularity of purpose. Perhaps the most comprehensive system which appears to fulfill the above approach requirements is the proposed ADA environment and tool box which is built around one specific programming language [SAI83].

Software tools will also be discussed in the next section with reference to syntactic analysis of source code.

3.4. SYNTACTIC ANALYSIS.

3.4.1. STATIC ANALYSIS.

A large number of source code tools summarised in Appendix 2 use some form of static analysis. In this section, possible products of static analysis are described. Firstly, some indication is given of why the most commonly analysed type of software is the program.

An application's source code can be arranged in terms of a hierarchy consisting of a system (a group of subsystems), subsystems (suites of programs), programs (groups of procedures), procedures (also called functions, sections or paragraphs) and statements. The software manipulated by a compiler is usually the source program, which is also, not surprisingly, the largest code grouping supported by most programming languages. Systems and subsystems tend to be described for a computer in interpreted JCL or through program calls embedded within programs. In this latter scheme, program execution control is decentralised and may be undocumented. Syntactic and other static analysis of software is very similar to compiling and hence has usually been orientated toward programs and the analysis one program at a time written using a particular programming language. Partial exceptions are

ERZEUGE of the SAMOS toolset and DOCU/TEXT which scan JCL. But these tools do not offer the extensive analysis and differing views of, say, a subsystem, which tools like AURUM, EDIERE (SAMOS) and MAP provide for programs. Information about file usage and update within a system can only be indirectly compiled from application source with most current tools (by running multiple programs through tools and combining the results) but such information may exist as design or specification data inside the application documentation system (see Section 3.2).

This identifies a particular problem that toolsets and environments need to resolve. Much structural information about an application must be held in JCL and source code. But this may be also present in design or other documents within the application's documentation system. Overlaps of information are inevitable because the documentation system describes the application and one of it's roles is to "flesh out" some structural data with, say, reasons for design decisions. For any overlap, there must be a clearly defined method for insuring and verifying consistency. Overlap can be reduced for low level design documents (e.g. structure charts) by generating them from source when necessary as mentioned in Section 3.2.1.

The functions of tools which are supported by static analysis are categorised below. The tools which supply a particular feature or function are bracketed.

1) Functions which report or enhance the physical layout and structure of source code.

a) Formating of a program. (CONTOUR, EDEIRE(SAMOS), RXVP).

b) Identification of a program's block structure and generation of module/procedure hierarchy charts. (EDEIRE(SAMOS), TEXJAX).

c) Production of a symbol table. (PECAN).

d) Structure based editing and statement syntax checking. (FRED, PECAN).

e) Structural deficiency report production. (SADAT).

2) Functions to specifically aid description or programmer understanding of a program's control flow. This is suggested as a general feature of a Source Analysis System (SAS) [DEU81].

a) Production of structure charts and structured flow graphs. (CHART, Bell's static analyser, MAP, PECAN, TEXJAX).

b) Generation of a program graph, reduced program graph or individual control flow paths. Program graphs (SADAT). Control Path tracing (EDEIRE(SAMOS) , Bell's static analyser, MAP, PECAN, RXVP, SCAN/370).

c) Generation of procedure connection diagrams or equivalently, a module invocation matrix. (AURUM, EDEIRE(SAMOS), PECAN, RXVP).

3) Functions describing data usage and aliasing within a program. This is also suggested as a general feature of a SAS [DEU81].

a) Generation of standard cross references for symbols used in a program. (RXVP).

b) Generation of data definition hierarchy diagrams. (PECAN).

c) Production or reports on data flow between modules and via common storage or external files. Data flow tracing (AURUM, Bell's static analyser, MAP). Common storage usage (RXVP).

4) Advanced functions involving comprehensive analysis of many program features.

a) Generation of program metrics. This may not be a very advanced function depending on the metrics selected for calculation (see Section 3.1.). (TEXJAX).

b) Annotation of source code or redocumentation. (SCAN/370, TEXJAX).

c) Standards checking. This may be the same as the reporting of structural defects above but it is possible to have very complex standards to insure a particular style of programming. (SUPERSTRUCTURE).

d) Program recoding or restructuring. This would usually be in response to the functions of structural defect identification or standards checking. (EDEIRE(SAMOS), SUPERSTRUCTURE).

e) Program instrumentation for dynamic analysis. (EDEIRE (SAMOS), RXVP, SADAT, SAS).

f) Execution monitoring and debugging. (PECAN).

g) Symbolic execution for infeasible path identification. (SADAT).

h) Generation of test cases. (SADAT).

i) Test assistance and results analysis. (RXVP, SAS).

j) Identification of source of errors from symptoms. (diagnostic tools in the ADA environment toolbox [SAI83]).

k) Ripple effect analysis for potential modifications. (proposed Ripple effect analyser in ME2).

From the above it can be seen that some functions are a simple product of syntactic analysis of source code (e.g. cross references, symbol table production and code block identification). Other functions display basic syntactic information in a descriptive and easy to read form (e.g. as program graphs or procedure connection diagrams). These functions actively support program understanding by maintainers. In addition, there are the more complex functions which Fairley [FAI78] describes as requiring sophisticated algorithms (e.g. isolated code determination and program recoding). Such algorithms usually need to use the results of syntactic analysis. Syntactic analysis of a program can thus be identified as a central feature in any system relying on static analysis. In the next subsection this analysis is further discussed.

3.4.2. A SYNTACTIC ANALYSIS TOOL.

In Section 3.3.2, a lack of functionality was identified as a problem with many current tools. Discussions in the previous subsection highlight general syntactic analysis as a specific function of many tools which could be isolated in a specific tool. The general process of analysing syntax is well known and is carried out in all compilers. Like a compiler, an analyser would need to be specific to a programming language or class of languages. The main unknown quantity for an analysis system is "what would it output for use by other tools ?".

Before considering possible outputs, the structure of a static analysis toolset or system will be outlined. The number of tools which use syntactic information suggest that this information be stored in common files or a database. A database containing syntactic data and usually other information forms the basis of Bell's Static Analyser, MAP, RXVP, SADAT, and SAS. The most efficient method of storing source code information is still the source file. The source file (or, more accurately, the formatted source file) remains the residence of instructions which are to be modified, recompiled and tested during a maintenance task. A syntactic database is meerly a device for streamlining access to source data for tools and, possibly, a maintainer directly. Information stored in this database is likely to be short lived. The database would be generated at the beginning of a particular maintenance task and, probably, regenerated if the source was modified.

Automatic update of database information as source is changed is a possibility. It requires the linking of the the syntactic database and source within an editor. This approach is taken in the EDIERE(SAMOS) and PECAN tools. For a prototype syntactic analyser supporting potentially many tools, the editor approach is too complex, although it does have advantages. For example, it allows editing at the structure chart level as in the SCG package of AIDES.

To use a syntactic database as a set of indices to a source file, a pointer mechanism is required. Although there are smaller elements, statements offer the finest view of source code needed for most static analysis. The most natural pointer to a statement in a source file is the line number assuming that the source is formatted with no more than one statement per line. A formatter which insures the above criterion, can also carry out standardisation of the layout of source code. This alone is seen as increasing readability ([LYO81], Geiger [EBE80]) and aiding modification [ELS82].

Figure 3.4. summarises the above discussion with a structure for a generalised static analysis system.

Colofello and Blaylock [COL85a] present three templates for maintenance information which were designed to be the output structures for the ME2 syntactic analyser. The output is supposed to be loaded into a maintenance knowledge base. The templates, with additional information, are listed below. Possible storage techniques for the control flow and data flow templates are also outlined.

Declarations Template.

1. Variable information. Includes a variable's name, defining module, type, range of valid values, initial values (if with definition), aliases and usage (e.g. computational or display). Useful data omitted are definition groupings among variables (data structures or hierarchies) and definition links between data structures and external files.

2. Module information. A module's name, parent or defining module, type (if defined), parameters, local entities (variables, modules, constants

A Possible Structure For A Static Analysis System

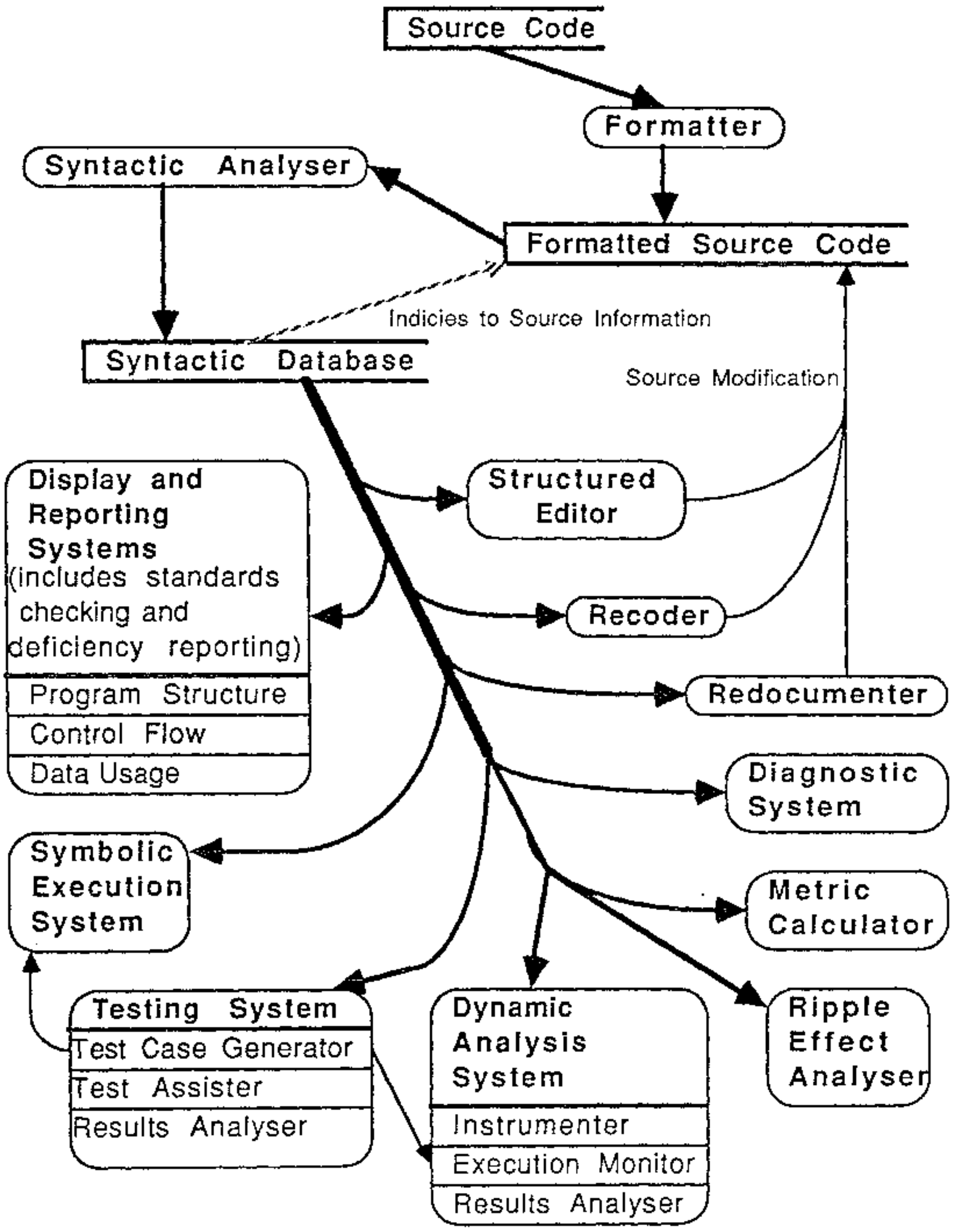


Figure 3.4

etc.), generic instantiation (as provided by ADA), overloading (i.e. system-defined functions or operators modified by the module) and abstractions (as for ADA packages). In some languages (e.g. PL/I), error detection (ON conditions) and error handling would also be candidates for inclusion in module information.

3. Constant information. A constant's name, defining module, value and where it is referenced.

4. Overload operators. New names assigned to operators. Also in module information.

5. User-defined type information. A type's name, defining module, definition and subtypes.

6. Label information. A label's name, location, and visibility data.

7. External file information. A file's name, type, use, access, device and sentinel. As mentioned above related data structures should be given.

Control Flow Template.

1. Identification of blocks and modules. A block is defined as a sequence of consecutive instructions. This assumes that instructions (statements) are

somehow referenced in the source (e.g. by line number). This identification is really not part of control flow (CF). It defines the layout of the program and, for modules, this information could be included in the declarations template.

2. Identification of inter-block CF.

3. Identification of inter-module CF.

4. Identification of inter-process CF. Exact inter-process CF is determined at run time and is beyond the scope of static analysis.

Complete CF in a program is shown by a set of CF graphs (see Section 3.1.1.2.). This set comprises a graph for each module and an overall graph. Within all these CF graphs, module calls are identified as special non-branching nodes (each referencing a particular CF graph). All inter-block and inter-module CF data is stored in these graphs and hence they are a complete storage medium for CF information.

Data Flow Template.

1. Basic data flow identification. Locations where variable's are used and / or modified.

2. Inter-block data flow. Variables within scope (i.e. reaching) and used (i.e. live) within a block. Use-definition chains for variables and locations of ASSERTs (conditions) which involve particular variables.

3. Inter-module data flow. Parameters and parameter passing within modules and global variables used in a module.

Possible data flows in a program can be generated from melding the CF graphs with statement information about which variables are used and modified. Thus, variable use within statements is a method of storing data flows. Variable use within conditions (ASSERTs) is important for any detailed predicate (branching) analysis in a program. Potentially, such analysis could determine isolated code segments.

The above templates duplicate a certain amount of program data. A more refined list of entities and information to hold in a syntactic data base is :

a) All information in the modified declarations template above.

And in addition :

b) Within each variable entity; the statements referencing it, the conditions using it, the modules over which it has scope.

c) Within each module entity; the line numbers and statements it covers, other modules it calls and parameter passing details.

d) A statement entity containing: statement line numbers, variables referenced and other data usage information, conditions used, containing module and CF data. (CF data is a method of storing the CF graphs statement by statement. A flexible form of this data is a CF-type for the statement and a set of statements branched to next).

e) A condition entity containing; the condition itself (literally), statements using it, and variables used.

Much of this syntactic information is relatively low-level and unsynthesised. This is because the results of general syntax analysis in maintenance are to be used in conjunction with source code as well as for the production of high-level views of, say, program data flow. The relevance of some of the information above will vary depending on the language used in a source program, as will precise definitions of some entities (e.g. modules). However, the collection of most of this data using syntax analysis is applicable to most "conventional" programming languages.

CHAPTER 4. DEVELOPMENT OF A PROTOTYPE SYNTACTIC ANALYSER.

From Section 3.4, a syntactic source code analyser (SSCA) is a necessary part of many maintenance tools and is worth centralising in one tool. The rest of this thesis is devoted to developing a prototype SSCA system which takes source code and creates a database of source information (SSCA DB) for access by tools. The machine available for this development exercise is a VAX 11/750 running the ULTRIX-32 operating system. The production of a SSCA is seen as a method for promoting future research at Massey University into other areas of static analysis.

There are three major decisions which must be made for this development :

1. Which programming language to base the analysis system on ?

2. What to store in the data base ?

3. How to analyse the source code ?

These three questions are answered and discussed in the next three sections.

4.1. CHOICE OF A PROGRAMMING LANGUAGE.

To examine a program in any detail an analysis system must be specific to a programming language or class of programming languages. It is possible for analysis or other tools to have the facility to handle a broad range of languages. An example is the Toolbuild system for LL(1) grammar languages [INC86]. However, for a prototype system, a specific language (and a somewhat simplified one) is desirable. Multi-lingual syntax analysis is left for future research.

Morrissey and Wu [MOR79] state that by far the most widely used programming language is COBOL. Approximately 50% of all programs, suggested to be about 75 billion lines [CAN86b], are written in this business data processing language. The U.S. federal inventory alone contains around a quarter of a million COBOL programs [FIO84]. A complete static analysis system based of some version of COBOL thus may be testable with a wide range of programs, would be relevant to most programmers and could potentially be commercially successful.

There are many versions of COBOL to choose from. COBOL was conceived in May 1959 [PHI74] and developed by the CODASYL group. By 1976, Robinson [ROB76] reports eight

COBOL variants officially defined by the Programming Language Committee of CODASYL and, in addition two official American standards, ANS 68 (USAS X3.23-1968) and ANS 74 (ANSI X3.23-1974). The language is still evolving with new standards being proposed and produced. ANS 74 contains a nucleus and eleven functional modules (some with at least two distinct levels). Most manufacturers who produce compilers, implement the nucleus along with some versions of some modules. Hawkins and Harandi [HAW79] calculate that there are potentially 104,976 official subsets of the language. However, all COBOL variations have much in common and programmers tend to use incompatible portions of language definitions infrequently. For example, Torsun and Al-Jarrah [TOR79] found in a sample of 340 commercial programs that the simplest format of each of the MOVE, IF, GO TO, PERFORM and ADD verbs accounted for 82.5% of Procedure Division (PD) statements.

To obtain a simplified version of COBOL, the approach taken in this thesis was to reduce an advanced proposed ANSI COBOL standard in terms of functionality and usage of COBOL features. The standard chosen was the Draft Proposed Revised X3.23-Sept. 1981 language definition [COB81]. A feature of this standard is the introduction of a large number of end-verb constructs for PD statements. Adherence to these statement constructs may allow syntactic analysis of some programs written in future COBOL standards. The next subsection outlines the reduced language described in Appendix 3.

4.1.1. DEVELOPMENT OF A REDUCED COBOL.

The language definition of September 1981 [COB81] identifies the following functional groups of features in COBOL :

1) The Nucleus
2) The Sequential I-O Module
3) The Relative I-O Module
4) The Indexed I-O Module
5) The Inter-Program Communication Module
6) The Segmentation Module
7) The Sort-Merge Module
8) The Source Text Manipulation Module
9) The Debug Module
10) The Report Writer Module
11) The Communication Module

It was decided to produce a language consisting largely of the first six of these groups. Some reasons for omitting Modules 7 to 11 were :

a) The facilities are used infrequently (e.g. the SORT verb represented 0.04% of all verbs in 340 programs [TOR79]).

b) These features appear to be helpful additions to COBOL rather than an intrinsic part of the language. This is shown by the number of COBOL compilers which omit them or use alternative facilities (e.g. the report generator (COBRG) of Dec system 10 COBOL 6 [DIG69]).

In a non-prototype SSCA, it would be expected that some of these modules would be included. The Sort-Merge module is a clear candidate because its features, like the SORT verb, are still reasonably common (two SORT references in 22 programs [TOR81]), although used sparingly compared to other verbs. It is though that analysis of modules 7 to 11 would not be unlike analysis of earlier modules. Again taking the SORT feature as an example, control flow between the input and output procedures of the SORT statement appears to be unusual but it is not unlike a sequential pair of PERFORM UNTIL statements. SORTs and MERGEs are actually somewhat simpler than PERFORMs because nesting is prohibited.

The Source Text Manipulation module (COPY and REPLACE statements) seem inappropriate to any program analysis system other than in pre-analysis source adjustment. This is because syntactic (and static) analysis is being carried out to reflect structures within the source and properties of the object code. Unexpanded COPY and unconverted REPLACE statements add to confusion for both users and implementors of a SSCA.

Of the six remaining language definition modules, various features were simplified. These features and omission considerations are given in Appendix 3A.

4.1.2. SOME PROPERTIES OF COBOL.

The COBOL language has a few peculiar properties which set it apart from many other high level languages (e.g. PL/I or Pascal). Most such properties reflect original language design decisions which were made over 25 years ago. These features, which will affect the analysis of COBOL and the information obtained from COBOL programs, are examined in this subsection.

An influential property of COBOL is the isolation of data definitions from data manipulation and program control statements. Data definition in a program is sited primarily in the data division but also in the identification and environment divisions, whereas control and data manipulation lies in the PD. This property is both a strength and weakness of COBOL. The main weakness is that localised functions (paragraphs) cannot have local variables as all variables are global [EVA82]. A consequence of this is that paragraphs and other statement groupings cannot have formal parameters. Procedural abstraction is thus restricted in COBOL (though programs may be parameterised and invoked via CALL statements).

In COBOL programs, a large portion of the source lines are data definitions. Often many programs share common definitions through COPY files. The definitions in each COPY file usually relate to a particular application data file. COBOL data isolation has thus supported a type of data abstraction through program definitions being linked external data files and programmers being influenced to think about programs in terms of data definition usage as well as procedures and actions. Thus, COBOL has indirectly affected software environments and techniques. This is most evident in data base technologies where schemas often look very like COBOLs file control and record description entries.

Segregation of data definition from procedures suggests that compilation, analysis or formatting could be carried out separately in the data definition divisions and the PD. Operations, other than simple formatting, would need to be connected via some form of symbol table.

Another COBOL feature is the melding of the functions of labels and modules (isolated groups of statements) in the PD. Labels indicate branch-to positions for use by GOTO instructions. The GOTO is the standard assembler control flow branching statement. Modules are a higher level method of delineating statement groups. In a program, this allows the direct instructing that a statement set should be executed a number of times possibly depending on some condition and then control returned away from the set. The PERFORM verb is COBOL's instruction mechanism and sections, groups of consecutive sections, paragraphs and groups of consecutive paragraphs form the modules. Unfortunately, these modules are out-of-line from their instruction mechanisms within a COBOL program which reduces clarity.

Modules can be used other than directly with GOTO and PERFORM statements. Programmers often use labels (sections/paragraphs) to break up a program's PD into manageable portions. For example, on average 27% of labels are unreferenced in a program [TOR79] which means the corresponding modules must be either indirectly performed (label name doesn't appear in the PERFORM statement, performed through) or dropped into (by execution without return of previous modules, possibly achieved using a GOTO statement) to be executed.

The label/module melding feature can thus be used to create highly confusing programs where particular groups of statements are directly performed, indirectly performed ,gone to and dropped into. Further confusion is possible by multiple program end-points (EXIT PROGRAM or STOP RUN statements) being used in an ad hoc manner. COBOL does not enforce structured programming techniques and a useful product of even a simple static analysis system would be identification of a program's control flow structure.

To allow clearer programming styles, the proposed September 1981 standard [COB81] contains provision for in-line statement blocks within PERFORM statements. Such blocks have always been available in IF and other similar statements and are a limited alternative to modularisation. It remains to be seen whether these new features will be widely implemented and used. On the whole, COBOL programmers appear to have a particular aversion to change. For instance, GOTO statements are more common in programs than PERFORMs [TOR79], especially in heavily executed portions of programs [TOR81].

4.2. DATA BASE CONTENT.

In this subsection, precise details are given of information from COBOL source to be included in the SSCA DB. What is stored is partially dependent on which Data Base Management System (DBMS) is chosen for the database. In order to make this decision, information about what structures exist among COBOL source entities, as well as, some details of what the SSCA DB is likely to be used for, must be compiled.

4.2.1. COBOL ENTITIES AND RELATIONSHIPS.

Section 3.4 presents a basis for the identification of COBOL entities. From the extended declarations template, it is noted that six types of information group instances are relevant to COBOL : variable, module, label, external file, statement, and condition. In the following subsections these information groups are translated into one or more entities relevant to COBOL.

4.2.1.1. LANGUAGE DEFINITION ENTITIES.

The entities described below represent groupings of program information resulting primarily from the language definition in Appendix 3B.

VARIABLE and VARIABLE-88 Entities

In COBOL, variable and file definitions are isolated in the Identification Division (ID), Environment Division (ED) and Data Division (DD) from CF and other DF details in the PD (see Section 4.1.2.). To reflect this property entities containing definition information are separated from entities describing references to data items.

From the format for data-description-entries (Appendix 3B), there are two distinct variations in definitions of COBOL variables. These are the level-88 variables which define conditions on pre-allocated storage and "normal" variables which define reference names to storage which may or may not have been pre-allocated. This suggests two entities VARIABLE and VARIABLE-88 to hold instances of variable definition information.

FILE and FILE-VARIABLE Entities

File control and file-description data are simply stored in instances of the FILE entity. Direct PD references to files are through the file-name or data records (variables). Files in the PD can be handled in a similar manner to variables. Variables can be used or updated when referenced in statements. Files (file-names) are updated, in the sense that data within them changes, by DELETE, REWRITE and WRITE statements and used by CLOSE, OPEN, READ, and START statements. It should be noted that REWRITE and WRITE file references are implicit. Unfortunately, there are file actions other than the use and update above relevant to understanding what happens to a file in a program. These actions are summarised by update of file variables such as the LINAGE-COUNTER (for sequential LINAGE files) and the implicit file pointer. To simplify often implicit references to these items in the PD, an entity FILE-VARIABLE is defined.

IDENTIFIER-LIT Entity

Some data type items used in a program are not explicitly declared. Literals fall into this category as do references to objects external to a program such as subject identifiers of the CALL and CANCEL statements. An entity IDENTIFIER-LIT is defined to hold instances of these items. The reasons for explicitly storing literals are exemplified by considering the value of an occurrence number for a variable A - this same value often appears within conditions of PERFORMs controlling update of A. Literals and identifiers such as implementor-name (ED), block sizes and record sizes (ED or FDs) which are unlikely to be referenced in PD statements in the above manner are considered not form instances of this entity.

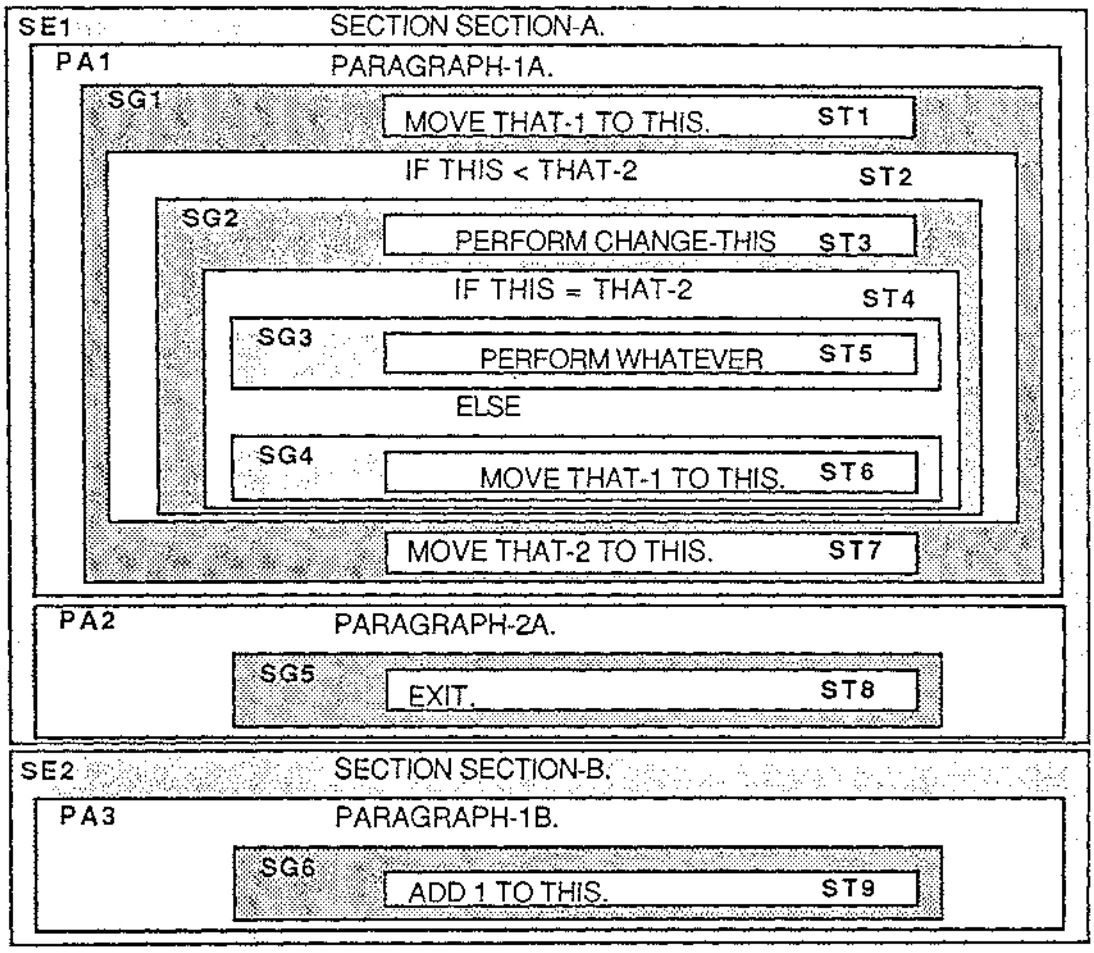
<u>PROGRAM</u>, <u>SECTION</u>, <u>PARAGRAPH</u>, <u>STATEMENT-GROUP</u> <u>and</u> <u>STATEMENT</u> <u>Entities</u>

Labelled modules in COBOL consist of sections and paragraphs (see Section 4.1.2.). Another grouping of statements is the in-line block (mentioned in Section 4.1.2.) which groups nested statements. A block can also have sub-blocks within it. Sentences, defined as a set of consecutive statements ending in a full stop, are another statement grouping system in COBOL. An in-line block reflects potential sequential control flow among it's statements, whereas little useful control flow information is identified by sentences. For this reason it was decided to define a STATEMENT-GROUP entity as a special kind of block. A statement-group is a set of statements at the same level of nesting which would be executed sequentially so long as no GOTO, STOP RUN or EXIT PROGRAM statements were encountered (see figure 4.2.1.). This assumes that execution of a statement A is completed when appropriate statement-groups within A (i.e. embedded statements) have been executed. Unliked a sentence, some statement-groups may contain more than one statement which ends in a full stop. Thus the static structure of the PD of a program can usually be modelled by a hierarchical sequence of instances of SECTION, PARAGRAPH, STATEMENT-GROUPS and STATEMENT entities. This hierarchy must sometimes be modified, however, since sections are optional and statement-groups are composed of statements but may also reside within statements (e.g. an IF statement contains one or more statement-group). Figure 4.2.1. illustrates a possible structure among entity instances. To complete this hierarchy a PROGRAM entity containing only one instance of general program information is also defined.

CONDITION Entity

The CONDITION entity exists to hold conditions which occur in the PD of a program. A particular condition may be used in several statements and, in addition to the direct text of a condition, data about which variable, variable-88, file-variable and identifier-lit instances are used is recorded. Conditions are associated with CF branching and thus are important for any tools performing CF analysis or modification (e.g. Ripple Effect Analysers and Recoders). Conditions need to be stored literally to aid more detailed analysis. For example, conditions "X(10) < 3" and "X(I) > 5" when viewed statically, may or may not refer to tests on the

Examples of Section, Paragraph, Statement-Groups and Statement Instances



Abbreviations		
	ST	Statement
	SG	Statement-Group
	PA	Paragraph
	SE	Section

Figure 4.2.1

same value. Symbolic execution could be used by some later tool to identify under what conditions I was 10.

4.2.1.2. LANGUAGE DEFINITION ATTRIBUTES.

Potentially, there are many attributes for language definition entities. For instance, the complete text of variable definitions and statements could form attributes. However, this information duplicates what is already efficiently stored in the program source code. From Section 3.4.2, the SSCA DB exists to allow easy access to source code data for tools. A major obstacle for maintainer comprehension of source code is delocalisation of information [LET86]. For example, complete local understanding of a COBOL paragraph tells a maintainer almost nothing about the definitions of variables referenced, what happens to these variables elsewhere in the program or how the paragraph could come to be executed. The main way the SSCA DB can support access to source information is to accurately record the relationships between source code items. These items are reflected as entities in the database.

From the discussion above, attributes of entities should largely consist of :

1. Attributes to identify specific instances of an entity (these may form keys for entity

relationships or instance ordering).

2. Attributes to link entities to source code items (i.e. line-numbers).

3. Attributes or relationships to link entities to entities.

Appendix 4A lists the entities, attributes and relationships for all entities in Section 4.2.1.1. Appendix 4A also contains implementation structures for these entities. Implementation aspects are considered in Section 4.2.3.

In Appendix 4A, some language definition entities contain attributes other than those mentioned above. These are used to store small amounts of information which would probably be required by tools or other users of source data. This information includes :

PROGRAM source and object computer-names, segment limit, currency-sign and decimal point.

FILE implementor name, organisation, access mode, block size and measure, record size and variability, whether the file is optional and whether duplicate alternative keys are allowed.

VARIABLE level, usage, section (FILE, WORKING-STORAGE or LINKAGE),occurrence number and picture and whether the occurrence is ascending or descending.

SECTION segment number.

STATEMENT verb and end-verb and two other attributes which identify

a) for OPEN statements whether the open is input, output, I-O or extend; and for PERFORM statements whether the condition evaluation is before or after.

b) for PERFORM and SEARCH statements whether there is a varying item or more that one varying item.

CONDITION text (as discussed in the CONDITION entity of this subsection) and type. The type identifies whether the condition is a normal boolean expression, conditional state (ON SIZE ERROR, ON OVERFLOW, INVALID KEY, AT END or WHEN OTHER), TIMES clause (PERFORM format 2 statements), DEPENDING ON clause (GO TO format 2 statements), or group of WHEN clauses (EVALUATE statements). Corresponding type values are ; " "(empty), STATE, TIMES, DEPEN and EVAL.

4.2.1.3. NAVIGATION AND USAGE ENTITIES.

The entities of the previous subsections omit data about where program divisions and non-PD sections begin and end and what VARIABLEs, FILEs, FILE-VARIABLEs, SECTIONs, PARAGRAPHs, STATEMENT-GROUPs and CONDITIONs are referenced in what STATEMENTs. The SOURCE, CONDITION-USAGE and DATA-USAGE entities are defined below to hold this information. For the latter two entities there are many possible entity structures involving different attributes and arrangements. Reasons behind the chosen structures, which are presented in Appendix 4A, are also examined.

SOURCE Entity

Variables, files, sections, paragraphs and statements offer direct links through line numbers to program source information. To further aid navigation around a program for users of a SSCA DB one instance per source program of a SOURCE entity is defined. This entity holds line numbers for program divisions, sections (excluding PD sections) and the PROGRAM-ID paragraph.

CONDITION-USAGE and DATA-USAGE Entities

The CONDITION instances record data references within conditions. For other portions of PD statements data references must also be stored. The DATA-USAGE entity is defined to hold such references. An entity (CONDITION-USAGE) is also needed to identify use of conditions within statements.

Torsun and Al-Jarrah [TOR79] identify that by far the most commonly used assignment statements involve the MOVE (format 1) and ADD (format 1) verbs. In both these statements, a set A of identifiers and/or literals (VARIABLEs, FILE-VARIABLEs or IDENTIFIER-LITs) is used to update a set B of identifiers (VARIABLEs or FILE-VARIABLEs). When examining data flow in these and other statements, it is common to work backward from an output value. A typical maintenance question is "how did this variable come to be assigned this value ?" (this question is closely related to question 4 of the maintenance enquires in the next subsection). In the context of the above statements, this question can be answered by noting that a particular member of set B is updated by set A. From this discussion, the relationships a DATA-USAGE entity could reasonably be involved in are :

1. A relationship with an updated (or potentially updated) VARIABLE, FILE or FILE-VARIABLE.
2. A relationship with a list of VARIABLEs, FILEs, FILE-VARIABLEs or IDENTIFIER-LITs which are used (possibly for the update in 1).

More than one DATA-USAGE instance can exist for a statement. An update descriptor attribute is also required for DATA-USAGE instances to identify whether the update is a normal one, involves corresponding data structures (e.g. ADD format 3), is an INITIALIZE statement update (in which case the update information resides in the data division), is absent (i.e. set B is empty and A is just used), is optional and set A is empty (e.g. for data-names in a CALL statement), or does not use other data items (set A is empty as for the ACCEPT statement). Corresponding descriptor values are : " "(empty), CORR, INIT, NOUP, OPUP and NOUS.

The most common control statements in COBOL programs are the IF, GOTO (format 1) and PERFORM (format 1) statements [TOR79]. Thus, the IF verb would appear to be a major user of conditions in programs. IF statements are characterised by a condition branching to either two statement-groups or one statement-group and the next statement after the IF. By default, control resumes at this next statement after execution of the IF in any event. Thus, identification of this next statement is really part of the order of statements (i.e. it is related to the STATEMENT entity) rather than the use of a condition (CONDITION-USAGE). Hence relationships for the CONDITION-USAGE entity are :

1. A condition relationship (the CONDITION instance used to determine the branching).

2. A primary branch-to relationship (link to a STATEMENT-GROUP, SECTION or PARAGRAPH).
3. A secondary branch-to relationship (link to a STATEMENT-GROUP, SECTION or PARAGRAPH).

The primary branch-to item is gone to when the condition is true. A condition use descriptor attribute identifies whether the condition is unprefixed, prefixed with an UNTIL (PERFORM formats 2 and 3) or prefixed with a WHEN (SEARCH formats 1 and 2). Values of the condition use descriptor are null, UNTIL or WHEN accordingly. The secondary branch-to item is optional, but if present, identifies a statement-group gone to when the condition is false (in IF statements) and a section or paragraph performed through (i.e. from the primary item through to the secondary item) in PERFORM statements. In GOTO (format 1) statements both the condition and secondary branch-to item are empty. A branch descriptor is required to identify these different branching schemes and it has value null (no secondary branch-to), ELSE, THRU or GOTO.

4.2.1.4. RELATIONSHIPS BETWEEN ENTITIES.

Figures 4.2.2 to 4.2.5 contain diagrams which depict the relationships among entities identified in previous subsections. These figures summarise relationship information from Appendix 4A. Portions of relationships are described as mandatory if an entity instance must form a relationship (e.g. in fv-file, a file-variable must have an associated file) or optional otherwise (e.g. in value, an identifier-lit may be used in the value clause of a number of variables or it may not). If many instances of an entity exist for one instance of a relationship then it is possible to demand an order among the entity instances. For example, the order of statements within a statement-group (st-stgroup) is important in terms of program control flow.

The figures also show multi-member and multi-owner relationships. An example of a multi-member relationship is sections-or-paras (figure 4.2.2.). Through this relationship a program many be identified as composed of a set of sections (sections being identified as sets of paragraphs through pa-section) or a set of paragraphs (if no sections exist). Either one set or the other must be present as sections-or-paras is mandatory for either sections or paragraphs. Programs without PD sections or paragraphs cannot contain PD statements and are prohibited as they are not very interesting. A multi-owner example is the linage-size relationship (figure 4.2.3.). This involves either a variable and many files or an identifier-lit and many files. Multi-relationships could be decomposed into two or more relationships. For example, linage-size combines a one-to-many relationship between the variable and file entities with another one-to-many relationship between the identifier-lit and file entities. It is considered more descriptive for SSCA DB users to identify multi-relationships rather than consider the separate sub-relationships.

Relationships Derived from the Program Structure

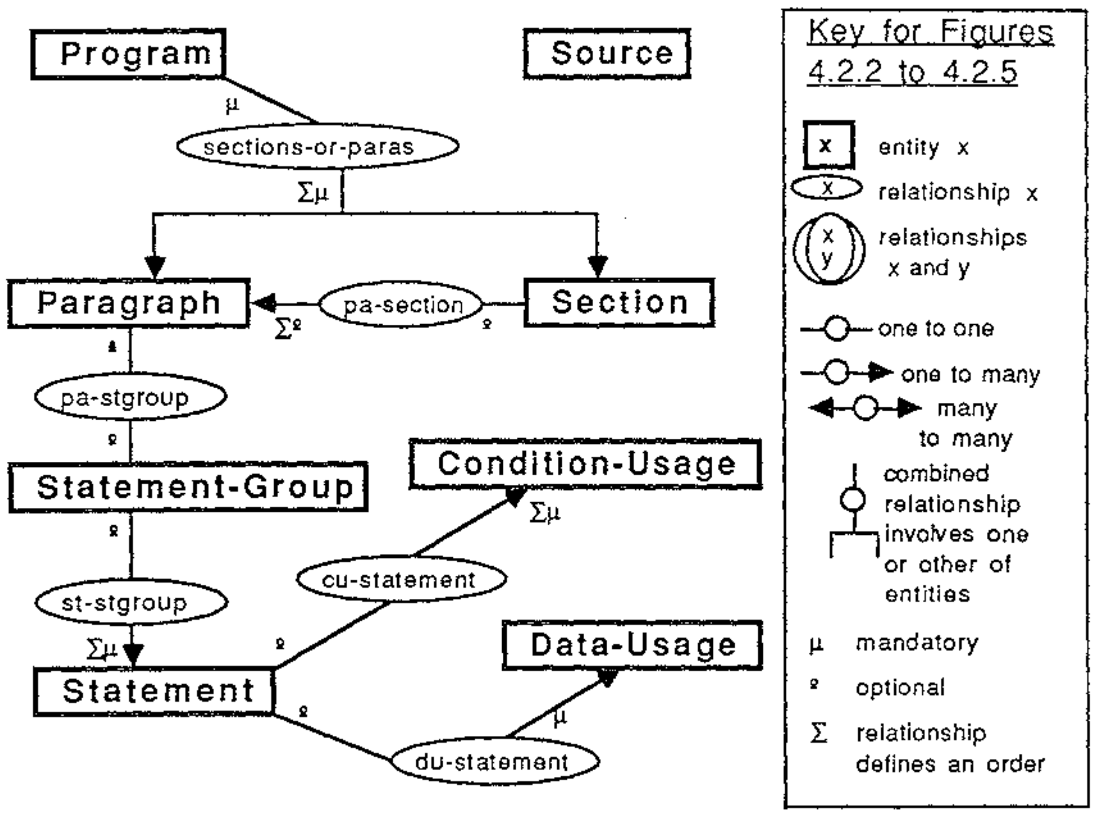


Figure 4.2.2

<u>Relationships Derived from Data Declaration</u>

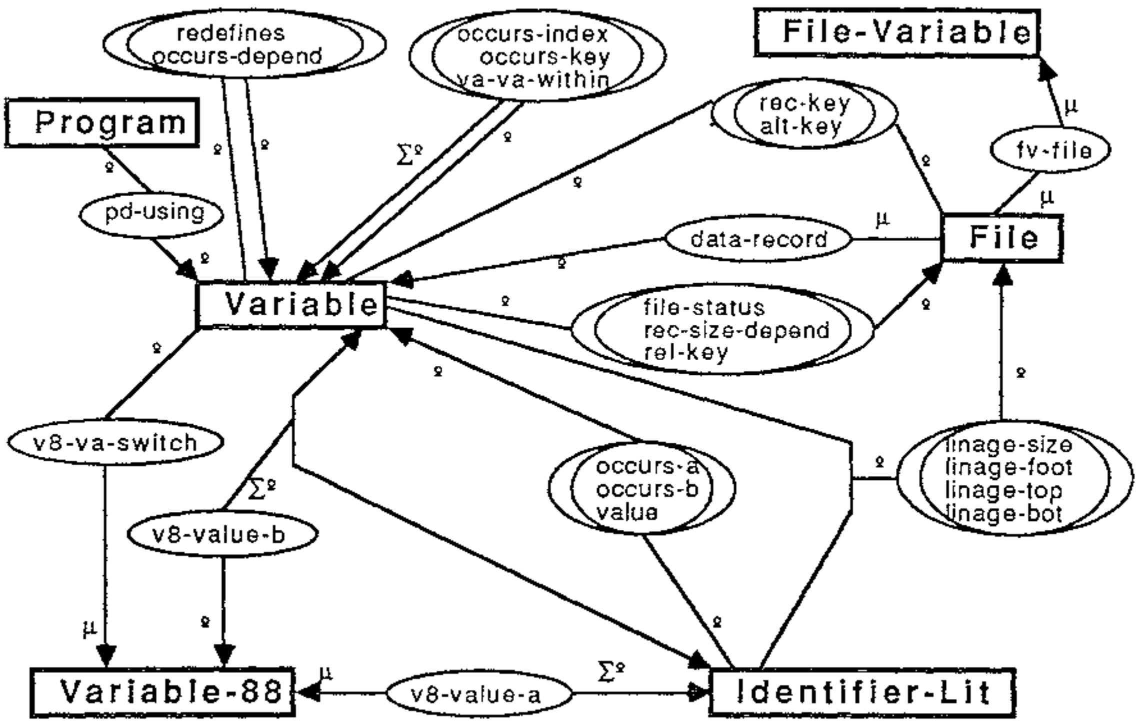


Figure 4.2.3

Relationships Derived from Branching

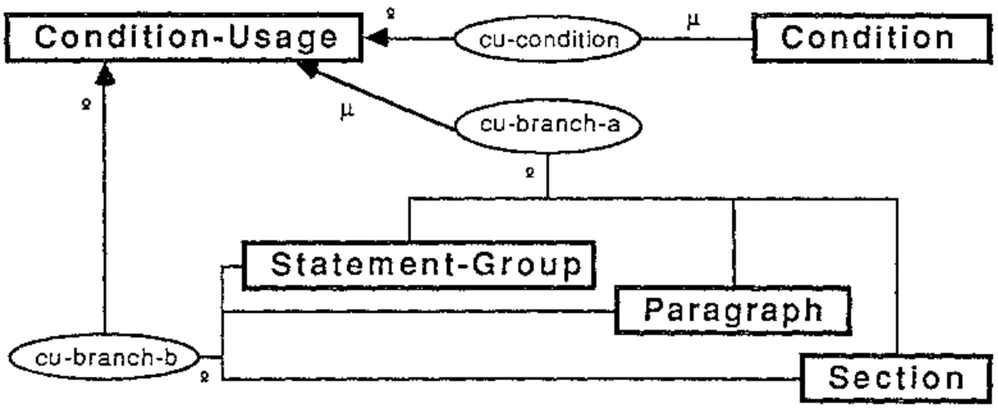


Figure 4.2.4

Relationships Derived from Data Reference

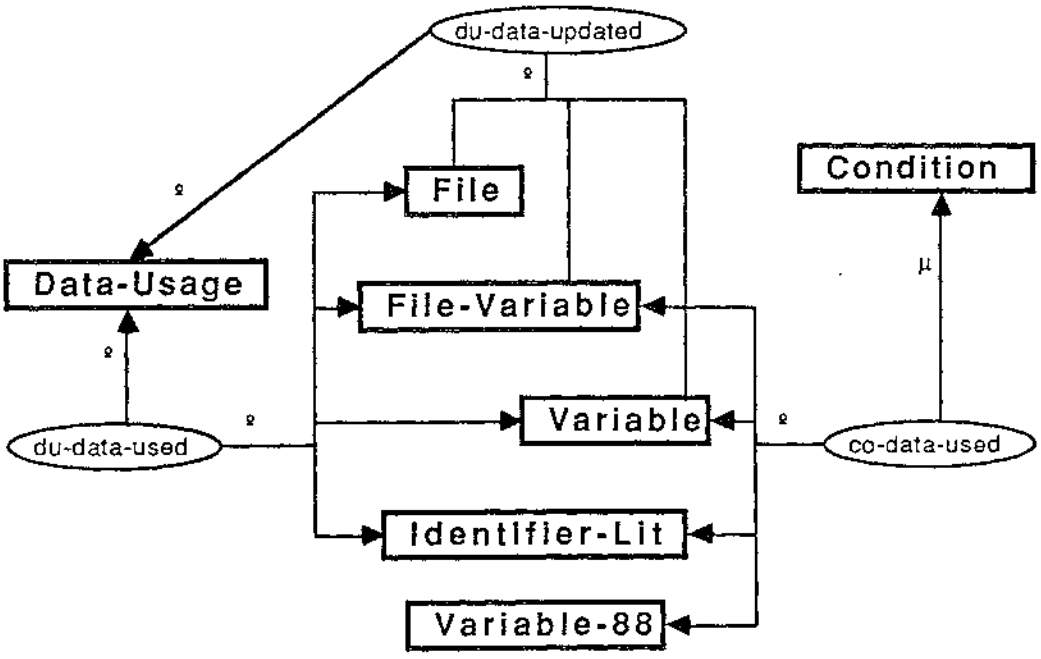


Figure 4.2.5

4.2.2. MAINTENANCE ENQUIRIES FOR A SSCA DB.

The design methodology presented by Teorey and Fry [TEO82] uses both the information structure and usage perspectives (ISP and UP) of database elements to determine a conceptual database design. In terms of this methodology, the design for the entities, attributes and relationships given in Sections 4.2.1.1 to 4.2.1.4 was derived mainly from the structure of COBOL programs (i.e. the ISP). However, some aspects of what the database will be used for have been considered (e.g. in developing a structure for the DATA-USAGE entity). The kinds of modifications a maintainer may be asked to perform are diverse and, accordingly, the kinds of questions which may be asked of a SSCA DB system are reasonably unpredictable. Characterising a UP for SSCA DB information would require an extensive survey of maintainers. Letovsky and Soloway [LET86] have carried out a small scale survey which involved programmers "thinking-aloud" while performing maintenance tasks but the collated results are unavailable at the time of writing.

As an alternative to UP determination, a small number of enquiries which focus on unlocalised features of COBOL source code are given below. These questions identify a necessary set of queries rather than a sufficient set and they will eventually be used to test the implemented SSCA DB system.

QUESTION 1 : What is the static hierarchical structure of the PD ? (i.e. what paragraphs are contained in what sections and how are they ordered ?).

QUESTION 2 : Given a paragraph-name (or section-name), what statements directly PERFORM, indirectly PERFORM or GOTO this paragraph and under what circumstances is it dropped into ? This is a first step to generating a module invocation matrix or set of CF graphs (or equivalently CF paths) for this program. These tasks should be carried out by specific tools which use the SSCA DB (see Section 3.4.2.).

QUESTION 3 : What is the static hierarchical structure of data definitions in this program ? (i.e. identification of data structures and their static associations with files).

QUESTION 4 : Given a variable-name (or file-name or file-variable-name), where is it referenced (used or updated) and what variables, files, file-variables, variable-88s, identifier-lits is it associated with in the PD (dynamically) ? This cross-reference is a first step toward data tracing and program slicing (identifying a reduced PD which demonstrates the PD of a program from the view of what happens to and affects a particular variable or variable set [WEL82]). These tasks are for later tools which will use CF graphs and data definition structures in addition to this cross reference information.

4.2.3. SSCA DATABASE IMPLEMENTATION.

4.2.3.1. A DATABASE MANAGEMENT SYSTEM.

Figure 4.2.6 summarises the types of relationship present among the entities of Appendix 4A. The types of relationships are relevant to determining an appropriate database management system (DBMS) with which to implement the SCCA DB. Two types of relationship directly affect the choice of DBMS architecture. These are the many-to-many (of which there are 7 or 20%) and the combined (of which there are 11 or 31%) relationships.

Combined relationships occur largely because of the choice of entities. If a DATA-ITEM entity was formed from melding variable, identifier-lit, file, file-variable and variable-88 instances and a LABEL entity

Types of Relationships

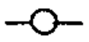
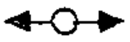
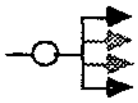
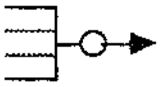
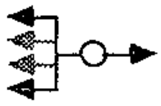
	Relationship	Number
	one to one	3
	one to many	17
	many to many	4
	multi-member one to many	1
	multi-owner one to many	7
	multi-owner multi-member many to many	3
	Total	35

Figure 4.2.6

from melding statement-group, paragraph and section instances then all combined relationships would be single relationships. However, it is believed that the original entities reflect useful elements within a COBOL program and melding entities would cause confusion and would be inefficient to implement if variable record structures are unavailable in the chosen DBMS (e.g. identifier-lits have one attribute whereas variables have seven).

There are three major architectures for DBMSs; hierarchical, network and relational. Many-to-many and combined relationships are difficult to implement in a hierarchical system and the numbers of these relationships discount the use of such a DBMS. Network systems implement relationships as sets. A set instance has one owner and potentially many members and can thus store one-to-one and one-to-many relationships. Combined relationships form multi-member sets. Many-to-many relationships each require the introduction of a "link" entity and two one-to-many sets to be implemented. The number of extra relationships or sets require for a network DMBS is thus calculated (from Figures 4.2.2 to 4.2.5) to be seven. However, appropriate melding of entities, as suggested in the previous paragraph, could reduce the overall number of sets.

In relational DBMSs, relationships are stored by relations (entities) holding foreign keys. Implementation of many-to-many relationships may still involve link relations and combined relationships can be directly represented by stipulating that keys for statement-groups, paragraphs and sections (and similarly variables, files, file-variables, identifier-lits and variable-88s) share a common domain. Many relational systems also offer built-in interactive query facilities which would allow simple verification of database information when developing a prototype SSCA. These features may also allow answering of some of the questions presented in Section 4.2.2. Thus a prototype SSCA DB would appear to be best implemented using a relational DBMS although a network system seems just as suitable for a non-prototype.

Clearly from Section 3.4, a basic requirement for a SSCA DBMS is to provide ready access to database information for tools. Similarly for database creation, the SSCA must be able to send information to the DBMS. The DBMS must thus interface in some manner with programming languages suitable for tool (and possibly analyser) implementation.

Using the above criteria, it was decided to use the INGRES Version 7.10 DBMS to implement the SSCA DB. INGRES is a relational system which contains both an interactive enquiry service (QUEL) and C language interface (EQUEL). C would also seem to be a useful language for implementing the prototype formatter and syntactic analyser as source analysis features are supported through LEX and YACC preprocessors.

4.2.3.2. RELATIONS AND IMPLEMENTATION CONSIDERATIONS.

As the SSCA DB is to be an INGRES database then a number of adjustments and additions must be made to the entities of Appendix 4A. This subsection explains the additional implementation attributes (which consist largely of foreign keys describing relationships) and additional relations. Standard INGRES modifications required for all entity and attribute names was to make them all a maximum of 12 characters, lower case and to change embedded "-" characters to "_".

KEYS AND LINK RELATIONS.

Keys are required for the implementation of relationships involving all entities except program and source. Ordering in a relationship can be achieved by making the keys numeric (or in some other way ordered). A particular order is required among sections (sections-or-para relationship), paragraphs (sections-or-paras and pa-section), statements (st-statement), condition-usages (cu-statement), values of variable-88s (v8-value-a and v8-value-b; although this is only to establish (value-a,value-b) pairings), occurs keys of variables (occurs-key), occurs indices of variables (occurs-index) and subvariables of variables (va-va-within). Problems with multiple orders for variables and identifier-lits are resolved by 3 actions :

1. The creation of link relation v8_value to implement the v8-value-a and v8-value-b relationships.

2. The creation of link relation va_occurs to implement the occurs-index and occurs-key relationships and store the attribute occurs_asc. This new relation is keyed to determine the order of keys or indices.

3. The realisation that a variable needs only to know its parent to store the va-va-within relationship. Thus va-va-within needs only to be one-to-many but is still ordered. This order is the only ordering constraint on variable instances and is thus achievable via keys.

From the above discussion and adapting the approach of using the same domain for keys of at least two groups of relations (see previous subsection), a keying system was derived for relations. A major portion of the keying system is illustrated by the following table.

Relation Sub-Keys

Variable	01	Statement	07
Condition	02	Variable-88	08
Condition-Usage	03	Identifier-Lit	09
Section	04	Data-Usage	10
Paragraph	05	Statement_Group	11
Va-Occurs	06		

It was decide to use the same domain for all keys. A key thus comes in two parts : the relation sub-key which defines the relation pointed at (identified above) and the instance sub-key which defines the particular tuple of the relation. A minimum size for an instance sub-key is several thousand as statements and variables are plentiful in COBOL programs [TOR79]. The combined key fits neatly into a 16 bit integer (i2 INGRES format). The top 4 bits forming the relation sub-key and the bottom 12 bits the instance sub-key. This gives a maximum of 4095 instances for relations. However, a non-prototype SSCA DB system could make use of relation sub-keys 00, 12, 13, 14 and 15 to extend the maximum number of statements and/or variables. There are minor machine dependent implementation problems with retrieval of 16 bit INGRES integers into standard 32 bit C integers when the top-bit (16th) is

1. In this case, the C integers are padded out with 1s in bits 32 to 17. A solution is to add 200000 octal to the integer if it is less than 1.

A one-to-many (or one-to-one) relationship between entities A and B respectively can be implemented by storing a key to an instance of A in every B tuple. With the exceptions of relationships sections-or-paras, pd-using and data-record (see Other Considerations), all one-to-one and one-to-many associations between entities of Appendix 4A (and the link relations) were implemented in this fashion.

As mentioned in Section 4.2.3.1, many-to-many associations can be implemented via link relations. In the actions above, link relations v8_value and va_occurs were identified to store four many-to-many relationships. Two remaining many-to-many relationships are co-data-used and du-data-used. A link relation was created for each of these relationships.

OTHER CONSIDERATIONS.

Several other modifications to entities, attributes and relationships were carried out during identification of INGRES relations. These are described below.

As relations section and paragraph are each ordered on a particular key (the order being as per occurrences in the source code), then relationship sections-or-paras which identifies this same information is redundant. Relationship pd-using is a simple one-to-many relationship between program and variable. Direct implementation would mean storing a program key in each variable instance. However, as there are only a maximum of five pd-using variables [COB81], it is more efficient to hold five variable keys within the one instance of program. In file-description-entries of the DD, data records are implicitly associated with files. A file's later data records implicitly redefine the first data record. It was decided to implement the implicit redefinitions directly as instances of the redefines relationship. This reduces data-record to a one-to-one relationship which is best stored as a variable key in each file instance.

The file entity contains many attributes and is in many relationships. In an effort to reduce the number of fields in the file relation a relation linage_file was created to hold linage information (page size and footing, top and bottom positions) for sequential linage files.

Since many ULTRIX-32 print functions fail for character fields greater than 128 wide, it was decided to use 128 as an absolute maximum attribute size for INGRES relations. This meant breaking il-token of identifier-lit into il_token_a and il_token_b (each of 80 chars) and cond-text of condition into cond_text_a, cond_text_b, cond_text_c and cond_text_d (each of 64 chars).

This completes explanation of the relations and most of the attributes of Appendix 4A. In the next section, mechanisms to build the INGRES relations from a COBOL program are examined and a small set of attributes to hold metric information are identified.

4.3. ANALYSIS OF SOURCE CODE.

The previous section identifies much of information which is required to be derived from a COBOL program. This section is concerned with the methods employed to carry out this derivation and possible extra by-products (i.e. some counts of information for metric production) which could also be generated.

4.3.1. SSCA SUBSYSTEMS AND IMPLEMENTATION CONSIDERATIONS.

Although SSCA system implementation employs compiler production tools, a SSCA is not a compiler. A function which should be foreign to a SSCA (unless linked directly to an editor) is production of detailed messages for source syntax errors and complex error recovery. The reasons for this are to reduce duplication of processing already available in compilers and to discourage attempted analysis of incomplete source code. The source code being analysed is assumed to have already been checked by a compiler. There is a possible problem here for the prototype SSCA being developed in this thesis. No compiler exists for the reduced COBOL language of Appendix 3B. However, a specific COBOL program may use language features present in some other COBOL compiler. Thus, the documenting of errors and error recovery should be rudimentary in the prototype SSCA although syntax checking remains a necessary by-product of syntactic analysis.

From Section 3.4.2, the two main components in a SSCA system are the formatter and the syntactic analyser itself. The formatter produces formatted source code from which the analyser produces the SSCA DB. Both the formatter and analyser need to identify language tokens (such as verbs, identifiers, punctuation etc.) and build up the concepts of statements, paragraphs, sections and divisions. In COBOL, the layout of statements on source code lines is relatively unrestricted. Portions of statements can appear anywhere in columns 12 to 73 (Area B) of lines. Statements can be broken over any number of lines and interspersed with comment and blank lines. A statement token recognising technique could be based on ignoring all comment lines, columns 1 to 11, linefeeds and multiple blanks. However, COBOL division, section and paragraph names must begin in columns 8 to 11 (Area A) and comments and continuation are indicated by characters in the indicator field (column 7). The mixture of fixed and freely laid-out objects in COBOL suggests separate recognition mechanisms for these objects.

A method is to parse source code twice in the SSCA system. Once to identify paragraphs (of the PD), sections and divisions, and again with the formatter. The formatted source code only requires one parse with an analyser so long as information such as section/paragraph names are retained and their line numbers updated. An obvious place to retain this information is the SSCA DB in instances of source, section and paragraph. The first initial parse has been called the section/paragraph extractor (SPEX) subsystem.

Piping is a method of data transfer between executing processes. Piping between processes A and B is similar to executing A which creates an intermediate data file, and then executing B which uses this file and finally disposing of the file. However, piping supports concurrent process execution and the intermediate file exists only as a run-time buffer to be written to by A and read by B. A pipe between A and B is thus a fast and effective means of data transfer between these processes. In terms of pipes, process A and B are considered filters.

Process communication through the piping of default input and output data between processes is simple to achieve and test in ULTRIX. Hence, it was decided to use this technique extensively in the development of all subsystems. Thus, subsystems were decomposed into functions which could act on data one after another to create the desired results.

4.3.2. THE SPEX SUBSYSTEM.

Figure 4.3.1 shows the design of the SPEX subsystem and

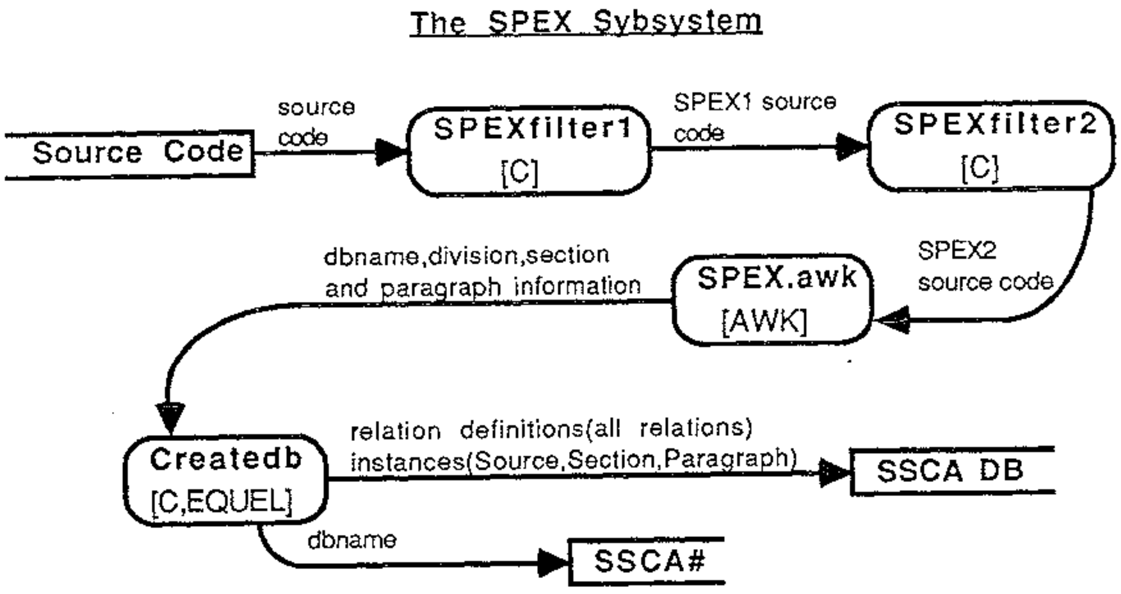


Figure 4.3.1

the tools used in its implementation.

SPEXfilter1 and SPEXfilter2

The first two filters (SPEXfilter1 and SPEXfilter2) identify, adjust for, and warn about simple errors in the source code. All warnings or error messages for components of all subsystems are written to the standard error pipe (stderr). Error checking done in SPEXfilter1 is :

1) Checking that source characters are alphanumeric, space, punctuation, tab or linefeed characters (invalid characters are converted to spaces).

2) Checking for lines longer that 80 characters (longer lines are truncated).

Other processing carried on in SPEXfilter1 involves replacing tab characters by the appropriate number of spaces and making sure that the last line of the source code ends in a linefeed (a linefeed is inserted if not present, this makes processing easier for later filters).

The only check in SPEXfilter2 is to verify that indicator fields of non-empty lines contain valid indicators (" ","*","/" or "-"). Invalid indicators are replaced with spaces. Other actions of SPEXfilter2 are the removal of line-numbers (columns 1-6), the conversion of all lower case alphabetic characters to upper case and the shortening of comments (although the comment lines themselves remain, all characters other than spaces and the indicator field are removed). The shortening of comments merely removes source characters unnecessary in the SPEX subsystem.

These first two filters check for simple errors and tailor the source code for later filters. The basic checks are also needed in the Format subsystem and, for safety, the Analyse subsystem. To speed up SSCA implementation, the first two filters of all three subsystems are all modified versions of one another. The reason for isolating the filter processing above in two filters instead of one is that this is a requirement for FORMfilter1 and FORMfilter2 as well as ANALfilter1 and ANALfilter2 (see next subsections).

SPEX.awk

To set up instances of source, section and paragraph, information is required to be retrieved from the source code. The extraction mechanism is the SPEX.awk filter. This filter is written in the AWK pattern matching language and produces a list of lines containing a linenumber and one character code. The code identifies whether the linenumber pertains to the Data Division, File Section or some other division, section or paragraph. Additional information for the PROGRAM-ID paragraph of the ID includes the program name (after which the SSCA DB will be named). For PD sections and paragraphs, additional information includes the section/paragraph name and for sections, the segment number. AWK is line orientated and a consequence of this is that COBOL program layouts must be restricted to excluding the breaking of division, section and paragraph tokens over several lines. Hence, the following is illegal :

```
000020 DATA
000021      DIVISION.
```

This restriction seems a small price to pay for the use of the AWK language but could be removed in a non-prototype SSCA by recoding the extractor filter in LEX. Such a LEX filter would be much more complicated as all source objects must be pattern matched in some way because the default for unmatched objects is to echo them in the output (the opposite default occurs in AWK). Other pattern matching facilities are GREP and SED. These appear to be line orientated like AWK (and hence similar restrictions could apply) but the actions available after matching has occurred are more limited. Actions of LEX and AWK are written in C-like languages which are very flexible. AWK, even with its restrictions, is thus a convenient language for the implementation of a prototype information extractor.

Createdb

The final processing in the SPEX subsystem is done in Createdb. Createdb must set up instances of source, section and paragraph relations using data supplied from SPEX.awk. The SSCA DB must, of course, be created and the relations above defined. It is convenient to define all SSCA relations here as well. The SSCA DB will be named after the program name. An INGRES restriction means that the program name must be translated so that all alphabetic characters are lower case. Later subsystems will be required to retrieve and update SSCA information. A simple method of passing the database name to these subsystems (to avoid them having to extract it from the source code) is to dump the name in a standard file known about throughout the SSCA system. This standard file was called SSCA#. If software tools are later developed which use SSCA DBs then the SSCA# file could be expanded. This file could contain the names of all SSCA databases which currently exist (at any point in time, at most one could be marked as in a state of creation). Customised access to the INGRES function destroydb would also be required so that the SSCA# file is kept updated.

When carrying out its processing, a number of error conditions can be encountered in Createdb. These include identifying that the COBOL program being examined has no ID, program-id, DD or PD or has other problems such as divisions being duplicated or incorrectly ordered. Database errors such as an already existing database of the same name are also possible. All errors found cause error messages to be sent and the destruction of any database which has been created.

4.3.3. THE FORMAT SUBSYSTEM.

Formatting is a means-to-an-end in terms of a prototype SSCA system and the main emphasis is to insure that there is, at most, one COBOL statement per line (see Section 3.4). Enhancing the layout of a program is a secondary consideration. Program layout is constrained in COBOL, especially by out-of-line performed paragraphs. There is often no way a program can be formatted to the complete satisfaction of a programmer used to, say, a block structured language. Formatting without major restructuring of a COBOL program is thus largely cosmetic. The format subsystem is illustrated in figure 4.3.2.

The Format Sybsystem

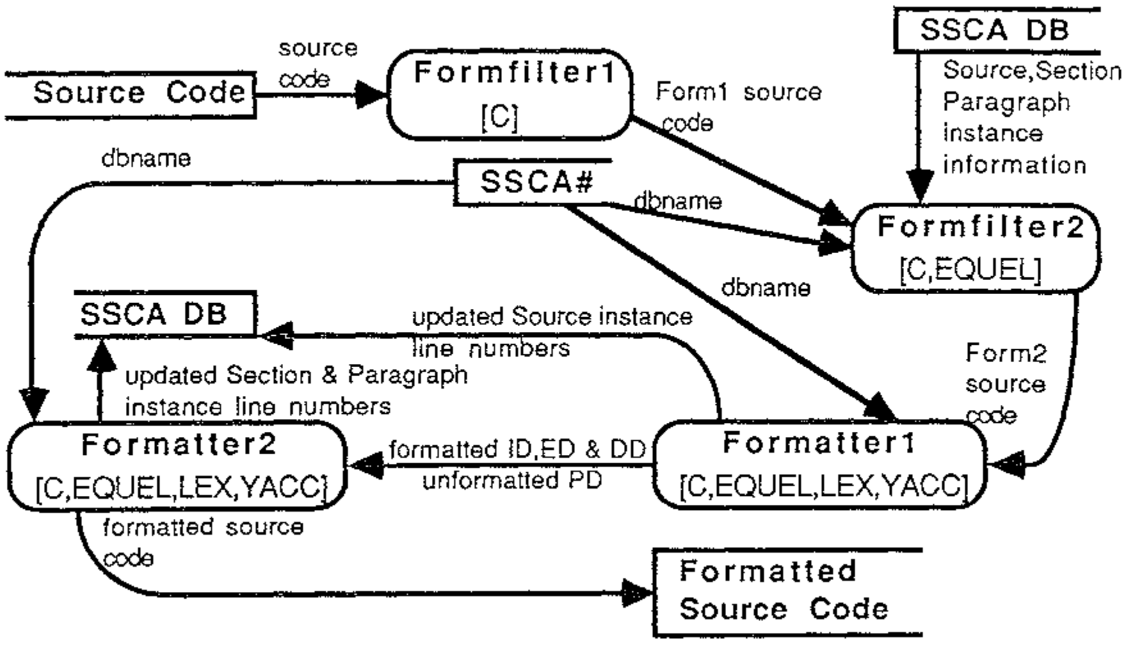

Source Code
source code
Formfilter1 [C]
Form1 source code
SSCA DB
Source,Section Paragraph instance information
dbname
SSCA#
dbname
Formfilter2 [C,EQUEL]
dbname
SSCA DB
updated Source instance line numbers
updated Section & Paragraph instance line numbers
Form2 source code
Formatter2 [C,EQUEL,LEX,YACC]
formatted ID,ED & DD unformatted PD
Formatter1 [C,EQUEL,LEX,YACC]
formatted source code
Formatted Source Code

Figure 4.3.2

FORMfilter1 and FORMfilter2

The first two filters of the format subsystem carry out the same error checking of SPEXfilter1 and SPEXfilter2 as the previous subsystem. In FORMfilter1, in addition to the processing of SPEXfilter1, trailing spaces are removed from all lines which have an even number of quote (") characters. This activity coupled with specific actions carried out when the continuation indicator is encountered in FORMfilter2 allows the reforming of all tokens broken by continuation. The process of reforming continued tokens relies on no comments existing between a line and its continuation line. This forms a restriction on the programs used in this SSCA system. A possible way around this restriction would be to buffer up comments in a temporary file while rebuilding tokens. This solution could have the side-effect of moving comments around among groups of tokens in the program and seems inappropriate in a prototype SSCA anyway.

Like SPEXfilter2, FORMfilter2 removes line numbers and converts lower case characters to upper case. Because strings (non-numeric literals) and comments will appear in the formatted source code, upper casing of these items is excluded. Other actions of FORMfilter2 are the highlighting of comments and division, section and PD paragraph identifiers with tab characters so that they can be more easily recognised by the Formatter filters. In order to identify division and other identifiers, their line numbers are retrieved from the SSCA DB. For lines which are not comments or other highlighted symbol lines, linefeeds are removed as they are unnecessary for formatting.

A major problem in the Format subsystem is the handling of comments which are within statements. It was originally hoped to be able to remove all comments with some filter, format the source, and insert the comments back in. The main advantage of this approach is that instead of just token (word) recognition in the LEX procedures of Formatter1 and Formatter2, complete phrases like ON SIZE ERROR could be identified. This would make the rest of the Formatter filters much simpler. Unfortunately, a by-product of comment removal/insertion is the moving of all comments within a statement to either before or after the newly reformatted statement. The difficulty is that some comments, like the ones in the statement below, become less meaningful with this scheme.

```
    GO TO
          INITIALISE-LOOP
   *         status indicates begin again
          REENTER-LOOP
   *         status indicates resume
          FINISH
   *         status indicates return a result
          DEPENDING ON STATUS
```

Instead of the removal/insertion scheme, comments both within and between statements are accommodated in the LEX procedures of the Formatter filters. When a comment is encountered within a statement, the currently formatted line followed by the comment line is outputted. The next line continues from the last character position in the previous non-comment line. For example, a move statement containing a comment could be formatted to :

```
        MOVE A
       *  comment
               TO B.
```

Comments between statements are merely outputted.

Comments within statements are a problem in COBOL. There are many methods for handling them other than the one above but all methods seem to have drawbacks. What is required is a stricter comment philosophy than "put a comment line in whenever you feel like it". The main reason why strict commenting philosophies are not given or enforced in languages seems to be that comments are ignored by compilers. With the advent of formatters and other source code manipulators the use of internal documentation in programming languages needs formalisation.

Formatter1 and Formatter2

The Formatter filters format statements which appear between comment and division, section and paragraph identifier lines. Formatter1 produces formatted output for the ID, ED and DD portions of a program and passes this, along with raw PD information to Formatter2. Formatter2 prints the already formatted parts of a program (keeping a count of the number of lines) and formats and prints the program's PD. This division of workload is possible because of data definition / procedure segregation in COBOL and was suggested in Section 4.1.2. Both filters update line numbers of Source, Section and Paragraph instances in the SSCA DB.

The two major components of both Formatters are a LEX token recognising procedure (yylex) and a YACC grammar parsing procedure (yyparse). These components are melded together with other procedures such as printline (to produce a numbered formatted line of source code) and yyerror (to report a YACC parsing error, remove the SSCA DB and abort formatting) in a C program. Yylex identifies language tokens (e.g. verbs), program identifiers (e.g.variable names) and highlighted lines and performs suitable actions. As mentioned above, comments are directed to the printline procedure from yylex. Yyparse uses the symbols from yylex to form groups of tokens (phrases) and statements. Using this information, yyparse executes the process of composing formatted source code.

A Summary of the formatting carried out in Formatter1 and Formatter2 is given below.

Division, section and PD paragraph identifiers are formatted to begin in column 8. The PROGRAM-ID and FILE-CONTROL paragraphs are also formatted in this fashion. In addition, a blank line appears before each line containing any of the above identifiers. The PROGRAM-ID paragraph also contains the program name on the same line. Other paragraphs in the ID and ED begin in column 10. Elements of these paragraphs begin on the same line as the paragraph name. Exceptions are the SEGMENT-LIMIT, CURRENCY and DECIMAL-POINT clauses in the ED which, if present, are laid out one per line each beginning in column 12.

In all divisions, if a statement, clause or some other portion of a statement will not fit on one line (normally columns 12 to 73) it is broken at a convenient word and continued (indented by two spaces relative to the starting line of the statement or clause) on the next line. A non-numeric literal token may be longer than the line size available and token continuation using the continuation indicator is necessary. In this case the literal is broken if possible so that the last character on a line is not a space. This is to attempt to avoid problems which may be encountered if the formatted source code is ever edited. Many editors remove trailing spaces from lines (usually by default). Indicated continuation lines have the same indentation as any other statement continuing lines.

Lines which begin file-control, file-description, 77-level-description and 01 level data-description entries start in column 12. Clauses for these entries follow line by line, each indented two further spaces to column 14 (largely as shown in Appendix 3B). Exceptions are the PICTURE (PIC) and VALUE clauses of data-description-entries (01 & 77). These clauses begin in columns 40 and 55 respectively and, if they are the only clauses of an entry, an attempt is made to place the whole entry on one line. A nested data-description-entry begins two spaces in from the start of the previous entry but the same clause layout is maintained. The maximum amount of nesting of data-description-entries which is shown by indenting is 10 entries or 20 indentation spaces. An 88-level entry begins at same column as it's previous entry and has it's VALUE clauses starting at column 40 (if possible the first one is put on the same line as the condition name).

In the PD, section and paragraph identifying lines are formatted to contain no portions of statements. Lines with section naming tokens may also contain a segment number. The PD token itself is on the same line as the start of any USING clause.

Unnested PD statements begin in column 12. All end-verb tokens are formatted onto new lines and indented the same amount as the corresponding verb (a period may also be present on this line). Inside statements formatting is carried out mostly on a clause basis. That is, particular clauses cause formatting in a different manner than the standard method of grouping tokens on a line until it is full or the last statement token is encountered. Many statements do not have any formatting clauses (e.g. ACCEPT and both formats of the GOTO verb). There are general formatting clauses and several exceptions. A general formatting clause, when encountered, causes the clause token and subsequent non-verb tokens to be placed on a new line (indented two spaces relative to the start of the current statement) and following embedded statements to be indented another two spaces. For example :

```
ADD A TO B
  ON SIZE ERROR
    MOVE "X" TO STATUS
END-ADD
```

Verbs and clause tokens of the general type are listed below. There can be a number of such clauses in any statement (e.g. SEARCH can have WHEN and WHEN OTHER clauses) and the embedded statement group may appear empty (e.g. EVALUATE or SEARCH with a number of WHENs but only one statement group following).

Verb	Formatting Clause Tokens
ADD(formats 1 & 3), COMPUTE, DIVIDE(format 1), MULTIPLY (format 1), SUBTRACT (formats 1 & 3)	ON SIZE ERROR
ADD(format 2), DIVIDE (formats 2 & 3), MULTIPLY (format 1), SUBTRACT(format 2)	GIVING, ON SIZE ERROR
CALL, STRING, UNSTRING	ON OVERFLOW
DELETE, REWRITE, START, WRITE(format 2)	INVALID KEY
EVALUATE	WHEN, WHEN OTHER
INITIALIZE	REPLACING
INSPECT(format 1)	TALLYING, REPLACING
INSPECT(format 2)	CONVERTING
PERFORM(format 3)	VARYING, AFTER
READ	AT END, INVALID KEY
SEARCH(formats 1 & 2)	AT END, WHEN
WRITE(format 1)	AT EOP

Exceptions to the general formatting clause are found in IF and PERFORM (formats 1 & 2) statements. In these statements, if embedded statements are discovered (by identifying a verb token) they are indented two spaces relative to the start of the original statement so long as certain clauses have not been found so far. These clauses are the ELSE clause in IF statements and the TIMES and UNTIL clauses of PERFORM (format 2) statements. The ELSE clause is placed at the same level of indenting as its IF and subsequent embedded statements are indented as above (two spaces) unless the first of these is another IF statement. A nested IF statement following an ELSE clause begins on the same line as the ELSE and its embedded statements are indented only two spaces relative to the previous IF.

```
IF A = B THEN
  ADD 2 TO A
  MOVE B TO C
ELSE IF A < 9
  MOVE 0 TO A.
```

A PERFORM (format 2) statement with TIMES or UNTIL clauses gives the same results as the statements with the general formatting clauses above. There are some minor recognition problems with identifying the TIMES clause as it begins with an integer or variable. For the purposes of formatting IF statements, NEXT SENTENCE is regarded as both a verb and embedded statement.

The nesting of PD statements coupled with the use of end-verbs can make it difficult to determine which statements are still in scope at some points in a program. Scope determination is require for formatting. A stack containing verbs and amounts of indentation of statements currently in scope is used to hold this information in Formatter2. In Formatter1 a similar stack is a convenient method for controlling the format of nested data-description-entries. This stack holds level numbers instead of verbs.

From the descriptions above which use phrases such as ON SIZE ERROR instead of either SIZE ERROR or ON SIZE ERROR, it is clear that some phrase standardisation has been carried out. Phrase standardisation within source code is a natural task for a formatter and has been implemented in the yyparse procedures of both Formatter1 and Formatter2. Standard COBOL phrases and their equivalents which are manipulated by the yyparse procedures are given in Appendix 4B.

4.3.4. THE ANALYSE SUBSYSTEM.

From figure 4.3.3., the Analyse subsystem has a very similar structure to the Format subsystem. As identified in Section 4.3.1, Analyser1 and Analyser2 are required to perform token recognition and parsing just as the Formatter filters are. The actions after identification of, say, a COBOL statement are, of course, different.

ANALfilter1 and ANALfilter2

ANALfilter1 is identical to FORMfilter1 except that warning messages issued by this filter indicate a different filter-name. ANALfilter2 has this same change compared to FORMfilter2. Other changes in

The Analyse Sybsystem

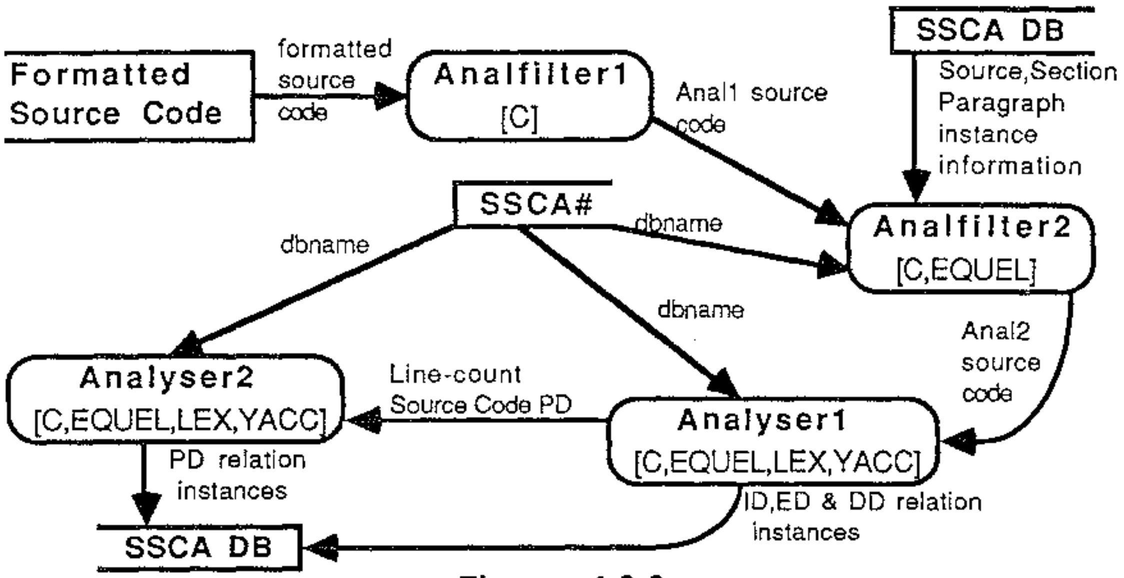


Figure 4.3.3

ANALfilter2 are that linefeeds on non-highlighted symbol lines are no longer removed. This is because the Analyser filters, unlike the Formatter filters, must keep track of current line numbers so they can be stored with related information in the SSCA DB. If the Format subsystem is working correctly, all error checking in these filters is redundant. However, in a prototype SSCA system it is easier and safer to leave these checks in place.

<u>Analyser1</u> <u>and</u> <u>Analyser2</u>

As indicated above the token identifying portions of yylex and grammar parsing portions of yyparse are almost identical in Formatter1 and Analyser1 and in Formatter2 and Analyser2. In yylex of both analyser filters, comment line handling after recognition is unnecessary (other than incrementing the line count). Because source code analysed in Analyser1 is not passed on to Analyser2, line count information must be passed to Analyser2 for it to correctly initialise the line number count. This information precedes a program's PD in the pipe between the Analyser filters. Yylex of Analyser2 must thus retrieve and utilise the line count information. In yyparse of both Analyser filters, identification of equivalent COBOL phrases and standardisation of these phrases is unnecessary as the source code is already formatted.

The task of Analyser1 and Analyser2 is to create and update (as new source information is identified) instances of SSCA DB relations. In both filters, achieving this aim is mostly straight forward. For instance, an instance of relation File will be created when the file's file-control-entry is parsed in the ED and updated when the file's FD is parsed in the DD. The main problems occur with both data-description-entry and PD statement nesting. In nesting, references to data items or statements still in scope are implicit and hence, retrieval of the corresponding relation instances for update is complicated. The method employed to handle similar problems in Formatter2 was a stack. The stack mechanisms of the Formatter filters are adapted for used in their corresponding Analyser filters. Instead of level numbers, variable instance identifiers (va_num) are used in Analyser1 and instead of verbs and indentation counts, statement instance identifiers (st_num) are used in Analyser2. However, in Analyser2 a further stack is required. This is because when analysing nested statements, not only are there implicitly referenced statement instances potentially requiring update (e.g. if the end-verb is encountered), there are condition_usage (cu) instances whose cu_br_desc and cu_branch_b attributes are unknown at the time of instance creation (this occurs when cu branches are statement-groups e.g. in IF-ELSE statements). The cu stack holds the cu identifiers (cu_num) and corresponding statement instance identifiers (st_num) for cu instances currently "in scope". When the end of a statement which has embedded statements is encountered (i.e. an end-verb or period), both the cu and statement stacks must, at least, be popped of statement related information (and possibly instances updated). It is possible to have many cu instances from one statement "in scope" at any one time (e.g. grouped WHEN clauses of EVALUATE or SEARCH statements).

4.3.5. METRIC CALCULATION AND THE SSCA.

A worth-while feature of any general static analysis system is the collection of complexity metrics (Section 3.1.4.). In terms of the structure of such a system (see Figure 3.4), actual metric calculation is carried out by the metric calculator tool. However, a SSCA can support the production of some metrics by recording certain source measurements during syntactic analysis and storing these measurements in the SSCA DB. The metric calculator is directly considered in the development of the prototype SSCA because a simplified version of it is likely to be the next implemented tool after the SSCA. This simplified metric calculator should, at least, produce the standard metrics for a program and its development would promote further research in software complexity measures. Support for production of even simple software metrics underlines the importance of attempting to quantify elements of maintenance. Without measures for elements such as software complexity it is not possible to effectively budget for or manage resources in maintenance.

In this subsection, a small subset of the metrics in Section 3.1 which appear relevant and suited to COBOL programs are identified. Definitions of these metrics for COBOL are also presented. Values of metrics from this subset are suggested as part of the output of a metric calculator. The implications of supporting metric production in the SSCA and SSCA DB are then explored.

4.3.5.1. COBOL METRICS FOR THE METRIC CALCULATOR.

Of the large range of metrics given in Section 3.1, only standard, instruction mix, program form and data reference metrics will be considered. One of the reasons for this is that many of the other metrics are better supported by tools other than a syntactic analyser. For instance, Piwowarski's N and Harrison and Magel's scope number and ratio (Section 3.1.2.3) could be derived from control flow (CF) graphs which are products of program CF analysis in the "Display and Reporting Systems" set of tools (see Figure 3.4). A reason for excluding consideration of complex metrics when implementing a SSCA system is that complete support for their calculation would compromise the main purpose of a SSCA which is to create a database of general program information with links to the formatted source code. Many complex metrics are composed of lesser metrics whose calculation may be supported anyway. For example McTap's S metric (Section 3.1.3.) uses instruction mix and data reference metrics.

STANDARD METRICS

All standard metrics (in one form or other) of Section 3.1.1. would be required output of any metric calculator because they are the most well known measures of program complexity. Halstead's E and McCabe's V(G) have been suggested to be closely related [CUR79] [SCH81] [SUN81] but later measurements discount this when variations for size are removed [CRA85]. The forms of metrics which appear useful for COBOL programs are given below.

LOC : defined as the total number of non-blank lines in the source code. Blank lines are ignored because the COBOL programs being analysed have already been formatted (i.e. the blank lines have not been entered by any programmer and their number is totally dependent on other elements in the program, such as the number of divisions, sections and paragraphs). LOC is composed of the submeasures LOCD, LOCP and comments (see submeasures definitions below).

V(G) : Which corresponds the CYC-MID (Section 3.1.1.2.). See submeasures definitions below.

E, D and length : Halstead's effort, difficulty and length measures (Section 3.1.1.3.). See submeasures definitions below.

Submeasure definitions for the above metrics are :

LOCD, LOCP : These are the number of non-blank (non-comment) lines in the PD (LOCP) and in all other divisions (LOCD). Two such measures seem appropriate because of COBOL's data definition / procedure segregation.

Comments : The number of comment lines in the source code.

n : The number of statements in the PD (equal to the number of verb occurrences).

e : The number of branch occurrences in the PD.

For COBOL, CASE statements are the EVALUATE, GO TO DEPENDING and SEARCH statements. The number of branches in these statements equals the number of alternatives.

n1 : The number of unique operators in the PD.

An operator is defined as a verb, phrase (set of words), or individual token (word or punctuation). Individual tokens must not be operands (see below), parts of phrases or commas. Commas appear inappropriate to count as operators in COBOL as they are often interchangeable with spaces. Phrases are a problem to define as no standard set of phrases exists for COBOL. Consider, for instance, whether RECORD IS VARYING IN SIZE identifies one phrase or a token (RECORD) and the IS VARYING IN SIZE phrase. However, phrases like RECORD IS VARYING IN SIZE and RECORD VARYING are synonymous even though their sub-tokens differ. Hence, in any token measuring count, such as n1, these phrases should be considered identical and indivisible. Such phrases have been standardised by the Format subsystem anyway. Appendix 4B contains the standard formatting phrases used in the prototype SSCA system.

n2 : The number of unique operands in the PD.

An operand is defined as a reference to a variable, variable-88, file, identifier-lit, file-variable or label (paragraph or section). Paragraph and section references in COBOL are not considered operators as it is difficult to use them as substitutes for intrinsic procedures or functions as parameters and local variables are unavailable. Subscripting is considered to be an extra occurrence of an operand although the brackets are punctuation and hence, are operators. Qualification of a data item may be optional, so when it is present, it seems appropriate to regard qualification tokens as part of the reference to an item (i.e. included in the operand instance).

N1 : The number of operator occurrence in the PD. See operator definition above.

N2 : The number of operand occurrence in the PD.

Operands are defined above. It should be noted that labels such a section and paragraph names are not regarded as operand occurrences in keeping with not counting declarations, although references to these labels are operand occurrences (see operand definition above).

OTHER METRICS.

The submeasures defined above represent many measures suggested in Sections 3.1.2.1, 3.1.2.2 and 3.1.2.4. However, there are other instruction mix, program form and data reference metrics or equivalent measures.

These are given below. A definition of equivalent measures is exemplified by the numbers of PD statements (n) and IF statements (#IFs) being equivalent to the percentage of IFs metric identified by Zolnowski and Simmons [ZOL80].

```
# - is an abbreviation for "number of"

#sections - sections in the PD
#paragraphs - paragraphs in the PD
#period occurrences - periods in the PD
#end-verb occurrences
average identifier name length - an identifier
        is a program data item

#CALLs
#unique CALLs
#IFs
#outer IFs - i.e. #IFs which are unnested as
        laid out in the program
#GOTOs
#label references - section and paragraph
        references
#labels defined but not referenced
#MOVEs
#PERFORMs
variable spans - for particular variables
        referenced in the PD
```

In the above list many measures have been omitted. This was sometimes done because of duplication. For example, #verb occurrences equals n, NCSL equals LOCD + LOCP, and DSL equals DOC (ignoring blank lines). Omissions were also made because of non-applicability. For example, #blank lines (an argument against blank line measures is given in the LOC definition), average #indentation spaces per line (same argument as for #blank lines), average variable definitions per module and percentage of symbolic constants (symbolic constants are not available in the reduced COBOL of this thesis). For COBOL, terms such as function and module, were taken include both sections and paragraphs. Hence, FD (#function definitions) equals #sections + #paragraphs, and the average length of modules equals LOCP / #paragraphs for paragraphs and LOCP / #sections for sections. #CF breaks (e of CYC-MIN) was left out as it should be approximately e from Torsun and Al-Jarrah's analysis of COBOL programs [TOR79].

4.3.5.2. SSCA AND SSCA DB IMPLICATIONS.

Having identified a set of desirable program measures, it must be determined whether the current SSCA DB can support calculation of all metrics in this set. Calculation of the measures in terms of SSCA DB information is given below.

LOCD - Unsupported (although the number of lines in the PD is known).

LOCP, Comments - Unsupported.

n - The number of Statement instances.

e - Can be derived from Condition-usage instances.

n1, N1 - Unsupported.

n2, N2, variable spans - These can be derived from Data-usage, Co-data-used and Du-data-used instances. (Name lookup would require the use of File, Variable, Variable-88, Identifier-Lit, Section and Paragraph instances).

#sections - The number of Section instances.

#paragraphs - The number of Paragraph instances.

#period occurrences - Unsupported.

#end-verb occurrences - The number of Statement instances with end-verbs.

average identifier length - Can be derived from File, Variable, Variable-88, Identifier-Lit, Section and Paragraph names.

#CALLs, #GOTOs, #IFs, #MOVEs, #PERFORMs - The number of Statement instances with a particular verb.

#outer IFs - Can be derived from IF Statement and Paragraph instances.

#label references, #labels defined but not referenced - These can be derived from Section, Paragraph and Condition_usage instances.

#unique CALLs - Can be derived from CALL Statement, Data-usage and Du-data-used instances.

Unsupported metrics are LOCD, LOCP, comments, n1, N1 and #period occurrences. Information for these metrics lies within the formatted source code but not currently in the SSCA DB. For all of these metrics except LOCD and comments not in the PD, a useful breakdown of their values is given by PD paragraph. This suggests the extra metric attributes of locp, pa_comments, u_operators, operators, and periods for the Paragraph relation. LOCD and the number of comment lines in the ID, ED and DD suggest metric attributes for either the Program or Source relations. The Source relation was chosen to hold the attributes locd and comments. Definitions for the metric attributes above are given in the entities of Appendix 4A. These metric attributes are supported in the SSCA by extra processing in the Analyse subsystem. Collection of metric information is carried out in the printline and yylex procedures of both Analyser1 and Analyser2 and the yyparse procedure of Analyser2.

CHAPTER 5. CONCLUSIONS.

This final chapter presents conclusions about the aspects of software maintenance and maintenance support examined and discussed in earlier chapters. These conclusions are divided into sections on maintenance in general, software metrics, maintenance support through tools, the prototype SSCA system, and a general conclusion.

5.1. MAINTENANCE IN GENERAL.

The main themes of Chapter 2 are the definition of maintenance and an outline of its perceived problems.

Section 2.1 identifies a generally accepted definition for maintenance although it seems too broad to be very useful. This is especially the case when attempting to classify system extensions. An extension to an existing application can be regarded as new development, or enhancement, or adaptive maintenance. However, modifications to existing parts of an application to cater for an extension would appear from definitions to be adaptive maintenance. Section 2.2 examines difficulties with subcategorisation of maintenance. There seems to be a general lack of solid and useful definitions throughout the maintenance area.

Because of its position at the end of the system life cycle, maintenance may inherit many problems from development. Obvious examples are poor application documentation and bugs in programs. Less clear are the affects of ill-trained users and poor program design. More survey evidence is required about these factors. This evidence could provide quantitative benefit information for development techniques such as structured design. At the moment, factors like documentation are known to affect maintenance costs but no information such as, "every X dollars spent on development documentation will, on average, save Y dollars off the maintenance budget" is available.

In addition to problems caused by the lack of clear definitions and by external influences, there are certainly internal problems in most maintenance tasks related to actually modifying software. These problems are illustrated by the difficulty in formulating a set of steps to aid a maintainer in modifying software (see Section 2.4). A clear understanding of the program to be changed, and to a lesser extent of the application, are definite factors affecting success in program modification. Even with a complete knowledge of a program's purpose and design, the design of modifications is affected by the role and goals of program maintenance as perceived by the maintainer. If a particular program is expected to have a remaining operational life of only 6 months, then the goals of maintenance on this program might exclude modification design to minimise structural decay and might concentrate on development of an effective, quickly implemented and more easily verified program patch. For programs which are expected to have much longer life-times and survive numerous modifications, it is suggested in this thesis that the goal of preserving good design principles during maintenance would usually outway the short term goal of implementing modifications to solve a perceived problem in the shortest possible time. However, there is little evidence available to support or contradict this hypothesis.

Overall, in almost all aspects of maintenance, more research is required. However, there are overlaps with areas such as development methodologies and tools, programming language design, documentation systems and DP management techniques.

5.2. SOFTWARE METRICS.

Investigations into static measures of software complexity have a part to play in identifying and quantifying factors critical to maintenance productivity. It is clear from Section 3.1 that a massive number of complexity metrics have been proposed. What is required is large numbers of evaluations of, at least, a small group of metrics. These metric values, together with actual maintenance costs and size / volume values for the maintenance tasks being performed, would allow some determination of what factors influence productivity. Static complexity metrics are considered to be measures of the difficulty in dealing with a piece of software. However, measures for the extent of changes to be made and additional code to be created in a maintenance task are few. After modification, task sizes could be ascertained by counting the number of lines in the modified program which are different form the original. However, for newly added functions and procedures, their complexity metrics might be a better indication of coding effort. These measures may be useful for identifying productivity factors but, in terms of cost estimation, other task size / volume metrics which can be calculated earlier in a maintenance task are required. Hopefully, the productivity factors themselves will suggest such metrics.

Complexity metric research would seem to have a long way to go before its products are suitable for cost estimation in maintenance. Re-iterating the suggestions from Section 3.1.4, this research is supported by compilers or specific tools which automatically evaluate a set of complexity metrics. The SSCA system whose development is summarised in this thesis is a first step toward a metric calculating tool as well as several other software tools.

5.3. MAINTENANCE SUPPORT THROUGH TOOLS.

Observations from Section 3.3.2 indicate there is a need for a more integrated approach to maintenance support tools. An integrated maintenance support environment is the ideal but how it should be constructed is largely unknown.

Application documentation is most often used in the operational and maintenance areas of the system life cycle and so a complete maintenance support environment may actually contain a documentation support system. At the very least, portions of a maintenance environment will be closely involved with interrogating and updating application documents. A clear conclusion is that documentation and maintenance support are inter-related and, in many areas, overlap. In wider perspectives, Anderson [AND81] and Saib [SAI83] interweave an application documentation system with a combined development and maintenance support environment.

With reference to the integration of maintenance support tools, the contention of this thesis is that a data base, such as the SSCA DB, is a reasonable method of linking some tools which manipulate or use source code. The centralising and formalising of access to source code allows these tools to achieve a higher degree of specialisation. Supporting a toolbox system based around static analysis of source code appears well-founded as the logic within source programs is fundamental to computer systems and maintenance and the programmer or maintainer's view of this code is primarily static in nature (interactive debugging being an exception). A consequence of the source code analysis approach, at least in a prototype toolbox system, is the production of tools for a specific programming language. COBOL was chosen as the language whose programs would be analysed in the static analysis system of which the SSCA and SSCA DB are components.

Building tools for an archaic (in computing time-scales) programming language could be considered as similar to installing plumbing in the pyramids. However, COBOL is still extensively used today and, by the shear volume of current code, will be around for at least the next 10 years. Production of commercial software tools for COBOL is a method of speeding up the diffusion rate of new software engineering innovations. The diffusion rate is expected to slow through the 1980s according to Morrissey and Wu [MOR79].

Most suggested maintenance tools aid maintainers to understand aspects of an application system. Most research in tool production must thus consider how maintainers (tool-users) form an understanding of software and perform modifications on it. Comprehension of common methods of software understanding and modification in any programming language provide valuable input into areas such as production of 4GLs and support environments. If maintenance tools are in use in an application system written in so-called 3GLs, this should force any suggested replacement or partial replacement for these 3GLs to provide at least as much maintenance support as currently offered. Tool production for current languages thus indirectly enhances future maintenance facilities.

5.4. THE PROTOTYPE SSCA SYSTEM.

5.4.1. SSCA DEVELOPMENT.

The method used for designing the SSCA in Section 4.3 was basically a melding of the actions which needed to be carried out and the tools available for the development. Clear objectives of the SSCA in terms of data base information were defined in Section 4.2. The tools used include simple input / output (i/o) pipelines, the AWK and C programming languages and YACC and LEX preprocessors. All these tools proved extremely useful in speeding up the implementation process.

For the use of AWK and i/o pipelines, there was a small price to pay. AWK is line orientated and consequently has trouble identifying sequences of tokens broken by blank or comment lines. I/o pipelines had the most effect on system design. Simple i/o pipes implement sequential data flows only. Breaks in the sequential nature of the data flows are implemented as data stores such as SSCA# or SSCA DB (see Figures 4.3.1 to 4.3.3). Data flow branching could have been implemented by sending, waiting and receiving mechanisms in filters but this is complicated and error-prone.

When confined to sequential filtering (except for data stores), the main subsystem design decision was which actions to group in which filters so that :

a) Either simple i/o piping or data stores (files or data bases) were the communication channels between filters.

b) Processing was simplified within filters (e.g. if indicator fields of lines are checked in a filter, some comment and continuation actions are relatively simple to implement as well)

c) Each filter has a reasonably logical set of actions.

There is also an order among some actions which must be maintained. For example, the two actions below regroup tokens broken by line continuation, but they must be carried out in the order 1 then 2.

1) removal of trailing spaces from lines not containing unfinished strings.

2) removal of columns 1 to 7 and proceeding spaces (and '"' if continuing a string) from continuation lines, and linefeeds from all lines except division, section, paragraph headings and comments.

Problems encountered when developing the SSCA system suggest alternative methods of storing source code and comments. From Section 4.3, the comments and layout (e.g. 80 character line orientation) of programs cause particular problems for tools which manipulate source code. Even in free-format programming languages (e.g. Pascal), source code tends to be commented and laid out for clarity when displayed on 24 line by 80 character screens or 132 character per line printers. For software tools which need to recognise elements of source code or automatically rewrite portions of programs, it seems useful to store a program (in a free format language) as a set of statements delimited by linefeeds (with no maximum line length). Statement nesting would needed to be indicated, probably by using control characters. Comments could be mapped onto a set of statements using line numbers. Ideally program comments would be stored separately in the application documentation system. Isolation of comments from program code has advantages for comment update and integration with other application documentation. To view, print or modify source code, the formatting services of a language-orientated editor (LOE) would be required to structure source information and comments for display on a particular device. Allowing higher program views (e.g. structure charts) rather than just formatted statements, could also be a feature of the LOE. The context editor EDIERE (one of the SAMOS tools [EBE80]) has these facilities for conventional source code. There are overheads with source code / comment separation especially in terms of pointer adjustment for comments when source statements are added, deleted or moved.

5.4.2. USE OF THE SSCA DB.

This topic is best summarised using some simple examples. Four questions given in Section 4.2.2 were suggested as enquiries a maintainer may make. Partial answers to the first two questions formulated as QUEL (INGRES's interactive enquiry language) queries are given below.

QUESTION 1. Part of this question translates as "given a section sname, what are its paragraphs ?". The answer is given by :

```
range of s is section
range of p is paragraph
retrieve (p.pa_name)
  where p.pa_section = s.se_num
    and s.se_name = "sname"
```

QUESTION 2. "Given a paragraph pname, at what lines in the source code is it directly PERFORMed or gone to ?" is answered by :

```
range of p is paragraph
range of c is cond_usage
range of s is statement
retrieve (s.st_position)
  where s.st_num = c.cu_statement
    and p.pa_name = "pname"
    and ( c.cu_branch_a = p.pa_num
    or c.cu_branch_b = p.pa_num )
```

Another part of question 2 was the identification of paragraphs and sections dropped into (possibly from GO TOs to earlier paragraphs/sections). A partial answer for paragraphs is to identify paragraphs for which the previous paragraph is gone to and no unconditioned GO TOs, STOP RUNs or EXIT PROGRAMs exist in this previous paragraph. The query is :

```
range of p is paragraph
range of q is paragraph
range of c is cond_usage
range of s is statement
range of t is statement
retrieve (q.pa_name)
  where p.pa_num = q.pa_num - 1
    and c.cu_branch_a = p.pa_num
    and c.cu_statement = t.st_num and t.verb = "GO TO"
    and any(s.st_stgroup by p.pa_stgroup
      where s.st_stgroup = p.pa_stgroup and
      (s.verb = "GO TO" or s.verb = "STOP RUN"
       or s.verb = "EXIT PROGRAM") ) = 0
```

The solution above takes no account of the previous paragraph being dropped into itself and conditioned GOTOs or STOP RUNs do not necessarily mean the next paragraph can be dropped into. For example, the statement below has the same branching effect as a unconditional GOTO.

```
IF A = B
        GO TO C
ELSE
        GO TO D.
```

The last query also shows a simulation of negated existential quantification in QUEL using the aggregate function "any".

The QUEL queries above were presented as examples to show that SSCA DB information is useful when accessed by an interactive enquiry system. These queries could have just as well have been EQUEL statements. However, it can be seen that most useful interactive enquires are very complex and cumbersome to type. For a maintainer, the best solution in the interactive environment is a preformulated library of queries from which he can select and then invoke. Such a selection process is currently possible in QUEL through the editing mechanism for queries.

Question 1 was actually stated in Section 4.2.2. as "what is the static hierarchical structure of the PD ?". This suggests that ordered lists of sections and paragraphs are not what was requested. Hierarchical diagrams would show the static relationships much better. Similarly, control flow paths indicated in the answers to question 2 are best described as a set of digraphs. These observations imply that the enquiry interface should really only be a method for tools (including display and report tools) to retrieve SSCA DB information. In a complete static analysis system, maintainers should interact exclusively through the user interfaces of tools.

When running the SSCA system to build the SSCA DB it is noted that appending of instances of relations one by one is very time consuming in INGRES. It would be possible, in some cases, to build an intermediate file of instances and copy them into the data base all at once. This would speed up data base creation. Speed and constraints in the INGRES DBMS, such as the need to use strange methods to simulate negated existential quantifiers, indicate that other DBMSs for the SSCA DB would be worth exploring in later SSCA systems. The DBMS chosen could again be relational or could be a network system so long as reasonable enquiry system was supported (as concluded in Section 4.2.3.1.).

5.5. GENERAL CONCLUSION.

The development of a prototype SSCA system has shown that a data base of program information is a feasible, practicable and worthwhile foundation for a basic set of maintenance tools. Enhanced access to program information encourages development of further components of the basic toolset and other related tools.

APPENDIX 1 - THE PURGE PROGRAM AND ITS MODIFICATIONS

```
 Source File:  <STUD2>CS302X>MISC>WRITEUP>PURGE.CBL
 Compiled on:  FRI, AUG 22 1986 at 14:30
          by:  CBL rev 19.3.5  06/12/85.09:38
 Options are:  LISTING BINARY OPTIMIZE U(PPER)CASE
               FORMATTED_DISPLAY
```

```
*
*****************************************************
*  PURGE Program from "Principles of Program Design,*
*     page 135, problem 11 by M.A. Jackson [JAC75]  *
*****************************************************
*
 Identification Division.
 Program-id. PURGE.
 Environment Division.
 Input-Output Section.
 File-Control.
     Select infile assign to PFMS.
     Select outfile assign to PFMS.
     Select errorfile assign to PFMS.
 Data Division.
 File Section.
    FD infile Compressed
     value of file-id is "infile".
    01 inrec                        pic x(80).
    FD outfile Compressed
     value of file-id is "outfile".
    01 outrec                       pic x(80).
    FD errorfile compressed
     value of file-id is "errorfile".
    01 errorrec                     pic x(80).
* ...
 Working-Storage Section.
* file record structures
    01 rec1.
      02 r1-branch                  pic 9999.
      02 r1-type                    pic 9999.
      02 r1-info                    pic x(72).
    01 rec2.
      02 r2-branch                  pic 9999.
      02 r2-type                    pic 9999.
      02 r2-info                    pic x(72).
    01 rec3.
      02 r3-branch                  pic 9999.
      02 r3-type                    pic 9999.
      02 r3-info                    pic x(72).
    01 error-heading.
```

```
      02 e-heading                  pic x(30)
             value " ERROR DATA FOR BRANCH-NUMBER ".
      02 e-heading-branch           pic 9999.
      02 filler                     pic x(46).
* other variables
    01 end-of-input-switch   pic x  value "F".
      88 end-of-input               value "T".
*****************************************************
 Procedure Division.
*
 Main.
     perform Start-para.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if not end-of-input
         perform Get-2-cards
         perform Main-loop
             until r1-branch = 0.
     perform Finish-para.
     stop run.
*
 Start-para.
     open input infile.
     open output outfile.
     open output errorfile.
*
 Finish-para.
     close infile.
     close outfile.
     close errorfile.
*
 Main-loop.
     if r1-branch not = r2-branch
         perform Branches-not-equal
     else
         if r1-branch = r3-branch
             perform More-than-2-cards
             perform Get-2-cards
         else
             if not(r1-type numeric and r2-type numeric)
                 perform Type-problem
                 perform Get-2-cards
             else
                 if not(r1-type < r2-type)
                     perform Type-problem
                     perform Get-2-cards
                 else
                     perform Print-good-set
                     perform Get-2-cards.
*
```

```
 Branches-not-equal.
     perform Print-error-heading-card.
     move rec2 to rec1.
     move rec3 to rec2.
     if not end-of-input
         perform Read-rec3.
*
 More-than-2-cards.
     perform Print-error-heading-card.
     write errorrec from rec2.
     write errorrec from rec3.
     perform Error-branch-loop
         until r1-branch not = r3-branch.
*
 Type-problem.
     perform Print-error-heading-card.
     write errorrec from rec2.
*
 Error-branch-loop.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec1
     else
         if r1-branch = r3-branch
             write errorrec from rec3.
*
 Get-2-cards.
     move rec3 to rec1.
     if not end-of-input
         perform Read-rec2
         if not end-of-input
             perform Read-rec3.
*
 Read-rec2.
     read infile into rec2
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec2.
*
 Read-rec3.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec3.
*
 Print-good-set.
     write outrec from rec1.
     write outrec from rec2.
*
Print-error-heading-card.
     move r1-branch to e-heading-branch.
     write errorrec from error-heading.
     write errorrec from rec1.
*
```

```
Source File:  <STUD2>CS302X>MISC>WRITEUP>P-A1.CBL
Compiled on:  FRI, AUG 22 1986 at 14:29
         by:  CBL rev 19.3.5  06/12/85.09:38
Options are:  LISTING BINARY OPTIMIZE U(PPER)CASE
              FORMATTED_DISPLAY

 *
 ***********************************************
 *  PURGE-A1 Program - handles the first modification*
 *                        to PURGE                   *
 ***********************************************
 *
  Identification Division.
  Program-id. PURGE-A1.
  Environment Division.
  Input-Output Section.
  File-Control.
      Select infile assign to PFMS.
      Select outfile assign to PFMS.
      Select errorfile assign to PFMS.
  Data Division.
  File Section.
     FD infile Compressed
      value of file-id is "infile".
     01 inrec                          pic x(80).
     FD outfile Compressed
      value of file-id is "outfile".
     01 outrec                         pic x(80).
     FD errorfile compressed
      value of file-id is "errorfile".
     01 errorrec                       pic x(80).
 * ...
  Working-Storage Section.
 * file record structures
     01 rec1.
       02 r1-branch                    pic 9999.
       02 r1-type                      pic 9999.
       02 r1-info                      pic x(72).
     01 rec2.
       02 r2-branch                    pic 9999.
       02 r2-type                      pic 9999.
       02 r2-info                      pic x(72).
     01 rec3.
       02 r3-branch                    pic 9999.
       02 r3-type                      pic 9999.
       02 r3-info                      pic x(72).
     01 error-heading.
       02 e-heading                    pic x(30)
              value " ERROR DATA FOR BRANCH-NUMBER ".
       02 e-heading-branch             pic 9999.
       02 filler                       pic x(46).
```

```
* other variables
    01 end-of-input-switch   pic x  value "F".
      88 end-of-input               value "T".
**************************************************
 Procedure Division.
*
 Main.
     perform Start-para.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if not end-of-input
         perform Get-2-cards
         perform Main-loop
             until r1-branch = 0.
     perform Finish-para.
     stop run.
*
 Start-para.
     open input infile.
     open output outfile.
     open output errorfile.
*
 Finish-para.
     close infile.
     close outfile.
     close errorfile.
*
 Main-loop.
     if r1-branch not = r2-branch
         perform Branches-not-equal
     else
         if r1-branch = r3-branch
             perform More-than-2-cards
             perform Get-2-cards
         else
             if not (r1-type numeric and r2-type numeric)
                and not (r1-type = "MOD1" and
                 r2-type = "MOD1")
                 perform Type-problem
                 perform Get-2-cards
             else
                 if not(r1-type < r2-type) and
                    not (r1-type = "MOD1" and
                    r2-type = "MOD1")
                     perform Type-problem
                     perform Get-2-cards
                 else
                     perform Print-good-set
                     perform Get-2-cards.
*
```

```
 Branches-not-equal.
     perform Print-error-heading-card.
     move rec2 to rec1.
     move rec3 to rec2.
     if not end-of-input
         perform Read-rec3.
*
 More-than-2-cards.
     perform Print-error-heading-card.
     write errorrec from rec2.
     write errorrec from rec3.
     perform Error-branch-loop
         until r1-branch not = r3-branch.
*
 Type-problem.
     perform Print-error-heading-card.
     write errorrec from rec2.
*
 Error-branch-loop.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec1
     else
         if r1-branch = r3-branch
             write errorrec from rec3.
*
 Get-2-cards.
     move rec3 to rec1.
     if not end-of-input
         perform Read-rec2
         if not end-of-input
             perform Read-rec3.
*
 Read-rec2.
     read infile into rec2
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec2.
*
 Read-rec3.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec3.
*
 Print-good-set.
     write outrec from rec1.
     write outrec from rec2.
*
 Print-error-heading-card.
     move r1-branch to e-heading-branch.
     write errorrec from error-heading.
     write errorrec from rec1.
*
```

```
Source File:  <STUD2>CS302X>MISC>WRITEUP>P-A2.CBL
Compiled on:  FRI, AUG 22 1986 at 14:30
         by:  CBL rev 19.3.5  06/12/85.09:38
Options are:  LISTING BINARY OPTIMIZE U(PPER)CASE
              FORMATTED_DISPLAY

*
**************************************************
*  PURGE-A2 Program - handles the first modification*
*                        to PURGE                 *
**************************************************
*
 Identification Division.
 Program-id. PURGE-A2.
 Environment Division.
 Input-Output Section.
 File-Control.
     Select infile assign to PFMS.
     Select outfile assign to PFMS.
     Select errorfile assign to PFMS.
 Data Division.
 File Section.
    FD infile Compressed
     value of file-id is "infile".
    01 inrec                           pic x(80).
    FD outfile Compressed
     value of file-id is "outfile".
    01 outrec                          pic x(80).
    FD errorfile compressed
     value of file-id is "errorfile".
    01 errorrec                        pic x(80).
* ...
 Working-Storage Section.
* file record structures
    01 rec1.
      02 r1-branch                     pic 9999.
      02 r1-type                       pic 9999.
      02 r1-info                       pic x(72).
    01 rec2.
      02 r2-branch                     pic 9999.
      02 r2-type                       pic 9999.
      02 r2-info                       pic x(72).
    01 rec3.
      02 r3-branch                     pic 9999.
      02 r3-type                       pic 9999.
      02 r3-info                       pic x(72).
    01 error-heading.
      02 e-heading                     pic x(30)
             value " ERROR DATA FOR BRANCH-NUMBER ".
      02 e-heading-branch              pic 9999.
      02 filler                        pic x(46).
```

```
* other variables
    01 end-of-input-switch   pic x  value "F".
      88 end-of-input               value "T".
**************************************************
 Procedure Division.
*
 Main.
     perform Start-para.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if not end-of-input
         perform Get-2-cards
         perform Main-loop
             until r1-branch = 0.
     perform Finish-para.
     stop run.
*
 Start-para.
     open input infile.
     open output outfile.
     open output errorfile.
*
 Finish-para.
     close infile.
     close outfile.
     close errorfile.
*
 Main-loop.
     if r1-branch = r2-branch and
        r1-branch not = r3-branch
        and r1-type = "MOD1" and r2-type = "MOD1"
         perform Print-good-set
         perform Get-2-cards
     else
         if r1-branch not = r2-branch
             perform Branches-not-equal
         else
             if r1-branch = r3-branch
                 perform More-than-2-cards
                 perform Get-2-cards
             else
                 if not (r1-type numeric and
                    r2-type numeric)
                     perform Type-problem
                     perform Get-2-cards
                 else
                     if not(r1-type < r2-type)
                         perform Type-problem
                         perform Get-2-cards
                     else
```

```
                         perform Print-good-set
                         perform Get-2-cards.
*
 Branches-not-equal.
     perform Print-error-heading-card.
     move rec2 to rec1.
     move rec3 to rec2.
     if not end-of-input
         perform Read-rec3.
*
 More-than-2-cards.
     perform Print-error-heading-card.
     write errorrec from rec2.
     write errorrec from rec3.
     perform Error-branch-loop
         until r1-branch not = r3-branch.
*
 Type-problem.
     perform Print-error-heading-card.
     write errorrec from rec2.
*
 Error-branch-loop.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec1
     else
         if r1-branch = r3-branch
             write errorrec from rec3.
*
 Get-2-cards.
     move rec3 to rec1.
     if not end-of-input
         perform Read-rec2
         if not end-of-input
             perform Read-rec3.
*
 Read-rec2.
     read infile into rec2
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec2.
*
 Read-rec3.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec3.
```

```
*
 Print-good-set.
     write outrec from rec1.
     write outrec from rec2.
*
 Print-error-heading-card.
     move r1-branch to e-heading-branch.
     write errorrec from error-heading.
     write errorrec from rec1.
*
```

```
Source File:  <STUD2>CS302X>MISC>WRITEUP>P-B1.CBL
Compiled on:  FRI, AUG 22 1986 at 14:42
        by:   CBL rev 19.3.5  06/12/85.09:38
Options are:  LISTING BINARY OPTIMIZE U(PPER)CASE
              FORMATTED_DISPLAY
```

```
*
*****************************************************
*  PURGE-B1 Program - handles the second            *
*                     modification to PURGE         *
*****************************************************
*
 Identification Division.
 Program-id. PURGE-B1.
 Environment Division.
 Input-Output Section.
 File-Control.
     Select infile assign to PFMS.
     Select outfile assign to PFMS.
     Select errorfile assign to PFMS.
 Data Division.
 File Section.
    FD infile Compressed
     value of file-id is "infile".
    01 inrec                        pic x(80).
    FD outfile Compressed
     value of file-id is "outfile".
    01 outrec                       pic x(80).
    FD errorfile compressed
     value of file-id is "errorfile".
    01 errorrec                     pic x(80).
* ...
 Working-Storage Section.
* file record structures
    01 rec1.
      02 r1-branch                  pic 9999.
      02 r1-type                    pic 9999.
      02 r1-info                    pic x(72).
    01 rec2.
      02 r2-branch                  pic 9999.
      02 r2-type                    pic 9999.
      02 r2-info                    pic x(72).
    01 rec3.
      02 r3-branch                  pic 9999.
      02 r3-type                    pic 9999.
      02 r3-info                    pic x(72).
    01 error-heading.
      02 e-heading                  pic x(30)
             value " ERROR DATA FOR BRANCH-NUMBER ".
      02 e-heading-branch           pic 9999.
      02 filler                     pic x(46).
```

```
* other variables
    01 end-of-input-switch   pic x  value "F".
      88 end-of-input               value "T".
*************************************************
 Procedure Division.
*
 Main.
     perform Start-para.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if not end-of-input
         perform Get-2-cards
         perform Main-loop
             until r1-branch = 0.
     perform Finish-para.
     stop run.
*
 Start-para.
     open input infile.
     open output outfile.
     open output errorfile.
*
 Finish-para.
     close infile.
     close outfile.
     close errorfile.
*
 Main-loop.
     if r1-branch not = r2-branch and
        r1-type not = "MOD2"
         perform Branches-not-equal
     else
         if r1-branch = r3-branch
             perform More-than-2-cards
             perform Get-2-cards
         else
             if not (r1-type numeric and r2-type numeric)
                and r1-branch = r2-branch
                 perform Type-problem
                 perform Get-2-cards
             else
                 if not(r1-type < r2-type)
                    and r1-branch = r2-branch
                     perform Type-problem
                     perform Get-2-cards
                 else
                     if r1-type not = "MOD2"
                         perform Print-good-set
                         perform Get-2-cards
                     else
```

```
                         write outrec from rec1
                         move rec2 to rec1
                         move rec3 to rec2
                         if not end-of-input
                             perform Read-rec3.
*
 Branches-not-equal.
     perform Print-error-heading-card.
     move rec2 to rec1.
     move rec3 to rec2.
     if not end-of-input
         perform Read-rec3.
*
 More-than-2-cards.
     perform Print-error-heading-card.
     write errorrec from rec2.
     write errorrec from rec3.
     perform Error-branch-loop
         until r1-branch not = r3-branch.
*
 Type-problem.
     perform Print-error-heading-card.
     write errorrec from rec2.
*
 Error-branch-loop.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec1
     else
         if r1-branch = r3-branch
             write errorrec from rec3.
*
 Get-2-cards.
     move rec3 to rec1.
     if not end-of-input
         perform Read-rec2
         if not end-of-input
             perform Read-rec3.
*
 Read-rec2.
     read infile into rec2
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec2.
*
 Read-rec3.
     read infile into rec3
         at end
```

```
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec3.
*
 Print-good-set.
     write outrec from rec1.
     write outrec from rec2.
*
 Print-error-heading-card.
     move r1-branch to e-heading-branch.
     write errorrec from error-heading.
     write errorrec from rec1.
*
```

```
Source File:  <STUD2>CS302X>MISC>WRITEUP>P-B2.CBL
Compiled on:  FRI, AUG 22 1986 at 14:43
        by:   CBL rev 19.3.5  06/12/85.09:38
Options are:  LISTING BINARY OPTIMIZE U(PPER)CASE
              FORMATTED_DISPLAY
```

```
*
***************************************************
*  PURGE-B2 Program - handles the second          *
*                     modification to PURGE       *
***************************************************
*
 Identification Division.
 Program-id. PURGE-B2.
 Environment Division.
 Input-Output Section.
 File-Control.
     Select infile assign to PFMS.
     Select outfile assign to PFMS.
     Select errorfile assign to PFMS.
 Data Division.
 File Section.
    FD infile Compressed
     value of file-id is "infile".
    01 inrec                          pic x(80).
    FD outfile Compressed
     value of file-id is "outfile".
    01 outrec                         pic x(80).
    FD errorfile compressed
     value of file-id is "errorfile".
    01 errorrec                       pic x(80).
* ...
 Working-Storage Section.
* file record structures
    01 rec1.
      02 r1-branch                    pic 9999.
      02 r1-type                      pic 9999.
      02 r1-info                      pic x(72).
    01 rec2.
      02 r2-branch                    pic 9999.
      02 r2-type                      pic 9999.
      02 r2-info                      pic x(72).
    01 rec3.
      02 r3-branch                    pic 9999.
      02 r3-type                      pic 9999.
      02 r3-info                      pic x(72).
    01 error-heading.
      02 e-heading                    pic x(30)
             value " ERROR DATA FOR BRANCH-NUMBER ".
      02 e-heading-branch             pic 9999.
      02 filler                       pic x(46).
```

```
* other variables
    01 end-of-input-switch   pic x  value "F".
      88 end-of-input               value "T".
**************************************************
 Procedure Division.
*
 Main.
     perform Start-para.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if not end-of-input
         perform Get-2-cards
         perform Main-loop
             until r1-branch = 0.
     perform Finish-para.
     stop run.
*
 Start-para.
     open input infile.
     open output outfile.
     open output errorfile.
*
 Finish-para.
     close infile.
     close outfile.
     close errorfile.
*
 Main-loop.
     if r1-branch not = r2-branch
         perform Branches-not-equal
     else
         if r1-branch = r3-branch
             perform More-than-2-cards
             perform Get-2-cards
         else
             if not (r1-type numeric and r2-type numeric)
                 perform Type-problem
                 perform Get-2-cards
             else
                 if not(r1-type < r2-type)
                     perform Type-problem
                     perform Get-2-cards
                 else
                     perform Print-good-set
                     perform Get-2-cards.
*
 Branches-not-equal.
     if r1-type = "MOD2"
         write outrec from rec1
     else
```

```
             perform Print-error-heading-card.
         move rec2 to rec1.
         move rec3 to rec2.
         if not end-of-input
             perform Read-rec3.
*
 More-than-2-cards.
     perform Print-error-heading-card.
     write errorrec from rec2.
     write errorrec from rec3.
     perform Error-branch-loop
         until r1-branch not = r3-branch.
*
 Type-problem.
     perform Print-error-heading-card.
     write errorrec from rec2.
*
 Error-branch-loop.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec1
     else
         if r1-branch = r3-branch
             write errorrec from rec3.
*
 Get-2-cards.
     move rec3 to rec1.
     if not end-of-input
         perform Read-rec2
         if not end-of-input
             perform Read-rec3.
*
 Read-rec2.
     read infile into rec2
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec2.
*
 Read-rec3.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec3.
*
 Print-good-set.
     write outrec from rec1.
     write outrec from rec2.
```

```
*
 Print-error-heading-card.
     move r1-branch to e-heading-branch.
     write errorrec from error-heading.
     write errorrec from rec1.
*
```

```
Source File:  <STUD2>CS302X>MISC>WRITEUP>P-B3.CBL
Compiled on:  FRI, AUG 22 1986 at 14:43
        by:  CBL rev 19.3.5  06/12/85.09:38
Options are:  LISTING BINARY OPTIMIZE U(PPER)CASE
              FORMATTED_DISPLAY
```

```
 *
*****************************************************
*  PURGE-B3 Program - handles the second            *
*                     modification to PURGE         *
*****************************************************
 *
 Identification Division.
 Program-id. PURGE-B3.
 Environment Division.
 Input-Output Section.
 File-Control.
     Select infile assign to PFMS.
     Select outfile assign to PFMS.
     Select errorfile assign to PFMS.
 Data Division.
 File Section.
    FD infile Compressed
     value of file-id is "infile".
    01 inrec                           pic x(80).
    FD outfile Compressed
     value of file-id is "outfile".
    01 outrec                          pic x(80).
    FD errorfile compressed
     value of file-id is "errorfile".
    01 errorrec                        pic x(80).
* ...
 Working-Storage Section.
* file record structures
    01 rec1.
      02 r1-branch                     pic 9999.
      02 r1-type                       pic 9999.
      02 r1-info                       pic x(72).
    01 rec2.
      02 r2-branch                     pic 9999.
      02 r2-type                       pic 9999.
      02 r2-info                       pic x(72).
    01 rec3.
      02 r3-branch                     pic 9999.
      02 r3-type                       pic 9999.
      02 r3-info                       pic x(72).
    01 error-heading.
      02 e-heading                     pic x(30)
             value " ERROR DATA FOR BRANCH-NUMBER ".
      02 e-heading-branch              pic 9999.
      02 filler                        pic x(46).
```

```
* other variables
    01 end-of-input-switch   pic x  value "F".
      88 end-of-input               value "T".
**************************************************
 Procedure Division.
*
 Main.
     perform Start-para.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if not end-of-input
         perform Get-2-cards
         perform Main-loop
             until r1-branch = 0.
     perform Finish-para.
     stop run.
*
 Start-para.
     open input infile.
     open output outfile.
     open output errorfile.
*
 Finish-para.
     close infile.
     close outfile.
     close errorfile.
*
 Main-loop.
     if r1-branch not = r2-branch
         perform Branches-not-equal
     else
         if r1-branch = r3-branch
             perform More-than-2-cards
             perform Get-2-cards
         else
             if not (r1-type numeric and r2-type numeric)
                 perform Type-problem
                 perform Get-2-cards
             else
                 if not(r1-type < r2-type)
                     perform Type-problem
                     perform Get-2-cards
                 else
                     perform Print-good-set
                     perform Get-2-cards.
*
 Branches-not-equal.
     perform Print-error-heading-card.
     move rec2 to rec1.
     move rec3 to rec2.
     if not end-of-input
         perform Read-rec3.
*
 More-than-2-cards.
     perform Print-error-heading-card.
     write errorrec from rec2.
     write errorrec from rec3.
     perform Error-branch-loop
         until r1-branch not = r3-branch.
*
 Type-problem.
     perform Print-error-heading-card.
     write errorrec from rec2.
*
 Error-branch-loop.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec1
     else
         if r1-branch = r3-branch
             write errorrec from rec3.
*
 Get-2-cards.
     move rec3 to rec1.
     if not end-of-input
         perform Read-rec2
         if not end-of-input
             perform Read-rec3.
*
 Read-rec2.
     read infile into rec2
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec2.
*
 Read-rec3.
     read infile into rec3
         at end
             move "T" to end-of-input-switch.
     if end-of-input
         move zeros to rec3.
*
 Print-good-set.
     write outrec from rec1.
     write outrec from rec2.
*
 Print-error-heading-card.
     if (r1-branch not = r2-branch) and r1-type = "MOD2"
```

```
         write outrec from rec1
     else
         move r1-branch to e-heading-branch
         write errorrec from error-heading
         write errorrec from rec1.
*
```

Data and Program Structures for Program PURGE

File Structures

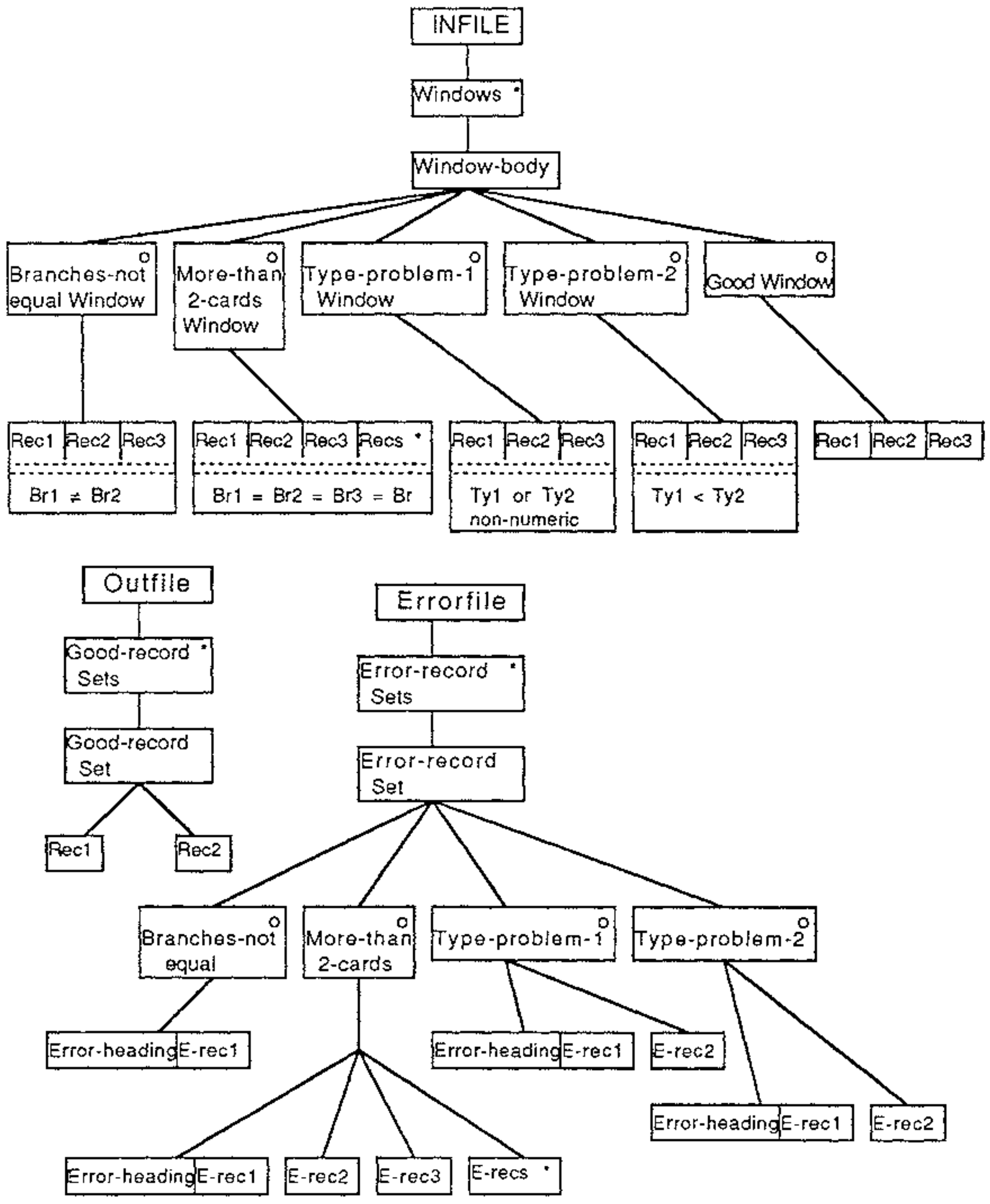


Figure A1.1

Program Structure

Note - COBOL paragraph names are identified on this diagram in bold type

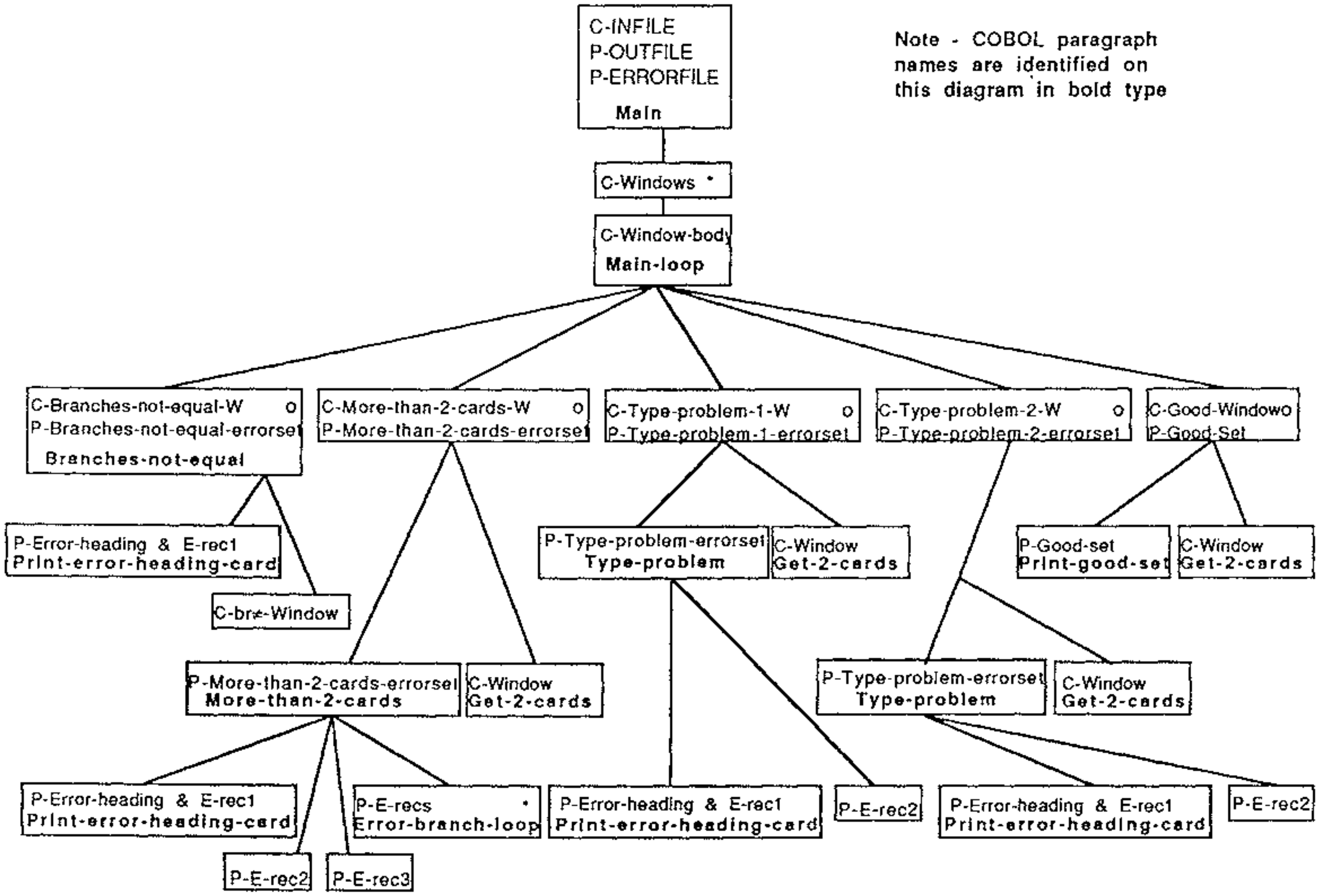


Figure A1.2

APPENDIX 2 - MAINTENANCE TOOLS

DOCUMENTATION SUPPORT TOOLS

Below several tools which support or can be applied to general or specific documentation are summarised.

FORTUNE

FORTUNE [RIC85] is a proposed project to develop a desk-top system for software engineers. This UNIX-based system is centred around documentation including text, design diagrams, source code, tables, charts, manuals and user documents. It's purpose is to support other IPSE systems by providing interfaces for system designers, programmers, technical writers and graphics designers to documentation. Interface features include editing, cross-reference searching, demand publishing (on high quality printers) and allowing document annotation by users. Author's tools for spelling checking and correcting, thesaurus and style checkers are also supported. This project is a method of enhancing documentation quality and appears to be directed more toward development information. FORTUNE will be tailorable to different development methodologies.

MIDOK

MIDOK (Kammerer [EBE80]) is a documentation management system for the systematic storage, retrieval and reproduction of information. MIDOK is essentially the user interface to documents held either in files or a GOLEM data base. The data base provides extensive searching and keywording functions although a reduced version of MIDOK can omit it. The structure of documentation from the user's view is hierarchical. Information about a particular subject is arranged into a concept tree of increasing specification which has as its leaf level individual documents (or information objects). There is considerable scope for different kinds of document and they may be identified (classified), unidentified, formatted or unformatted. Users may access, update or print documents, concepts or subjects via a menu-type dialogue. An accelerated "chaining" of dialogue commands is also supported.

SID

SID [BRI84] (System Information Data Base) is an automated documentation system for rapid and accurate entry, update and retrieval of documents. SID is based around a relational database. Elements stored in the system include details of the purposes, customer and who is responsible for maintenance in business application components as well as typical development information about component functions, data usage and invocational structures. The system supports retrieval of information as visual tables of contents (VTOC) and hierarchical input process output (HIPO) diagrams and readily displays the information inventory through indices of programs, files, tasks and catalogues of subsystems. As well as helping programmers understand and document an application, SID has been used to quickly identify which personnel have administrative, maintenance and operational responsibilities over which parts of a system. Eventually it is hoped to include in SID run and recovery instructions, file access rights, account restrictions, file retentions and other operations data.

SODOS

SODOS (Software Documentation Support) [HOR86] is a system to support the definition and manipulation of development documents. SODOS is a connecting user interface to an object based model of the Software Life Cycle (SLC) and documents held in a relational DBMS. Documents such as memos, notes, tutorials, formal documents (which depend on the development methodology used), manuals and source code are stored as instances of a DOCUMENT relation. The system allows documents to be created, updated and queried. Hierarchies and interrelationships can be defined between documents. Every document has associated keywords, components (e.g. figures) and a structure (i.e. format). Using the SLC model, SODOS provides for consistency and completeness checks among documentation.

TOOL-BASED DEVELOPMENT AND MAINTENANCE ENVIRONMENTS

Some environments usually consisting of sets of integrated tools are briefly outlined below.

ADA Environment Tool Box

When suggesting future tools to be incorporated into an ADA environment, Saib [SAI83] identifies three major groups: multipurpose, software production and management. Software production tools are requirements processor, specification processor, design analyser, coding assistant, standards checker, compiler, static analyser, linking loader, configuration manager, test assister and verifier. Management activities supported by tools are planning, staffing, directing, organising and status reporting. The major multipurpose tool which integrates the whole proposed environment is a data base manager which controls a single data base for all environment activities. A feature of the connection between design and coding tools in development is the generation of a program skeleton from the design description.

For most maintenance activities, the support tools are the same as for development. Exceptions include diagnostic mechanisms to aid in identifying the source of error symptoms in corrective maintenance. It is suggested that enhancement maintenance should be carried out by modifying first the requirement documents then the specifications and finally the code. This order facilitates identification of the extent of changes early in the enhancement task by comparison of original and modified program skeletons. Good documentation of the entire history of a project including test descriptions, data sets and results appears to be a corner-stone of environment support for maintenance (and development).

DREAM

DREAM [RID81] is a design system for development. The basis of DREAM is DDN (DREAM Design notation), which is a language which allows description of design decisions and their hierarchies and interrelationships. DDN is particularly suited to concurrent system design. DDN text for modules/programs is obviously referred to when coding but can also be used for integration and a type of modelling of the developing application. The DREAM system supports managed access and updating of fragments of DDN script stored in a central data base.

ME2

ME2 [COL85a] is a proposed prototype maintenance engineering environment. The environment consists of three parts; a knowledge base, an integrated toolset, and maintenance personnel. The proposed knowledge base contains a data base of maintenance information. Maintenance personnel interact with this information when performing their tasks. The knowledge base monitors the interaction and is supposed to "learn" from it, creating new implied relationships among the information. It is suggested that the toolset should contain the technical support tools outlined below. To derive a portion of the maintenance information required in the knowledge base, a prototype syntactic analyser for PASCAL has been developed.

Understanding tools : To enable a high level, problem-knowledge domain view (i.e. program requirements information) as well as lower level views of module imports, exports, syntax and semantics and module calling hierarchies.

Modification Management Tools : To link modification requirements to design and to perform software change control management.

Designing and Testing Tools : A ripple effect analyser is suggest and a subsystem for cost-effective regression testing is under investigation.

TOOLS FOR SOURCE CODE MANAGEMENT

Some tools exclusively for the management, access or manipulation of program source are identified below.

AURUM

AURUM (Wagner [EBE80]) is a system for representing visualisations of the structure of an application system. This tool uses application information such as source code, module libraries, cross-reference listings and procedure tracing paths for its analysis but the main emphasis is on displaying the analysis results using a graphics workstation machine and plotter. Application information is derived from other tools present on a mainframe. AURUM allows graphical observation of procedure/module linkages (both static and dynamic) and usage of language structures and data structures within a program. Source languages whose structures can be represented include PASCAL, FORTRAN and COBOL.

CHART and STRUC

CHART and STRUC [THA81] are graphical tools for computer science education. They are written in MIRA-2D and allow visualisation of processes and structures within a PASCAL program. CHART supports viewing the structure of a program by producing structured charts. STRUC shows the evolution of data structures during program execution and appears to be a minor advance on interactive debuggers. CHART is also a relatively simple student program understanding tool and it does not allow any direct program modification.

CONTOUR

CONTOUR [GIM80] is a program formatter for Pascal and C programs. This tool illustrates blocks and the scope of control structures in source code by outlining sections with "contour" lines or boxes. The formatted source is suitable for display on standard terminals unlike many graphical tools.

MAP

MAP [WAR82] is an interactive understanding tool designed to assist maintainers in understanding COBOL programs. MAP takes the source of a COBOL program, generates a data base of source information and allows various views and enquiries on the DB data. A program can be viewed as a structure chart (of CALLed and PERFORMed blocks) or local source code (a paragraph which has been zoomed in on). By displaying relevant lines of source, MAP supports control tracing (following control flow paths) and data tracing (following data aliasing and references). Tracing can be carried out both forward from a particular line number and backward. MAP also contains a scheme for comparing two versions of the same program and reporting differences. To enhance the simple MAP commands a facility for macro definition (called Script files) is available.

SCAN/370 and SUPERSTRUCTURE

SCAN/370 [RIC84] is a tool which can be applied to a COBOL program to produce a source listing containing embedded path data and reports that trace all logic paths within the source and identify dead code. SUPERSTRUCTURE [RIC84] is a more advanced tool which identifies unacceptable flaws in COBOL source (e.g. fall through execution of paragraphs) and rewrites a program using only structured constructs.

Interactive Static Analyser

Bell [BEL84] reports on the results of a beta test on an interactive static analyser and identifies where this tool fits into the overall maintenance environment. The analyser takes COBOL source and stores it in an on-line data base. Through this system three views of a program are possible using structure charts, the source code itself, or source differences (a program version comparison). For each view, selection and tracing of data and control flows is supported. This static analyser primarily aids programmer understanding.

TEXJAX

TEXJAX [RIC84] is an in-house static analyser and documenter for PL/I programs. It scans code and produces complexity metrics, structure charts, module hierarchy charts and annotated source code.

OTHER TOOLS

Tools which are associated with particular application documentation, other than exclusively source code, are summarised below. Some of these paragraphs identify systems which support large areas within application development or maintenance.

AIDES

AIDES [WIL79] (Automated Interactive Design and Evaluation System) is a proposed system to support application design in the form of structure charts of the Structured Design methodology. Two AIDES tools which have been developed are a structure chart graphics package (SCG) and a design quality metric package (DQM). The SCG allows the building of a structure chart on a graphics terminal. It automatically collects design information in a database and supports reproduction of designs. The DQM evaluates a structure chart produced by the SCG in terms of hierarchies of calling modules and module connectivity. Other parts of AIDES include features for automatic chart layout, design quality assurance, automatic test selection and documentation and configuration control.

The Programmer's Apprentice

The Programmer's Apprentice [WAT82] is system to assist in application design and implementation. This system documents an evolving program using two representations; a plan and the program text. Components of the apprentice are a library (plans of common program modules), an analyser (which produces plans from existing source), a drawer (to draw plans), a coder (which generates source from simple plans) and a plan editor (which interfaces to all other components and the user. The idea is to automate code production for simple programs and enable modification of these programs by merely modifying their plans. The system was designed to support development but clearly could be used in maintenance.

CASE

CASE [AME79] is a system development tool to support the functions of document production, module design and code production, module testing, configuration management and management report production. CASE is designed to provide status information for managers as well as various documentation and configuration data for programmers and designers. This system allows information to be entered and edited in a variety of files including text (for general documents), DADA (designs and diagrams), DBD (data base definitions), test descriptions, and CCS (Configuration Control systems) files. Using these files, testing can be directed, overall database schemas and subschemas designed and correct application configuration assured. In this system, program source is entered directly into DADA files. A measure of consistency can be achieved between program design and implementation in terms the functions used and data referenced using a cross file correlator.

DOCU/TEXT

DOCU/TEXT [RIC84] is a system redocumentation tool linked to JCL. It requires the JCL to fit into a fairly rigid format and seems to be most useful when combined with a customised JCL scanner. DOCU/TEXT provides the basis for automated production of application operator manuals.

FRED, ISADORE

FRED [SHI85] is a structured screen based editor and ISADORE is an associated automated reference librarian system. FRED, like other language orientated editors, is used to avoid syntax and context sensitive errors in source code. Error detection is performed by incrementally parsing user input. FRED achieves a degree of language independence in that all language orientated information is derived from description tables which can be adjusted for different dialects or languages. An advanced feature of FRED is that it allows users to find library code and construct interfaces to this code. ISADORE is responsible for managing the library routines whose existence is an attempt to reuse previously tested code. The library routines may be in several different languages and identification is by function. Hence descriptive documentation about routines must also be available in some form.

LIBMAN

LIBMAN [RIC84] is a custom library management system for control of source and load libraries across multiple sites. It is specifically designed to support repair and enhancement of production programs.

MICS

MICS (MVS Integrated Control System) [RIC84] is an operational logging system which gathers information from many sources on problems within an application system. The information is stored in a SAS data base from which reports can be generated.

PECAN

PECAN [REI84] is a program development system generator for algebraic programming languages. PECAN is built on a Brown workstation environment and uses tools including MAPLE, ASH, VT and WILLOW to provide graphical views of an evolving application. This system allows a developer to enter, edit or display a program through a syntax-directed editor, Nassi-Schneiderman flow chart, structured flowgraph (proposed extension) or procedure-connection diagram (proposed extension) view. An application program can also be displayed terms of its symbol table, data types, expressions and flow of control. PECAN uses an incremental compiler to build many of these views so that the analysis of a program is often partially static and partially dynamic. Execution views of variables and stacks provide for debugging and execution monitoring services within PECAN.

RXVP

RXVP (Geiger [EBE80]) is a automated test and documentation system for static analysis of FORTRAN programs. The system first standardises the format of a program and then generates documentation reports containing control flow and data usage information. For large programs, a data base is used to store program information between analysis and report production. The reports produced include a module invocation matrix, cross-reference data (identifying variable definition, use and updating), and a common block usage report. In addition to static analysis, RXVP can aid dynamic analysis. This is achieved by modifying the program to report reaching of branches during execution. Running the "instrumented" program in a test bed situation results in execution frequencies of parts of the program for the test data used. Thus, the completeness of test coverage can be assessed.

SADAT

SADAT [VOG80] is an automated testing tool applicable to single FORTRAN modules. This tool combines the functions of static analysis, dynamic analysis, test case generation and path predicate calculation. These functions are integrated via a common database and directed by a user. The database is generated through static analysis which also produces a program graph, reduced program graph and reports on structural deficiencies of the source. From the database, a near optimal group of test cases ensuring at least one execution of each node of the program graph can be derived and, by using symbolic execution, infeasible paths through the program are identified (path predicate calculation). Dynamic analysis is performed by automatically instrumenting the program (again using data base information), recompiling, loading with a monitoring subroutine and execution. Reports produced by dynamic analysis include frequencies of path and loop execution for given test data.

SAMOS tools

SAMOS (Software Adaption and Maintenance Organisation System) applies an integrated toolset to the task of firstly adapting mainly FORTRAN software to different environments, and then managing maintenance of the adapted systems. The toolset includes QUODOS, EDIERE, PROTRAN (a compiler construction tool for dialect/language translation), COMPARISON (for delta abstraction and analysis, see QUODOS), and ERZEUGE. Some of these tools are briefly described below. For further information see the SAMOS section of [EBE80] by Luegger, Fromm, Goecke and Roitzsch.

QUODOS and SCCS : SCCS (Source Code Control System) [ALL84] is a model for the modification management of text (source, documentation or JCL). Within this model original text exists along with chains of deltas (changes) representing modifications which have been made over time. Information about the author and creation date are held about each delta. An identification system defines modified versions of text in terms of release numbers (possibly major development phases) and within releases, level numbers. From the information a particular version of, say, a program, can be generated.

QUODOS is an automated tool for a similar control system to SCCS. There is a different approach in QUODOS with reverse deltas being stored, recording what changes to the current up-to-date version of text generate earlier versions. Benefits from this scheme include better delta synchronisation as line numbers can be more easily used to define changes. In SCCS, line numbering is complicated by renumbering when inserting new text.

EDIERE : EDIERE is a context editor designed to cope with maintenance and adaptation requirements for large programs. A generalised I/O interface in

EDIERE allows it to access diverse types of files with different organisations and structures. To enable editing and alternative "views" of these files EDIERE has a command structure based on SNOBOL4 pattern matching and allows extensive use of command macros. When editing source code, programming language orientated macros can pick out block structures, extended control structures and other source objects. EDIERE is thus useful for source analysis, systematic modification and source restructuring as well as standard editing. The static source analysis appears to be more tailored to control flow analysis than extensive data abstraction, but this may be because EDIERE has mainly been used in adapting FORTRAN systems to different machines.

ERZEUGE : ERZEUGE is an organisation tool for JCL and is basically a macro processor with some interactive features. It allows control of the modification process through parameterisation of change procedures and coordination of other tools. ERZEUGE macros can be used to abbreviate JCL commands or to extend the JCL to, say, automatically link compiling, loading and testing of a generalised module. This tool, based on interpreted macros, may be redundant as it's facilities are already present in some operation systems (e.g. UNIX)

SAS

Deutsch [DEU81] describes a general class of automated test assistance tools called Source Analyser Systems (SASs). SASs provide facilities for measuring performance of software and effectiveness of test cases. The five basic functions are :

1. Source analysis and database creation.
2. Generation of reports from static analysis describing software control and data features and identifying potential problems.
3. Insertion of software probes (instrumentation) for the collection of execution information.
4. Analysis and reporting of test results.
5. Generation of test assistance reports.

An integrated SAS would form a test environment useful in both development and maintenance. RXVP is an example of a complete SAS but most tools provide only some of test functions given above (e.g. CHART and TEXJAX).

APPENDIX 3 - A REDUCED COBOL LANGUAGE

APPENDIX 3A - OMISSIONS FROM STANDARD COBOL

A module by module account of disparities between the COBOL standard defined in [COB81] and Appendix 3A is given below. Abbreviations used are :

ID	for	IDENTIFICATION DIVISION
ED	for	ENVIRONMENT DIVISION
DD	for	DATA DIVISION
PD	for	PROCEDURE DIVISION

1. The Nucleus.

In an effort to remove machine configuration dependent aspect from the COBOL standard being produced, implementor-names which link objects such as files to external devices were omitted (except in the ASSIGN TO clause of file-control-entries). This led to the removal of mnemonic-names which are heavily related to implementor-names via the SPECIAL-NAMES paragraph. ALPHABET and SYMBOLIC CHARACTER statements of this paragraph and partial array references within a program were also omitted to simplify analysis of COBOL symbols and statements. The STOP verb is similar to implementor-names in that it defines an external execution link and so it too was removed. Level 66 (renames) variables were deleted because they are seldom used [TOR79]. Some of these changes imply other deletions in the I/O modules, but nucleus specific omissions are :

PROGRAM COLLATING SEQUENCE statements in the
    OBJECT-COMPUTER paragraph of the
    CONFIGURATION SECTION

Implementor-names, ALPHABET, or SYMBOLIC
    CHARACTERS statements in the SPECIAL-NAMES
    paragraph of the CONFIGURATION SECTION

Level 66 RENAMES statements in data-description-
    entries of the DD

FROM mnemonic-name clause for the ACCEPT verb in
    the PD

UPON clause for the DISPLAY verb in the PD

Mnemonic-name TO ON/OFF format for the SET verb
    in the PD

STOP verb in the PD (used to temporarily suspend
    program execution)

Reference to partial arrays by

```
subscript ( leftmost-character-position :
                                     [length] )
```

in any identifier

2. The I/O Modules.

The omissions of implementor, mnemonic and alphabet-names from the nucleus mean removal of some clauses in file-control-entries FDs and PD statements. The ASSIGN TO clause of file-control-entries remains because it is thought to be commonly used. In analysis, no action, other than identifying and storing the group of implementor-names in each ASSIGN clause will be carried out. As the PADDING CHARACTER clause of file-control-entries is peculiar to sequential I/O and not present in ANS 740, it was deleted. The DATA RECORD IS clause of FDs was also removed as it is redundant and marked for removal in later COBOL standards [COB81]. Specific omissions are :

PADDING CHARACTER IS clause in a file-control-entry
in the ED

RECORD DELIMITER IS clause in a file-control-entry
in the ED

CODE-SET clause in a file-control-entry in the ED

DATA RECORDS clause of FDs in the DD

VALUE OF clause in FDs in the DD

CODE-SET clause in FDs in the DD

ADVANCING mnemonic-name clause for the WRITE verb
in the PD

Out-of-line exception and recovery facilities were considered extras to COBOL which although used in many modules could be ignored in a prototype SSCA. Specific omissions are :

I-O-CONTROL paragraph of the INPUT-OUTPUT SECTION
(also part of the Sort/Merge module)

DECLARATIVES to END-DECLARATIVES block preceding
normal PD sections and paragraphs (also
part of the Debug and Report Writer modules)

USE verb in the PD (also part of the Inter-Program-
Communication, Debug and Report Writer modules)

3. <u>The Inter-Program Communication Module</u>.

Some inter-program communication features and the system for nesting programs within one compilable unit were omitted. This is because the prototype SSCA is concerned with building a database reflecting just one COBOL program. Limited communication facilities remain. For instance, the CALL statement is available and data can be sent or received via the LINKAGE SECTION. BY REFERENCE and BY CONTENT clauses of the CALL were left out because they are not present in ANS 74. The EXIT PROGRAM statement in the PD will be treated in a similar manner to the STOP RUN statement (i.e. in terms of the executing program it is as if it had been CALLed). Specific omissions are :

END-PROGRAM-HEADER entry

COMMON or INITIAL clauses of the PROGRAM-ID paragraph of the ID

EXTERNAL or GLOBAL clauses in FDs in the DD

EXTERNAL or GLOBAL clauses in data-description-entries in the DD

BY REFERENCE or BY CONTENT clauses for the CALL verb in the PD

4. The Sort/Merge Module.

The whole module is omitted. This includes removal of the following features :

Sort-merge-file-description-entries (SDs) in the FILE SECTION

MERGE verb in the PD

RELEASE verb in the PD

RETURN verb in the PD

SORT verb in the PD

5. The Source Text Manipulation Module.

The whole module is omitted. This includes removal of the following features :

COPY statement anywhere in the program

text-name IN/OF library-name qualification clauses in any statement

REPLACE verb in the PD

REPLACE OFF verb in the PD

6. The Debug Module.

The whole module is omitted. This includes removal of the following features :

WITH DEBUGGING MODE clause in the SOURCE-COMPUTER paragraph of the CONFIGURATION SECTION

7. The Report Writer Module.

The whole module is omitted. This includes removal of the following features :

REPORT SECTION or Report-file-description-entries (RDs) in the DD

IN/OF report-name qualification of an identifier, LINE-COUNTER or PAGE-COUNTER

GENERATE verb in the PD

INITIATE verb in the PD

SUPPRESS PRINTING verb in the PD

TERMINATE verb in the PD

8. The Communication Module.

The whole module is omitted. This includes removal of the following features :

COMMUNICATION SECTION or communication-description-entries (CDs) in the DD

IN/OF cd-name qualification of a condition or identifier

cd-name MESSAGE COUNT format for the ACCEPT verb in the PD

DISABLE verb in the PD

ENABLE verb in the PD

PURGE verb in the PD

RECEIVE verb in the PD

SEND verb in the PD

APPENDIX 3B - LANGUAGE DESCRIPTION

This is a reduced version of COBOL based on the Draft Proposed Revised X3.23 A.N.S. Programming Language COBOL, Sept. 1981 [COB81]

Meta Language Description

```
1) Object Grouping : { object1 object2 } - normal gp.
                     [ object1 object2 ] - optional gp.
2) Iteration of a group : { object1 }... - indicates 1
                                           or more object1s
                          [ object1 ]... - indicates 0
                                           or more object1s
3) Case (one or other of objects) : {| object1 |}
                                     | object2 |
                 - either object1 or object2
4) Case entries over more than one line :
        {| object1     |} - indicates object1 or
         | / object2 \ |          object2 object3
         | \ object3 / |
5) Mandatory objects : object1
```

Format for the Overall Program

```
identification-division

[ environment-division ]

data-division

procedure-division
```

Format for the Identification Division

```
IDENTIFICATION DIVISION.

PROGRAM-ID.     program-name.

[ AUTHOR.       [ comment-entry ]... ]

[ INSTALLATION.      [ comment-entry ]... ]

[ DATE-WRITTEN.      [ comment-entry ]...   ]

[ DATE-COMPILED.     [ comment-entry ]... ]

[ SECURITY.       [ comment-entry ]... ]
```

Format for the Environment Division

```
ENVIRONMENT DIVISION.

[ CONFIGURATION SECTION.

[     SOURCE-COMPUTER.     [ computer-name . ]]

[     OBJECT-COMPUTER.     [ computer-name
        [ SEGMENT-LIMIT IS segment-number ].]]

[     SPECIAL-NAMES.
        [ CURRENCY SIGN IS literal-4 ]
        [ DECIMAL-POINT IS COMMA ] .   ]

[ INPUT-OUTPUT SECTION.

  FILE-CONTROL.
   { file-control-entry }...
```

Format for file-control-entry

```
SELECT [ OPTIONAL ] file-name-1

   ASSIGN TO { implementor-name-1 }...

   [ RESERVE integer-1 [| AREA   |] ]
                       | AREAS  |

   [ [ ORGANIZATION IS] {| SEQUENTIAL |} ]
                         | RELATIVE   |
                         | INDEXED    |

   [ BLOCK CONTAINS [ integer-2 TO ] integer-3
                       {| RECORDS    |} ]
                        | CHARACTERS |

   [ ACCESS MODE IS {| SEQUENTIAL |} [ RELATIVE KEY
                     | RANDOM     |
                     | DYNAMIC    |
                                IS data-name-1] ] ]

   [ RECORD KEY IS data-name-1 ]

   [ ALTERNATE RECORD KEY IS data-name-2
                          [ WITH DUPLICATES ] ]...

   [ FILE STATUS IS data-name-2 ].
```

```
Format for the Data Division
============================

   DATA DIVISION.

   [ FILE SECTION.
        [ file-description-entry
               { record-description-entry }... ]... ]

   [ WORKING-STORAGE SECTION.
        [| 77-level-description-entry |]... ]
         | record-description-entry   |

   [ LINKAGE SECTION.
        [| 77-level-description-entry |]... ]
         | record-description-entry   |


Format for file-description-entry
=================================

   FD   file-name-1

   [ BLOCK CONTAINS [ integer-1 TO ] integer-2
                                    {| RECORDS    |} ]
                                     | CHARACTERS |

   [ RECORD
   {| CONTAINS integer-3 [TO integer-4] CHARACTERS |} ]
    | / IS VARYING IN SIZE [ [FROM integer-4]    \ |
    | | [ TO integer-5 ] CHARACTERS ]            | |
    | \ [ DEPENDING ON data-name-1 ]             / |

   [ LABEL {| RECORD IS   |} {| STANDARD |} ]
            | RECORDS ARE |   | OMITTED  |

   [ LINAGE IS {| data-name-4 |} LINES
                | integer-8   |
        [ WITH FOOTING AT {| data-name-5 |} ]
                           | integer-9   |
        [ LINES AT TOP {| data-name-6 |} ]
                        | integer-10  |
        [ LINES AT BOTTOM {| data-name-7 |} ] ]
                           | integer-8   |

   [ [ ACCESS MODE IS ] {| SEQUENTIAL |}
                         | RANDOM     |
                         | DYNAMIC    |
                    [ RELATIVE KEY IS data-name-8 ] ]
   [ RECORD KEY IS data-name-9 ]

   [ ALTERNATE RECORD KEY IS data-name-10
                    [ WITH DUPLICATES ] ]...

   [ FILE STATUS IS data-name-11 ] .
```

Format for record-description-entry
===================================

```
   record-description-entry ::= levelled set of

                                data-description-entries

   77-level-description-entry ::=

                                record-description-entry

                                 where level-number is 77
```

Format for data-description-entry
=================================

```
{| / 88 condition-name-1 {| VALUE IS    |}             \ |}
 | |                       | VALUES ARE |              | |
 | |   { literal-1 [ {|THROUGH|} data-name-4 ] }... .| |
 | \                   |THRU    |                     / |
 |                                                      |
 | / level-number [| data-name-5 |]                   \ |
 | |                | FILLER      |                    | |
 | |                                                   | |
 | |    [ REDEFINES data-name-6  ]                     | |
 | |                                                   | |
 | |    [ / {| PICTURE |} IS character-string \ ]      | |
 | |      \  | PIC     |                      /        | |
 | |                                                   | |
 | |    [ / [ USAGE IS ] {| COMPUTATIONAL |} \ ]       | |
 | |      |              | COMP          |  |          | |
 | |      |              | DISPLAY       |  |          | |
 | |      \              | INDEX         |  /          | |
 | |                                                   | |
 | |    [ / [ SIGN IS ] {| LEADING  |}          \ ]    | |
 | |      |             | TRAILING |           |      | |
 | |      \              [ SEPARATE CHARACTER ] /      | |
| |                                                  | |
| |    [ / OCCURS integer-1 [TO integer-2] TIMES\ ]  | |
| |      |    [DEPENDING ON data-name-7]      |      | |
| |      |    [ / {| ASCENDING  |}          \ |      | |
| |      |      |  | DESCENDING |          | |      | |
| |      |      \ KEY IS {data-name-3}... ].../ |    | |
| |      \    [ INDEXED BY {index-name-1}...} ] /    | |
| |                                                  | |
| |    [ / {| SYNCHRONIZED |} [| LEFT  |] \ ]        | |
| |      \  | SYNC         |   | RIGHT |  /          | |
| |                                                  | |
| |    [ / {| JUSTIFIED |} RIGHT \ ]                 | |
| |      \  | JUST      |        /                   | |
| |                                                  | |
| |    [ BLANK WHEN ZERO ]                           | |
| |                                                  | |
| \    [ VALUE IS literal-2 ] .                      / |
```

Format for the Procedure Division

```
   PROCEDURE DIVISION [ USING {data-name-1}... ] .

      { [ section-name SECTION [segment-number]. ]
             [ paragraph-name. [sentence]... ]... }...


       sentence ::= set-of-statements
```

Format for COBOL Verbs

```
ACCEPT identifier-1 [ FROM {| DATE        |} ]
                           | DAY         |
                           | DAY-OF-WEEK |
                           | TIME        |


Format 1  ADD {| identifier-1 |}...
               | literal-1    |

   TO { identifier-2 [ROUNDED] }...

   [ ON SIZE ERROR imperative-statement-1 [ END-ADD ] ]


Format 2  ADD {| identifier-1 |}... TO {| identifier-2 |}
               | literal-1    |         | literal-2    |

   GIVING { identifier-3 [ ROUNDED ] }...

   [ ON SIZE ERROR imperative-statement-1 [ END-ADD ] ]


Format 3  ADD {| CORRESPONDING |} identifier-1
               | CORR          |

   TO identifier-2 [ROUNDED]

   [ ON SIZE ERROR imperative-statement-1 [ END-ADD ] ]


CALL {| identifier-1 |} [ USING { data-name-2 }... ]
      | literal-1    |

   [ ON OVERFLOW imperative-statement-1 [ END-CALL ] ]
```

```
CANCEL {| identifier-1 |}
        | literal-1    |


CLOSE { file-name-1 [| / {| REEL |}[FOR REMOVAL] \ |] }...
                     | \   | UNIT |              / |
                     | / WITH {| NO REWIND |}    \ |
                     | \       | LOCK      |     / |


COMPUTE { identifier-1 [ROUNDED] }...
   = arithmetic-expression-1

   [ ON SIZE ERROR imperative-statement-1
                                       [ END-COMPUTE ] ]


CONTINUE


DELETE file-name-1 RECORD

   [ INVALID KEY imperative-statement-1 [ END-DELETE ] ]


DISPLAY {| identifier-1 |}... [ WITH NO ADVANCING ]
         | literal-1    |


Format 1  DIVIDE {| identifier-1 |}
                  | literal-1    |
   INTO { identifier-2 [ROUNDED] }...

   [ ON SIZE ERROR imperative-statement-1
                                        [ END-DIVIDE ] ]


Format 2  DIVIDE {| identifier-1 |} {| INTO |}
                  | literal-1    |   | BY   |
   {| identifier-2 |}
    | literal-2    |

   GIVING { identifier-3 [ ROUNDED ] }...

   [ ON SIZE ERROR imperative-statement-1
                                        [ END-DIVIDE ] ]
```

```
Format 3  DIVIDE {| identifier-1 |} {| INTO |}
                 | literal-1    |    | BY   |

   {| identifier-2 |} GIVING identifier-3 [ ROUNDED ]
    | literal-2    |

   REMAINDER identifier-4

   [ ON SIZE ERROR imperative-statement-1
                                       [ END-DIVIDE ] ]


EVALUATE {| identifier-1 |}... { { WHEN
           | literal-1    |
           | expression-1 |
           | TRUE         |
           | FALSE        |
   {| ANY                                              |}...
    | condition-1                                      |
    | TRUE                                             |
    | FALSE                                            |
    |                                                  |
    | / [NOT] {{| identifier-2             |}       \  |
    | |          | literal-2                |       |  |
    | |          | arithmetic-expression-2  |       |  |
    | |     [ / {| THROUGH |}                 \ ]} |  |
    | |       |  | THRU    |                  |    |  |
    | |       |   {| identifier-3            |}|   |  |
    | \       \    | arithmetic-expression-3 | /   /  |
    |                                                  |
        }... imperative-statement-1 }...

   [ WHEN OTHER imperative-statement-2 ]

   [END-EVALUATE]


EXIT.


EXIT PROGRAM


Format 1  GO TO procedure-name-1


Format 2  GO TO {procedure-name-1}... DEPENDING ON
                                         identifier-1
```

```
IF condition-1 THEN {| {statement-1}... |}
                     | NEXT SENTENCE    |

   [| ELSE {statement-2}... [END-IF] |]
    | ELSE NEXT SENTENCE             |
    | END-IF                         |


INITIALIZE {identifier-1}...

   [ REPLACING {| ALPHABETIC          |}
                | ALPHANUMERIC        |
                | NUMERIC             |
                | ALPHANUMERIC-EDITED |
                | NUMERIC-EDITED      |
                           DATA BY {| identifier-2 |} ]
                                    | literal-2    |


Format 1  INSPECT identifier-1

   [ TALLYING { identifier-2 FOR
     {| / CHARACTERS [{| BEFORE |}          \ |}...}...]
      | |              | AFTER  |           | |
      | |     INITIAL {| identifier-3 |}]...| |
      | \              | literal-3    |     / |
      |                                       |
      | / {| ALL     |} { {| identifier-4 |}\ |
      | |  | LEADING |     | literal-4    | | |
      | |      [ {| BEFORE |} INITIAL       | |
      | |         | AFTER  |                | |
      | |        {| identifier-5 |} }...}...| |
      | \         | literal-5    |          / |
      |                                       |

   [ REPLACING
     {| / CHARACTERS BY {| identifier-6 |}       \ |}...]
      | |                | literal-6    |        | |
      | |    [ {| BEFORE |}  INITIAL             | |
      | |       | AFTER  |                       | |
      | |        {| identifier-7 |}]...          | |
      | \         | literal-7    |               / |
      |                                            |
      | / {| ALL     |} { {| identifier-8 |} BY  \ |
      | |  | LEADING |     | literal-8    |      | |
      | |  | FIRST   |                           | |
      | |    {| identifier-9 |} [ {| BEFORE |}   | |
      | |     | literal-9    |     | AFTER  |    | |
      | |    INITIAL {| identifier-10 |} ]...}...| |
      | \             | literal-10    |          / |
Format 2  INSPECT identifier-1 CONVERTING

   {| identifier-2 |} TO {| identifier-3 |}
    | literal-2    |      | literal-3    |

   [ {| BEFORE |} INITIAL {| identifier-4 |} ]...
      | AFTER  |           | literal-4    |


Format 1  MOVE {| identifier-1 |} TO { identifier-2 }...
                | literal-1    |


Format 2  MOVE {| CORRESPONDING |}  identifier-1
                | CORR          |

   TO identifier-2


Format 1  MULTIPLY {| identifier-1 |}
                    | literal-1    |

   BY {identifier-2 [ROUNDED]}...

   [ ON SIZE ERROR imperative-statement-1
                                      [ END-MULTIPLY ] ]


Format 2  MULTIPLY {| identifier-1 |}
                    | literal-1    |

   BY {| identifier-2 |}
       | literal-2    |
             GIVING { identifier-3 [ ROUNDED ] }...

   [ ON SIZE ERROR imperative-statement-1
                                      [ END-MULTIPLY ] ]


OPEN {| INPUT { file-name-1 [ WITH NO REWIND ] }... |}...
      | OUTPUT { file-name-2 [ WITH NO REWIND ] }...|
      | I-O    { file-name-3 }...                   |
      | EXTEND { file-name-4 }...                   |


Format 1  PERFORM [ procedure-name-1
                     [ {| THROUGH |} procedure-name-2 ] ]
                        | THRU    |


   [ imperative-statement-1 END-PERFORM ]
```

```
Format 2  PERFORM [ procedure-name-1
                    [ {| THROUGH |} procedure-name-2 ] ]
                       | THRU    |

   [| / {| identifier-1 |} TIMES                   \ |]
    | \  | integer-1    |                          / |
    | / [ WITH TEST {| BEFORE |} ] UNTIL condition-1\ |
    | \             | AFTER  |                     / |

   [ imperative-statement-1 END-PERFORM ]


Format 3  PERFORM [ procedure-name-1
                    [ {| THROUGH |} procedure-name-2 ] ]
                       | THRU    |

   [ WITH TEST {| BEFORE |} ]
               | AFTER  |

   VARYING {| identifier-1 |} FROM {| identifier-2 |}
            | index-name-1 |        | index-name-2 |
                                    | literal-2    |
      BY {| identifier-3 |} UNTIL condition-1
          | literal-3    |
      [ AFTER {| identifier-4 |} FROM {| identifier-5 |}
               | index-name-4 |        | index-name-5 |
                                       | literal-5    |
           BY {| identifier-6 |} UNTIL condition-2 ]...
               | literal-6    |

   [ imperative-statement-1 END-PERFORM ]


READ filename-1 [ NEXT ] RECORD [ INTO identifier-1 ]

   [ KEY IS data-name-2 ]

   [ {| AT END      |} imperative-statement-1
      | INVALID KEY |
                                         [END-READ] ]


REWRITE record-name-1 [ FROM identifier-2 ]

   [ INVALID KEY imperative-statement-1 [END-REWRITE] ]
```

```
Format 1  SEARCH identifier-1

   [ VARYING {| identifier-2 |} ]
              | index-name-2 |

   [ AT END imperative-statement-1 ]

   { WHEN condition-1 {| imperative-statement-2 |} }...
                       | NEXT SENTENCE          |

   [END-SEARCH]


Format 2  SEARCH ALL identifier-1

   [ AT END imperative-statement-1 ]

   WHEN { {| / data-name-2 IS {| EQUAL TO |}    \ |} }
           | |                 | =        |     | |
           | |                                  | |
           | |     {| identifier-3          |}  | |
           | |      | literal-3             |   | |
           | \      | arithmetic-expression-3|  / |
           |                                      |
           | condition-name-1                     |

     [ AND {| / data-name-4 IS {| EQUAL TO |}    \ |} ]...
            | |                 | =        |     | |
            | |                                  | |
            | |     {| identifier-5          |}  | |
            | |      | literal-5             |   | |
            | \      | arithmetic-expression-5|  / |
            |                                      |
            | condition-name-2                     |

   {| imperative-statement-2 |}
    | NEXT SENTENCE          |

   [END-SEARCH]


Format 1  SET {| index-name-1 |}... TO {| index-name-2 |}
               | identifier-1 |         | identifier-2 |
                                        | integer-2    |


Format 2  SET { index-name-1 }...

   {| UP BY   |} {| index-name-2 |}

    | DOWN BY |   | integer-2    |
START

   file-name-1 [ KEY IS {| EQUAL TO     |} data-name-1 ]
                        | =             |
                        | GREATER THAN  |
                        | >             |
                        |NOT LESS THAN  |
                        | NOT <         |

   [ INVALID KEY imperative-statement-1 [END-START] ]


STOP RUN


STRING { {| identifier-1 |}...
          | literal-1    |

   DELIMITED BY {| identifier-2 |} }...
                | literal-2     |
                | SIZE          |

   INTO identifier-3 [ WITH POINTER identifier-4 ]

   [ ON OVERFLOW imperative-statement-1 [END-STRING] ]


Format 1  SUBTRACT {| identifier-1 |}...
                   | literal-1    |

   FROM { identifier-2 [ROUNDED] }...

   [ ON SIZE ERROR imperative-statement-1
                                        [ END-SUBTRACT ] ]


Format 2  SUBTRACT {| identifier-1 |}...
                   | literal-1    |

    FROM {| identifier-2 |}
          | literal-2    |

   GIVING { identifier-3 [ ROUNDED ] }...

   [ ON SIZE ERROR imperative-statement-1
                                        [ END-SUBTRACT ] ]
```

```
Format 3  SUBTRACT {| CORRESPONDING |} identifier-1
                    | CORR          |

   FROM identifier-2 [ROUNDED]

   [ ON SIZE ERROR imperative-statement-1
                                           [ END-SUBTRACT ] ]


UNSTRING identifier-1
   [ DELIMITED BY [ALL] {| identifier-2 |}
                         | literal-2    |
                     [ OR [ALL] {| identifier-3 |} ]...]
                                 | literal-3    |

   INTO { identifier-4 [ DELIMITER IN identifier-5 ]

                    [ COUNT IN identifier-6 ] }...

   [ WITH POINTER identifier-7 ]

   [ TALLYING IN identifier-8 ]

   [ ON OVERFLOW imperative-statement-1 [END-UNSTRING] ]


Format 1  WRITE record-name-1 [ FROM identifier-2 ]
   [ {| BEFORE |} ADVANCING
      | AFTER  |
              {| / {| identifier-3 |} {| LINE  |} \ |} ]
               | \  | integer-3    |   | LINES |  / |
               |                                    |
               | PAGE                               |

   [ AT {| END-OF-PAGE |} imperative-statement-1
         | EOP         |
                                         [END-WRITE] ]


Format 2  WRITE record-name-1 [ FROM identifier-2 ]

   [ INVALID KEY imperative-statement-1 [END-WRITE] ]
```

Qualification Formats

Identifiers

```
   data-name-1 [ {| IN |} data-name-2 ]...
                  | OF |
                                [ {| IN |} file-name-1 ]
                                   | OF |

    [ ( {| integer-1                                   |}
         |                                             |
         | / {| data-name-2  |} [ {| + |} integer-2 ] \|
         | \  | index-name-1 |     | - |              /|

      [ , {| integer-3                          |}... ) ]
           |                                    |
           | / {| data-name-3  |}             \ |
           | |  | index-name-2 |              | |
           | |         [ {| + |} integer-4 ]  | |
           | \            | - |               / |
```

Paragraphs

```
   paragraph-name [ {| IN |} section-name ]
                     | OF |
```

Linage-Counters

```
   LINAGE-COUNTER {| IN |} file-name
                   | OF |
```

Conditions

```
   condition-name-1 [ {| IN |} data-name-2 ]...
                       | OF |
                                [ {| IN |} file-name-1 ]
                                   | OF |

   [ ( {| integer-1                                   |}
        |                                             |
        | / {| data-name-2  |} [ {| + |} integer-2 ] \|
        | \  | index-name-1 |     | - |              /|

      [ , {| integer-3                          |}... ) ]
           |                                    |
           | / {| data-name-3  |}             \ |
           | |  | index-name-2 |              | |
           | |         [ {| + |} integer-4 ]  | |
           | \            | - |               / |
```

Miscellaneous Formats

Relational Conditions

```
 {| identifier-1           |} IS [NOT] {| GREATER THAN|}
  | literal-1              |            | >           |
  | arithmetic-expression-1|            | LESS THAN   |
                                        | <           |
                                        | EQUAL TO    |
                                        | =           |

                     {| identifier-2            |}
                      | literal-2               |
                      | arithmetic-expression-2 |

      [ {| AND |} [NOT] {| GREATER THAN |}
         | OR  |         | >            |
                         | LESS THAN    |
                         | <            |
                         | EQUAL TO     |
                         | =            |

                     {| identifier-3            |} ]...
                      | literal-3               |
                      | arithmetic-expression-3 |
```

Simple Conditions

```
   [NOT] {| / identifier-1 IS                        \ |}
          | |             [NOT] {| NUMERIC        |} | |
          | |                    | APHABETIC      |  | |
          | |                    | APHABETIC-LOWER|  | |
          | \                    | APHABETIC-UPPER|  / |
          |                                            |
          | / arithmetic-expression-1 IS             \ |
          | |                   [NOT] {| POSITIVE |} | |
          | |                          | NEGATIVE |  | |
          | \                          | ZERO     |  / |
          |                                            |
          | condition-name-1                           |
```

Complex Conditions

```
   [NOT ( ]... {| simple-condition-1     |} [ ) ]...
                | relational-condition-1 |

       { {| AND |} [NOT ( ]...
          | OR  |
               {| simple-condition-2     |} [ ) ]... }...
                | relational-condition-2 |
```

General Condition

```
   {| simple-condition-1     |}
    | relational-condition-1 |
    | complex-condition-1    |
```

APPENDIX 4 - COBOL SOURCE CODE INFORMATION

APPENDIX 4A - ENTITIES AND ATTRIBUTES FOR INFORMATION IN A COBOL PROGRAM

Abbreviations.

general abbreviations : Y|X - either Y or X

{X} - any number of X tokens

A{X}B - between A and B X tokens

[X] - optional; specified for relationships only.

descriptors : id - general identifier

ln - source line number

fg - flag

tokens : co condition, cu condition-usage,

du data-usage, fi file, fv file-variable,

il identifier-lit, pa paragraph,

pr program, se section,

sg statement-group, so source,

st statement, va variable,

v8 variable-88.

storage characteristics :

c? - 1 occurrence of ? characters

xc? - x occurrences of ? characters

i? - 1 occurrence of a ? byte integer

xi? - x occurrences of ? byte integers

```
          Token pointers are assumed to have
          storage characteristic i2.
     relationships :    --   one to one
                        -->  one to many
                        <--  many to one
                        <--> many to many
```

LANGUAGE DEFINITION ENTITIES.

```
ENTITY-NAME : program                    token : pr
Description : Records general program information
  from the ID and Configuration Section of the ED.
  Pd-using identifies linkage variables.
Instances : One only
Attributes : pr-name(pr,c31),
  source-computer(id,c31), object-computer(id,c31),
  segment-limit(i2), currency-sign(c1),
  decimal-point(fg,c1).
Relationships : [pd-using(pr --> va)],
  sections-or-paras(pr --> se|pa).

Implementation Attributes : pd_using(0{pva,i2}5).
Relation : program( pr_name = c31,
  source_comp = c31, object_comp = c31,
  seg_limit = i2, currency_sn = c1, decimal_pt = c1,
  pd_using_1 = i2, pd_using_2 = i2, pd_using_3 = i2,
  pd_using_4 = i2, pd_using_5 = i2 )
```

```
--------------------------------------------------------
ENTITY-NAME : file                          token : fi
Description : Records definition information about
  particular files. Contains file-control-entry
  (ED) and file-description-entry (DD) data as
  well as links to record-description-entries.
Instances : One per file
Attributes : fi-name(fi,c31),
  fi-control-position(ln,i2), fd-position(ln,i2),
  optional(fg,c1), implementor(id,c31),
  organization(S|R|I,c1), block-size(2i2),
  block-measure(C|R,c1), rec-size(2i2),
  rec-varying(fg,c1), access(S|R|D,c1),
  alt-duplicates(fg,c1).
Relationships : [rec-size-depend(fi <-- va)],
  [file-status(fi <-- va)],
  data-record(fi --> va),
  [(Sequential Files  linage-size(fi <-- va|il),
   linage-foot(fi <-- va|il),
   linage-top(fi <-- va|il),
   linage-bot(fi <-- va|il))],
  [(Relative Files rel-key(fi <-- va))],
  [(Indexed Files rec-key(fi -- va),
   alt-key(fi -- va))],
  fv-file(fi --> fv), [du-data-updated(fi --> du)],
  [du-data-used(fi <--> du)].
```

Implementation Attributes : fi_num(i2),
  rec_size_dep(pva,i2), file_status(pva,i2),
  data_record(pva,i2), rel_key(pva,i2),
  rec_key(pva,i2), alt_key(pva,i2).

Relation : file(fi_num = i2, fi_name = c31,
  fic_position = i2, fd_position = 12,
  optional = c1, implementor = c31,
  organization = c1, block_size_a = i2,
  block_size_b = i2, block_meas = c1,
  rec_size_a = i2, rec_size_b = i2, rec_varying = c1,
  rec_size_dep = i2, access = c1, file_status = i2,
  data_record = i2, rel_key = i2, rec_key = i2,
  alt_key = i2, alt_dups = c1)

ENTITY-NAME : **variable** token : va

Description : Records data-description information
  from non-88 variables defined in all sections of
  the DD.

Instances : Many

Attributes : va-name(va,c31), level(i2),
  def-position(ln,i2), usage(C|D|I,c1),
  def-section(F|W|L,c1), occurs-asc({fg,c1}),
  picture(id,c30).

Relationships : [redefines(va --> va)],
  [va-va-within(va <--> va)],

```
  [(Occurs occurs-a(va <-- il), occurs-b(va <-- il),
     occurs-depend(va <-- va), occurs-key(va <--> va),
     occurs-index(va <--> va))],
  [(Non-occurs Non-redefinition value(va <-- il))],
  [pd-using-data(va <-- pr)],
  [rec-size-depend(va --> fi)],
  [file-status(va --> fi)], [data-record(va <-- fi)],
  [lineage-size(va --> fi)], [linage-foot(va --> fi)],
  [linage-top(va --> fi)], [linage-bot(va --> fi)],
  [rel-key(va --> fi)], [rec-key(va -- fi)],
  [alt-key(va -- fi)], [v8-va-switch(va --> v8)],
  [v8-value-b(va <--> v8)], [co-data-used(va <--> co)],
  [du-data-updated(va --> du)],
  [du-data-used(va <--> du)].

Implementation Attributes : va_num(i2),
  redefines(pva,i2), va_va_within(pva,i2),
  occurs_a(pil,i2), occurs_b(pil,i2),
  occurs_dep(pva,i2), value(pil,i2).
Relation : variable( va_num = i2, va_name = c31,
  level = i2, def_position = i2, usage = c1,
  def_section = c1, redefines = i2, va_va_within = i2,
  occurs_a = i2, occurs_b = i2, occurs_dep = i2,
  picture = c30, value = i2 )

----------------------------------------------------

ENTITY-NAME : variable-88                token : v8
```

Description : Records data-description information
  from 88 variables defined in all sections of the
  DD.

Instances : Many

Attributes : v8-name(v8,c31).

Relationships : v8-value-a(v8 <--> il),
  [v8-value-b(v8 <--> il|va)],
  v8-va-switch(v8 <-- va), [co-data-used(v8 <--> co)].

Implementation Attributes : v8_num(i2),
  v8_va_switch(pva,i2).

Relation : variable_88(v8_num = i2, v8_name = c31,
  v8_va_switch = i2)

ENTITY-NAME : **file-variable**   token : fv

Description : Records LINAGE-COUNTER information for
  sequential LINAGE files and implicit file pointers
  for all files.

Instances : Many.

Attributes : fv-type(L|P,c1).

Relationships : fv-file(fv <-- fi),
  [co-data-used(fv <--> co)],
  [du-data-used(fv <--> du)],
  [du-data-updated(fv --> du)].

Implementation Attributes : fv_num(i2), fv_file(pfi,i2).

Relation : fi_variable(fv_num = i2, fv_type = c1,
  fv_file = i2)

ENTITY-NAME : **identifier-lit**   token : il

Description : Records (Non-variable, non-file-variable,
  non-variable-88, non-file) data-names, identifiers
  and literals used in the program.

Instances : Many

Attributes : il-token(il,c160)

Relationships : [lineage-size(il --> fi)],
  [linage-foot(il --> fi)], [linage-top(il --> fi)],
  [linage-bot(il --> fi)], [occurs-a(il --> va)],
  occurs-b(il --> va)], [value(il --> va)],
  [v8-value-a(il <--> v8)], [v8-value-b(il <--> v8)],
  [co-data-used(il <--> co)],
  [du-data-used(il <--> du)].

Implementation Attributes : il_num(i2).

Relation : ident_lit(il_num = i2, il_token_a = c80,
  il_token_b = c80)

ENTITY-NAME : **condition**   token : co

Description : Records PD condition and conditional
  expression information.

Instances : Many

```
Attributes : cond-text(id,c256),
  cond-type(~|STATE|TIMES|DEPEN|EVAL,c5).
Relationships : co-data-used(co <--> va|v8|fv|il),
  cu-condition(co --> cu).

Implementation Attributes : co_num(i2).
Relation : condition( co_num = i2, cond_text_a = c64,
  cond_text_b = c64, cond_text_c = c64,
  cond_text_d = c64, cond_type = c5 )

---------------------------------------------------
ENTITY-NAME : section                  token : se
Description : Records PD section information. Default
  seg-number is 0.
Instances : Many.
Attributes : se-name(se,c31),  seg-number(i2),
  se-position(ln,i2).
Relationships : sections-or-paras(se <-- pr),
  [cu-branch-a(se --> cu)], [cu-branch-b(se --> cu)],
  pa-section(se --> pa).

Implementation Attributes : se_num(i2).
Relation : section( se_num = i2, se_name = c31,
  seg_number = i2, se_position = i2 )

---------------------------------------------------
ENTITY-NAME : paragraph                token : pa
```

Description : Records PD paragraph information.

Instances : Many.

Attributes : pa-name(pa,c31), pa-position(ln,i2).

Relationships : pa-section(pa <-- se),
  sections-or-paras(pa <-- pr), pa-stgroup(pa -- sg),
  [cu-branch-a(pa --> cu)], [cu-branch-b(pa --> cu)].

Implementation Attributes : pa_num(i2),
  pa_section(pse,i2), pa_stgroup(psg,i2).

Metric Attributes : locp(i2), pa_comments(i2),
  u_operators(i2), operators(i2), periods(i2).

Relation : paragraph(pa_num = i2, pa_name = c31,
  pa_position = i2, pa_section = i2, pa_stgroup = i2,
  locp = i2, pa_comments = i2, u_operators = i2,
  operators = i2, periods = i2)

ENTITY-NAME : **statement-group**　　　　token : sg

Description : Records PD statement group information.
  A statement-group is a group of consecutive
  statements (in terms of the same nesting level)
  within a paragraph or statement.

Instances : Many

Attributes : sg-type(-|NEXT SENTENCE|EXIT,c13).

Relationships : [pa-stgroup(sg -- pa)],
  [st-stgroup(sg --> st)], [cu-branch-a(sg --> cu)],

  [cu-branch-b(sg --> cu)].

Implementation Attributes : sg_num(i2).

Relation : stgroup(sg_num = i2, sg_type = c13)

ENTITY-NAME : **statement** token : st

Description : Records PD statement information.

Instances : Many

Attributes : verb(id,c10), end-verb(id,c12),

  st-position(ln,i2),

  st-attr-a(I|O|I-O|EX|before|after,c6),

  st-attr-b(-|varying|multvar,c7).

Relationships : st-stgroup(st <-- sg),

  [cu-statement(st --> cu)], [du-statement(st --> du)].

Implementation Attributes : st_num(i2),

  st_stgroup(psg,i2).

Relation : statement(st_num = i2, verb = c10,

  end_verb = c12, st_position = i2, st_attr_a = c6,

  st_attr_b = c7, st_stgroup = i2)

NAVIGATION AND USAGE ENTITIES.

ENTITY-NAME : **source** token : so

Description : Records line numbers which map the source

  code. Line numbers for the IDENTIFICATION DIVISION, PROGRAM-ID paragraph, ENVIRONMENT DIVISION, CONFIGURATION SECTION, INPUT-OUTPUT SECTION, DATA DIVISION, FILE SECTION, WORKING-STORAGE SECTION, LINKAGE SECTION and PROCEDURE DIVISION are stored.

Instances : One only

Attributes : iden-div(ln,i2), pid-para(ln,i2), envi-div(ln,i2), conf-sec(ln,i2), inou-sec(ln,i2), data-div(ln,i2), file-sec(ln,i2), wost-sec(ln,i2), link-sec(ln,i2), proc-div(ln,i2).

Metric Attributes : locd(i2), comments(i2).

Relation : source(iden_div = i2, pid_para = i2, envi_div = i2, conf_sec = i2, inou_sec = i2, data_div = i2, file_sec = i2, wost_sec = i2, link_sec = i2, proc_div = i2, locd = i2, comments = i2)

ENTITY-NAME : **condition-usage** token : cu

Description : Records PD branch parameter information about statements.

Instances : Many

Attributes : cu-desc(-|UNTIL|WHEN,c5), cu-br-desc(-|ELSE|THRU|GOTO,c4).

Relationships : cu-condition(cu <-- co), cu-branch-a(cu <-- se|pa|sg),

```
  [cu-branch-b(cu <-- se|pa|sg)],
  cu-statement(cu <-- st).

Implementation Attributes : cu_num(i2),
  cu_condition(pco,i2), cu_branch_a(pse|ppa|psg,i2),
  cu_branch_b(pse|ppa|psg,i2), cu_statement(pst,i2).
Relation : cond_usage( cu_num = i2,
  cu_condition = i2, cu_branch_a = i2,
  cu_branch_b = i2, cu_desc = c5, cu_br_desc = c4,
  cu_statement = i2 )

----------------------------------------------------

ENTITY-NAME : data-usage                 token : du
Description : Records PD data use and update
  information about statements.
Instances : Many
Attributes : du-desc(-|CORR|INIT|NOUP|OPUP|NOUS),c4).
Relationships : [du-data-updated(du <-- va|fv|fi],
  [du-data-used(du <--> va|fv|fi|il],
  du-statement(du <-- st).

Implementation Attributes : du_num(i2),
  du_updated(pva|pfv|pfi,i2), du_statement(pst,i2).
Relation : data_usage( du_num = i2, du_updated = i2,
  du_desc = c4, du_statement = i2 )

----------------------------------------------------
```

LINK AND OTHER RELATIONS.

RELATION-NAME : **linage_file** token : lf

Description : Records linage information for sequential file with LINAGE clauses.

Instances : One per sequential LINAGE file

Relationships : fi-num(lf -- fi), linage_size(lf <-- va|il), linage_foot(lf <-- va|il), linage_top(lf <-- va|il), linage_bot(lf <-- va|il).

Implementation Attributes : fi_num(pfi,i2), linage_size(pva|pil,i2), linage_foot(pva|pil,i2), linage_top(pva|pil,i2), linage_bot(pva|pil,i2).

Relation : linage_file(fi_num = i2, linage_size = i2, linage_foot = i2, linage_top = i2, linage_bot = i2)

RELATION-NAME : **va_occurs** token : vo

Description : Records key and index lists from occurs clauses of variable definitions.

Instances : Many per occurs clause with keys or indices

Relationships : va-num(vo <-- va), occurs-key(vo <-- va), occurs_index(vo <-- va).

Implementation Attributes : vo_num(i2), va_num(pva,i2), occurs_asc(fg,c1), occurs_key(pva,i2), occurs_index(pva,i2).

Relation : va_occurs(vo_num = i2, va_num = i2,
  occurs_asc = c1, occurs_key = i2, occurs_index = i2)

RELATION-NAME : **v8_value**   token : vv

Description : Records lists of pairs of values
  associated with variable-88s.

Instances : Many per variable-88

Relationships : v8-num(vv <-- v8),
  v8-value-a(vv <-- il), v8-value-b(vv <-- va|il).

Implementation Attributes : v8_num(pv8,i2),
  v8_value_a(pil,i2), v8_value_b(pva|pil,i2).

Relation : v8_value(v8_num = i2, v8_value_a = i2,
  v8_value_b = i2)

RELATION-NAME : **co_data_used**   token : cd

Description : Records lists of data used in
  conditions.

Instances : Many per condition

Relationships : co-num(cd <-- co),
  co-data-used(cd <-- va|v8|fv|il).

Implementation Attributes : co_num(pco,i2),
  co_used(pva|pv8|pfv|pil,i2).

Relation : co_data_used(co_num = i2, co_used = i2)

```
-----------------------------------------------------

RELATION-NAME : du_data_used              token : dd

Description : Records lists of data used in

  data-usage instances of statements.

Instances : Many per data-usage instance

Relationships : du-num(dd <-- du),

  du-data-used(dd <-- fi|va|fv|il).

Implementation Attributes : du_num(pdu,i2),

  du_used(pfi|pva|pfv|pil,i2).

Relation : du_data_used( du_num = i2, du_used = i2 )

-----------------------------------------------------
```

APPENDIX 4B - STANDARD PHRASES FOR FORMATTING

Standard Phrase	Equivalents
SEGMENT-LIMIT IS	SEGMENT-LIMIT
CURRENCY SIGN IS	CURRENCY
	CURRENCY SIGN
DECIMAL-POINT IS COMMA	DECIMAL-POINT COMMA
ASSIGN TO	ASSIGN
AREAS	AREA
ORGANIZATION IS	ORGANIZATION
BLOCK CONTAINS	BLOCK
ACCESS MODE IS	ACCESS MODE
	ACCESS IS
RELATIVE KEY IS	RELATIVE KEY
	RELATIVE IS
RECORD KEY IS	RECORD KEY
	RECORD IS
	RECORD {in Environment Division only}
ALTERATE RECORD KEY IS	ALTERNATE RECORD KEY
	ALTERNATE RECORD IS
WITH DUPLICATES	DUPLICATES
FILE STATUS IS	FILE STATUS
	STATUS IS
RECORD CONTAINS	RECORD {within Data Division only}
RECORD IS VARYING IN SIZE	RECORD IS VARYING IN
	RECORD IS VARYING SIZE
	RECORD IS VARYING
	RECORD VARYING IN SIZE
	RECORD VARYING IN
	RECORD VARYING SIZE
DEPENDING ON	DEPENDING
LABEL RECORDS ARE	LABEL RECORD IS
LINAGE IS	LINAGE
WITH FOOTING AT	WITH FOOTING
	FOOTING AT
	FOOTING
LINES AT TOP	LINES TOP
	AT TOP
	TOP
LINES AT BOTTOM	LINES BOTTOM
	AT BOTTOM
	BOTTOM
VALUE IS	VALUE
	VALUES ARE
	VALUES

THRU	THROUGH
PIC	PIC IS
	PICTURE
	PICTURE IS
USAGE IS	USAGE
SIGN IS	SIGN
SEPARATE CHARACTER	SEPARATE
ASCENDING KEY IS	ASCENDING KEY
	ASCENDING IS
	ASCENDING
DESCENDING KEY IS	DESCENDING KEY
	DESCENDING IS
	DESCENDING
INDEXED BY	INDEXED
SYNCHRONIZED	SYNC
JUSTIFIED	JUST
BLANK WHEN ZERO	BLANK ZERO
VALUE	VALUE IS
CORRESPONDING	CORR
ON SIZE ERROR	SIZE ERROR
ON OVERFLOW	OVERFLOW
FOR REMOVAL	REMOVAL
WITH NO REWIND	NO REWIND
WITH LOCK	LOCK
INVALID KEY	INVALID
WITH NO ADVANCING	NO ADVANCING
	WITH NO
GO TO	GO
DATA BY	BY {within INITIALIZE statement only}
BEFORE INITIAL	BEFORE {within INSPECT statement only}
AFTER INITIAL	AFTER {within INSPECT statement only}
WITH TEST	TEST
KEY IS	KEY
AT END	END
DELIMITED BY	DELIMITED
WITH POINTER	POINTER
DELIMITER IN	DELIMITER
COUNT IN	COUNT
TALLYING IN	TALLYING
AT EOP	AT END-OF-PAGE
	END-OF-PAGE
	EOP
>	IS >
	IS GREATER THAN
	IS GREATER
	GREATER THAN
	GREATER
NOT >	IS NOT >

	IS NOT GREATER THAN
	IS NOT GREATER
	NOT GREATER THAN
	NOT GREATER
<	IS <
	IS LESS THAN
	IS LESS
	LESS THAN
	LESS
NOT <	IS NOT <
	IS NOT LESS THAN
	IS NOT LESS
	NOT LESS THAN
	NOT LESS
=	IS =
	IS EQUAL TO
	IS EQUAL
	EQUAL TO
	EQUAL
NOT =	IS NOT =
	IS NOT EQUAL TO
	IS NOT EQUAL
	NOT EQUAL TO
	NOT EQUAL

BIBLIOGRAPHY

[ALB83] ALBRECHT A.J., GAFFNEY J.E.
"Software Function, Source Lines of Code and Development Effort Prediction"
IEEE Transactions on Software Engineering,Vol.SE-9, No.6,Nov.1983

[ALL84] ALLMAN E.
"An Intro to SCCS"
ULTRIX-32 Supplementary Documents, Vol.II Programmers, Digital Equipment Corporation 1984

[AME79] AMEY W.S.
"The Computer Assisted Software Engineering (CASE) System"
4th International Conference on Software Engineering, Sept.1979

[AND81] ANDERSON R.E.
"Modular Documentation: a Software Development Tool"
AFIPS National Computer Conference,May 1981, pp401-405

[ARN82] ARNOLD R.S., PARKER D.A.
"The Dimensions of Healthy Maintenance"
6th International Conference on Software Engineering, Sept.1982 IEEE,pp10-27

[BAK80] BAKER A.L., ZWEBEN S.H.
"A Comparison of Measures of Control Flow Complexity"
IEEE Transactions on Software Engineering,Vol.SE-6, No.6,Nov 1980

[BAS82a] BASLI V.R., MILLS H.D.
"Understanding and Documenting Programs"
IEEE Transactions on Software Engineering,Vol.SE-8, May 1982,pp270-283

[BEL84] BELL F.J.
"Technology Transfer in the Maintenance Environment"
AFIPS National Computer Conference, 1984,pp229-234

[BER85b] BERRY R.E., MEEKINGS B.A.
"A Style Analysis of C programs"
Communications of A.C.M.,Vol.28,No.1,Jan.1985

[BOE73] BOEHM B.W.
"Software and its Impact : a Quantitive Assessment"

Datamation,May 1973,pp48-59

[BOE76] BOEHM B.W.
"Software Engineering"
IEEE Transactions on Computers,Vol.C-25,No.12, Dec.1976,pp1226-1240

[BOE81] BOEHM B.W.
"Software Engineering Economics"
Prentice-Hall 1981

[BOE84] BOEHM B.W.
"Software Engineering Economics"
IEEE Transactions on Software Engineering,Vol.SE-10, No.1,Jan.1984

[BRI83] BRICE L., CONNELL J.
"A Methodology for Minimizing Maintenance Costs"
AFIPS National Computer Conference,May 1983,pp113-121

[BRI84] BRICE L., CONNELL J.
"System Information Database: an Automated Maintenance Aid"
AFIPS National Computer Conference, 1984,pp209-216

[CAN86b] CANE A.
"Getting Programs back into shape"
The Dominion (New Zealand Newspaper),Dec 1st 1986,pp18

[CEN82] CENTRE J.W.
"A Quality Assurance Program for Software Maintenance"
AFIPS National Computer Conference,June 1982,pp399-407

[CHA79] CHAPIN N.
"A Measure of Software Complexity"
AFIPS National Computer Conference, 1979,pp995-1002

[CHA81] CHAPIN N.
"Productivity in Software Maintenance"
AFIPS National Computer Conference, 1981,pp349-352

[CHA85] CHAPIN N.
"Software Maintenance: a Different View"
AFIPS National Computer Conference, 1985,pp507-513

[COB81] TECHNICAL COMMITTEE X3J4 - COBOL
"Draft Proposed Revised X3.23 American National Standard Programming Language COBOL"
American National Standards Institute, Sept.1981

[COL85a] COLLOFELLO J.S., BLAYLOCK J.W.

"Syntactic Information Useful for Software Maintenance"
AFIPS National Computer Conference, 1985,pp547-553

[CON84] CONNELL J., BRICE L.
"Prolonging the Life of Software"
AFIPS National Computer Conference, July 1984, pp243-249

[COO78] COOPER J.D.
"Corporate Level Software Engineering"
IEEE Transactions on Software Engineering,Vol.SE-4, July 1978,pp319-325

[CRA85] CRAWFORD S.G., MCINTOSH A.A., PREGIBON D.
"An Analysis of Static Metrics and Faults in C Software"
Journal of Systems and Software, No.5, 1985 pp37-48

[CUR79] CURTIS B., SHEPPARD S.B., MILLIMAN P., BORST M.A., LOVE T.
"Measuring the Psychological Complexity of Software Maintenance Task with the Halstead and McCabe Metrics"
IEEE Transactions on Software Engineering,Vol.SE-5, May 1979,pp96-104

[DEU81] DEUTSCH M.S.
"Software Project Verification and Validation"
Computer, Vol.14,No.4,April 1981

[DIG69] DIGITAL
"DecSystem10 COBOL Programmer's Reference Manual"
DIGITAL EQUIPMENT CORPORATION, 1969

[EBE80] EBERT R., LUGGER J., GOECKE R.
"Practice in Software Adaptation & Maintenance"
North-Holland 1980

[ELS76] ELSHOFF J.L.
"An Analysis of Some Commercial PL/1 Programs"
IEEE Transactions on Software Engineering,Vol.SE-2, No.2,June 1976

[ELS78] ELSHOFF J.L.
"An Investigation into the Effect of the Counting Method Used on Software Science Measurements"
Sigplan Notices,Vol.13,No.2,Feb.1978,pp30-45

[ELS82] ELSHOFF J.L., MARCOTTY M.
"Improving Computer Program Readability to Aid

Modification"
Communications of A.C.M.,Vol.25,No.8,Aug.1982

[ELS84] ELSHOFF J.L.
"Characteristic Program Complexity Measures"
7th International Conference on Software Engineering, 1984 IEEE,pp288-293

[EVA82] EVANS M., BANKEN S.E.
"Software Engineering for the COBOL Environment"
Communications of the A.C.M.,Vol.25,No.12,Dec 1982

[FAI78] FAIRLEY R.E.
"Tutorial: Static Analysis and Dynamic Testing of Computer Software"
Computer,Vol.11,No.4,April 1978

[FEU79] FEUER A.R., FOWLKES E.B.
"Relating Computer Program Maintainability to Software Measures"
AFIPS National Computer Conference, 1979,pp1003-1012

[FIO84] FIORELLO M., CUGINI J.
"Is COBOL-8x Cost Effective ?"
AFIPS National Computer Conference, 1984,pp223-228

[FOR85] FORAGE G.
"Fourth-Generation Languages and Advanced Software Development Aids"
Data Processing,Vol.27,No.9,Nov 1985

[GIM80] GIMPEL J.F.
"CONTOUR -A Method of Preparing Structured Flowcharts"
Sigplan Notices,Vol.15,No.10,Oct 1980

[GUI83] GUIMARAES T.
"Managing Application Program Maintenance Expenditures"
Communications of the A.C.M.,Vol.26,No.10,Oct 1983

[HAL77] HALSTEAD M.H.
"Elements of Software Science"
North-Holland 1977

[HAN78] HANSEN W.J.
"Measurement of Program Complexity By the Pair (Cyclomatic Number,Operator Count)"
Sigplan Notices,Vol.13,No.3,Mar.1978,pp29-33

[HAR81a] HARRISON W.A., MAGEL K.I.
"A Complexity Measure Based on Nesting Level"
Sigplan Notices,Vol.16,No.3,Mar 1981

[HAR81b] HARRISON W.A., MAGEL K.I.
"A Topological Analysis of Computer Programs with less than 3 Binary Branches"
Sigplan Notices,Vol.16,No.4,Apr.1981,pp51-63

[HAR83] HARRISON W., MAGEL K., KLUCZNY R.
"Research in Software Maintenance"
Journal of Systems Management,July 1983,pp10

[HAR84] HARRISON W.A.
"Software Complexity Metrics"
Journal of Systems Management,July 1984,pp28

[HAR85] HARRISON W., COOK C.R.
"A Method of Sharing Software Complexity Data"
Sigplan Notices,Vol.20,No.2,Feb.1985

[HAR86] HARRISON W., COOK C.R.
"A Note on the Berry-Meekings Style Metric"
Communications of the A.C.M.,Vol.29,No.2,Feb.1986

[HAW79] HAWKINS T.J., HARANDI M.T.
"Towards more Portable COBOL"
The Computer Journal,Vol.22, 1979,pp290

[HEN81] HENRY S., KAFURA D.
"Software Structure Metrics Based on Information Flow"
IEEE Transactions on Software Engineering,Vol.SE-7, No.5,Sept.1981

[HOR79] HORSLEY T.R., LYNCH W.C.
"Pilot : a Software Engineering Case Study"
4th International Conference on Software Engineering, Sept.1979

[HOR86] HOROWITZ E., WILLIAMSON R.C.
"SODOS - A Software Documentation Support Environment - Its Definition"
IEEE Transactions on Software Engineering,Vol.SE-12, No.8,Aug.1986

[HOU83] HOUGHTON R.C.
"Software Development Tools : A Profile"
Computer,Vol.16,No.5,May 1983

[HUA78] HUANG J.C.
"Program Instrumentation and Software Testing"
Computer,Vol.11,No.4,Apr.1978

[HUT86] HUTCHINSON A.
"Some Practical Principles for Design of Maintainable

Systems"
The Computer Journal,Vol.29,No.1, 1986

[INC86] INCE D., WOODMAN M.
"The Rapid Generation of a Class of Software Tools"
The Computer Journal,Vol.29,No.2,April 1986

[JAC75] JACKSON M.A.
"Principles of Program Design"
Academic Press Inc. (London) Ltd, 1975

[JON86a] JONES C.
"Programming Productivity"
McGraw-Hill 1986

[JON86b] JONES R.
"Automated Tools for the Analyst"
Data Processing,Vol.28,No.5,June 1986

[LET86] LETOVSKY S., SOLOWAY E.
"Delocalized Plans and Program Comprehension"
IEEE Software,Vol.3,No.3,May 1986

[LIE78] LIENTZ B.P., SWANSON E.B., TOMKINS G.E.
"Characteristics of Application Software Maintenance"
Communications of the A.C.M.,Vol.27, 1978,pp466-471

[LIE80] LIENTZ B.P., SWANSON E.B.
"Software Maintenance Management"
Addison-Wesley 1980

[LIE81] LIENTZ B.P., SWANSON E.B.
"Problems in Application Software Maintenance"
Communications of the A.C.M.,Vol.24,Nov.1981,pp763-769

[LIE83] LIENTZ B.P.
"Issues in Software Maintenance"
Computing Surveys,Vol.15,No.3,Sept 1983

[LIN73] LINDHORST W.M.
"Scheduled Maintenance of Application Software"
Datamation, May 1973,pp64-67

[LIU76] LIU C.C.
"A look at Software Maintenance"
Datamation,Vol.22,Nov.1976,pp51-55

[LYO81] LYONS M.J.
"Salvaging your Software Asset(Tools Based
Maintenance)"
AFIPS National Computer Conference, 1981,pp337-341

[MAR83a] MARSH R.E.
"Application Maintenance: One Shop's Experience and Organization"
AFIPS National Computer Conference, 1983,pp146-153

[MAR83b] MARSELOS N.L.
"Human Investment Techniques for Effective Software Management"
AFIPS National Computer Conference,May 1983,pp131-136

[MAR83c] MARTIN J., MCCLURE C.
"Software Maintenance : The Problem and Its Solutions"
Prentice-Hall N.J., 1983,pp5-7

[MAT86] MATHIS R.F.
"The Last 10 Percent"
IEEE Transactions on Software Engineering,Vol.SE-12, No.6,June 1986

[MCC76] MCCABE T.J.
"A Complexity Measure"
IEEE Transactions on Software Engineering,Vol.SE-2, No.4,Dec.1976

[MCK84] MCKEE J.R.
"Maintenance as a Function of Design"
AFIPS National Computer Conference, 1984,pp187-193

[MCN84] MCNEILE A.T.
"A Model of Programming Activity"
Computer Bulletin,Sept.1984

[MCT80] MCTAP J.L.
"The Complexity of an Individual Program"
AFIPS National Computer Conference, 1980,pp767-771

[MOR79] MORRISSEY J., WU L.
"Software Engineering - An Economic Perspective"
4th International Conference on Software Engineering, Sept.1979

[MUN78] MUNSON J.B.
"Software Maintainability: a Practical Concern for Life-Cycle Costs"
Proc. Compsac,Nov.1978,pp54-59

[MYE77] MYERS G.J.
"An Extension to the Cyclomatic Measure of Program Complexity"
Sigplan Notices,Vol.12,No.10,Oct 1977,pp62-64

[NAR84] NARROW B., KELLY I.
"Two Perceptions of Software Maintenance Performed by On-Site Contractors"
AFIPS National Computer Conference,July 1984,pp235-245

[OGD72] OGDINOG J.L.
"Designing Reliable Software"
Datamation,Vol.18,July 1972,pp71-78

[PAI77] PAIGE M.R.
"On Partitioning Programs Graphs"
IEEE Transactions on Software Engineering,Vol.SE-3, No.6,Nov 1977

[PAN78] PANZL D.J.
"Automatic Software Test Drivers"
Computer,Vol.11,No.4,April 1978

[PAR85a] PARIKH G.
"Discovering the World of Software Maintenance: Selected Readings"
Sigsoft Software Engineering Notes,Vol.10,No.5, Oct 1985

[PAR85b] PARIKH G.
"Software Maintenance: Penny Wise, Program Foolish"
Sigsoft Software Engineering Notes,Vol.10,No.5, Oct 1985

[PHI74] PHILIPPAKIS A., KAZMIER L.
"Information Systems Through COBOL"
McGraw-Hill, 1974

[PIW82] PIWOWARSKI P.
"A Nesting Level Complexity Measure"
Sigplan Notices,Vol.17,No.9,Sept 1982

[POW84] POWERS M.J., ADAMS D.R., MILLS H.D.
"Computer Information Systems Development: Analysis and Design"
South-Western Publishing Co., 1984,p38

[PRA84] PRATHER R.E.
"An Axiomatic Theory of Software Complexity Measure"
The Computer Journal,Vol.27,No.4,Nov.1984

[PUN75] PUNTER M.
"Programming for Maintenance"
Data Processing,Sept.1975,pp292-294

[RED86] REDISH K.A., SMITH W.F.

"Program Style Analysis: A Natural By-Product
of Program Compilation"
Communications of the A.C.M.,Vol.29,No.2,Feb.1986

[REE82] REES M.J.
"Automatic Assessment Aids for Pascal Programs"
Sigplan Notices,Vol.17,No.10,Oct.1982

[REI84] REISS S.P.
"Graphical Program Development with PECAN
Program Development System"
Dept.Comp.Sci.,Brown Univ.,Rhode Is.,Technical Report
CS-84-04,Mar.1984

[REU81] REUTTER J.
"Maintenance is a Management Problem and a
Programmers Opportunity"
AFIPS National Computer Conference, 1981,pp343-347

[RIC83] RICHARDSON G.Y., BUTLER W.C.
"Organizational Issues on Effective Maintenance
Management"
AFIPS National Computer Conference,May 1983,pp155-161

[RIC84] RICARDSON G., HODIL E.D.
"Redocumentation: Addressing the Maintenance Legacy"
AFIPS National Computer Conference, 1984,pp203-208

[RIC85] RICHMOND I.
"Documentation Support - a Route to Full Life Cycle
Productivity"
Data Processing,Vol.27,No.9,Nov.1985

[RID81] RIDDLE W.E.
"An Assessment of DREAM"
Software Enginnering Environments,H.Hunke,
North-Holland 1981,pp191-221

[RIG69] RIGGS R.
"Computer Systems Maintenance"
Datamation,Nov.1969,pp227-232

[ROB76] ROBINSON P.
"Advanced COBOL : ANS 74"
Macdonald and Jane's/American Elsevier 1976

[SAI83] SAIB S.H.
"Future ADA Environments"
AFIPS National Computer Conference, 1983,pp57-63

[SCH79] SCHNEIDEWIND N.F.

"Software Metrics for Aiding Program Development and Debugging"
AFIPS National Computer Conference, 1979,pp989-994

[SCH81] SCHNEIDER G.M., SEDLMEYER R.L., KEARNEY J.
"On the Complexity of Measuring Software Complexity"
AFIPS National Computer Conference, 1981,pp317-322

[SCH83] SCHNEIDER G.R.E.
"Structured Software Maintenance"
AFIPS National Computer Conference, 1983,pp138-144

[SHE81] SHIEL B.A.
"The Psychological Study of Programming"
Computing Surveys,Vol.13,No.1,Mar.1981

[SHE83] SHEN V.Y., CONTE S.D., DUNSMORE H.E.
"Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support"
IEEE Transactions on Software Engineering,Vol.SE-9, No.2,Mar.1983

[SHI85] SHILLING J.
"FRED : A Program Development Tool"
Dept.Comp.Sci.,Univ.Illinois, Report UIUCDCS-R-85-1224,Sept.1985

[STA84] STANDISH T.A.
"An Essay on Software Reuse"
IEEE Transactions on Software Engineering,Vol.SE-10, No.5,Sept.1984

[SUN81] SUNOHARA T., TAKANO A., UEHARA K., OHKAWA T.
"Program Complexity Measure for Software Development Management"
5th International Conference on Software Engineering, Mar.1981,pp100-106

[TAI84] TAI K.C.
"A Program Complexity Metric Based On Data Flow Information In Control Graphs"
7th International Conference on Software Engineering, 1984 IEEE,pp239-248

[TAT85] TATE G., HAYWARD R.
"Software Engineering Issues in the Use of Fourth Generation Languages"
Massey University Computer Science Report 85/5, ISSN0112-630X,July 1985

[TAU83] TAUTE B.J.

"Quality Assurance and Maintenance Application Systems"
AFIPS National Computer Conference,May 1983,pp123-129

[TEA85] TEAGUE L.C., PIDGEON C.W.
"Structured Analysis Methods for Computer Information Systems"
SRA, 1985,p42

[TEI77] TEICHROEW D., HERSHEY E.
"PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems"
IEEE Transactions on Software Engineering,Vol.SE-3, No.1,Jan.1977

[TEO82] TEOREY T., FRY J.
"Design of Database Structures"
Prentice-Hall, 1982

[THA81] THALMANN D., MAGNENAT-THALMANN N.
"Computers in Education"
North-Holland 1981

[TIN83] TINNIRELLO P.C.
"Improving Software Maintenance Attitudes"
AFIPS National Computer Conference, 1983,pp107-122

[TIN84] TINNRELLO P.C.
"Software Maintenance in Fourth Generation Environments"
AFIPS National Computer Conference,July 1984,pp251-257

[TOR79] TORSUN I.S., AL-JARRAH M.M.
"An Empirical Analysis of COBOL Programs"
Software Practice and Experience,Vol.9,No.5,May 1979

[TOR81] TORSUN I.S., AL-JARRAH M.M.
"Dynamic Analysis of COBOL Programs"
Software Practice and Experience,Vol.11,No.9,Sept 1981

[VES83] VESSEY I., WEBER R.
"Some Factors Affecting Program Repair Maintenance: An Empirical Study"
Communications of A.C.M.,Vol.26,No.2,Feb.1983

[VOG80] VOGES U., GMEINER L., VON MAYRHAUSER A.A.
"SADAT - An Automated Testing Tool"
IEEE Transactions on Software Engineering,Vol.SE-6, No.3,May 1980

[WAR82] WARREN S.

"MAP : a Tool for Understanding Software"
6th International Conference on Software Engineering, Sept.1982 IEEE,pp28-37

[WAT79] WATERS R.C.
"A Method for Analyzing Loop Programs"
IEEE Transactions on Software Engineering,Vol.SE-5, No.3,May 1979

[WAT82] WATERS R.C.
"The Programmer's Apprentice : Knowledge Based Program Editing"
IEEE Transactions on Software Engineering,Vol.SE-8, No.1,Jan.1982

[WEI74] WEISSMAN L.
"Psychological Complexity of Computer Programs: an Experimental Methodology"
Sigplan Notices,June 1974,pp25-36

[WEL82] WELSER M.
"Programmer uses Slice when Debugging"
Communications of A.C.M.,Vol.25,July 1982,pp446-452

[WIL79] WILLIS R.R., JENSEN E.P.
"Computer Aided Design of Software Systems"
4th International Conference on Software Engineering, Sept.1979

[WOO79] WOODWARD M.R., HENNELL M.A., HEDLEY D.
"A Measure of Control Flow Complexity in Program Text"
IEEE Transactions on Software Engineering,Vol.SE-5, No.1,Jan.1979

[YAU80] YAU S.S., COLLOFELLO J.S.
"Some Stability Measures for Software Maintenance"
IEEE Transactions on Software Engineering,Vol.SE-6, Nov.1980,pp545-552

[ZOL80] ZOLNOWSKI J., SIMMONS D.B.
"Measuring Program Complexity in a COBOL Environment"
AFIPS National Computer Conference, 1980,pp757-766

[ZOL81] ZOLNOWSKI J.C., SIMMONS D.B.
"Taking the Measure of Program Complexity"
AFIPS National Computer Conference, 1981,pp329-336

[ZVE82a] ZVEGINTZOV N.
"What Life ? What Cycle ?"
AFIPS National Computer Conference,June 1982,pp561-568