

LINEAR LIBRARY
C01 0068 1247



FLOW: A programing environment using diagrams.

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

FLOW: A programing environment using diagrams.

By Jeffrey W M Dooley.

**A Thesis prepared under the supervision of
Associate Professor S.R.Schach
in fulfilment of the requirements for the degree of
Master of Science
in Computer Science
at the University of Cape Town.**

October 1983.

The University of Cape Town has been given
the right to reproduce this thesis in whole
or in part. Copyright is held by the author

Abstract

A graphical language is developed as a generalization of the structured flowcharts proposed by Nassi and Shneiderman. This language can be used in the specification of procedures, procedure interfaces and data structures. A software production support environment is then developed using this language which is capable of producing systems in FORTRAN IV, COBOL and Pascal. The environment integrates new and existing tools and facilitates and encourages the design, coding and testing of well structured systems using the methodology of stepwise refinement.

A central component of the environment is a software production data base which holds the programme source as well as control information pertaining to the state of development of the system and interfaces of the various programme modules being created within the system. A helpful syntax directed editor for the graphical language is used to update the data base. Programme specifications are extracted from the data base by a number post-processors which produce target code for the required high level languages as well as system documentation.

Some of the practical experience gained over a three year period is described and suggestions for the extension of the current environment and topics for future research are presented.

Acknowledgements

I wish to express my gratitude to Professor Schach for his encouragement and helpful suggestions, and to my colleagues who used FLOW during its various stages of development and who contributed their comments for inclusion in this work.

I am deeply grateful for the encouragement and patience of my wife Michelle and I dedicate this thesis to her.

Contents

Abstract	ii
Acknowledgements	iii
Chapter 1 Introduction	1
Chapter 2 Background	7
2.1 Computer Assisted Software production .. environments	7
2.2 The use of tools within the software .. development cycle	10
2.3 Summary	17
Chapter 3 FLOW diagrams	21
3.1 Modifications to NSDs	22
3.2 FLOW-DL: The language for specifying .. GNSDs	31
3.3 Using GNSDs for DATA definitions	39
Chapter 4 The FLOW methodology of software	53
4.1 The software production model	53
4.2 Software production - The movie	55
Chapter 5 The FLOW environment	94
5.1 FLOW environment architecture	94
5.2 The environment manager	98
Chapter 6 The Software production data base	104
6.1 Design philosophy	104
6.2 GNSDs as tree structures	106
6.3 Data base structure and implementation	110
Chapter 7 FLOW-EDIT: A syntax directed editor for	119
FLOW-DL.	
7.1 Design Philosophy	119
7.2 Simplicity and ease of use	121
7.3 Helpfulness	133
7.4 Applicability to GNSD editing	139
7.5 Ensuring syntactical correctness of the .. data base	141
7.6 Display terminal constraints	145
7.7 Implementation notes	146
Chapter 8 Post-processors	149
8.1 Generation of target code	150
8.2 Programme trace and stub facilities ...	162
8.3 Maintainability	167
8.4 Implementation notes	171
Chapter 9 Practical experience in the use of the	175
FLOW environment	
9.1 The prototype	175
9.2 The FLOW environment	184

Chapter 10	Portability issues	186
10.1	Hardware	187
10.2	Software	193
10.3	Software generated	195
10.4	The 'non portability' issue	197
Chapter 11	Conclusion and topics for future re- search	199
11.1	Conclusions	199
11.2	Topics for future research	201
Bibliography	214
Appendix A	221
B	227

1. Introduction.

The FLOW environment is a computer aided software production system which makes extensive use of diagrams. From earliest times, graphical representations have been used as an effective and efficient medium for expressing and communicating 'technical' ideas. As a means of expression, pictures have been used to visualise, clarify and document the thoughts of their creators. The architect, for example, draws a 'rough sketch' of a new building. This is the first tangeable form of a concept which will later be refined until more detailed descriptions (blueprints, perspective drawings etc.) are produced. Only after the picture has served this 'creative' need will it be used to communicate these concepts to others. At that stage the architects will, for example, hand over to the builders the building plans, whilst the perspective drawings will be shown to their clients. There are other examplas of the use of pictures in expressing and communicating ideas that can be found in the fields of civil, electrical, and mechanical engineering. The ability of a civil engineer, for example, to design and communicate the details of a bridge in words alone, would be seriously questioned. In the field of software engineering, pictures have also been used, enabling software designers to conceptualise their designs, and to communicete these designs to the software production teams.

With continued use, pictorial representations for specific requirements have become stylised and standardised. Hieroglyphics,

heraldic design, road signs, building plans, topographical maps, and engineering blue-prints are but some examples of this. The stylising and standardising process has had the effect of making pictures even more effecieint and effective from both the expressive and communicative points of view (sometimes at the cost of making them unintelligible to outsiders). In the field of software engineering, for example, a number of diagram forms have evolved covering various aspects of the software design. The definition of possible programme execution paths is often sketched using a flowchart. Early flowchart formats have been standardised whilst new flowchart forms have evolved which are more compact and easier to draw [Nass73] [Chap74].

The ability of computers to draw and manipulate pictures rapidly together with their ability to perform mathematical calculations can be put to use in producing 'computerised sketch pads' which are superior in many respects to their traditional counterparts. In engineering and architecture, Computer Aided Design (CAD) systems have been developed and these have generally improved the productivity of the designer. One of the principles of CAD, namely that the computer may be used to assist the designer in concpetualising and experimenting with design alternatives can be applied to computer aided software design as well.

Whilst 'drawing' is usually thought of in terms of the physical motions of applying pen to paper (or stylis to digitiser) this is not always necessary in communicating the details of pictures to a computer. The format of the pictures being produced may

often be defined by a notation or 'language' which the computer can then be programmed to 'understand'. For example, when two or three dimensional objects are being designed, these may be defined in the terminology of co-ordinate geometry. Heraldry provides a good example of a well developed 'language' for defining a special kinds of pictures, with complete and unambiguous descriptions being given in a few words.

The logical extension to communicating designs to the computer, is to allow the computers to assist in the building of the objects designed. This is generally referred to as 'Computer Aided Manufacture' (CAM). For example, in the manufacture of some components, 'numerically controlled' (NC) machines (lathes, milling machines etc.) are used. The interfacing of Computer Aided Design with Manufacture (CAD/CAM) is possible with the NC machine control information being generated from the component design 'drawing.'

From the foregoing discussion, the following characteristics of 'technical' pictures emerge

- Technical pictures are of benefit to the designer in both a creative and communicative role
- Computers can be used to draw and manipulate pictures rapidly
- 'Languages' exist for defining pictures

- Pictures may be used to communicate with the computer.
- Computers are able to translate pictures into other forms.

With this knowledge, a software production environment, FLOW, was built which makes full use of the advantages provided by pictures. The 'stylised' pictures used are a generalised form of the diagrams proposed by Nassi and Shneiderman, the designs represented are software designs, and the translated form of the design is the software source code.

Objectives

The objective in developing the FLOW environment was to create a computer assisted software production environment capable of producing systems with source code in COBOL, FORTRAN and Pascal, which would, at the same time be capable to achieving a twofold increase in development productivity, and a halving of maintenance costs on small to medium scale projects. (Small to medium scale projects are here taken as being projects with an estimated development time of 1 to 10 man years, and an implementation time of 6 months to 2 years).

This modest productivity goal was set against the backdrop of previous attempts to attain 'orders of magnitude' improvements. Attaining this objective was considered significant in the light of research which has shown that the overall increase in DP

productivity over the last ten years has been a meagre 3 to 7 percent per annum in spite of widespread acceptance of tools and methodologies such as structured programming [Dist80].

Bibliography

- [Chap74] Chapin N. **New format for flowcharts.** Software Practice and Experience, Vol 6 No 4. pp.341-357, 1974.
- [Dist80] Distaso J R. **Software Management - A Survey of the Practice in 1980.** Proceedings of the IEEE, Vol 68 No 9 pp. 1103-1119, September 1980
- [Nass73] Nassi I, Shneiderman B. **Flowchart Techniques for Structured Programs.** ACM SIGPLAN Notices, Vol 8 No. 8 pp. 12-26, August 1973.

2. Background

The FLOW environment is a computer assisted software production environment using graphical representations of procedures and data. As background information for the chapters which follow, a brief description of some current computer assisted software productions environments, tools for assistance in the software production cycle, and automated systems for graphically representing software are presented. The references given in this chapter may be consulted for a more detailed description of these topics.

2.1 Computer Assisted Software Production Environments.

One of the ways in which the production of software may be made more efficient is through the creation of an environment which is conducive to improved productivity. In order to provide such an environment, the many facets of the environment including the aspects of the physical environment (office plants, desks, visual display terminals and the coffee machine) need to be examined and evaluated. The applicability of computers is so diverse that it can be reasonably expected that any environment may be improved with the aid of computers in one way or another. For example, computers are already being used to control the temperature, humidity and lighting in many buildings, and the creation of more congenial working conditions is certainly possible with computer assistance.

Programming environments

An area of particular interest is harnessing computers in order to increase software productivity. This can be achieved by providing improved tools for software development. Traditionally, tools such as text editors, high level language compilers and debuggers have served software developers. Although tools have been in use and been refined over a period of three decades, they have been applied mainly to assist with programming. The creation of tools to meet other aspects of software production, and the integration of tools which are able to interact with one another is a fairly recent innovation.

These integrated tool-sets have been loosely referred to as 'programming environments'. Amongst those which have emerged are the PROGRAMMER'S WORKBENCH [Ivie77], INTERLISP [Teit81a] and PASES [Shap81]. The PROGRAMMERS WORKBENCH is used in general programme development, whilst INTERLISP is an environment which is used specifically for programming in the language LISP [Teit81a]. PASES is another environment used specifically for programming in Pascal. Buxton [Buxt80] and Houghton [Houg83] cite a number of other environments currently in use, and the next few years are likely to see many more developments such as the MAPSE and APSE for the ADA language [DoD80].

The creation of a 'computerised software production environment' is most often achieved by applying one or more formal

methodologies of software production in conjunction with a number of computerised tools. This concept of an amalgam or collection of tools and techniques constituting an environment (as opposed to a single tool supporting all functions) is described in much of the literature, although comprehensive environments, with tools covering the entire spectrum of software production, have yet to be developed. Boehm [Boeh76] discussed a number of these computerised tools from the Software Engineering viewpoint, listing 'current practice', 'current frontier technology' and 'future trends'. Some five years later, similar discussions of 'current practice', with more predictions for the future by Howden [Howd81], Standish [Stan81], and Zelkowitz [Zelk81] show that there is still some faith in the concept of a comprehensive environment, (even though little practical advancement has been seen over this period).

The term 'Computer Assisted Software Production Environment' is used here in place of the more usual 'programming environment' or 'programming support environment'. This is done to emphasise that software production encompasses more than just programming. Glass, in his proposal for a minimum standard software tool-set [Glas82], reviewed tools applicable to software production which included tools for requirements definition, design, implementation, checkout, maintenance, documentation and project management. All of these activities have in fact been computerised to some extent. Unfortunately many of these tools operate independently of one another, and provide assistance for one or two activities only.

The Software Production Data Base

Often where proposals for an integrated software production environment are made, such as in the proposal of Howden [Howd81], the central component is the software production data base. In the ideal case, the Software Production Data Base will contain all of the information pertaining to the software production process; the various tools within the environment access this data base without having to interface directly with other tools. Data base technology is sufficiently well developed to cater for the needs of integrated software production tools, although the particular data base system used will determine to some extent the ease with which the various tools can be integrated.

2.2 The use of tools within the software development cycle.

Analysis and Specification

The first phase undertaken in a software production effort is the analysis and specification of the system to be developed. Tools covering this aspect of the software production cycle should ideally be included in an integrated environment. Computer assisted specification systems currently available use a formal, high level input language which is used to define the functions and the data of the system. Specification systems such as PSL/PSA [Teic77] include definitions of the system input/output, system

structure, data structure and derivation, system size, system volumes, system dynamics and system properties. In PSL/PSA these specifications are written in the Problem Statement Language (PSL). The PSL specifications are then analysed by the Problem Statement Analyser (PSA) to produce a number of reports. Among these are the Contents Comparison Report which gives an analysis of the similarity between inputs and outputs, the Data Process Interaction Report which is used to show information flow and determine unused data or objects, and the Process Chain report which shows the dynamics of the system. Formal Functional Descriptions and Data Requirements may also be produced.

There are many other ways in which specifications can be written. One possibility is to make use of a formal 'pseudo code' which can be accepted by a 'specification compiler'. This approach is described by Van Leer [VanL76] and has been used in the TOPD system [Snow78]. Another approach which is commonly used, is the graphical representation of system designs. A number of graphical specification systems have been developed such as Structured Analysis [Ross77] (used with the Structured Analysis and Design Technique (SADT) [Dick78]) and the Tell system [Heba78]. In Structured Analysis a special notation is developed using a series of boxes and arrows to represent inputs, outputs, controls and mechanisms. In TELL 'icons' are developed to represent procedures and data objects and these are connected by a number of relationships. It is of interest to note that all of the analysis and specification systems mentioned above employ the top-down methodology of stepwise refinement by functional decomposition

[Wirt71] and that the methodology is assisted and re-inforced by the tools themselves.

Programming and debugging.

Following on from the analysis and design stage (although not necessarily after its completion), the next stage in the software production cycle is programming and debugging. Programming was one of the first aspects of software production to become computerised (using tools such as compilers and editors) and was also one of the first aspects to be integrated into a 'programming environment'. Computer assistance may be applied to many aspects of programming ranging from the entry of programme source to the management and control of the programme libraries and various versions of source and object files.

A number of 'Smart Editors' have been developed to assist with programme entry and modification. These editors have some knowledge of the syntax of the language being edited, and make the creation of syntactically incorrect programmes almost impossible. Some editors include an interpreter for the language being edited, and thereby provide an interactive and iterative development facility. In these systems, the user may execute parts of the programme at an intermediate state of completeness, and may modify the programme source during interruptions of execution. One such editor is the 'Program Synthesiser' used at Cornell University [Teit81b] [Teit81c] which has proved to be useful

for both programming and teaching programming in a student environment. Another similar system is CAPS [Wilc76] which was designed primarily as a teaching tool with the emphasis on helping students to diagnose and understand their errors.

The major application of syntax directed editors, has been in the editing of programmes for block structured languages. These editors usually make use of 'templates' or skeleton syntactic block structures (e.g. `if [...] then [...] else [...] end`) where the key words (or structural aspects) are immutable. An alternative to this design for syntax directed editors is presented by Morris and Schwartz [Morr81] in which editing and syntax checking takes place in two stages, allowing the user freedom to create programmes which are temporarily incorrect. Although most syntax directed editors have concentrated on procedure structures, the techniques used are equally applicable to general data structures including documents [Fras81], [Wood81].

The editing of programmes represented graphically (i.e. by means of flowcharts) has been used by 'the Graphical Editor' [NG78], PIGS [Pong80], GRASP [Work83] and DUIF [Van083]. In the editors developed by Ng and Pong, the Nassi Shneiderman Diagram (NSD) representation of programmes is used for procedure definitions. GRASP uses 'D-Charts' which are able to represent both specifications and procedures. DUIF introduces a new flowchart form especially useful in representing data structures, but also able to represent procedures.

The above discussion has concentrated on programming in procedural languages. It should be noted that a great many non-procedural systems (including the so called fourth generation languages), and software packages (report writers, query languages etc.) exist which are widely used for the creation of 'programmes' for everyday applications. Many of these tools generate source code which may be subsequently modified and tailored to meet specific needs. Many of these tools are proprietary software supplied by hardware vendors, and in most cases operate only on one specific type of hardware. A number of code generation utilities such as screen generators which automatically produce the code to display and accept data on display terminal screens, and application generators which generate code for well defined applications also exist. Since these code generation tools and fourth generation languages are likely to be used for the bulk of the routine data processing applications in future, there is obviously a need to integrate them into the software production environment.

Verification, quality assurance, and automated testing.

System testing takes place at two levels. At the first level, testing is performed in an attempt to find and eliminate programme errors. Test data generators have been in use for some time for 'batch' data processing work, and some interactive test assistants have also been developed [Chap82].

At the second level, testing may be performed as part of quality

assurance. Once a system has been developed and debugged, it is important that it be verified against the original specification and thoroughly tested. In an integrated environment, the verification and quality assurance aspects of software production would make use of the analysis and design information produced at the initial stages of the development cycle (presumably stored on the software production data base).

Current verification systems such as RXVP [Deut81] have been developed capable of analysing systems both statically and dynamically. Functions performed by RXVP include producing graphs of control structures, insertion of 'software probes' to monitor system performance and behaviour, and measuring the 'completeness of testing.' The use of 'instrumenting' systems to determine 'system coverage' [Haun80] is another simple yet effective quality assurance tool.

Documentation

Documentation requirements exist throughout the software development cycle. Amongst the documents produced are project plans, functional and technical specifications, system design documents, programme flowcharts, and user and procedure manuals.

Generalised computer assisted documentation systems exist including word-processors and photo-typesetting systems. Some documentation systems, such as 'the Document Editor' [Walk81] have been designed

specifically for technical documentation. A number of documentation systems have also been developed to cater for specific stages within the software development cycle. For example, some of the specification and testing tools mentioned in the preceding discussion such as PSL/PSA and RXVP include documentation aids, and in fact the output generated by most tools may be included as part of system documentation.

One of the main purposes of documentation, particularly programme documentation, is to reduce the effort required to make modifications subsequent to a particular piece of software going into production. Automated programme documenters can enhance the readability of programmes by reformatting them or presenting them in one or other graphical form. Documentation tools such as 'the Linear Flowchart Generator' [Roy76], GREENPRINTS [Be1a80a], and CONTOUR [Gimp80] all reproduce on a line-printer, the original source code 'embedded' in some graphical representation. There are also a number of 'prettyprinting' systems [Mike81],[Rubi83] which display the programme source interactively on the terminal screen whilst it is being modified in order to provide a better overall view of the programme than is normally obtained using conventional editors.

Management

Perhaps the most important aspect of any software production effort is the management of the project. Generalised production management systems have been developed for assisting in the

manufacturing and construction industries, and the principles and computer aided tools involved, such as critical path analysis, project costing, and resource allocation, may be equally applied to the 'software manufacture and construction industry'.

Software production does however require some specialised management activities and correspondingly specialised tools for assistance. Amongst the problems of software production which management faces, are the quantitative determination of the intermediate states of development in terms of completeness and performance, and the measurement of the quality of the emerging and complete product.

As in the case of documentation, some of the tools covering a particular aspect of the software development cycle such as PSL/PSA also provide management information.

2.3 Summary

From this brief discussion, it should be obvious that there is a broad scope for the application of 'comprehensive' computer assisted software production environments. Growing awareness of the problems of software production has prompted a start to be made in this regard, and some directions for research efforts have been provided [Oste81].

Bibliography

- [Bela80a] Belady L A, Evangelisti C.J, Power L.R. **Greenprint—a graphic representation of structured programmes.** IBM Systems Journal, Vol 19 No 4 pp. 542-553 Nov, 1980.
- [Boeh76] Boehm B W. **Software Engineering.** IEEE Transactions on Computers Vol 25 No 12 December, 1976.
- [Buxt80] Buxton J.N. **An informal bibliography on programming support environments** ACM SIGPLAN Notices, Vol 15 No 12 pp. 17-30, December 1980.
- [Chap82] Chapman D. **A Program Testing Assistant.** Communications of the ACM, Vol 25 No 9 pp. 625-634 September 1982.
- [DoD80] United States Department of Defence. **Requirements for ADA Programming Support Environments, Stoneman.** 1980.
- [Deut81] Deutsch M S. **Software project verification and Validation.** IEEE COMPUTER, Vol 14 No 4 pp. 54-70, April 1981.
- [Dick78] Dickson M E, McGowan C L, Ross D T. **Software Design using SADT.** Structured Analysis and Design, Infotech Internation Limited, Volume 2 pp. 101-114, 1978.
- [Fras81] Fraser C W. **Syntax-Directed Editing of General Data Structures.** ACM SIGPLAN Notices, Vol 16 No 6 pp. 17-21, June 1981.
- [Glas82] Glass R L. **A Minimum Standard Software Toolset.** ACM SIGSOFT Software Engineering Notes, Vol 7 No 4 pp. 3-13, October 1982.
- [Gimp80] Gimpel J F. **Contour — a method of preparing structured flowcharts.** ACM SIGPLAN Notices, Vol 15 No 10 pp. 35-41, October 1980.
- [Heba78] Hebacker P G, Zelles S N. **Tell — A system for graphically representing software designs.** Research Report RJ2351, IBM Research Laboratory, San Jose, September 1978.
- [Houg83] Houghton R C. **Software Development Tools, a Profile.** IEEE COMPUTER Vol 16 No 5 pp. 63-70, May 1983.
- [Howd81] Howden W. **Contemporary software development environments.** ACM SIGSOFT Software engineering notes, Vol 6 No 4 pp. 6-15, August 1981.

- [Huan80] Huang J C **Instrumenting Programs for Symbolic-Trace Generation**. IEEE COMPUTER Vol 13 No 12. pp 17-23, December 1980.
- [Ivie77] Ivie E L. **The Programmer's Workbench - A Machine for Software Development**. Communications of the ACM, Vol 20 No 10 pp. 746-753, October 1977.
- [Mike81] Mikelson M. **Prettyprinting in an Interactive programming environment**. ACM SIGPLAN Notices, Vol 16 No 6 pp. 108-116, June 1981.
- [Morr81] Morris M.J., Schwartz M D. **The design of a Language directed editor for Block structured languages**. ACM SIGPLAN Notices, Vol 16 No 6 pp. 28-33, June 1981.
- [Ng78] Ng N. **A Graphical Editor for Programming Using Structured Charts**. Research Report RJ2344, IBM Research Laboratory, San Jose, September 1978.
- [Oste81] Oseterweil L. **Software Environment Research: Directions for the Next Five Years**. IEEE COMPUTER, Vol 14 No 4 pp. 35-43, April 1981.
- [Pong80] Pong M.C. **A System for Programming with Interactive Graphic Support**. MPhil thesis, Univ Hong Kong 1980.
- [Ross77] Ross D T. **Structured Analysis (SA): A language for Communicating Ideas**. IEEE Transactions on Software Engineering, Vol 3 No 1 pp. 16-34, January 1977.
- [Roy76] Roy P, St Dennis R. **Linear Flowchart Generator for a Structured Language**. ACM SIGPLAN Notices, Vol 11 No. 11 pp. 58-64, November 1976.
- [Rubi83] Rubin L F. **Syntax-directed Pretty Printing - A First Step Towards a Syntax Directed Editor**. IEEE Transaction on Software Engineering Vol 9 No 2 pp. 119-127, March 1983.
- [Shap81] Shapiro E, et al. **Pases: a Programming Environment for PASCAL**. ACM SIGPLAN Notices, Vol 16 No 8 pp. 50-57, Aug 1981.
- [Snow78] Snowdon R A, Henderson P. **The TOPD System for Computer Aided System Development**. Structured Analysis and Design, Infotech Internation Limited, Volume 2 pp. 285-305, 1978.
- [Stan81] Standish T. **Advanced Development Support Systems**. Software engineering notes ACM SIGSOFT. Vol 6 No 4 Aug 1982.

- [Teic77] Teichroew D, Hershey E A. **PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems.** IEEE Transactions on Software Engineering, Vol 3 No 1 pp. 41-48, January 1977.
- [Teit81a] Teitelman W, Masinter L. **The Interlisp Programming Environment.** IEEE COMPUTER, Vol 14 No 4 pp. 25-33, April 1981.
- [Teit81b] Teitelman T. **The Why and Wherefore of the CORNELL Program Synthesizer.** ACM SIGPLAN Notices, June 1981.
- [Teit81c] Teitelman T, Reps T. **The CORNELL program synthesizer.** Communications of the ACM, Vol 24 No 9 pp. 563-573, Sept 1981.
- [VanL76] Van Leer P. **Top-down Development using a Program Design Language.** IBM Systems Journal, Vol 15 No 2 pp. 155-170, 1976.
- [Van083] Van Oost E M J C. **DUIF: A Data-Oriented Flowchart Environment.** ACM SIGPLAN Notices, Vol 18 No 2 pp. 69-75, February 1983.
- [Walk81] Walker J H. **The Document Editor: A Support Environment for Preparing Technical Documents.** ACM SIGPLAN Notices, Vol 16 No 6 pp. 44-55, June 1981.
- [Wilc76] Wilcox T R, et al. **The Design of a Table Driven Interactive Diagnostic Programming System.** Communications of the ACM, Vol 19 No 11 pp. 609-616, November 1976
- [Wirt71] Wirth N. **Program development by Stepwise Refinement.** Communications of the ACM, Vol 14 No 4 pp. 221-227 April 1977.
- [Wood81] Wood S R. **Z-The 95% program editor.** ACM SIGPLAN Notices, Vol 16 No 6 pp. 1-7, June 1981.
- [Work83] Workman D A. **GRASP: A Software Development System using D-Charts.** Software proactice and experience Vol 13 pp. 17-32, 1983
- [Zelk81] Zelkowitz M. **High Level Language Programming Environments.** ACM SIGSOFT Software Engineering Notes, Vol 16 No 4 pp. 36-43, August 1981.

3. FLOW Diagrams

The FLOW system is a software production environment which makes extensive use of annotated diagrams for expressing and experimenting with software designs. The diagrams used are derived from Nassi Shneiderman Diagrams (NSDs) [Nasi73], and include some of the modifications made to these by Chapin [Chap74].

This particular 'style' of diagram was chosen because

- it provides a compact and concise method of depicting the procedures of structured programmes.
- it has received widespread support as a flowcharting technique.
- it can be generated and manipulated on computer devices.
- it can be extended to depict data structures

In order to generate NSDs on a computer, and to represent data structures, a generalised form of NSD has been developed, which will be referred to as a 'GNSD.'

3.1 Modifications to NSDs

Some of the modifications made to NSDs in creating GNSDs are identical to those adopted in the implementation of the linear flowchart generator of Roy and St Dennis [Roy76], and the programme generators of Ng [Ng78] and Pong [Pong80], and were obviously required by these authors for the same reasons as were found during implementation of the FLOW environment, namely to allow their generation by computers. These modification had two purposes:

- to reduce the number of lines needed in order to be able to display the flowchart on a typical 24 X 80 character CRT display. (The problem here is that the display length is limited to 24 lines, and it is therefore important that the information content displayed on the screen be as high as possible).
- to reduce the length of printed listings. Experience has shown that listings of the NSD produced on a line printer should not be significantly longer than the conventional programme source code listing if it is to become a viable replacement to programme listings as the programmer's working document.

The GNSDs as implemented for both CRT displays and line printer listings have been designed to make use of the standard character sets provided by these devices. GNSD constructs together with their corresponding 'structured programming' counterparts (given

adjacently in Pascal) are as follows;

(a) Process block.

The process block is the basic component of the GNSD.

The block is shown as

GNSD	Pascal
<pre> +-----+ +-----+ </pre>	<pre> begin end; </pre>

As in structured programming, blocks may be nested.

(b) Sequence

Sequence is shown as a series of lines or blocks enclosed within a box.

GNSD	Pascal
<pre> +-----+ +-----+ +-----+ +-----+ </pre>	<pre> begin end; begin end; begin end; </pre>

(c) Iteration

Iteration is shown as

GNSD

```

+-----+
| <condition> |
|/+-----+
|/! ... |
|/! ... |
+-----+

```

Pascal

```

while <condition> do
  begin
    ...
    ...
  end;
(*endwhile*)

```

(d) Selection

Four constructs for selection are used in GNSDs, namely unary selection, binary selection and two constructs for n-ary selection).

Unary selection**GNSD**

```

+-----+
| IF ? <condition> |
+-----\T/-----+ \F/-----+
| ... | |
| ... | |
+-----+-----+

```

Pascal

```

if <condition> then
  begin
    ...
    ...
  end;
(*endif*)

```

Binary selection

```

+-----+
| IF ? <condition> |
+-----\T/-----+ \F/-----+
| ... | | |
| ... | |
| | | ... |
| | | ... |
+-----+-----+

```

```

if <condition> then
  begin
    ...
    ...
  end
else
  begin
    ...
    ...
  end;
(*endif*)

```

N-ary selection (construct 1)**GNSD**

```

+-----+
| Case <expression> |
+--\case1/+-\case2/+-\case3/+-\case4/+-+
| ... | | | | |
| | | | |
| | | | |
| | | | |
+-----+

```

Pascal

```

case <expression> of
  case1: begin
    ...
  end;
  case2: begin
    ...
  end;
  case3: begin
    ...
  end;
  case4: begin
    ...
  end
end;(*case*)

```

This first construct is applied only where a limited number of selection possibilities exist. Where more than 4 possibilities exist a notation similar to that of Roy and St. Dennis [Roy76] is used as follows

N-ary selection (construct 2)

```

+-----+
| Case <case variable> |
| +-----+ |
| case 1 > |
| | ... |
| | ... |
| +-----+ |
| case 2 > |
| | ... |
| | ... |
| +-----+ |
| case n > |
| | ... |
| | ... |
+-----+

```

Although there exists the possibility of confusion arising from having two seemingly different notations for the same construct, the second notation should be seen as a horizontal arrangement of the first, and not as a different construct.

The similarity between the representations, of unary (IF-THEN), binary (IF-THEN-ELSE) and n-ary (CASE) selections in GNSDs is intentional.

In support of the horizontal layout of n-ary selection, it should be noted that this construct is consistent with other representations of procedures such as pseudo code [Jens79] and has the advantage over the vertical arrangement in that blocks are presented in the same order as they would appear in the source code listings of the Pascal programme coded from them. The horizontal layout of n-ary selection for $n > 1$ has the additional advantage of overcoming one of the major criticisms of NSDs, namely the rapid diminishing of block sizes when selections are used [Bela80b].

The horizontal arrangement appears in other graphical representations such as CONTOUR [Gimp80] and GREENPRINTS [Bela80a], and is obviously a very useful construct.

(e) Procedure invocation

One of the problems of producing NSDs on printers and terminals is that the symbol used to denote procedure invocation is an ellipse. This is acceptable for hand-drawn diagrams (for which NSDs were designed), but is not possible without graphic devices for computer generation. Although an attempt could be made to draw a crude elliptical shape using the standard printer character set, this would require more than one line to be printed for a procedure call. For the reasons given above, a more compact representation was chosen for GNSDs as follows

```
+-----+
|      .....      |
|      <<procname>> |
|      .....      |
+-----+
```

or

```
+-----+
|      .....      |
|      <<procname/parameter-list>> |
|      .....      |
+-----+
```

where `procname` is the procedure name
 `parameter-list` is the list of calling arguments

(f) Input/output

The GNSD view of input/output is as a special form of procedure call. The representation of input/output operation is as follows

```

+-----+
|      .....      |
|      << io-op io-chan/io-list-and-format>>      |
|      .....      |
+-----+

```

where io-op is either INP for input, or OUT for output,
 io-chan is the channel on which the io occurs, and
 io-list-and-format is the list of variables and/or
 formats

As an example, an output statement is written as

```
<< OUT printer/ " THE ANSWER IS ",X:F6.2,Y:F6.2,Z:F6.2>>
```

(g) Procedure definition

One of the modifications to NSDs made by Chapin [Chap74] is that the procedure name appears in an elliptical shape placed above the procedure. The obvious intention is that the ellipse should form the link between the invocation and the definition of the procedure. Following this idea of Chapin's, the procedure invocation and the procedure definition of GNSDs use the same symbols, so that procedure definitions are shown as

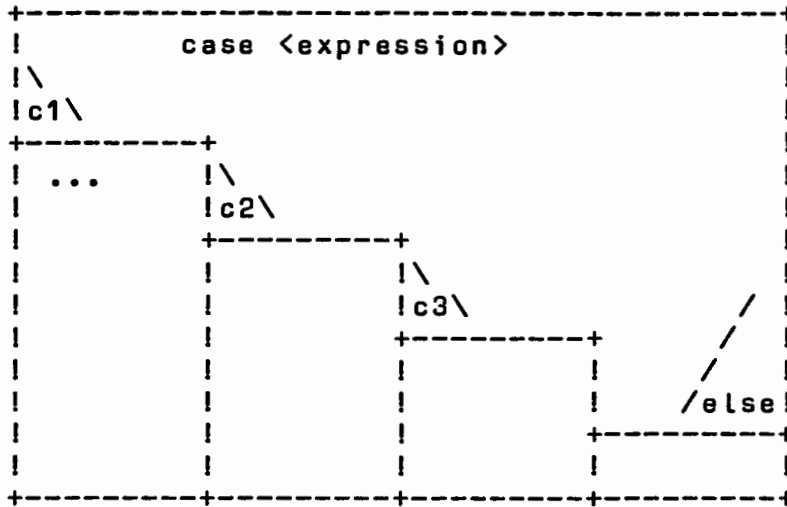
```

+-----<< procname >>-----+
| ..... |
+-----+

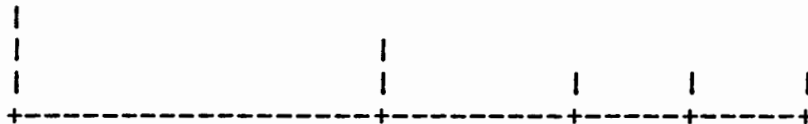
```

Applicability of GNSDs

Although GNSDs were designed primarily to be used for printers and display terminals, the GNSD modifications are equally applicable to hand-drawn flowcharts. The advantages of GNSDs over the original representation is readily observed in the case of the CASE construct; the original case construct of Nassi and Shneiderman being



If one compares the GNSD with the PASCAL source code counterparts, the flowchart listings are on average considerably shorter. This fact is more pronounced at the termination of nested IF-statements which would be shown on the GNSD as for example



without loss of ambiguity, whereas the Pascal code with comments used to determine the scope of blocks would be shown as

```

        end;
      (*endif*)
    end;
  (*endif*)
end;
(*endif*)

```

Pascal does allow programmes to be written with multiple statements on the same line, and thus Pascal programmes could be written

very compactly if desired. Furthermore, the 'pseudo operations' as used above ((*endif*)) are inserted only to enhance readability, and may be omitted. However for the equivalent clarity, the argument that the GNSDs are more compact than the source code still holds.

3.2 FLOW-DL: The language for specifying GNSDs.

The definition of a new 'language' which uses graphic generating function keys instead of key words is central to the implementation of the FLOW environment. It was deemed necessary to design a new language in order to provide a unifying factor between the various components proposed for the creation of the FLOW environment. In the implementation of the environment that followed, FLOW-DL was used as the input language for the syntax directed editor FLOW-EDIT, and was translated into FORTRAN, COBOL or Pascal source code, and 'hard-copy' flowchart listings by a number of post-processors.

Design criteria

FLOW-DL was designed as a block structured language with a syntax similar to that of Pascal (except that the blocks are represented graphically).

The design criteria used in defining FLOW-DL were:

- to facilitate the interactive construction of GNSDs using graphics generating function keys instead of keywords.

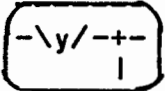
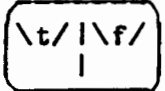
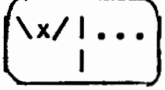
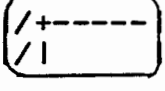
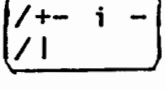
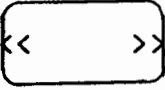
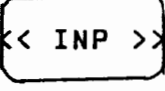
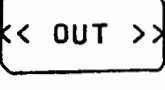
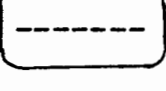
- to remove some of the error-prone syntax found in languages such as Pascal and COBOL. e.g. ??? "how to get schlonked by Pascal" SIGPLAN notices 17 (Dec 1982) pp 31-40???

- to provide shortened (single character) keywords for those aspects of the language not catered for by function keys.

- to be general enough to facilitate the generation of multi-lingual code.

FLOW-DL Function Keys

The function keys used by FLOW-DL are as follows

operation	key symbol	key-name
unary selection		IF-THEN
binary selection		IF-THEN-ELSE
n-ary selection		CASE
repetition		DO-WHILE
iteration		ITERATE
procedure call		CALL
input		INPUT
output		OUTPUT
end		END

In what follows, function keys will be denoted by the key name enclosed in square braces, e.g. [DO-WHILE] will represent the repetition key.

Operation of function keys

Function keys together with lines of text are used to create GNSDs. The [IF-THEN] [IF-THEN-ELSE] [CASE] [DO-WHILE] and [ITERATE] keys initiate the start of a new diagram element within the current block boundaries. When used for creating a GNSD on a display terminal, the initial block boundaries are defined by the edges of the terminal screen. The text line immediately following one of these function keys is taken as a logical expression or iteration expression (depending on the key pressed). The second and subsequent text lines will be placed within the body of the GNSD created by the function key. Text lines within the GNSD body may be either assignment statements or comments. The [CALL] [INPUT] and [OUTPUT] keys may also be used.

If a [CALL] [INPUT] or [OUTPUT] key is used, the line of text that follows defines the called procedure name and parameters, or input/output channel, parameters and format, as appropriate.

The [END] key may also be used within a GNSD body to terminate the current block. Since GNSD blocks may be nested, the [IF-THEN] etc. keys may also be used, each requiring one or more [END] keys to terminate blocks created.

As an example of flowchart construction using these keys, the sequence of function keys and statements shown on the left may be used to generate the flowchart on the right.

Key sequence	GNSD produced
\$ Initialize	! \$ Initialize !
x { 1	! x { 1 !
s[1] { 1	! s[1] { 1 !
[ITERATE] i = 2 to n	! i = 2 to n !
l[x] { l[x] + 1 + p[i]	! l[x] { l[x] + 1 + p[i] !
[IF-THEN] l[x] > z	! l[x] > z !
x { x + 1	! x { x + 1 !
s[x] { i	! s[x] { i !
[END]	! /+-----\y/-----+ \n/ -+ !
[END]	! /+-----+ !

FLOW-DL Statements

Although the FLOW-DL syntax rules are derived from PASCAL, there are some variations worth noting.

- FLOW-DL uses the ' { ' symbol to denote assignment instead of the ' := ' used by Pascal. (FLOW-DL uses two other symbols in a slightly different 'assignment' context, namely the ' } ' and ' : ' symbols. The usage and motivation for these is discussed later under the discussion on 'procedural data structures').

- The ' \$ ' symbol is used to indicate the beginning of a comment. The end of the comment is the end of the line, and the Pascal problem of 'unclosed comments' does not

arise [Cail82]. The '\$' provides an 'in-line' commenting facility (similar to that of FORTRAN 77), which encourages the writing of highly readable code. 'Proper' comment lines are allowed (COBOL and FORTRAN IV style) by beginning a new line with a \$.

- Each statement is entered on a separate line (in FORTRAN style). The end of a line is marked with the characters <newline> or <carriage-return> i.e. the natural line terminators. This 'one statement per line' policy is in the interests of readability. Continuation over more than one line is denoted by the '&' symbol at the end of the line of a partially completed statement.

FLOW-DL Procedure and I/O invocation

The most important differences between FLOW-DL and Pascal in the invocation of procedures and I/O are

- The parameter list in a procedure call is separated from the procedure name by the '/' symbol. The [CALL] function key generates the construct << >>, so that the entering of a procedure call in FLOW-DL requires the [CALL] key followed by the procedure name and parameters e.g. '[CALL]XYZ / R,S,T' results in the construct '<<XYZ / R,S,T>>' being created.

- The I/O channel, I/O variable list and format specification

are separated by the ' / ' character. The [INPUT] or [OUTPUT] function keys define the I/O operation.

FLOW-DL Data types

For data declarations, the predefined data types have been shortened in FLOW-DL to a single character as follows:

I for Integer
F for Floating point (Real)
L for Logical (Boolean)
A for ASCII (Character string).

Data declaration statements are given in the form

<variable> { <type><format>

e.g. count { I3
 mean-value { F6.3

Further aspects of FLOW-DL data declarations are discussed in subsequent sections of this chapter.

FLOW-DL as a productivity tool

FLOW-DL as a language contributes to increased productivity for the following reasons:

- FLOW-DL is based on NSDs, and since NSDs were designed specifically for structured programming, the user of FLOW-DL is forced to use structured techniques. It is generally

accepted that structured programming leads to a significant increase in programming and maintenance productivity, and FLOW-DL 'inherits' this benefit.

- The graphical representation of FLOW-DL clearly shows the scope of each iteration and selection block. The unambiguous display of the extent (scope) of each block eliminates many of the programme nesting errors that can be made during coding, and, more importantly, eliminates most nesting errors made during programme modification. The graphical representation, together with the simplified syntax (namely the omission of line and block separators such as ; and end), overcomes the common problem where the programme source code indentation (i.e. programmer's intended meaning) does not corresponding to the programme semantics (i.e. compiler's interpretation). A good (and extremely common) example of this problem is the misplaced period in a COBOL nested IF statement. The importance of a language feature which assists the programmer in producing both syntactically and semantically correct programmes cannot be overstated.

- The use of 'function keys' and shortened key-words eliminates spelling errors without the language becoming cryptic. (In fact, if one considers that a repetition function key generates a large amount of 'scope' information on each subsequent statement by virtue of the graphical output produced, FLOW-DL is, on the contrary, verbose.)

3.3 Using GNSDs for DATA definitions

As noted earlier, GNSDs provide a compact and concise representation of procedure structures. It is contended that these representations (GNSDs) are equally applicable to the representation of data structures.

The Nassi Shneiderman constructs for representing sequence, selection, and iteration are extended in GNSDs to represent data structures as follows:

- the **sequence** construct is used to represent variables

for example:

```
+-----+
| tax-ref { 13_digits |
| name    { 10_chars  |
+-----+
```

- the **iteration** construct is used to represent arrays

for example:

```
+-----*Table*-----+
| i=1..50 |
|/+-----+
|/!Table(i) { 5_chars!
+-----+
```

- the **selection** construct is used to represent variant records. (Binary or N-ary selection may be required).

For example:

```
+-----+
|                variant                |
|--\usage a/---+---\usage b/---|
| a { 3_chars | b { 3_digits |
+-----+
```

- the **procedures definition** construct is used to represent records

For example

```
+-----<<name>>-----+
| surname { 30_chars |
| initials{ 4_chars |
+-----+
```

- the **procedure invocation** construct is used to represent sub-records within records.

For example:

```
+-----<<personnel record>>-----+
|                <<name>>                |
| age { 3_digits |
|                <<address>>            |
+-----+
```

The 'sub-record' concept requires some explanation. The term 'sub-record' is used to define a record containing a record as a component. For example, Pascal records may contain 'sub-records' as in the following example.

```

personnel = record  pname:name;
                age:0..150;
                paddress:address
                end;

```

where name is defined as

```

name = record  surname:array[1..30] of char;
                initial:array[1..4] of char
                end

```

The 'sub-record' invocation may be replaced by the actual definition of the 'sub-record' to produce the following notation

```

+-----<<personnel record>>-----+
|                                     |
| +-----<<name>>-----+           |
| | surname { 30_chars           | |
| | initials { 4_chars           | |
| +-----+                       | |
|                                     |
| age { 3_digits                 | |
| +-----<<address>>-----+       |
| | i=1..4                       | |
| |/+-----+                     | |
| |/! addressline(i) { 26_chars   | |
| +-----+                       | |
+-----+

```

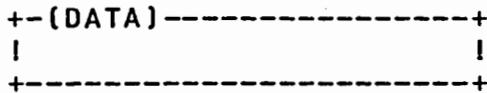
The important point to note in this GNSD data notation of 'sub-records' within records is that it is the equivalent of the procedural construct of 'internal procedures' in languages such as Pascal.

This 'sub-record' notation is not new. Group items are defined in languages such as COBOL and the preceding GNSD for 'personnel record' would be coded in COBOL as

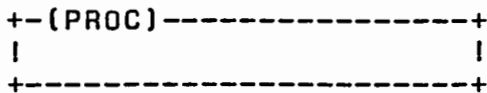
```

01 PERSONNEL-RECORD.
  02 NAME.
    03 SURNAME          PIC X(30).
    03 INITIALS        PIC X(4).
  02 AGE                PIC 9(3).
  02 ADDRESS            OCCURS 4 TIMES.
    03 ADDRESS-LINE    PIC X(24).
    
```

In order to avoid confusion between the data and procedure GNSDs, since the GNSD constructs may be used to represent both, the GNSD data structures are shown as



and the GNSD procedures are shown as



Application of GNSD data structures

The use of GNSDs for data representation may appear somewhat contrived and of little practical value. However the usefulness of this representation is borne out in two important applications.

Firstly, Data GNSDs are a useful tool for documenting data structures. Since the data structures are enclosed within boxes, the scope of blocks of data (records), such as the content of a COBOL OCCURS (shown as an iteration in GNSD), and COBOL 01 to 49 group levels (shown as nested procedures) is readily observed. This is of particular importance where the data definition section of a programme listing may extend over more than one page of source. The Data GNSD notation is also useful in showing those parts of the data structure which are REDEFINED in COBOL, EQUIVALENCED in FORTRAN, or are variant records in Pascal (shown as selection in GNSD), since the REDEFINED etc. records are contrasted against the portions of the data structure which redefine them.

The second, and perhaps more significant application of Data GNSDs is that it is possible to implement data structures as the equivalent Procedure GNSDs using procedures for producing and consuming data streams. This leads to a number of applications of 'procedural' data structures which are described below. The FORTRAN IV code produced by FLOW-F4 in the implementation of records, described in chapter 8, is one such example.

3.4 Procedural Data structures.

The similarity between procedures and data structures which is obvious from the GNSD representations leads to some observations on the procedural nature of data structures.

This procedural nature of the data structures is demonstrated by considering the following pair of assignment statements

```
+--(PROC)-----+
!  previous_salary { current_salary  !
!  current_salary  { 12345.67       !
+-----+
```

In the first statement, a value (say 12000.00) is produced by 'current_salary', and assigned to 'previous_salary'. In the second statement, the 'current_salary' consumes the value 12345.67. Thus data structures may be classified as either producers or consumers of data, depending on which side of the assignment symbol they appear.

To represent this data structure, the following notation is used

```
+-(data)-----+
! current_salary : 5.2_digits !
+-----+
```

read as 'current_salary produces and consumes a number 5 digits in length with 2 decimal digits'. (The symbol ' : ' representing

'consumes and produces')

It is not always true, however, that a data structure can be both a producer and a consumer. For example, the following data structure is given for (an approximation of) a well known constant which is a 'produce only' data structure

```

+--(data)-----+
! pi } 3.14159      !
+-----+

```

This notation can be used to define Pascal constants, FORTRAN parameters etc.

Some 'write only' data structures can also be defined. For example if a data structure is defined representing a line printer, the writing of a line to the printer could be defined as

```

+--(proc)-----+
! line_printer { printext  !
+-----+

```

The producer data structure concept defined above may be extended even further to include the concept of 'virtual data production' as demonstrated in the example of a sales record defined as

```

+--(data)----<<Sales_record>>-----+
! ....                                !
! costprice : 10.2_digits             !
! tax = costprice * 0.06 } 10.2_digits!
! ....                                !
+-----+

```

In this example, 'tax' is derived from 'costprice', but the data structure is not required to have storage allocated to hold the value of 'tax'. The notation used here for the 'production' of 'tax' gives both the formula for calculating the tax amount, as well as the definition of the tax 'producer' format.

As stated earlier, data structures and procedures can be considered equivalent. The equivalence of the 'procedural' data structures and conventional procedures is demonstrated in the following

```

+--[data]-----<<personnel_record>>-----+
| ...                                         |
| name : 20_chars                             |
+-----+
|   i=1..4                                   |
|/-----+
|/! address[i] : 10_chars                    |
+-----+
| status : 7_chars                           |
+-----+
|           status = 'married'               |
+-----\y/-----+-----\n/-----+
| wifename: 20_chars                           |
+-----+

+--[proc]---<<produce_personnel_record>>---+
| ...                                         |
|   <<OUT/buffer/name / 20_chars>>           |
+-----+
|   i=1..4                                   |
|/-----+
|/! <<OUT/buffer/address[i] / 10_chars >> |
+-----+
|   <<OUT/buffer/status / 7_chars>>         |
+-----+
|           status = 'married'               |
+-----\y/-----+-----\n/-----+
| <<OUT/buffer/wifename/: 20_chars>>|
+-----+

```


- Variable length records: In this case the data structure would consist of an iteration up to the size of the record.

- Classical 'abstract' data objects such as stacks: The implementation of abstract data types is easily achieved, and the use (in the procedure) of the abstract data object so defined becomes an assignment statements as in the example:

```
stack { value           $ equivalent to 'push'
value { stack          $ equivalent to 'pop'
```

- Self-validating data structures: It should be possible for the data structure to validate the data during a 'consume' operation.

Of these possibilities for 'smart' data structures, only the last one (self validating data structures) presents any practical problem, namely how to handle the exception case. (The fact that a large number of the current language implementations fail to handle exceptions such as number overflow, underflow and type conversion other than by reporting a 'runtime error' aggravates the problem). One solution is to write the exception handling procedure as part of the data structure itself. (This is possible because of the procedural nature of data structures!) Another possible solution is to provide an 'error return' as a standard part of an assignment statement (similar to the 'CALL OVERFLOW' provided in FORTRAN). Thus, assignment statements would be represented as

```

+--(proc)-----+-----+
| personnel_rec { card_reader    >> err >>      |
| ...                |                          |
+-----+-----+

```

The blurring of the distinction between data structures and procedures leads to a radical change in design and programming methodologies. In fact, it is exactly this attribute of data structures on which the 'data oriented' design methodologies are based.

For example, this view of data structures as procedures concurs with Jackson's System Development method (JSD) [Jack78] [Came82] and Jackson's Structured Programming (JSP) techniques [Jack75] [Jack76], as well as other schools of the data oriented approach such as the Structured System Design of Warnier and Orr [Orr77]. For example, in JSD, the data objects are 'modelled' by procedures. The 'procedural data structures' can also be used to model data structures. On the other hand, JSP analyses the inputs and outputs of a data processing problem, and arranges them on a hierarchy diagram using sequences, selections and iterations. Procedures are then written corresponding to the diagram. The inputs and outputs could just as easily be considered 'procedural data structures'.

The GNSD approach may therefore be used for both procedure oriented and data oriented design and programming methods. In fact, if the 'procedural data structure view is taken, the two are seen to be equivalent; this suggests that 'hybrid' approaches

are possible. This topic is discussed briefly in chapter 11.

3.5 Procedure interfaces

If one considers how procedures actually operate, it can be seen that they too act as consumers and producers of data when they pass and return parameters. The GNSD notation adopted for data structures is extended for use in interface definitions. The interface of a procedure (if parameters are used) is shown as

```
+---(i/f)-----+
! number } 10.2_digits      !
! root   { 10.2_digits      !
! counter: 10_digits        !
+-----+
```

where number is an input parameter (i.e. a producer only, and therefore treated as a constant), root is an output parameter (i.e. consumer, and therefore restricted to appearing on the left hand side of assignment statements), and counter is both an input and output parameter (which may be used as an ordinary variable). The notation (i/f) is used to distinguish the interface from the data and procedure definition GNSDs.

The full definition of a programme is therefore given by three separate GNSD boxes, namely, the interface definition, the data definition and the procedure definition. Each of these will in turn consist of component GNSD boxes.

Bibliography

- [Bela80a] Belady L A, Evangelisti C.J, Power L.R. **Greenprint - a graphic representation of structured programmes.** IBM Systems Journal, Vol 19 No 4 pp. 542-553 Nov, 1980.
- [Bela80b] Belady L A. **Modifiability of large software systems.** IBM Research Report RC8525, August 1980.
- [Cail82] Cailliau R. **How to Avoid Getting Schlonked by Pascal.** ACM SIGPLAN Notices, Vol 17 No 12 pp. 31-40, December, 1982.
- [Came82] Cameron J.R. **Two Pairs of Examples in the Jackson Approach to System Development.** Proceedings of 15th Annual Hawaii International Conference on System Sciences, 1982.
- [Chap74] Chapin N. **New format for flowcharts.** Software Practice and Experience, Vol 6 No 4. pp.341-357, 1974.
- [Gimp80] Gimpel J F. **Contour - a method of preparing structured flowcharts.** ACM SIGPLAN Notices, Vol 15 No 10 pp. 35-41, October 1980.
- [Jack76] Jackson M A. **Constructive Methods of Programme Design.** Lecture Notes in Computer Science, Vol 44, Springer Verlag Inc, New York, 1976.
- [Jack78] Jackson M A. **Information systems: Modelling, Sequencing and Transformations.** Preceedings of the Third International conference on Software Engineering, May 1978.
- [Jack75] Jackson M A. **Principles of program Design.** Academic Press, 1975.
- [Jens79] Jensen R W, Tonies C C. **Software Engineering.** Prentice Hall, Engelwood Cliffs, New Jersey, 1979.
- [Nass73] Nassi I, Shneiderman B. **Flowchart Techniques for Structured Programs.** ACM SIGPLAN Notices, Vol 8 No. 8 pp. 12-26, August 1973.
- [Ng78] Ng N. **A Graphical Editor for Programming Using Structured Charts.** Research Report RJ2344, IBM Research Laboratory, San Jose, September 1978.
- [Orr77] Orr K T. **Using Structured System Design.** Structured System Development, Yourdon Press, pp. 81-106, 1977.
- [Pong80] Pong M.C. **A System for Programming with Interactive Graphic Support.** MPhil thesis, Univ Hong Kong 1980.

[Roy76] Roy P, St Dennis R. **Linear Flowchart Generator for a Structured Language**. ACM SIGPLAN Notices, Vol 11 No. 11 pp. 58-64, November 1976.

4. The FLOW methodology for software production.

4.1 The software production model

One of the requirements of a software production environment is that it should encourage and support one or more software production methodologies. The methodology of top down design, coding and testing based on functional decomposition [Your75] was chosen as the software production technique to be supported explicitly by the FLOW environment. This methodology was selected because of its general acceptance in the software engineering field, and also because many of the components needed to implement it lend themselves to computerisation.

In order to understand and build an effective support environments, it is useful to build a model of the software production process or processes to be supported. Many different models have been proposed for the activities which constitute the software production process e.g. [Jens79]. In the model chosen for the implementation of the FLOW environment, the major activities are identified as

- the definition of requirements (functional specification)

- the designing of a software product which will meet the requirements (system design)

- the implementation of the design (programming and debugging)
- the building of the complete system (system integration)
- testing of the product against the design specifications (system testing)
- the description of the product (documentation)

These activities are performed iteratively with some additional activities being introduced to achieve top-down design, coding and testing.

The following flow diagram describes the software production process as applied within the FLOW environment, and will be referred to as the FLOW model for software production.

```

+-----+
| requirements-definition |
| top-level-system-design |
+-----+
| while software product is incomplete |
|/+-----+
|/| programme to the current level of refinement |
|/| define interfaces to next level of refinement |
|/| build intermediate system at current level of refinement |
|/| test and debug system at current level of refinement |
|/+-----+
|/| is emerging software product ok ? |
|/+-----\Y/-----+-----\N/-----+
|/| document current level | review |
|/| refine design | backtrack |
+-----+
| produce user documentation |
+-----+

```

Of the activities defined in the model, the following are given explicit support by the FLOW environment.

- system design
- programming
- intermediate system building
- debugging and testing
- system documentation

4.2 Software production - the movie.

In order to demonstrate how the FLOW environment supports the software production process given in the model, the use of the FLOW environment will be shown by means of some 'frames' of the 'movie' as displayed on the terminal screens by the FLOW programmes during this process.

The development of a laboratory quality control system (which will be called QC) is used as an example. This example deals with the development of one module of QC called QCPLLOT. The purpose of this example is to demonstrate some of the features of FLOW-DL and the FLOW environment, and the design and development of QCPLLOT contains deliberate errors and design flaws.

The requirements for QCPLLOT may be summarised as follows:

QCPLLOT is required to read the means and standard deviations of batches of 'control' standards (i.e. substances with known parameters) as well as the parameters determined by various analyses for these controls. Control charts must then be plotted for the various controls as selected. QCPLLOT must return an error parameter to its calling procedure if some exception occurs.

System Design

To begin our development of QCPLLOT, the environment manager, FLOW, is invoked. FLOW displays the following screen:

```
* * * FLOW Environment * * *
```

```
Enter Command:EDIT QCPLLOT
```

```
For Help type: HELP
```

```
Current name is:
```

The command **EDIT QCPLLOT** is given. (User responses are displayed in full intensity by the FLOW programmes, and system generated output is shown at half intensity). The editor, FLOW-EDIT is used to create and modify the descriptions of modules within the environment. FLOW passes control to FLOW-EDIT which displays:

```
* * * FLOW EDIT * * *
```

```
Name QCPLLOT is new ...
```

```
Enter Type: .TS
```

```
.TM - Type is Main procedure
```

```
.TS - Type is Sub procedure
```

```
.TC - Type is Copy Procedure
```

```
.TD - Type is Data block (global)
```

```
For Help type .H
```

```
Current name is: QCPLLOT
```

Since the name 'QCPLLOT' is being used for the first time, its type must be specified. The four types of FLOW-DL source that may be selected are

- Main procedures: These procedures correspond to Pascal, COBOL and FORTRAN main programmes.

- Sub procedures: These procedures correspond to subroutines in FORTRAN, internal procedures in Pascal, and paragraphs or sections in COBOL. (note that COBOL subroutines are not used)

- Copy procedures: These procedures are lines of commonly used source code that may be included at any point within a procedure definition. They are used in that same way as the COBOL COPY statement used in the procedure division.

- Data blocks: These are data definitions which will typically be used in more than one programme or procedure. They are useful in defining record layouts, and may be considered as data base schemas in some respects. In the FORTRAN context, they correspond to named common blocks. In the COBOL context they correspond to COPY statements used in the WORKING STORAGE SECTION.

Since QCPLLOT is to be a sub procedure, .TS is entered. This creates an entry in the system data base for the name QCPLLOT, and marks it as a sub-procedure.

Having defined the type of QCPLLOT, the following screen appears:

```
* * * FLOW EDIT * * *
```

```
Enter Command:[]
```

```
For Help type .H
```

```
Current name is:QCPLLOT
```

When the type specification is given, a 'skeleton form' is created on the software development data base. In this case a 'sub-procedure' was specified which created four 'blocks' on the data base namely, a functional specification block, an interface definition block, a data definition block, and a procedure definition block. The contents of these blocks are entered in any sequence, and the user may jump from one block to another at any stage by giving a single command. In this example, the interface definition is to be entered first and the **display-interface** command is given.

The empty interface specification block is displayed.

Name:QCPLOT	Command:DISPLAY
--<<Interface definition>>-----	
[]	

The cursor (shown as []) indicates the place at which text is to be entered. In the case of QCPLOT, a single parameter ERROR is to be returned through the interface. The definition of the parameter is entered using data definition statements of FLOW-DL the FLOW environment language.

Name:QCPL0T

Command:INSERT

```
--<<Interface definition>>-----  
ERROR ] I3      $ Return a 3 digit error code  
[]
```

Note that the symbol '] ' in FLOW-DL means that the data item is an output parameter. The next block of the specification that is to be entered is the functional specification. The command **display functional specification** is given. The 'skeleton form' of the functional specification presents the topics that should be entered as a 'fill in the blocks' screen.

```

Name:QCPLOT      Command:DISPLAY
-  

--<<functional specification>>-----
+--<<Functional Description>>-----+
+-----+
+--<<Input Description>>-----+
+-----+
+--<<Output Description>>-----+
+-----+
[ ]

```

At this stage the cursor is positioned outside of the description blocks. The cursor may be moved to any of the blocks for text to be inserted using the commands **Move Up**, **Move Down**, **Move Left** and **Move Right**. These commands may also be followed by an integer giving the number of rows/columns to be moved. e.g. **Move up 6** moves the cursor up 6 rows.

A status line, shown at the top line of the display, gives the name of the procedure being edited, as well as the last command entered. The last command may be repeated by pressing the carriage-return key. This is very useful in cursor positioning; after

selecting the direction in which to move, repeated movement of the cursor is simplified to pressing one key. Once the cursor has been positioned, text may be entered.

Name:QC PLOT

Command:DOWN 1

```

--<<functional specification>>-----
+--<<Functional Description>>-----+
| QC PLOT is the laboratory Quality Control system |
| PLOT utility. QC on different analytical methods |
| is achieved by analysing substances with known |
| parameters. The results of these 'controls' |
| are plotted on a Shewart Chart with the control |
| lines drawn for the mean +- 2 standard deviations. |
| Statistics are provided by the manufacturers of the |
| 'control' substances. |
+-----+

+--<<Input Description>>-----+
+ [ ] +

+--<<Output Description>>-----+
+-----+

```

Further descriptions may be added at a later stage. Presuming that no further descriptions is to be given, the command **Display data** is entered to access the data definition block.

```
Name: QCPL0T  Command: DISPLAY
```

```
--<<Data Definition>>-----
```

```
[ ]
```

The data definition given in the design phase, will typically consist only of comments and some 'global data' declarations. Comments are preceded by the symbol \$. Global data declarations are included by preceding the global data block name with the symbol %.

```
Name: QCPL0T      Command:INSERT      % BLOCK NOT FOUND
```

```
--<<Data Declaration>>-----
$      ***** GLOBAL DATA *****
$ Include the standard variable names for I/O devices
%IONAMES
$ Include the global data defined for the QC results and QC
$ means, standard deviations, and control substance names
%QCDEFN
%QCRSULT
```

FLOW-EDIT verifies the existence of all global data blocks. When %QCRSULT was entered, for example, an attempt was made to find it in the entry point table. Since it is not defined (it should have been spelled QCRESULT), an error message is given on the status line. The cursor is then positioned to the incorrect line, and a spelling correction is made.

As well as checking for the existence of data blocks which are to be 'included', FLOW-EDIT also checks that they are of the correct type i.e. defined as global if they are to be used in the data definition block.

After defining the global data to be used by QCPL0T, the command **display procedure** is given.

Name:QCPLOT	Command:DISPLAY
---<<Procedure>>---	
[]	

In the application of the methodology defined by the software production model, the design specifications are entered as skeleton flowcharts using FLOW-DL 'comments' and a few basic structures.

For our procedure design specification of QCPLOT, we wish to specify some initialization to open the QC data files etc. After entering a comment to this effect, the [CALL] function key is pressed.

Name:QCPLOT

Command:CALL

--<<Procedure>>-----

\$ Initialize

<<[]

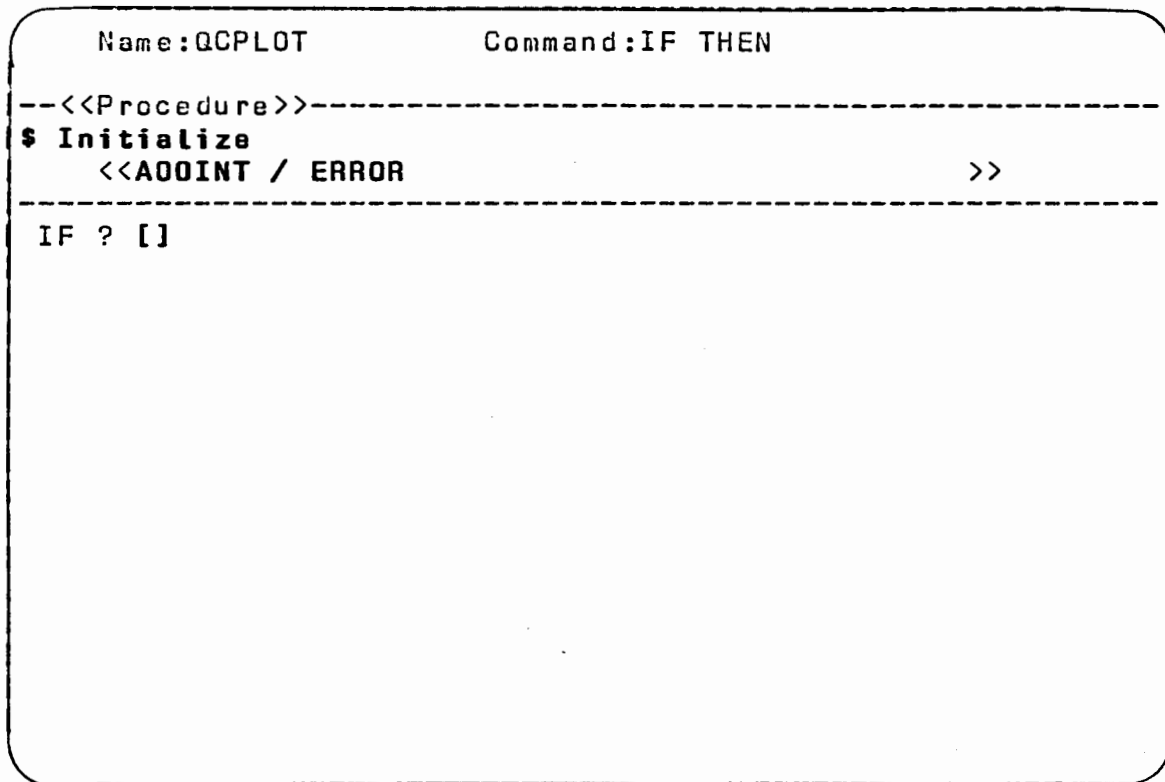
>>

The initialization procedure AOOINT (which has not yet been defined) is to be used, and AOOINT is to return an error parameter.

```
Name:QCPLLOT          Command:CALL
--<<Procedure>>-----
$ Initialize
  <<A00INT / ERROR          >>
```

Since the procedure name A00INT is not currently defined in the entry point table, a message to this effect is displayed on the status line. The cursor is then moved to the status line for the type to be specified. Next FLOW-EDIT will ask for the 'new' interface specification to be given. The call A00INT/ERROR is then checked against this specification.

Continuing with the design, the next step is to indicate that the error condition is to be tested. The [IF-THEN] function key is pressed, and the IF ? prompt is displayed:



The conditional statement for the selection is then entered, and the beginnings of the two condition blocks (Y and N) appear.

Note that the N block will not be used, and is therefore reduced in width to provide more space for the Y block.

Name:QCPLLOT	Command:IF THEN
--<<Procedure>>-----	
\$ Initialize	
<<ADDINT / ERROR	>>

IF ? ERROR = 0	
	\Y/-----+ \N/-----
[]	

After entering some more 'design comments', the [WHILE] function key is pressed to specify that plots must be produced until the user has obtained all the plots that are required. FLOW-EDIT displays the start of the while block:

```

Name:QCPLLOT          Command:WHILE
-----<<Procedure>>-----
$ Initialize
  <<ADDINT / ERROR          >>
-----
IF ? ERROR = 0
-----\Y/-----+ \N/-----
$ Present the user with the system identification |
$ using the general purpose screen display utility |
-----+-----
WHILE [] |

```

The WHILE condition is entered, as 'CHOICE <> 3', where a choice of 3 will indicate no more plotting required. FLOW-EDIT continues building the 'while block'.

```

Name:QCPLLOT          Command:INSERT
-----<<Procedure>>-----
$ Initialize
  <<ADDINT / ERROR          >>
-----
IF ? ERROR = 0
-----\Y/-----+~\N/-----
$ Present the user with the system identification |
$ using the general purpose screen display utility |
-----+-----
  WHILE CHOICE <> 3 |
/-----+-----
/|[] |

```

As each line is inserted in the while block, the symbol |/| will appear to remind the user that he is still within the while block. The vertical lines of the other blocks are also extended as each line is entered. After some more design details have been entered, the final procedure design appears as:

```

Name:QCPLLOT          Command:INSERT

--<<Procedure>>-----
$ Initialize
  <<A00INT / ERROR          >>
-----
IF ? ERROR = 0
-----\Y/-----+ \N/-----
$ Present the user with the system identification
$ using the general purpose screen display utility
-----+-----
  WHILE CHOICE <> 3
  /+-----+-----
  /|$ Display a menu giving the options
  /|$     1 - Plot a single control chart
  /|$     2 - Plot all controls
  /|$     3 - Exit
  /|$ Plot control charts as required
  /|   <<A05-QCP-PLOT          >>
  -----+-----
$ Finalize
  <<A10FIN          >>
[ ]

```

Note that two [END] function keys were required to be entered following the comment \$ Plot control charts... The first [END] terminated the WHILE block, and the second terminated the IF-THEN blocks. This completes the design phase.

The design, depicted above is stored in the system data base. This design 'document' will be used later for a number of purposes.

- the post-processors will make use the interface definition of QCPLLOT as well as the 'dummy' specifications of modules such as AOODINT to generate 'stubs' for these modules. These stubs will then be incorporated into the system under construction to facilitate top down testing. 'Stubs' are described in more detail in 'Intermediate System Building' below.

- the 'design' skeleton will be fleshed out to produce the code for the procedure. Thus the source code is a refinement of the design document, and not a separate document. This has the advantage that the source code contains design information such as the functional specification and the designer's original comments on the skeleton procedure (if they are not removed by the coder).

- Once the interfaces of procedures have been defined (a design activity), these interfaces are available to all other modules which will invoke them. This facilitates interface checking.

Since the system is designed, coded and tested top-down, modules will generally be invoked before they are defined. The first reference to a procedure e.g. AOODINT forces the creation of a 'dummy' specification. When the time comes to design the actual module, these 'dummy' interfaces will be replaced by the real interface definitions, and the parameter names will possibly be redefined. The system data base keeps track of

the modules referenced by other modules, as well as the status of each module. The status of a module may be:

- dummy: The module has been referenced by some other module, but has not yet been designed. Stubs will be generated using the dummy interface if required.

- designed: Design information, including interface definitions, has been given, but coding is still required. The post-processors will generate stubs using the proper interface definition.

- coded: Some code (not necessarily all) has been entered for the module. The post-processors will generate executable code for the coding given.

- interface error: If the interface definition of some module is changed, then all modules referencing that module are flagged as 'error in interface'. This applies both to 'designed' and to 'coded' modules. Post processors will report this error if any attempt is made to generate code for these modules.

Programming

So far the FLOW environment has been shown in use during the system design phase of the QCPLLOT module of the QC system. The system design document has been in the form of a GNSD with some supporting 'comments'. The FLOW environment allows this system design document to be extended to become the programme source code. The programmers task is simplified in that the interface definition and programme functional description already exist. The designer may also have provided a skeleton procedure. In either case, the programmer is merely required to produce the detailed procedural statements, and to add the local, and perhaps some (of the omitted) global data declarations.

If a skeleton procedure is provided as part of the design, as in the case of the QCPLLOT module, this skeleton will contain a number of comments. The programmer may choose to retain these comments in the final version of the procedure, thus providing correspondence between the original design document and the source code. Of course, in the worst case, the programmer may have to discard the skeleton procedure provided in the design document and start from scratch.

Continuing with the development of QCPLLOT, the editor, FLOW-EDIT is again used, this time to enter the programme source code. A hardcopy listing of the design document for QCPLLOT, produced by FLOW-CHART, would usually have been produced before

programming commences, although the original design document may be re-displayed by FLOW-EDIT. Supposing that the command **display procedure** is given to display the design of the procedure definition of the module. After positioning the cursor at the desired place, the command **make space** is entered. This command causes the current block (i.e. the block in which the cursor has been placed) to be split at the point of the cursor, and extended towards the bottom of the screen, allowing a number of lines free for input.

```

Name:QCPLOT          Command:INSERT

<<ADDINT / ERROR          >>
-----\Y/-----+---\N/---
IF ? ERROR = 0
$ Present the user with the system identification
$ using the general purpose screen display utility
-----+-----+
WHILE CHOICE <> 3
/|-----+-----+
/|$ Display a menu giving the options
/|$      1 - Plot a single control chart
/|$      2 - Plot all controls
/|$      3 - Exit
/|[]
|
|$ Plot control charts as required

```

After the FLOW environment had been in use for some time, a library of utilities would have accumulated in the software production data base which could be incorporated into any new programmes. Supposing that a menu display utility, X07MNU, is to be used in the current programme to present the QC user with a list of options allowed, and to accept, validate and return the selected option. It can be presumed that the programmer would not know off hand what the parameters of the call to X07MNU would be. In order to use the menu utility, the [CALL] key would be entered followed by the name X07MNU with the ? symbol entered at the end to indicate that the parameters are unknown. This directs FLOW-EDIT to display X07MNU's interface definition for reference purposes. The cursor is then repositioned by FLOW-EDIT for the parameters to be entered.

```

Name:QCPLOT          Command:CALL
<<ADDINT / ERROR          >>
-----\Y/-----\N/-
IF ? ERROR = 0
$ Present the user with the system identification |
$ using the general purpose screen display utility |
-----+-----+
WHILE CHOICE <> 3 |
/!$ Display a menu giving the options |
/!$      1 - Plot a single control chart |
/!$      2 - Plot all controls |
/!$      3 - Exit |
/! <<Z07MNU?[] >> |
-- Z07MNU I/F -----
FILENAME { A6      $ Menu screen file name
OUCH { I2         $ Output channel for menu screen display
INCH { I2         $ Input channel for users response
CHOICE } I2       $ User's choice returned
MAXCHOICE { I2    $ Highest choice available on the menu
-----
$ Plot control charts as required

```

After the parameters for Z07MNU have been given, the interface information is cleared from the screen. Now the correct action must be taken according to the menu option chosen. The [IF-THEN-ELSE] key is pressed, and a new flowchart fragment is created within the current block.

```

Name:QC PLOT      Command:IF THEN ELSE      OPERATOR EXPECTED
      <<A00INT / ERROR                        >>
-----\Y/-----\N/-----
IF ? ERROR = 0
-----\Y/-----\N/-----
$ Present the user with the system identification |
$ using the general purpose screen display utility |
-----\Y/-----\N/-----
WHILE CHOICE <> 3 |
/|-----\Y/-----\N/-----
/|$ Display a menu giving the options |
/|$      1 - Plot a single control chart |
/|$      2 - Plot all controls |
/|$      3 - Exit |
/| <<Z07MNU/'QCPH01',INCH,OUCH,CHOICE,2 >> |
/|-----\Y/-----\N/-----
/| IF ? CHOICE 1 |

```

Obviously, the condition should have read 'CHOICE = 1'. The error message 'OPERATOR EXPECTED' is given, and the cursor is repositioned by FLOW-EDIT to the place at which the syntax error was detected. The correction is made by entering '= 1'.

The syntax checker detects only one error at a time (i.e. the first one). The position of the error is shown by the cursor, and a simple message displayed on the status line. The combination of a simple error message and a pointer to the error are considered preferable to verbose (and often misleading) diagnostics.

Corrections to incorrect lines are made by keying over the incorrect portion of the line. The correction is simplified as a result of the cursor being positioned automatically at the place of the error by FLOW-EDIT. Backward and forward positioning of the cursor within the incorrect line is permitted.

Suppose that we wish to enter only the major control logic of QCPLLOT at this stage of the programming in order to eliminate logic errors in the flow of control (which indeed there are as will be seen later). We complete the Y block, ignoring the plotting procedure, (except to insert a comment to remind ourselves of our intention to fill in the detail later) and press the [END] key. This terminates the Y condition, and places the cursor in the N block.

```

Name:QCPL0T          Command:END
<<ADDINT / ERROR          >>
-----\Y/-----\N/-----
IF ? ERROR = 0
|$ Present the user with the system identification
|$ using the general purpose screen display utility
-----+-----
WHILE CHOICE <> 3
/|+-----+
/|$ Display a menu giving the options
/|$   1 - Plot a single control chart
/|$   2 - Plot all controls
/|$   3 - Exit
/| <<Z07-MENU/'QCPM01',6,5,CHOICE,2 >>
/|+-----+
/| IF ? CHOICE = 1
/|+-----\Y/-----\N/-----+
/|$ Get plot number      |[]
/|+-----+
|
|$ Plot control charts as required

```

Now the comment 'loop through all plots' is given for the N condition, followed by the **exit** command.

Intermediate System Building.

In order to build an intermediate system, i.e. a system consisting of some top level modules with stubs for the incomplete lower levels, the following tasks must be performed by the user.

- all modules referencing another whose interface definition has been changed, must be modified accordingly.
- all modules which have been 'programmed' or modified must be re-generated and compiled.

- all undefined procedure references must be resolved.

FLOW provides a list of modules for which 'interface errors' occur, as well as for modules requiring re-generation and compilation.

In order to resolve undefined references, the FLOW environment allows stubs to be generated for the referenced modules. Stubs are generated for two classes of incomplete module. Firstly, a stub may be generated from the interface definition given in the system design; secondly, it may be generated from a 'dummy' interface which is created at the time of the initial procedure invocation. A stub performs the following tasks when executed:

- upon entry, it displays a message giving the name of the procedure and the values of all the 'input' and 'in-out' parameters.
- the person performing the test is requested to enter 'returned' values for the 'output' parameters.

Using the FLOW environment in the current example, the stub for A00INIT would be generated, and the source code of QCPL0T and the stub compiled. The compiled modules are then linked with the library to form an executable programme.

Debugging and Testing

The next phase in the development cycle is the debugging and testing of the programme in its current (intermediate) state of development. The FLOW environment provides the facilities during the testing phase, to generate imbedded 'trace' statements within the source code. No trace package is required, as each post-processor (the code generators of the FLOW environment) automatically produces the trace 'display' statements as part of the generated source code. The level of tracing required may be set by entering trace commands (post-processor directives) which are embedded within the FLOW-DL source. Tracing may be set to one of three 'granularities':

- 'course': displays the name of the procedure and values of the input parameters upon entry, and the output parameters upon return.

- 'medium': displays as for 'course' as well as displaying the entry into and exit from each block of the flowchart. Messages are given for each type of block e.g. TRUE and END-TRUE on selection, or BEGIN-WHILE and END-WHILE on iteration. In addition, at each selection or iteration, the actual clause (expression) is displayed.

- 'fine': displays as for 'medium', as well as displaying each statement in the order of execution, and the value to be stored in the case of an assignment statement.

Continuing with the QC example, the current version of our system may be tested. Having built a QC system at this intermediate level of development, and having placed a debugging directive in QCPL0T to produce level 2 tracing, we execute the QC system. The following is a portion of the 'trace' produced. (Responses entered from the keyboard are shown underlined).

```

*** PROCEDURE QCPL0T ENTERED ***
*** STUB ADDINT ENTERED ***
ERROR ? 0
IF ERROR = 0
  TRUE
  WHILE CHOICE <> 3
    BEGIN-WHILE
*** PROCEDURE Z07MNU ENTERED ***
FILENAME = QCPM01
OUCH = 6
INCH = 5
MAXCHOICE = 2
....
<the menu is displayed and option 1 chosen>
...
*** PROCEDURE Z07MNU RETURNED ***
CHOICE = 1
  IF CHOICE = 1
    TRUE
  END-TRUE
  END-WHILE
  BEGIN-WHILE
*** PROCEDURE Z07MNU ENTERED ***
...
<option 2 is chosen>
...
*** PROCEDURE Z07MNU RETURNED ***

```

```

CHOICE = 2
  IF CHOICE = 1
    FALSE
    FOR NPLOT = 1,MAXPLOT
      BEGIN-LOOP
      END-LOOP
    . . . .
  END-FALSE
END-WHILE
BEGIN-WHILE
*** PROCEDURE Z07MNU ENTERED ***
. . .
<option 3 is chosen>
. . .
*** PROCEDURE Z07MNU RETURNED ***
CHOICE = 3
  IF CHOICE = 1
    FALSE
    FOR NPLOT = 1,MAXPLOT
      BEGIN-LOOP
      END-LOOP
    . . . .
  END-FALSE
END-WHILE
END-TRUE
*** PROCEDURE QCPLLOT RETURNED ***
ERROR = 0
. . .

```

After trying options 1 and 2, we realize that when we enter option 3, we perform the operations of option 2 before terminating on the condition 'WHILE OPTION <> 3'. We obviously need to by-pass this condition. After calling FLOW-EDIT to enter this modification, and after positioning the cursor on the statement 'WHILE CHOICE <> 3', the command **display zooming in** is given. The block containing the WHILE is now drawn to the full screen width.

```

Name:QCPLLOT          Command:ZOOM

WHILE CHOICE <> 3
/|+-----+
/|$ Display a menu giving the options
/|$     1 - Plot a single control chart
/|$     2 - Plot all controls
/|$     3 - Exit
/|$     <<Z07MNU/'QCPM01',INCH,OUCH,CHOICE,2      >>
/|+-----+
/| IF ? CHOICE = 1
/|+-----+-----+-----+-----+-----+-----+-----+-----+-----+
/|$ Get plot i.d. and plot      |N|PLOT=1,MAXPLOTS
/|+-----+-----+-----+-----+-----+-----+-----+-----+-----+
/|                               |/|$ Plot N|PLOT
/|                               +-----+-----+-----+-----+-----+
/|                               +-----+-----+-----+-----+-----+
/|$ Plot control charts as required
/|+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

The modification requires a new selection block to cater for the case of CHOICE=3. To achieve this, we position the cursor on the block 'IF CHOICE = 1' and enter the command **clip 1** to clip the block and to file it away as clipping number 1. Clipped blocks are filed for possible later re-use during an edit session. The space on the screen previously occupied by the clipped block is cleared, and the new block is entered in its place. The new block is entered with a unary selection IF CHOICE <= 2. The block is drawn and the cursor positioned within the Y block. The content of the Y selection of this new condition is to be the clipping file 1. This is inserted using the **glue 1** command.

```

Name:QCPLOT          Command:INSERT

WHILE CHOICE <> 3
/+-
/|$ Display a menu giving the options
/|$     1 - Plot a single control chart
/|$     2 - Plot all controls
/|$     3 - Exit
/|     <<Z07-MENU/'QCPM01',6,5,CHOICE,2           >>
/+-
/| IF ? CHOICE <= 2
/+------\Y/-----+ \N/-
/|[]                                     |
/|                                     |
/|                                     |
/|                                     |
/+------
/|$ Plot control charts as required
-----

```

The glued block is drawn within the boundaries of the current block.

```

Name:QCPLLOT          Command:GLUE

WHILE CHOICE <> 3
/|-----+-----|
/|$ Display a menu giving the options
/|$      1 - Plot a single control chart
/|$      2 - Plot all controls
/|$      <<Z07-MENU/'QCPM01',6,5,CHOICE,2          >>
/|-----+-----|
/| IF ? CHOICE <= 2
/|-----+-----|
/| IF ? CHOICE = 1
/|-----+-----|
/|$ Get plot number          |NPLOT=1,MAXPLOTS
/|-----+-----|
/|                          |/|$ Plot NPLOT
/|-----+-----|
/|$ Plot control charts as required
/|-----+-----|

```

System Documentation.

As the software development methodology specifies, documentation is produced within the refinement 'loop'. The part of the system documentation which can be generated directly from the FLOW-DL source (held in the system data base.) is the hierarchy listing ('who calls who') and the detailed flow diagrams of each module. The detailed flow diagram comprises the design specification as it currently exists (it may have been changed during programming), and the detailed flowchart of the procedure.

To document what we have developed so far of QCPLLOT, we enter the command:

```
* * * FLOW Environment * * *
```

```
Enter Command:CHART
```

```
For Help type: HELP
```

```
Current name is:QCPLLOT
```

Since the current name is QCPLLOT, the simple command **CHART** is all that need be given. Having obtained a printed copy of our work so far, we are able to proceed with the next refinement...

...until the stage is reached when we are able to give our final command:

* * * FLOW Environment * * *

Enter Command: **BYE**

For Help type: **HELP**

Current name is: **Z99PLT**

Bibliography

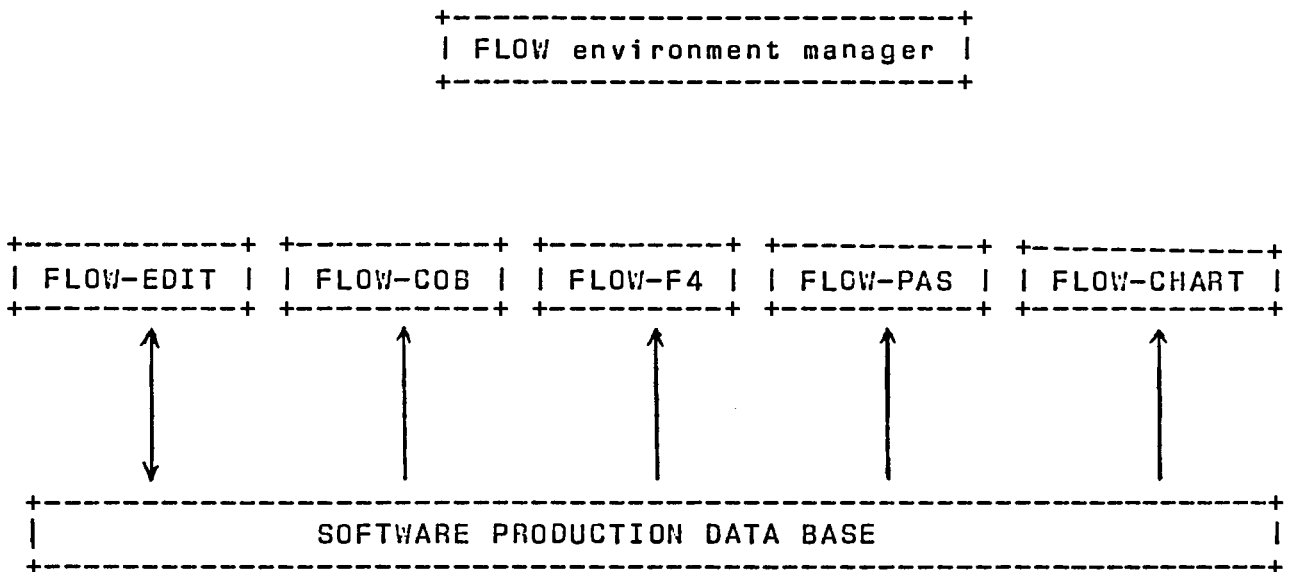
[Jens79] Jensen R W, Tonies C C. **Software Engineering**. Prentice Hall, Engelwood Cliffs, New Jersey, 1979.

[Your75] Yourdon E. **Techniques for Program Structure and Design**. Prentice Hall, Engelwood Cliffs New Jersey, 1975.

5. The FLOW environment.

5.1 FLOW environment architecture

The FLOW environment is constructed using a number of individual components. Before discussing these components in detail, the general architecture of the FLOW environment architecture should be considered. The environment encompasses a number of programmes which access a common software production data base as depicted in the following diagram.



(a) The environment manager.

The environment manager acts as the co-ordinator of the FLOW environment component programmes, and provides an interface between the environment and the operating system of the computer on which the environment exists. One of the functions of the environment manager is to allow the FLOW user to access 'system utilities' such as compilers and link-editors as part of an integrated environment.

(b) FLOW-EDIT: The FLOW-DL editor.

FLOW-EDIT is the syntax directed editor for FLOW-DL (the environment's source language). It allows Generalized Nassi Shneiderman Diagrams (GNSDs) to be specified, and operates directly on the software development data base, allowing GNSDs or blocks within GNSDs to be created, modified or deleted.

(c) FLOW-COB, FLOW-PAS and FLOW-F4

FLOW-COB, FLOW-PAS and FLOW-F4 are post-processors for the production of target code for the COBOL, Pascal and FORTRAN IV programming languages. Target code is generated by accessing the FLOW-DL source held on the software development data base and performing language-specific transformations. Since FLOW-EDIT performs syntax checking at the stage when the FLOW-DL is input, all GNSDs entered into the data base will be syntactically correct and there is therefore no further need for syntax checking by the post-processors.

(d) FLOW-CHART

FLOW-CHART is a post-processor for documentation generation. The post-processor operates in a similar manner to the language post-processors except that GNSDs are output. These GNSDs may then be printed on a line-printer. The GNSDs produced may be used for two purposes:

- they may be used to aid in the understanding of the system and its associated source code, i.e. provide system documentation in the traditional sense of 'blow by blow' flowcharts.

- they may be used by the designers and programmers as working documents i.e. act as a replacement for the design documents as well as for the FORTRAN, Pascal or COBOL language source code listings. When used for this purpose, the GNSDs may be considered as FLOW-DL source code listings, and FLOW-DL then becomes a true graphical programming language. The target code generated from the FLOW-DL source by the post-processor then becomes transparent to the FLOW-DL programmer in the same way that p-code, assembler-code, machine-code etc. are transparent to the Pascal, FORTRAN or COBOL programmer.

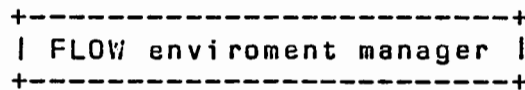
(e) The software development data base.

The FLOW-DL source and control information for the software system being built is kept in the software development data base.

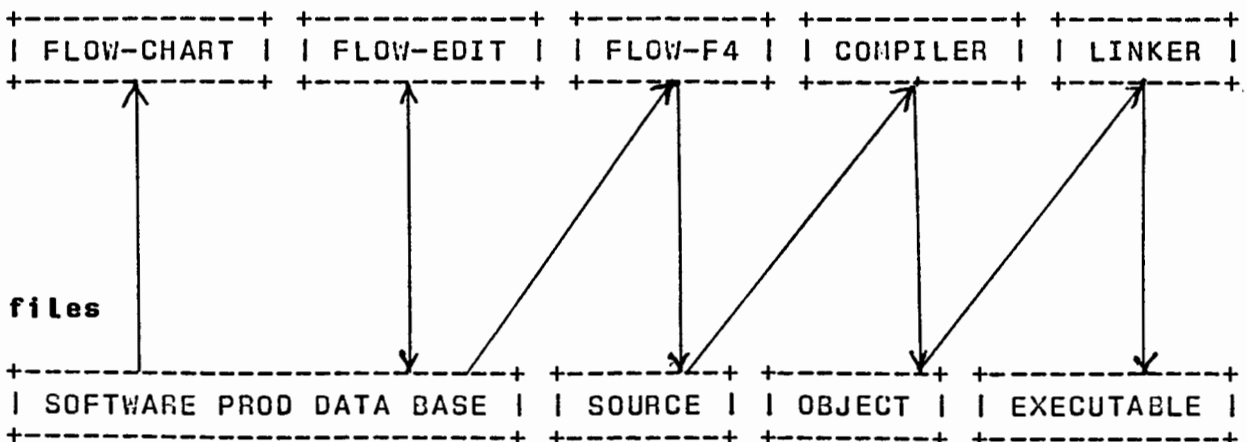
Tailored environments.

It is probable that all of the components of the FLOW environment will not be required by every user or group of users. For example, software systems are often developed using only one programming language. In this case, the environment could be tailored by including only those tools which are applicable. A typical environment configured for the development of FORTRAN systems which incorporates both FLOW and 'system' utilities and associated files, could be configured as follows:

components



system utilities



5.2 The environment manager.

The environment manager acts as the co-ordinator of the FLOW environment components in any configuration of the environment. Since the implementation of an environment manager is highly dependent on the operating system of the machine on which it is implemented, and would furthermore be tailored for each site and possibly even for each software project on which it was used, no details of any one specific environment manager are presented.

Whereas the components of an environment have a well defined task which is not likely to be extended, the environment manager is expected to 'grow' as each new tools is added to the environment.

The component tools which are used to construct a FLOW environment may be controlled by the environment manager in a number of ways

- a specialised environment manager may be used to controls a specific environment. Such an environment manager would ideally be used to control both FLOW components and system utilities, if this is possible.

- an existing environment manager may be used and the FLOW components incorporated into an existing software production environment and used together with the other software production tools.

- a 'manually controlled' environment may be established where the FLOW component programmes may be used independently. In this case, the programmes could be controlled by some interactive job control language with the necessary programme intercommunication being established manually.

Both a specialised and manually controlled environment were used during the design and testing of the FLOW components. A specialised environment manager for the production of FORTRAN systems is currently in use. This particular environment manager controls FLOW-F4, FLOW-CHART, FLOW-EDIT, a FORTRAN precompiler, a FORTRAN IV compiler, a linker and a text editor.

In what follows, the specialised environment will be discussed as this is the most likely implementation possibility. In general, an environment manager's responsibilities are:

- to provide an interface between the operating system and the remainder of the FLOW programmes (FLOW-EDIT, and the post-processors).

- to incorporate 'system' utilities such as compilers, link-editors and file handlers in such a way that they appear as an integral part of the environment, and thus provide a comprehensive environment for software production.

As an example of a facility that can be provided in an integrated environment, the environment manager may be made to 'remember' the name of the last entity of the software production data base accessed. Commands such as **EDIT AOOINIT** would invoke FLOW-EDIT to access the subroutine named AOOINIT, and subsequent commands such as **F4** and **COMPILE** would invoke FLOW-F4, and the system's FORTRAN compiler without having to specify the name AOOINIT again.

In order to be able to interface with the operating system and incorporate system utilities, the environment manager must be written in a language which allows 'operating-system calls'. This requirement would usually require using an assembler language, a systems programming language, or an enhanced 'standard' language. One version of a FLOW environment manager currently in use was written in an extended FORTRAN IV. In this version, access to the operating-system facilities is provided by means of calls to the system library.

It is important that the environment manager should make available to the user a comprehensive array of tools and facilities. Some of the (system dependent) functions which are useful in creating such an environment manager are

- the ability to suspend execution of the environment manager, initiate a son process, and to resume execution of the environment manager on termination of the son process.

- the ability to set up commands programmatically for the system utilities (compilers, link editors etc.), and to be able to pass these commands to the utilities. For example, a compiler may allow the name of the source and object files and the device or file for error messages to be specified. In this case, the environment manager could provide standard or generic names, and thereby save some data entry on the part of the user.

- the ability to create, delete and rename files with names which may be programmatically assigned. This facility is actually a pre-requisite in the current implementation due to the fact that components of the FLOW environment are written in Pascal which lacks of this facility. The environment manager is thus required to assign names externally to the Pascal components.

The FLOW component programmes which have been written in Pascal perform input/output using instructions such as **READLN(filename,-LINE)**, where **filename** is defined as **filename: FILE OF CHAR** and where **filename** is predefined. The way in which an environment manager communicates with these component programmes is via the content of these files. The environment managers task is to associate the predefined file-names with the appropriate files or devices as required.

A number of these predefined file-names exist, the most important being

- The interprocess communication file (IPC)

This file contains the name of the procedure to be edited, documented or to have source-code generated. The procedure name passed in this file is used to locate the procedure entry in the entry point table of the software production data base. The entry point table holds the root nodes of the FLOW-DL sub-trees for the procedure, and the editor and post-processors operate on these sub-trees.

- The terminal input file (INPUT)

This file must be linked to the current terminal keyboard when FLOW-EDIT is started i.e. the Pascal statement **READ (INPUT,character)** must return a character entered at the keyboard. This file is not used by any of the post-processors.

- The output file (OUTPUT)

This file must be linked to the current terminal screen when the FLOW-EDIT is started. The post-processors also make use of this file for post-processor output i.e. GNSDs are written to OUTPUT by FLOW-CHART; FORTRAN, Pascal and COBOL source code is written by FLOW-F4, FLOW-PAS and FLOW-COB respectively. When a post-processor terminates, this file may be renamed by the environment manager. A suggested procedure to be adopted for file re-naming is to create a new file name by adding a two letter 'extension' to the procedure name if possible. e.g. the FLOW-CHART output file for procedure 'QCPLLOT' would be named 'QCPLLOT.DC'.

- The error log file (ERRLOG)

This file may be linked to the terminal screen during post-processing. Messages for all errors detected during post-processing are written to ERRLOG, and would then appear on the user's screen. A second possibility is to link this file to an audit trail file.

The limiting of the system dependent functions of the FLOW environment to a single component has a number of advantages, the most important being that it allows the remainder of the FLOW components to be implemented in 'standard' Pascal. This improves the portability of the FLOW environment at the relatively small cost of a 'non portable' environment manager.

6. The Software Production Data Base.

6.1 Design Philosophy

The FLOW environment is implemented using a number of independent programmes each accessing a common 'software production data base'. The software production data base, contains all of the FLOW-DL source code as well as the control information pertaining to the system or systems being produced. The data base approach offers numerous advantages to the FLOW environment as a development tool.

Large systems may be developed.

The FLOW environment programmes all operate directly on the data base, and access a single line (record) of source code at a time. Unlike many editors, FLOW-EDIT does not read 'pages' of the programme source into memory, but rather performs all edit operations on the records 'in place'. Arbitrarily large programmes, limited only by disc storage requirements, may be thus developed. (This point is particularly important to the implementation of the environment on 16 bit mini-computers with each programme restricted to a maximum size 64K bytes).

Global (system) functions are facilitated.

- Since the data base contains the source code of the entire system under development, global functions are facilitated. Examples of these global functions are

- the interface checking facility, whereby the procedure invocation in one programme module is checked against the actual interface specification in the called module.

- information which is available during programme editing such as the descriptions of the call parameters defined in the interface specification, and the functional description of the called module.

Comprehensive control information is provided.

The system data base allows comprehensive control information to be recorded during system development. Module interdependencies are recorded as well as the status of each module.

Programmer teams are facilitated.

The system database facilitates team development of a system by allowing the possibility of simultaneous multi-user access. The comments made regarding interface checking are particularly applicable to team development where modules are produced by individual team members.

Components of the environment are implemented independently.

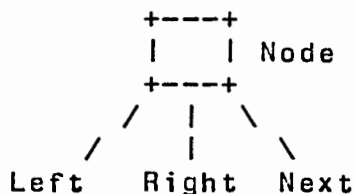
The data base allows the environments constituent programmes (the environment manager, FLOW-EDIT, FLOW-CHART, FLOW-COB, FLOW-PAS and FLOW-F4) to be bound (link edited) independently of one another, with the data base serving as the medium of communication. This again has implications for implementation on mini-computers, with the need for overlaying being avoided.

6.2 GNSDs as tree structures.

The major portion of the software production data base consists of the FLOW-DL source code lines. Structured programmes (including Generalised Nassi Shneiderman Diagrams (GNSDs)), are easily and conveniently represented and stored as a tree data structures. In the current implementation, a ternary tree representation is chosen for storing GNSDs, which caters for the five major types of GNSD constructs. The diagrams which follow show these basic GNSD constructs and their corresponding tree structures.

CHAPTER 6. **The Software Production Data Base**

Each node in the tree is shown as a box, corresponding to a block in the GNSD.

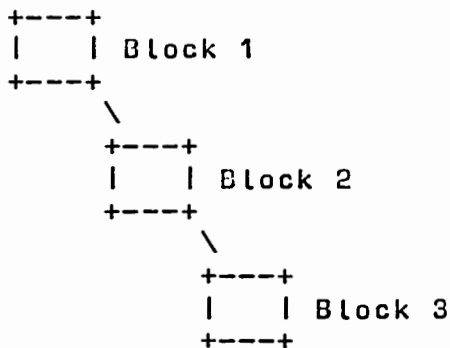
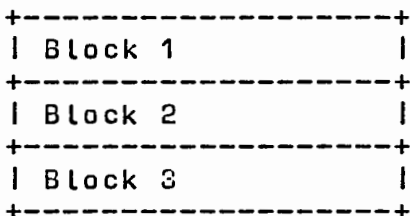


Each node may have one, two or three branches which will be referred to as the Next, Left and Right branches. (The names of the branches are taken from the binary selection GNSD application (see below) where all three branches are used).

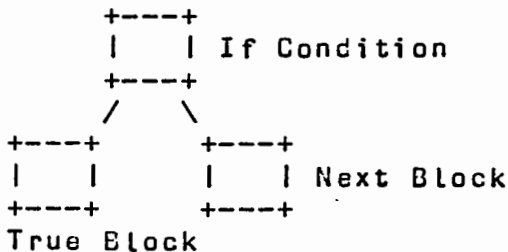
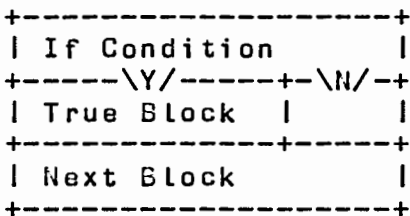
GNSD Construct

Tree Structure

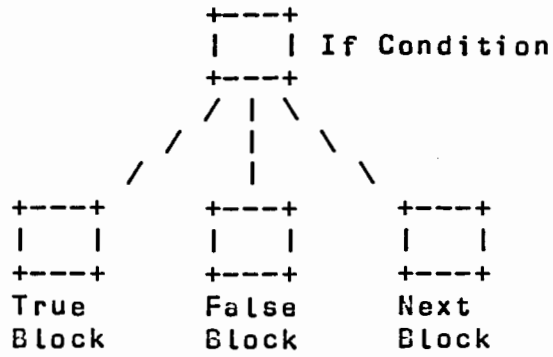
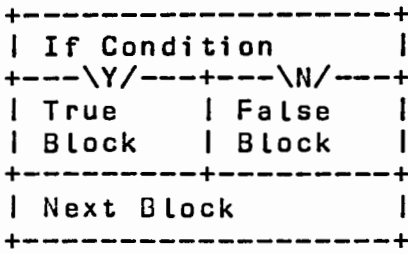
Sequence



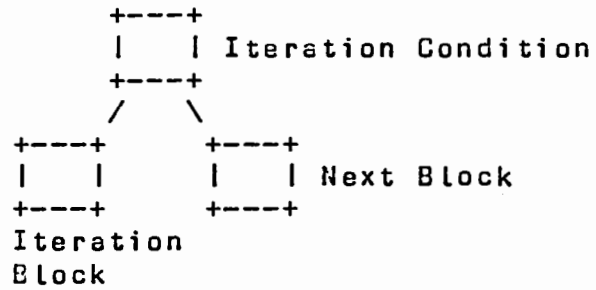
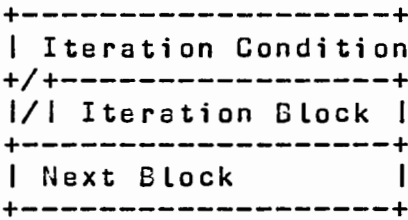
Unary Selection



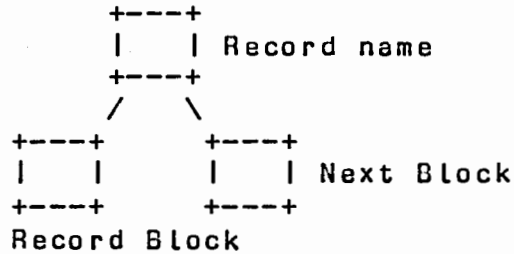
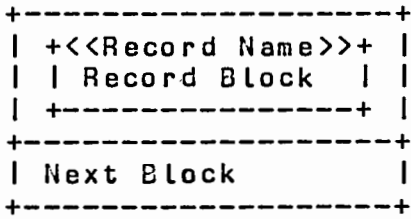
Binary Selection



Iteration



Record

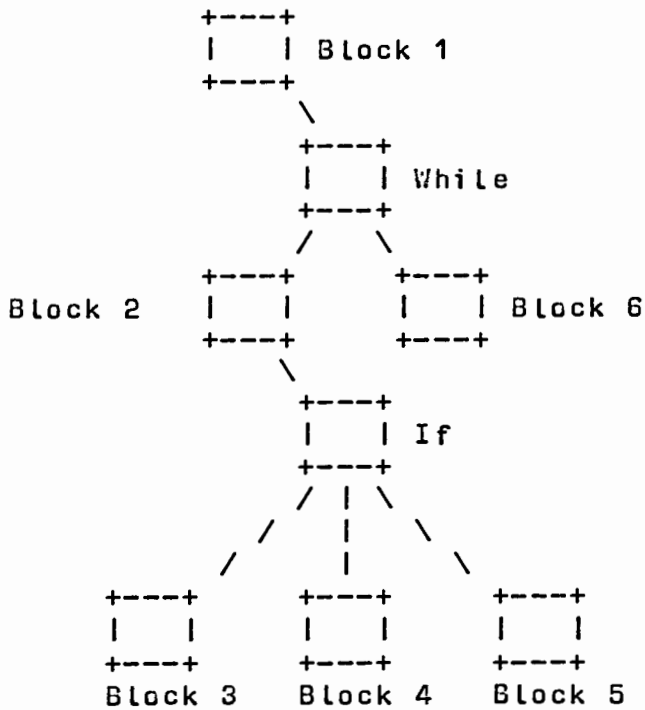


As an example of this representation, consider the following GNSD.

```

+-----+
| Block 1 |
+-----+
| while |
+ / +-----+
| / | Block 2 |
| / +-----+
| / | if |
| / +----- \ Y /-----+ \ N /-----+
| / | Block 3 | | Block 4 |
| / +-----+
| / | Block 5 |
+-----+
| Block 6 |
+-----+
    
```

This would be stored as a tree with the following structure:



6.3 Data base structure and implementation.

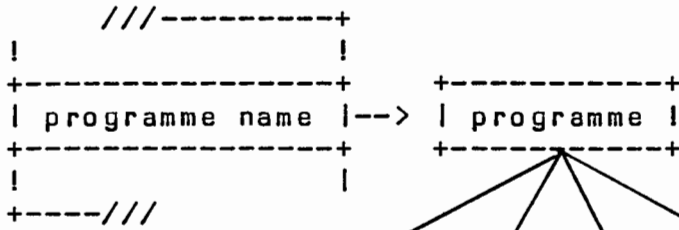
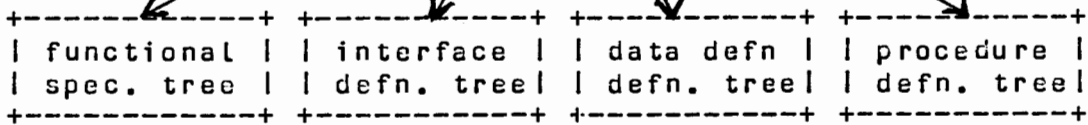
The data base contains the sources of a number of modules, and consists of two main parts, namely the FLOW-DL source code (GNSD) trees, and the entry-point table which locates the root nodes of each tree.

FLOW-DL source code (GNSD) trees.

Within the FLOW environment, programmes are considered as consisting of (up to) four GNSD trees emanating from the programme root node. The four GNSD trees are

- the functional specification tree
- the interface definition tree
- the data definition tree
- the procedure definition tree

These GNSD trees are accessed by the FLOW environment programmes independently of one another. Access to these trees is via the entry-point table (described later).

entry point table**GNSD trees**

The types of GNSD construct (node types) allowed within each of the four GNSD trees differs for each tree.

*** Functional specification tree**

Only sequence and record nodes are allowed within functional specification trees. The record construct is used to produce paragraph headings within the specification. Nested records produce an indentation of paragraphs. The skeleton functional specification which is generated when a new module is created, consists of three empty records labelled 'Functional Description', 'Input Description' and 'Output Description' respectively. The contents of these 'paragraph records' may be supplied and new paragraphs may also be defined.

*** Interface definition tree.**

The interface definition trees are restricted to sequence nodes which provide the lists of input and output parameters.

* Data definition tree.

As described in chapter 3, GNSDs may be used for representing data structures. The data definition trees consist of nodes for the GNSD constructs of sequences, records, binary and n-ary (but not unary) selections, and iterations. Sequences are used for elementary data items, selections are used to represent variant records, and iterations (controlled loops) are used to represent arrays.

* Procedure definition tree.

These trees may contain all of the GNSD constructs with the exception of records.

The datum to be associated with each node was decided upon after careful design, with the 'trade-off' between storage efficiency, processing efficiency, and ease of implementation being considered. Experimentation with a number of alternatives, and the development of a prototype system, provided valuable insight into the problems and benefits expected from various design alternatives.

One design decision which arose was determining the optimal structure for storing the content of the GNSD blocks. In the prototype implementation of the data base (which, incidentally, was held entirely in memory) the programme tree was implemented with a node for each GNSD construct, and the contents of the GNSD BLOCK was stored as a chain of lines of text. This appeared the obvious structure to adopt since blocks could contain zero or more lines of text. The node datum in the prototype therefore

Conversely, if a condition block were inserted somewhere after some line of text in a block, (Block 1 say), a new GNSD of a similar form to that given above will result. A node representing Block-3 would have to be created, and pointers to the text chain (previously the contents of Block 1), would have to be suitably adjusted depending on where the text was split for the creation of the condition block.

Thus the insert and delete operations are not obtained merely by grafting or pruning the GNSD tree as might be intuitively expected.

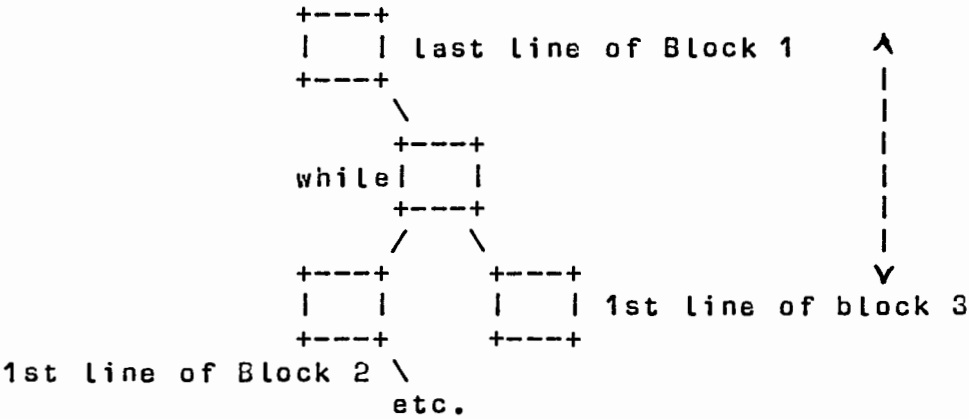
The current implementation is a simplification of the earlier approach. Text chains are eliminated, and instead, each line of text is implemented as a separate node. This is possible because, by definition, a number of lines in a block is actually a 'sequence' of blocks each containing one line.

+-----+ line 1 line 2 line 3 +-----+	<=>	+-----+ line 1 +-----+ line 2 +-----+ line 3 +-----+
--	-----	--

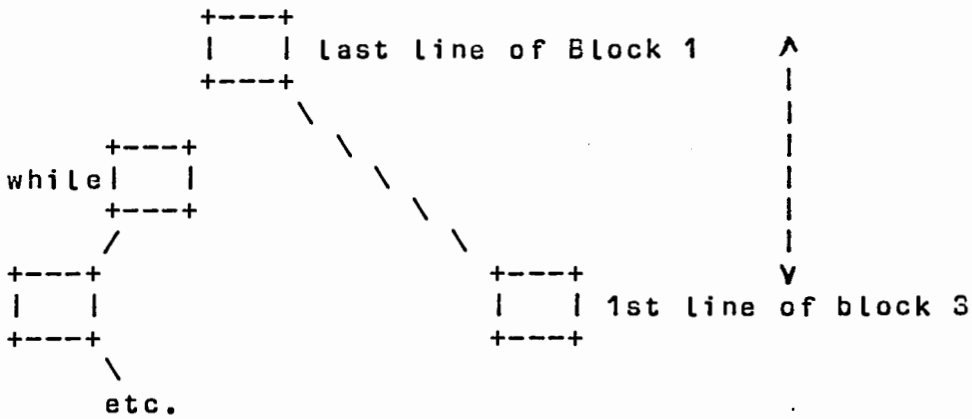
Thus, in the current implementation, the datum associated with the node is chosen to include the actual line of text instead of a pointer to a text chain. This simplifies the insertion and deletion operations and greatly reduces the complexity of

the entire system. For example, the deletion of the while-block in the previous example is achieved by unlinking the deleted block from the 'next block' chain, and re-linking.

before deletion



after deletion



Similarly, the problem of splitting text chains does not arise with this implementation when a new block is inserted, since the inserted block is simply linked between the two nodes.

The FLOW-DL tree nodes are held on the data base as records that are directly addressable by node number. The node records

contain the node type, tree pointers and FLOW-DL statements (text).

The entry-point table

The entry-point table portion of the data base provides access to the FLOW-DL source (GNSD trees). Entry point records are created within the entry-point table for each entity (module, global data block etc.) named at the time the entity is first created. The name of each entity is stored as a unique six character name. (The six character restriction is imposed by the FORTRAN subroutine naming rules).

The entry-point table also holds the entity type and status. There are currently four types of entities defined namely main procedures, sub procedures, copy procedures and global data blocks, each of which is associated with a different set of trees.

- Main procedures

Main procedures are constructed from three trees namely the functional specification , procedure and data definition trees. Interface definitions are not required in main procedures.

- Sub procedures

Sub procedures have four trees, namely functional specification, and interface, procedure and data definition trees.

- Copy procedures

Copy procedures have only a procedure definition tree.

- Data blocks

Data blocks have only a data definition tree.

The status held in the entry point table indicates the following:

- development status

This indicates whether the entity was generated automatically (when it was referenced by another module), or whether it has been manually edited. If the module has been manually edited, the status will also indicate whether or only comment lines have been entered on the procedure tree, implying that the only the design of the module has been entered, or whether some executable lines have been entered, implying that the module has been 'coded'. This status is used when stubs are generated.

- Interface error status

This indicates whether modifications have been made to the interface of some module which this module references which implies that the interfacing to modules referenced should be re-checked.

It should be noted that in the current implementation both the source data base and entry point table have been implemented in a crude (although efficient) manner using random files. The version of the Pascal compiler used in the current implementation provides random file access facilities, and the data base is implemented as a collection of random files. Where a data base management system or more sophisticated file handling is available, indexed files would be more appropriate particularly for the entry point table implementation.

7. FLOW-EDIT: A Syntax directed editor for FLOW-DL

7.1 Design Philosophy

When the idea of a software production environment using Nassi Shneiderman Diagrams was first conceived, it was realized that the editor would be the most difficult component to construct, not so much from the theoretical/technical aspects of the implementation, but rather from the user interface problems that would be encountered.

In order to gain insight into some of the human engineering factors concerned, as well as to experiment with some of the possible design alternatives, the basic requirements of the editor were specified, and prototyped. The prototype was then used on a number of projects over a two year period before it was re-written in its present form as FLOW-EDIT.

FLOW-EDIT was designed as an **interactive** GNSD Picture builder/data base editor. The interactive construction of GNSD pictures simultaneously with the editing of the corresponding GNSD tree on the data base is the prime requirement of FLOW-EDIT. The philosophy of FLOW-EDIT (as indeed of the entire FLOW environment) is to 'let the computer do the work for you.' Some of the principles adopted in the implementation of FLOW-EDIT were:

- FLOW-EDIT must be **easy to use**, and must operate in a simple and natural manner.

- FLOW-EDIT must be **helpful** and provide assistance wherever possible. Assistance may be in the form of dialogue entered between the user and the editor, or in the form of unsolicited feedback during editing.

- FLOW-EDIT must be **applicable to the editing of GNSDs**. This requirement implies that it should meet all of the editing requirements of the FLOW environment and therefore be equally concerned with the 'local' and 'global' requirements of the system being developed. The local requirements of the editor are that it must be an interactive GNSD picture builder; the global requirements are that it must serve as a software production data base editor.

- All source information on the data base must be maintained as **syntactically correct FLOW-DL source**. This requirement is imposed to allow post-processors to produce error free target code without the need for error checking.

- **Standard display terminals** (non graphic) must be able to be used as the 'editing work-station'.

A number of practical implementation constraints were imposed by both the hardware and software on which the editor was implemented. These constraints are dealt with in chapter 10 under 'portability' issues.

7.2 Simplicity and ease of use

A number of factors contribute to making FLOW-EDIT simple and easy to use. The diagrams of Nassi and Shneiderman, by virtue of their simplicity and clarity, is considered one of the major factors in this regard. Other contributory factors are the design of the FLOW-DL language for constructing GNSD pictures, the design of the FLOW-EDIT command language, and the consistency of FLOW-EDITs operation with respect to the user's conceptual model of the editing process taking place.

7.2.1 Command language

The FLOW-EDIT command language consists of only six basic command verbs; move, draw, clip, glue, make-space and exit. This simplicity is not accidental, but was achieved after continuously rationalising the command language as experience was gained in the practical application of FLOW-EDIT.

Command modes

It has been noted that the fewer the number of command 'modes', the simpler the user interface will be [Newm79]. Editors are generally implemented using a number of modes. For example, most text editors require an 'insert' and a 'command' mode, one of the reasons being that, text strings, which correspond to commands may be required to be inserted as part of the text. During 'insert' mode, all commands, with the exception of some

escape sequence, are ignored by these editors.

FLOW-EDIT has been implemented using only one mode, namely GNSD editing mode. The elimination of modes is made possible because of the well defined structure of the objects being edited (i.e. only GNSDs are being edited), and because the set of edit commands was chosen so that they do not intersect with the set of possible FLOW-DL statements (i.e. commands are clearly distinguishable from other input). The basic operations performed during GNSD editing are insertion, cursor movement, block display and block manipulation.

Insertion

The user interactively builds GNSDs on the screen by entering FLOW-DL statements or commands. FLOW-DL statements define the GNSD form and content, whilst edit commands dictate the editing operations to be performed on the GNSD. The terminal cursor is used to show the user where his current input will be inserted in the GNSD and also to identify the GNSD block which the editing operations will affect. For example, if a block of a GNSD currently displayed on the screen is to be deleted, the cursor would be positioned by the user to the beginning of that block, and the command given for the deletion to be performed.

As a consequence of eliminating modes, commands must be given 'in the middle' of the GNSD, which can result in part of the picture being overwritten. FLOW-EDIT automatically re-draws

any part of the picture which has been overwritten when the command is executed so that the GNSD picture remains intact.

A less obvious benefit of accepting either a command or a FLOW-DL 'picture building statement' at the current cursor position does is that it enables the user to focus his attention on only one region on the screen instead of having separate command and display areas.

Cursor movement

The cursor is positioned to various points within a GNSD displayed on the screen by the user entering commands to move it up, down, left or right. Not all points on the screen may be accessed by these commands, since part of the GNSD picture, such as the block boundaries, provide the 'graphics' information. Therefore in performing cursor movement, FLOW-EDIT determines the destination screen co-ordinate of a move command (computed as an offset from the current cursor position i.e. relative and not absolute co-ordinates), and if the destination is a valid 'edit point' on the screen, the cursor will be moved to that point. If it happens that the resultant co-ordinate is not a valid 'edit point', FLOW-EDIT will continue to search in the direction given by the command until such a point is found, or until no further possibilities exist. For example, if the command is given to **move up four lines**, and the line four lines above the current line is not a FLOW-DL statement, then the fifth, sixth,

seventh lines etc. above the fourth will be tested until a valid point is found. If no such point can be found, the cursor will remain where it was.

The symbol `[]` is used to show the valid 'edit points' on the following screen.

```

Name:QCPL0T          Command:Z00H
[]WHILE CHOICE <> 3
/+-----+
/|[]$ Display a menu giving the options
/|[]$      1 - Plot a single control chart
/|[]$      2 - Plot all controls
/|[]$      3 - Exit
/|      <<[]Z07-MENU/'QCPM01',6,5,CHOICE,2      >>
/+-----+
/|[]IF ? CHOICE = 1
/|      \Y/-----+-----\N/-----
/|[]$ Get plot i.d. and plot |[] FOR IPLOT=1,MAXPLOTS
/|[]-----+-----+
/|      |/|[]$ Plot NPL0T
/|      +[]-----+
/|      +[]-----+
/|[]$ Plot control charts |
[]-----+
[]-----+

```

Note that these 'edit points' are to the left of each line in a GNSD block, and that FLOW-EDIT will automatically place the cursor at this point if the user causes it to move to anywhere within the line.

Block display

As described in chapter 6, programmes are held as four separate trees on the software production data base. In order to edit one of these trees, it must first be displayed on the screen with the appropriate **display** command. FLOW-EDIT displays the GNSDs using the full width permitted by the screen which allows the maximum number of levels of block nesting to occur before the blocks become too small.

Three factors affect the ability to fully display a particular GNSD on the screen. These are the lengths of the lines in each individual block (i.e. the real block width), the total length of the GNSD, and the level of nesting that occurs.

If a particular line to be displayed is wider than the enclosing block in which it is to be displayed, the line is truncated and shown with a .. at the end. Thus in many cases, only the first part of a line is drawn.

If the GNSD being displayed is longer than the maximum number of vertical lines allowable on the screen, only the first full screen will be displayed. FLOW-EDIT will then await a further command. If the carriage-return key is pressed, the next 8 lines of the GNSD will be displayed on the screen and so on, with the screen image scrolling upwards to make space each time.

One method for displaying long GNSDs, which has been found useful

when additional FLOW-DL statements are required to be inserted, is to display the GNSD one block at a time. This facility is provided by the **display with stops** command. This command operates in exactly the same manner as the **display** command, except that FLOW-EDIT awaits further commands at the end of each block displayed. As before, the carriage-return key allows the display to continue, and scrolling will occur where necessary.

Deeply nested GNSD blocks, when displayed, will cause the displayed blocks to become too small for their content to be displayed on the screen. To overcome this problem, as each block is displayed, FLOW-EDIT determines its width, and when it would be impossible to display the content of a block, the block is displayed as an incomplete block, with only the first FLOW-DL statement of the block given to show its type e.g.

```
+-----+
|IF ? XVAR..|
+---.....|
+-----+
```

Note that the width of blocks decreases when new nested selections or nested iterations are displayed, so that displaying the first line of the block will provide some indication of the function of the empty block.

In order to view the contents of 'hidden blocks', the cursor may be positioned within the block, and a **display-zooming-in** command given. The block will then be redrawn using the full width of the display screen. Zoom commands are not restricted

to 'hidden blocks' only, but may be given when the cursor has been positioned to any desired block on the screen. Thus truncated lines can also be expanded by choosing an appropriate block on which to zoom.

In the implementation of FLOW-EDIT, a stack of the blocks (GNSD node pointers) at which **display-zooming-in** commands are given is maintained, so that when the user wishes to return to the previous level of perspective, a **display-zooming-out** command may be given which 'pops' the stack and then redraws the screen as a sub-tree rooted on that node.

Block manipulation

In addition to allowing GNSD blocks to be created on the screen, FLOW-EDIT also allows currently displayed blocks to be deleted or re-arranged. Since there is no edit mode (as part of the philosophy of simplifying the user interface), the contents of a block may be entered starting at the point in the GNSD identified by the cursor. An open space will often be available on the screen for the insertion to be made. For example, this will be the case when a new GNSD is created, or where an empty block exists within a selection construct. If however sufficient space is not available for insertion, the **make-space** command may be given, which will create additional free space on the screen by extending the currently identified block downwards toward the last line of the screen, and scrolling the screen display upwards if fewer than the specified number of lines

of space are available. When space is created between a sequence of FLOW-DL lines, the line at which the 'split' was made is re-drawn at the bottom of the screen (i.e. at the end of the space created). The user is thus made aware that what he is entering will be inserted before this line in the GNSD.

Blocks of GNSDs may be moved around using the **clip** and **glue** commands. The **clip** command clears the area of the screen which was occupied by clipped GNSD block, and holds the clipping while the cursor is moved about. Clippings may be re-inserted at some other point in the GNSD using the **glue** command. By successive clipping and glueing commands, blocks and lines of the GNSD may be re-organised. Since it is often useful to clip more than one block before glueing, the command **clip** may be specified with a clipping number. A clipping query command is provided which displays a list of the first lines of all currently clipped GNSD blocks. (Once again, the first line of say a **while** block will usually provide sufficient information to identify the block).

The implementation of the clip/glue facility follows directly from the pruning/grafting operations on the underlying GNSD tree whereby only one node needs to be pruned/grafted in order to prune/graft an entire sub-tree. The clippings file contains nothing more than a list of node numbers, and the re-display of the glued block is then simply a display of the GNSD rooted at this node.

One interesting omission from the FLOW-EDIT command repertoire is the unconditional delete command. Deletions must be performed using the **clip** command. Since clippings are not immediately discarded, the user may immediately issue the **glue** command, which in effect provides an 'undelete' facility. The 'non-finality' of deletions should improve the users confidence on using the editor and encourage experimentation by the user [Good81].

7.2.2 Consistency of FLOW-EDIT operation with respect to the user's conceptual model

As was expected, satisfying the human engineering aspects involved in the development of FLOW-EDIT had the effect of complicating the implementation and prohibiting the application of more 'elegant solutions'. One of the most important of the human factors applied in the implementation was maintaining consistency with respect to the user's conceptual model of the editing processes taking place. That is to say, a simple command language is not a sufficient condition for simplicity, if the commands do not operate as the user would expect them to.

The user's view (conceptual model) of GNSD editing is based on a two dimensional diagram displayed on the screen consisting of a number of blocks, with blocks appearing within blocks. The user expects to be able to point to regions of interest within this two-dimensional space, identifying areas to be 'edited', and then to modify the content of a block, or to add or delete

blocks, or to shuffle blocks around.

The implementation of the GNSD editing process, on the other hand, is based on operations permitted on the underlying tree structure of the GNSD as stored in the data base. The operations applicable to tree-structure editors allow only for 'movement' along branches of the tree, and for the grafting and pruning of sub-trees.

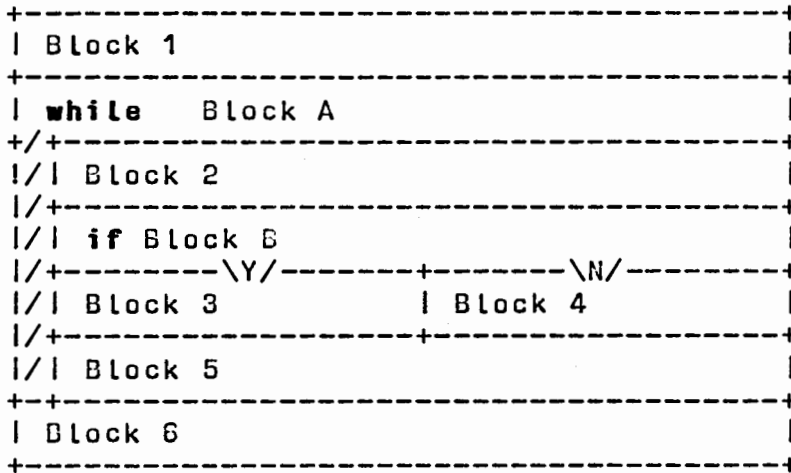
The problem to be solved is obviously one of the user's conceptual 'block' model not corresponding to the editor's underlying 'tree' model.

Since 'ease of use' of an editor is will depend on the editor reacting to commands in the way the user would expect, some way to reconcile the differences is needed. The fact that the differences between the two models described above are rather subtle (since in many instances operations on the two models do actually correspond), introduces the danger of the differences being considered trivial and thus being overlooked.

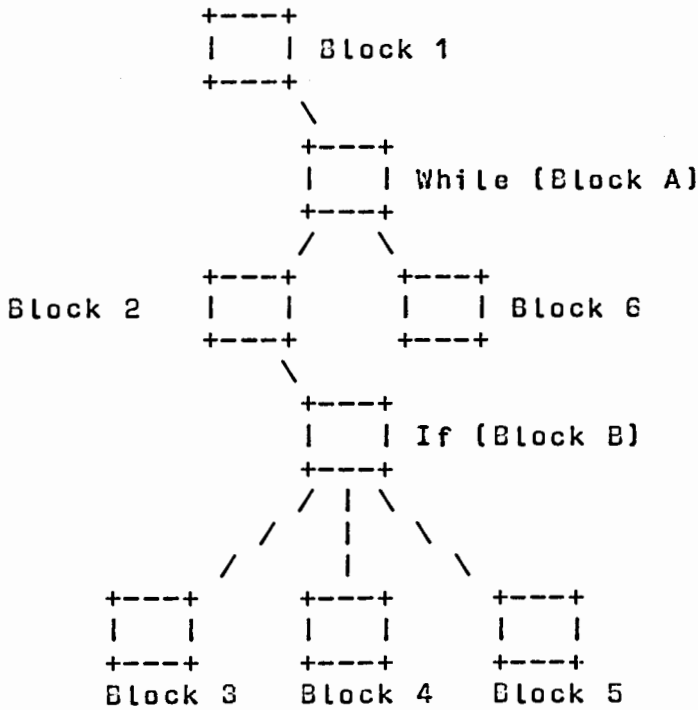
In order to demonstrate that the dichotomy of the models, although subtle, does present real problems, the following examples show what happens when the user wishes to position the cursor within the GNSD. The user will intuitively expect to be able to move his cursor in two dimensions to any block displayed on the screen using the commands **up**, **down**, **left** and **right**.

The following diagrams show a GNSD as it would be displayed on the screen, and the same GNSD as it would be held in the data base.

user's model



editors model



Suppose that the user has positioned the cursor to identify Block 2 and wishes to move the cursor **down**. It is clear from the diagrams that the block **down** from Block 2 is the selection block (Block B) on both the user's and editor's models, and so the command **down** in this case will result in the same operation irrespective of the model used. Now suppose the cursor is positioned at Block 6 and the user wishes it to move up. In this case, the block **up** from Block 6 is the Block 5 on the user's model, and Block A on the editor's model!

Since FLOW-EDIT must perform the edit operations specified by the user on both the screen image and on the data base in parallel, these conflicts are resolved by maintaining a descriptor table which represents the current content of the screen (i.e. user's model). Whenever the cursor is moved on the screen, this descriptor may be examined and the corresponding node of the GNSD tree obtained. Any subsequent editing operations performed on the screen may then also be performed on the data base. The particular representation of GNSD trees chosen, in which each line of text is treated as a block represented by one node of the GNSD tree in the data base, allows the screen descriptor table to be constructed very simply, needing to hold only the row of the screen on which the block is displayed (Y co-ordinate), its left and right boundaries (Xl and Xr co-ordinates), and the data base node number. This information is sufficient to achieve the required correspondence between the two models.

The descriptor table is updated whenever the screen contents changes, which occurs when blocks are inserted or deleted, or when 'scrolling' occurs. On scrolling, the Y co-ordinates of all blocks in the descriptor are updated and any block with its Y co-ordinate 'off the screen' is considered as deleted. To eliminate some of the overheads required in updating the screen descriptor, 'off screen' nodes are allowed to remain on the descriptor table until the available table space is exhausted. At that time, the table is compacted by removing all 'off screen' (including deleted) blocks. Searching through the table is performed in reverse order to reduce the search time, since blocks are added in the order in which they are displayed, and therefore those blocks which are displayed first are the most likely to be 'off the screen'.

7.3 Helpfulness

The motivation for the development of the FLOW environment was to gain substantial productivity improvements. FLOW-EDIT's contributions to productivity improvement are largely attributable to the help provided to the user during the editing process. Assistance may be solicited by the user during the building of a GNSD, or may be provided when FLOW-EDIT deems it necessary (e.g. on error detection).

Line by line syntax checking.

It is common practice for programmers working in an interactive computer environment to use the compiler to detect errors in their programmes. The result is a continual swapping between editor and compiler interspersed with a scribbling of notes concerning the (first few) errors detected by the compiler. This problem is solved with syntax directed editors which provide line by line error checking and result in programmes which are entered and will compile 'error free'. Experience in a commercial environment has shown that syntax errors are not as common as are generally believed, and apart from module interfacing and variable name spelling errors, professional programmers make very few errors in the syntax of the programming language for which they are writing. Consequently, it is believed that the benefit of the syntax checking is that it provides of a convenient means of identifying variables and procedure arguments etc. for interface checking and descriptions. (Were this identification possible by alternate means, the syntax checker could have been omitted from FLOW-EDIT without fear of a major loss in productivity).

Each line of text entered from the keyboard is passed to the syntax checker which determines whether it is an edit command or valid FLOW-DL statement. The FLOW-DL and editor command grammars are combined into a single LL(1) grammar thereby allowing the syntax prompting and correction facilities to be common to both commands and FLOW-DL statements. If the line entered is not in the combined grammar, an error message is displayed,

and the user is allowed to correct the text.

The syntax checking module of FLOW-EDIT uses a recursive descent recognising procedure for the combined FLOW-DL and edit command grammar [Lew76]. The procedure is implemented without a lexical analysis phase. The text line is parsed character by character, with a pointer to the current character being maintained. Lewis's procedure has been modified such that as soon as an error is detected in the line, syntax checking aborts, and an error message and the pointer are passed back to the edit module. The edit module then displays the error message and positions the cursor at the point at which the error was detected thereby allowing the user to correct the line starting at a likely point at which the error occurred, namely the point at which it was detected.

The use of lengthy (and possibly misleading) error diagnostics has been avoided. The combination of a terse error message and a cursor showing the place where the error was first detected has been found to give sufficient assistance to the user in correcting the error. The following are some examples of error messages produced together with the FLOW-DL which generated

them. The `_` symbol shows the place where the cursor will be positioned after error detection.

error message	FLOW-DL source line
1.] EXPECTED	ARRAY[IX_ <u>_</u> { ARRAY[[IX-1]]
2. VARIABLE UNDEFINED	XLARGE { M <u>A</u> XVL
3. VARIABLE/CONSTANT EXPECTED	XDIST { (IX- <u>_</u>)/IY
4. ILLEGAL CONSTANT	.SORTVAL { 1 <u>B</u> RRRAY[IX]

A symbol table is built for all variables declared in the data declaration and interface specification. This allows undefined variable names, as in example 2 above, to be detected. Note that in the case of apparently misspelled variable names, the cursor is positioned to the beginning of the name, and not the point where the syntax checker determined the error.

Module interface checking.

The syntax checker makes use of the entry point table and the interface definition trees on the data base described in chapter 6 for verification of procedure names and parameters.

Module interface checking is considered to be one of the major benefits of FLOW-EDIT, since the checking of interfaces of externally defined modules cannot be performed by traditional compilers, as these compilers are unable to access the source of the other modules. Likewise, the link-editor, which traditionally is

the only component aware of all modules constituting a programme, is usually only concerned with resolving addresses. Thus traditional methods of producing compiled software require interfaces to external modules to be verified by inspection.

Interface checking is implemented by identifying the called module name during the parsing of the line, locating the interface specification tree of the called module in the software production data base (via the entry point table), and matching the types of the called parameters against those given in the interface specification. The power of this method derives from the fact that the interface is checked against the actual definition used in the referenced module, and not against a user supplied definition (as for example in the case of EXTERNAL definitions in extended versions of Pascal).

Interface descriptions

As part of the principle of providing assistance, FLOW-EDIT will provide information on module interfaces if requested. To obtain this information, the user must enter his call statement in the normal way with a ? character appended to the procedure name. The feedback information is displayed in the space immediately following the current line, and is deleted once a syntactically

correct call statement has been entered. For example, the interface description of the OPENF module is displayed as follows

```

          <<CALL OPENF?                               >>
----- OPENF  I/F DEFINITION -----
CHANNEL { I4          $ CHANNEL ON WHICH TO OPEN FILE
FILENAME { A20       $ NAME OF FILE TO BE OPENED
IERR  } I4          $ ERROR PARAMETER RETURNED
-----

```

The use of 'in-line' comments on declarations can be used to provide additional information on the variables used in the interface. Producing the interface specification interactively eliminates the documentation 'look up' time that would normally be required (presuming that the documentation actually existed), and has a major advantage of communicating up to date information, particularly where more than one programmer is working on the construction of a large system. Interface specifications will exist for all procedures currently being developed as part of the emerging system, possibly long before the actual module is written, and these interface specifications may be called up as needed. Specifications may also be entered on the data base for library procedures.

The interface description facility may also be used to create new interface definitions if the user finds that no interface exists for the module he intends calling (as would be the case in top-down coding). If a ? is given on a procedure name, and the procedure cannot be found in the entry point table, the user is asked if he wants to create a new interface. The definition

of this interface may then be entered, and will be stored on the data base as a 'stub' for later expansion. (Note that 'stub' code may be generated from these interfaces if required (see chapter 8)).

Variable descriptions

FLOW-EDIT can provide the user with descriptions of variables defined within its data declaration during the editing of procedures in a similar manner to the way in which interface descriptions are made available. Any variable name given in a FLOW-DL line may be followed by the ? character. If the variable has been previously declared, the declaration of that variable will be displayed. The symbol table contains the symbol, type and GNSD node pointer. To display the description, FLOW-EDIT merely accesses that node on the data base, and displays it. If the variable has not been declared, FLOW-EDIT will ask the user if it should be defined, and then prompt the user for the declaration. This will result in the declaration being automatically appended to the data definition tree.

7.4 Applicability to GNSD editing

The requirements of FLOW-EDIT are that it should operate as a GNSD picture builder for construction of independent modules whilst at the same time being concerned with the goals of the total software system being constructed.

In the global sense, FLOW-EDIT operates on the entire software production data base. As discussed, the entry point table and interface definition sub-trees are used to verify the syntax of procedure calls. New interfaces (for stubs) are also created during the edit process.

There are a number of other 'global' facilities provided by FLOW-EDIT which assist with the system development effort.

Cross reference lists

During the process of interface checking, FLOW-EDIT updates the 'called by' list in the entry-point table of the module referenced, and the 'calls list' of its own entry point table entry. This allows a cross reference listing to be produced, thereby providing a useful piece of documentation to assist in the understanding of the system. If the interface specification of a module is altered, all modules in the 'called by' list are automatically flagged as 'interface in error'.

Global Data Declarations and Copy Procedures

In order to support 'global' development, FLOW-EDIT provides the facilities to declare 'global data' and 'copy procedures'. These definitions are analogous to the COBOL COPY statement. Global data provides the means to define common data definitions which may be used by a number of different programmes as 'schemas', or by a number of different modules in the same programme as

'common blocks'. Copy procedures provide standard fragments of code (or in this case, fragments of GNSDs) which may be used repetitively within a system.

Global data declarations and copy procedures are referenced in the data and procedure trees of a module respectively by special FLOW-DL statements which specify their name. This provides a 'virtual sub-tree' within the relevant tree which is expanded during the symbol table building and module syntax checking stages of FLOW-EDIT.

7.5 Ensuring syntactical correctness of the data base

FLOW-EDIT is essentially a syntax directed editor which allows GNSD trees held in the software production data-base to be inserted, modified or deleted. The theoretical basis of syntax directed editors is well developed and presents no real design challenge. GNSDs can be expressed by a context free grammar, with the GNSD constructs as terminals, and the blocks as non-terminals. The GNSD will be complete once all of the non-terminals (blocks) have been expanded. An editor will maintain the syntactical correctness of a GNSD represented by its parse tree, if the edit operations are restricted such that only syntactically correct and complete sub-trees of this tree to be inserted, deleted or modified.

In order to show that syntactical correctness is maintained, it is sufficient to show that syntactical correctness is maintained at the completion of each insertion, deletion or modification. A brief description of these operations follows.

Insertion

FLOW-EDIT checks the syntax of each FLOW-DL statement entered at the time it is entered and thus the datum associated with each node on the data base will always be syntactically correct. The symbol table required for the checking of the procedure definition tree is built from the corresponding data definition and interface definition trees whenever the procedure definition tree is entered.

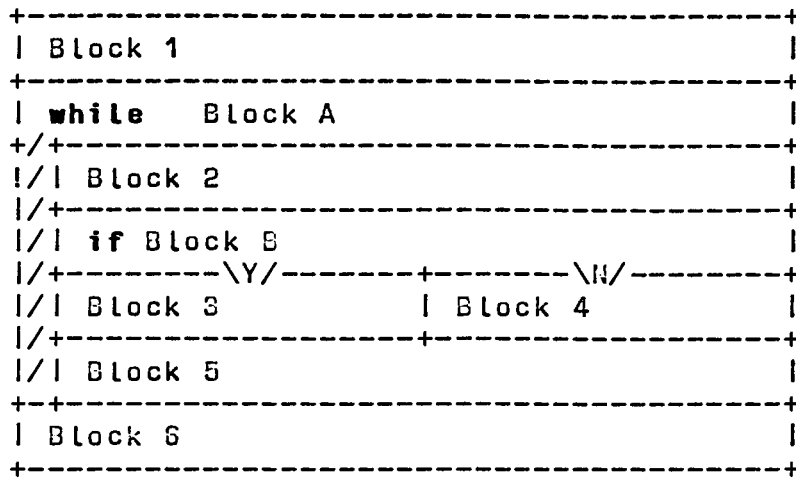
Syntactically correct GNSD trees require that each 'block generating' function key (**while if-then-else** etc.) be terminated with an **end** key. Since GNSDs are built interactively as FLOW-DL statements are entered, it may occur that one or more **end** function keys will be outstanding at any one point in time during an insert operation. FLOW-EDIT stacks all 'outstanding end requirements' (e.g. a **while** requires one **end** whereas an **if-then-else** requires two), and pops this stack whenever an **end** is given. All outstanding **ends** are then generated automatically by FLOW-EDIT when the user gives a command such as **move**, **display** or **exit**, thus satisfying the syntax requirements before moving, displaying, exiting etc. One other feature of the implementation which 'gaurantees' syntactical correctness even if FLOW-EDIT terminates on error during

insertion, is that the sub-trees are not built 'in-place' (i.e. not immediately linked to their eventual parent node), but rather they are constructed as free standing trees, and grafted on the required parent node only when the insertion has been successfully completed.

Clipping and glueing

Complete tree structures on the data base are maintained during **clip** and **glue** operations by allowing only complete sub-trees on the GNSD tree to be clipped or glued. (This restriction has the added advantage that it lends itself to an almost trivial implementation). The user is not provided with the facility to specify the clipping of multiple blocks for two reasons

- 1) this would allow syntactically incorrect GNSDs to be stored on the data base, violating the system design constraint
- 2) this would violate the principle of maintaining consistency with the user's conceptual model, as can be seen from the following example where the effect of a command such as **clip 2 blocks** given from within Block A or Block B or even Block 5 may not be correctly predicted by the user.



From this diagram, the user would probably realize that blocks 3 and 4 are contained within block B, but may have difficulty in determining that there are three blocks in sequence within Block A.

Modification

If the text associated with a GNSD node is modified, this text is once again passed to the syntax checker. The node datum includes the node type (assignment statement, if condition, procedure call etc.) which allows the appropriate checking to be done on this text.

If the GNSD trees for procedure definitions or functional descriptions is modified (deleted or re-arranged), syntax checking of the individual node data are not required. If however the interface specification or data definition trees are modified, the procedure tree node datum will be re-checked for syntax errors.

7.6 Display terminal constraints

Screen size and resolution

As one of the design criteria, FLOW-EDIT is required to provide for the editing of GNSDs on terminals used for traditional programme editing i.e. non graphic terminals. Such terminals usually have a display format of 80 X 24 characters and consequently the space available for displaying GNSDs is servererly limited. Not only are very few lines (rows) allowed on these screens, but horizontal lines drawn on the screen for to delimit GNSD blocks cannot be drawn between lines of text (as can be done on graphics terminals), and will therefore occupy one whole line. Some modifications of Nassi Shneiderman Diagrams were described in chapter 3 to allow for a more compact representation of these diagrams and thus reduce their length. Suppression of redundant lines used in drawing the blocks on the screen may also be achieved by allowing GNSD blocks to share common boundaries. The implementation of a line suppression scheme provides a good example of a practical consideration which detracted from the implementation of an 'elegant' solution; displaying a GNSD on the screen can be achieved by a pre-order traverse of the GNSD tree, with horizontal lines (marking the beginning and end of blocks) being drawn whenever a leaf is reached, or a node with multiple branches (while, if etc.) is encountered. For the practical reasons mentioned, a number of switches have to be maintained to decide whether boundary sharing should be

done. No general rules can be formulated for boundary sharing, since not all boundaries are shared in all cases. For example, certain boundaries which could be shared, actually represent empty GNSD blocks, and if line suppression is performed, the positioning to one of these empty blocks, (e.g. to insert a line) would be ambiguous. Thus a number of ad hoc measures are implemented in order to maintain an aesthetically pleasing yet functional display of GNSDs

7.7 Implementation notes.

Most of the FLOW environment development effort has gone into the development of the editor. The design principles were applied and the implementation constantly revised over a three year period as experience was gained.

The fact that GNSDs are well structured objects, allows the editor to be implemented in a very compact form. The editor contains a number of 'tree operators' (procedures) which are combined to produce the required functions. For example, a procedure exists to display a 'sub-tree' given the data base node number and screen window size. This procedure is used for the display of the four GNSD trees (interface definition, functional description, data definition and procedure definition), for zooming in and out, and for re-drawing clipped blocks of a GNSD within another block using the glue command. The editor compiles to only 8000 P-code instructions in spite of the fact

that the editor was optimized for maintainability and not for size.

Bibliography

- [Good81] Good M. **Etude and the Folklore of User Interface Design.** ACM SIGPLAN Notices, Vol 16 No. 6, June 1981.
- [Lewi76] Lewis P M, Rosenkrantz D J, Stearns R E **Compiler Design Theory.** Addison Wesley Publishing Company Inc. Phillipines 1976.
- [Newm79] Newman W.M., Sproul R.F. **Principles of Interactive Computer Graphics.** [2nd Edition] McGraw-Hill, New York, 1979.

8. Post-Processors.

The FLOW environment was designed for the development of systems in some of the most commonly used high-level programming languages, namely FORTRAN IV, COBOL and Pascal. This is made possible through the creation of a data base containing an intermediate language (FLOW-DL) which may be accessed by independent post-processors and translated into the appropriate target code. (In what follows, the post-processors' point of view is used to establish the terminology, so that **source-code** refers to the FLOW-DL held on the software development data base, and **target-code** refers to the post-processor output).

A suite of three 'language' post-processors and a 'documentation' post-processor have been implemented. These post-processors are FLOWF4 producing the FORTRAN-IV target code, FLOWCOB producing COBOL, FLOWPAS producing Pascal, and FLOWCAHRT producing GNSD documentation.

The post-processors are required to perform two functions. These functions are firstly to generate target code from source held on the software production data base, and secondly to provide a 'trace' facility to be used during the debugging and testing stages of development.

8.1 Generation of target code.

Post-processors produce target code by translating the FLOW-DL source-code held on the software production data base into equivalent code for each of the target languages. Provided that the translation of the source code is performed correctly, any FLOW-DL source code implementation of a procedure may be obtained in the target languages for which a post-processor exists.

8.1.1 FLOW-DL

FLOW-DL has been described in Chapter 3 as being a GNSD definition language. The GNSD picture drawing facilities provided only part of the motivation for FLOW-DL. The second consideration was to provide some level of independence from the high level languages for which target code was to be produced.

In designing FLOW-DL, a choice had to be made from amongst the possible facilities (i.e. language constructs) to be incorporated into this new language in order to provide a translation from the new source into the target languages in as simple a manner as possible. The possibilities at the one extreme were to design a language incorporating only those facilities common to all of the object language facilities (i.e. the intersection set). At the other extreme, the choice was to design a language incorporating all of the facilities available in the target languages (i.e. the union set).

The advantage of designing a language with facilities common to all of the target languages is that it allows a direct mapping of the source language onto the target language and thus simplify the task of post-processor writing. This approach would however lead to smaller and smaller subsets of the language facilities being provided as more target languages were added.

Designing a language with all of the facilities provided by all of the target languages on the other hand, would have a major disadvantage in that a number of features not directly provided in the target language (such as records in FORTRAN) would have to be simulated in order to obtain the required result. Furthermore, this would result in an increasingly large set of source language facilities as more target languages were added.

Obviously both extremes are impractical, and a middle course of action was followed, namely selecting a 'useful' subset selected from the union set. Thus some of the richness found in many of the languages was retained.

Since FLOW-DL was designed for the building of GNSD pictures, all of the constructs required by structured programming are provided. FLOW-DL also provides the data definitions which are possible with GNSDs. The following table shows the GNSD constructs and the extent to which simulation is necessary in order to implement these constructs in each of the target languages.

The classifications given in the table (X, S and L) were based on the ability to translate directly from the source code (FLOW-DL) into the target code without the need to introduce additional control mechanisms such as GO TO and PERFORM statements.

GNSD Construct	PASCAL	FORTRAN-IV	COBOL

procedure definition			

Sequence (of blocks)	X	S	L
Iteration (controlled)	X	X	L
Iteration (do while)	X	S	L
Unary selection (if then)	X	L	L
Binary selection (if then else)	X	S	L
N-ary selection (case)	X	L	L
Procedure call	X	X	X

data definition			

variables	X	X	X
arrays	X	X	X
records	X	S	X
variant records	X	S	L

X = No simulation required			
L = Limited simulation required.			
S = Full simulation required			

As an example of these classifications, the implementation of the sequences construct in COBOL is considered.

A 'sequence' may be defined as a number of blocks of any construct placed in series. For example, the sequence of three blocks (two simple and one selection block) is shown as follows

```

+-----+
| Statement A                               | |<-- Block 1
+-----+
| if condition1                             | |
+-----\Y/-----+-----\N/-----+ |<-- Block 2
| Statement B           | Statement C       | |
+-----+-----+-----+-----+
| Statement D                               | |<-- Block 3
+-----+

```

This sequence can be translated into COBOL as

```

STATEMENT-A
IF CONDITION-1
    STATEMENT-B
ELSE
    STATEMENT-C.
*   ENDIF
STATEMENT-D

```

In this example, a direct translation into COBOL was possible. If however the previous GNSD forms part of some larger GNSD, such as in the following example, there is no direct translation.

```

+-----+
| if condition 2                             | |
+-----\Y/-----+-----\N/-----+
| Statement A                               | |
+-----+-----+-----+-----+
| if condition 1                             | |
+-----\Y/-----+-----\N/-----+
| Statement B           | Statement C       | |
+-----+-----+-----+-----+
| Statement D                               | |
+-----+

```

COBOL may therefore be considered as a 'block structured' language with only one level of nesting allowed. Accordingly, the sequence construct in COBOL is classified as (L) in the table given.

8.1.2 The FLOW-DL transformations

The post-processors are required to perform the translations from FLOW-DL source to FORTRAN, COBOL or Pascal target code. This operation is implemented at two levels. At the first level, FLOW-DL 'statements' (i.e. the text associated with each node in the GNSD tree) is translated. In the second level, the FLOW-DL structure (GNSD sub-trees) is translated which provides the necessary flow of control between statements.

Statement transformations

Statement transformations are direct translations of the text held on each node within the GNSD tree with each node generating one line of source code. In the simple case, the construct in which the line occurs does not affect the translation.

For example the FLOW-DL comment

\$ initialize all counters

is translated into

C INITIALIZE ALL COUNTERS	(FORTRAN IV)
* INITIALIZE ALL COUNTERS	(COBOL)
(* initialize all counters *)	(Pascal)

One of four sets of transformations may be selected depending on in which of the four GNSDs trees the node occurs. For example, in the case of the functional description tree, all the text of all nodes are translated into comments.

Typical procedure definition tree statements are

```
PI { 3.14159
CIRCUM { PI * DIAMTR
```

which are translated into

```
PI = 3.14159          (FORTRAN IV)
CIRCUM = PI * DIAMTR
```

```
MOVE 3.14159 TO PI. (COBOL)
COMPUTE CIRCUM = PI * DIAMTR.
```

```
PI := 3.14159;      (Pascal)
CIRCUM := PI * DIAMTR;
```

A typical data definition translation is

```
XLARGE { F7.5
```

which becomes

```
REAL XLARGE          (FORTRAN IV)
```

```
01 XLARGE PIC 9(1)V9(5). (COBOL)
```

```
XLARGE:REAL;        (Pascal)
```

Each node on the data base has a construct type (sequence, selection etc.) which, together with the GNSD tree type uniquely identifies the transformation routine to be applied to the node. For example,

the the 'Boolean' transformation will always be performed on text of the node associated with selection or iteration (do while) in the procedure definition tree.

Structure transformations

Structure transformations are performed in order to obtain the functionally equivalent flow of control as specified in the GNSD. Where the language constructs facilitate a direct mapping of one FLOW-DL source construct onto one target construct, as in the case of Pascal, these transformations consist of simply inserting the correct 'keywords' before and after the [translated] node text. For example the following GNSD would be translated into Pascal by performing the pre-order traverse of the underlying GNSD procedure sub-tree, and supplying the correct Pascal 'key word' as each node in the tree is encountered.

GNSD

```

+-----+
| while condition 1 |
|/+-----+
|/| if condition 2 | |
|/|-----\Y/-----\N/-----|
|/| statement A | Statement B |
|/|-----+
|/| statement C |
+-----+
| statement D |
+-----+

```

Pascal

```

WHILE CONDITION 1 DO
  BEGIN
    IF CONDITION 1 THEN
      BEGIN
        STATEMENT A;
      END
    ELSE
      BEGIN
        STATEMENT B;
      END;
    (*END IF*)
    STATEMENT C;
  END;
(*END WHILE*)

```

Where the target language does not provide as simple a mapping of control structures, these control structures must be simulated using a number of other control constructs which are provided in that target language. In this case the post-processors make use of some 'inherited' information acquired during the tree traverse in order to make the correct translation. For example the FORTRAN IV post-processor (FLOW-F4) uses the unconditional GO TO and labelled CONTINUE statements to achieve the flow of control. Labels are placed on a stack as each branch is encountered in the GNSD tree, and un-stacked when a branch has been fully

expanded. The following example shows the how labels are used in iterations.

GNSD

```

+-----+
| while condition 1 |
|/+-----+
|/| while condition 2 |
|/|/+-----+
|/|/| statement A |
|/|/| statement B |
|/|/| ... |
+-----+

```

FORTRAN IV

```

5   IF (.NOT.(CONDITION 1)) GO TO 10
15  IF(.NOT.(CONDITION 2)) GO TO 20
      STATEMENT A
      STATEMENT B
      ...
      GO TO 15
20  CONTINUE
      GO TO 5
10  CONTINUE

```

In those cases where the target language provides a construct similar to (but not the same as) the requirement of GNSDs, this construct may be used. For example, COBOL has a 'do until' which is similar to the GNSD 'do-while'. The COBOL target code generated by FLOWCOB for the previous iteration example would be as follows

```

        IF NOT (CONDITION1) PERFORM LOOP-1 UNTIL CONDITION1.
...
...
LOOP-1.

        IF NOT (CONDITION2) PERFORM LOOP-2 UNTIL CONDITION2.
LOOP-2.
        STATEMENTA
        STATEMENTB
        ...

```

The 'IF NOT (CONDITION) PERFORM...UNTIL ...' is introduced to overcome the 'one trip' implementation of PERFORM statements found in some COBOL compilers.

It should be noted that the left node of the iteration construct in the GNSD (which is the root of the loop 'body') is not expanded during the same traverse of the GNSD tree. Instead this node (and its associated sub-tree) is queued together with a label which defines the COBOL paragraph (such as LOOP-1) for expansion on a later traverse. Thus the FLCWCOB post-processor attempts to process the GNSD tree in pre-order, but whenever a PERFORM is needed to implement the construct, the un-processed sub-tree beneath that node is regarded as a new root node for later expansion. After each traverse of the tree, the queue is examined, and any as yet un-processed sub-trees are then taken as the root to be processed. The use of a queue ensures that the target code produced follows in a logical sequence.

Implementing Data Structures

In translating procedure GNSDs it was possible to implement a two level translation process, since the translation of the

text of the node was not affected by the structure in which it occurred. This approach is not always possible in translating data structures. For example, a special case of statement translation exists where the statement translation depends on the structure in which the statement occurs such as that required to implement FORTRAN arrays. Arrays are specified in the following form in the data definition GNSD

GNSD

```

+-----+
| for dummyvar1 = n1,m1 |
|/+-----|
|/| for dummyvar2 = n2,m2 |
|/|/+-----+
|/|/| arrayA:R6.3 |
|/|/| arrayB:I4 |
|/|/| ... |
+---+-----+

```

This GNSD would be translated into FORTRAN IV by FLOWF4 to produce:

```

REAL ARRAYA(N1:M1,N2:M2)
INTEGER ARRAYB(N1:M1,N2:M2)

```

There is obviously no way that the post-processor could obtain the array subscripts simply by inspecting the node on the GNSD tree. The subscript information must therefore be remembered from the preceding **for** statement. FLOWF4 stacks the loop constraints (n1,m1 etc.) as each iteration construct is encountered in the data definition tree. While this stack contains some information, each data item encountered is translated as an array declaration.

Another example of special translations occurs in the use of procedural data structures (introduced in chapter 3). The principle of procedural data structures is that any data structure represented by a GNSD can be simulated by producer and consumer procedures. This principle has been used in order to provide a 'record' facility for FORTRAN IV. Special translations are made in the procedure tree whenever record assignments are encountered. Thus

```
RECA ( RECB
```

is translated into two call statements

```
CALL RECB(PRODCE)  
CALL RECA(CONSME)
```

where PRODCE and CONSME are parameters to the sub-routines 'implementing the records'. (The subroutines of course only implement the assignment operator for the records).

Special translations are also made on the data definition tree. This tree is in fact processed twice by the FLOWF4. On the first pass, the tree produces the normal data declarations which makes the fields of the record available to the main procedure. On the second pass (which is performed after producing the code for the procedure definition) the records nodes are processed and one FORTRAN subroutine is generated for each record defined. Record nodes are identified during the first pass and queued for later expansion.

The record definitions (which occur in the data definition tree)

are expanded similarly to if they were procedure trees, except that special line translations are made for each data item declared.

Thus

```
+-----<<RECA>>-----+
| IX=1,20                    |
|/+-----+                 |
|/| XLARGE:F6.2              |
|/| YLARGE:F6.2              |
+-----+
```

is translated into

```
      SUBROUTINE RECA(CPFLAG)
      REAL XLARGE(1:20)
      REAL YLARGE(1:20)
      COMMON/COMM01/XLARGE,YLARGE
      DO 5 IX=1,20
          CALL F4REC(XLARGE(IX),CPFLAG)
          CALL F4REC(YLARGE(IX),CPFLAG)
5      CONTINUE
      END
```

where F4REC is a special sub-routine which reads or writes from an internal buffer, depending on the value of CPFLAG.

8.2 Programme trace and stub facilities

The FLOW environment is designed to include support for the system building and testing phases of the software production process. One of the ways in which support is given by the FLOW environment in this regard is the provision of a programme trace and stub generation facility.

Trace facility

The language post-processors provide a trace facility by inserting 'trace code' into the target code during the code generation process. Consequently tracing is achieved without resorting to a 'trace package'. As mentioned in chapter 4, the level of tracing required is set by trace commands (post-processor directives) embedded within the FLOW-DL source. The level of tracing to be performed may be set at one of three 'granularities':

- 'coarse': displays the name of the procedure and values of the input parameters upon entry, and the output parameters upon return.
- 'medium': displays as for 'coarse' as well as displaying the entry into and exit from each block of the flowchart.
- 'fine': displays as for 'medium', as well as displaying each statement in the order of execution, and the value to be stored in the case of an assignment statement.

Any number of post-processor directives may be set within one procedure, so that the level of tracing may be altered according to the requirements of the user.

The interface definition of the GNSD is used by the post-processors in order to derive the 'coarse' level of trace code. Trace statements are generated at the beginning and end of the procedure

body of the target code for the input and output variables respectively. For example the OPENF interface definition

```
+--<<INTERFACE DEFINITION>>-----+
| INCHAN { I4  $ INPUT CHANNEL NUMBER      |
| FILENAME { A6 $ NAME OF FILE TO BE OPENED |
| ERRCODE } I4  $ STANDARD SYSTEM ERROR CODE |
+-----+
```

would be used by FLOWPAS to produce the following trace code.

```
BEGIN
  WRITELN('*** PROCEDURE OPENF ENTERED ***');
  WRITELN('INCHAN = ',INCHAN:4);
  WRITELN('FILENAME = ',FILENAME:6);
  ....
  body of procedure OPENF
  ....
  WRITELN('ERRCODE = ',ERRCODE:4);
  WRITELN('*** PROCEDURE OPENF RETURNED ***')
END; (*OPENF*)
```

The 'medium' level of tracing is obtained by allowing the post-processors to generate additional code in order to display messages which will appear whenever the beginning or end of each GMSD selection or iteration block is encountered. The statement associated with the selection or iteration node is also displayed so that the user may easily identify the GMSD block. As an example,

'medium' tracing for the following GNSD

```

+-----+
|  if XVAL = 1                               |
|-----\Y/-----+-----\N/-----|
|  ...                                     | ... |
|-----+-----|
|  ...                                     |
+-----+

```

would be generated by FLOWF4 as

```

      WRITE(6,1)
1     FORMAT("IF XVAL = 1")
C     IF XVAL = 1
C     --
      IF (.NOT.(XVAL.EQ.1) GOTO 10
          WRITE(6,2)
2     FORMAT("    TRUE")
          ...
          GO TO 15
C     ELSE
C     ----
10    WRITE(6,11)
11    FORMAT("    FALSE")
          ...
C     END IF
C     --- --
15    WRITE(6,16)
16    FORMAT("END IF")
          ...

```

The 'fine' trace level requires the post-processor to produce code displaying each assignment statement found in the body of the GNSD together with the value to be stored. Thus the GNSD block

```

+-----+
|  ...                                       |
|  SALESTAX { COST * TAXRATE                |
|  ...                                       |
+-----+

```

would be used by FLOWCOB to produce

```

...
COMPUTE SALESTAX = COST * TAXRATE .
DISPLAY 'SALESTAX { COST * TAXRATE --- ',SALESTAX .
...

```

within the procedure division of a COBOL programme.

Stub facility

The creation of programme 'stubs' has been found to be a very useful tool during the top down development of a system. The FLOW environment forces the creation of an interface specification before a procedure can be referenced so that interface definitions will exist for all referenced procedures. The post-processors are able to use these interface definitions to generate 'stubs' automatically for the as yet 'unwritten' procedures. The generation of a stub is a similar to the generation of 'coarse' tracing described above, with the exception of the values of the parameters defined as output parameters (if any) which are requested from the user instead of being displayed.

Thus the previous OPENF interface definition could be used to generate the following Pascal stub

```

BEGIN
  WRITELN('*** PROCEDURE OPENF ENTERED ***');
  WRITELN('INCHAN = ',INCHAN:4);
  WRITELN('FILENAME = ',FILENAME:6);
  WRITE('ENTER ERRCODE : ');
  READLN(ERRCODE);
  WRITELN('*** PROCEDURE OPENF RETURNED ***')
END; (*OPENF*)

```

This stub, when used in an the testing of an intermediate system, allows the user to manually enter the return parameter, and thus allows top-down testing to proceed.

8.3 Maintainability

Although target code created within the FLOW environment may be used transparently during software production (i.e. generated without the user ever being required to read it), it is possible that the code produced will have to be maintained in its output (target code) form. To cater for this need, the post-processors have been designed to produce highly readable output in a standardised format.

Translation from the higher order language FLOW-DL into the target language is a purely mechanical process. The post-processor is therefore able to adhere strictly to programming standards which may be defined for the target language. A number of recommendations for standards and good programming practice exist e.g.[Lee80], [Ledg76], and many of these recommendations have been implemented in the current post-processors.

Prettyprinting

If 'mechanically produced' target code is to be maintained (in its output form), it must be made 'readable.' The post-processors achieve this requirement by providing indentation in the code produced and by inserting helpful comments wherever possible.

Indentation is used to show the scope of each iteration and selection construct. The indentation is easily implemented in the post-processor by keeping track of the level of nesting of blocks within the GNSD being translated.

Readability is improved by inserting comments which are used to show the 'original' statement as opposed to the translated form whenever this is significantly different from the original. For example, the FORTRAN comment

```
C   IF XLARGE <= 100.0
C   --
```

precedes the translated line

```
IF (.NOT.(XLARGE.LE.100.0)) GO TO 25
```

Comments are also used to introduce 'pseudo statements' into the target code. These 'pseudo statements' enhance the effectiveness of indentation in defining the scope of the translated GNSD blocks. For example, the scope of a selection is terminated by the **end if** 'pseudo statement' in all three of the post-processor target languages. Similarly where an **else** is not provided, comments

are used. Thus an 'if statement' would be shown as

```

C   IF XLARGE <= 100.0
C   ---
C   IF (.NOT.(XLARGE.LE.100.0)) GO TO 25
C       ....
C       ....
C   GO TO 30
25  CONTINUE
C   ELSE
C   -----
C       ....
C       ....
30  CONTINUE
C   ENDIF
C   -----

```

Users of the above form of FORTRAN listing have found it easy to ignore the GOTOs and CONTINUEs, and to read only those control structures underlined.

Flow charts

In order that programme documentation be of any use during the software maintenance phase of the software development life cycle, it is essential that this documentation be correct and up to date. (The same remark is also true in the development phase!) Since it is highly unlikely that the manually created documentation of medium sized systems will correspond with the 'as implemented' system at any one point in time (to say nothing of large scale systems), system builders have tended to discard traditional documentation (flowcharts etc.) and turned to writing 'self documenting' code.

To alleviate the documentation production problem, a special purpose post-processor, FLOW-CHART, capable of drawing GNSDs directly from the source code on the software production data base has been included as part of the environment.

The use of FLOW-CHART in generating 'hard copy' GNSDs has been found to give a considerable saving in time previously spent on this activity during the development phase, (if indeed documentation was previously done during development) as well as in savings during maintenance phase by virtue of the 100% correspondence between the FLOW-CHART documentation and the high-level language target code (since they are both generated from the same source code).

The implementation of FLOW-CHART is straight-forward and uses many of the techniques described earlier for language post-processors. The four GNSD trees are translated into four separate diagrams labelled 'functional description', 'interface definition', 'data definition' and 'procedure definition' respectively. It is obviously not necessary for FLOW-CHART to perform any statement translations since the statements in the GNSD to be produced are in FLOW-DL form (although the language specific transformations as used in the high-level language post-processors could easily be applied if language specific GNSDs are required for users who are not familiar with FLOW-DL notation).

FLOW-CHART traverses four GNSD trees in pre-order sequence producing one or more line of the GNSD as each node is expanded. Unlike FLOW-EDIT, lines of text are not truncated if they exceed the width of the block in which they occur, but rather they are allowed to extend into the adjacent GNSD blocks. To avoid lines overwriting one another, each node containing text is printed on a separate line. When the size of the GNSD block becomes too small, the node being processed is treated as a leaf of the GNSD tree for the purposes of the current diagram. The sub-tree rooted on this node is then queued for subsequent processing

One of the problems encountered in implementing FLOW-EDIT was how to reduce the length of the output document in order to make optimal use of the terminal screen space. The solutions found for this problem are applicable to FLOWCHART, and it has been found in practice that the GNSDs which are produced by are considerably shorter than the corresponding target code produced by the other post-processors. This makes the FLOWCHART listings practical working documents.

8.4 Implementation notes

The implementation of post-processors has been found to be surprisingly easy. As expected, the Pascal post-processor was the one most easily implemented due to its block structure and a direct correspondence of language constructs to GNSD constructs.

With the exception of the implementation of the record facility, (which was done mainly for experimental reasons) the FORTRAN post-processor was also found to be straight-forward. This was due to the fact that the GNSD control structures in FORTRAN are implemented using unconditional GO TO and CONTINUE statements.

The implementation of the COBOL post-processor posed some problems due to the desire to produce 'structured' target code. (If this were not the case, GO TOs could also have been used in the COBOL target code.)

The following table shows the relative sizes of the post-processors using the number of P-code instructions produced by the compiler for each as a measure.

post-processor	size
FLOW-F4	4500
FLOW-PAS	2500
FLOW-COB	3500
FLOW-CHART	2000

Note that the size of all of the above post-processors could be reduced (if required) by optimizing their source code since, as in the case of FLOW-EDIT, a large amount of code is repeated within the post-processors to enhance their maintainability.

Experience with the current post-processors have shown the utility of the FLOW-DL data base approach, and the writing of new post-processors for procedure oriented languages should present no real problems.

Bibliography

- [Ledg80] Ledgard H.F. **COBOL Under Control**. Communications of the ACM, Vol 19 No 11 pp. 601-608 November 1976.
- [Lee80] Lee G, et al. **Fortran Programming Standards**. ACM SIGPLAN Notices, Vol 15 No 2 pp. 51-63, February 1980.

9. Practical experience in the use of the FLOW environment.

9.1 The prototype

The current implementation of the FLOW environment followed two years of experimentation using a prototype system. The prototype, called FLOWTRAN, was implemented in FORTRAN IV, and produced FORTRAN IV target code. Although the current environment has been re-designed and re-written in Pascal, it is instructive to discuss the prototype, since most of the practical experience in programming environments using Nassi Shneiderman Diagrams has been derived from use of this system.

FLOWTRAN had limited capabilities, being able to translate only the procedure definition constructs of GNSDs. The interface definition of the programme was entered in response to prompts ('enter procedure name', 'enter parameters', etc.), and the data definitions were entered using function keys which selected the FORTRAN data types (REAL, INTEGER etc.), followed by the name of the data item being declared. The use of function keys for the data type names reduced the number of keystrokes required for entering this part of the programme, and eliminated spelling errors in the data definition. Function keys were also used to select the GNSD construct in the procedure definition. Statements entered in the procedure definition (i.e. the content of the GNSD boxes) were required to be valid FORTRAN IV statements, were not checked for syntax errors, and were directly output

by the code generation section of FLOWTRAN without any form of intermediate code.

FLOWTRAN demonstrated the practicality and usefulness of an environment using on GNSDs, and provided a vehicle for experimenting with design alternatives. The practical experience gained in the use of FLOWTRAN was invaluable in enabling the author to decide subsequently on which facilities should form part of a good software production tool. This practical experience is considered to have been particularly important in the human interface design, and has hopefully lead to a well engineered human interface.

The prototype was used by the author and three colleagues on projects of varying size and complexity. Two of the software systems developed using the prototype are discussed below. A short description of the function of each of these systems is given in order to provide the necessary perspective in evaluating any potential improvement in productivity.

File transfer system.

The file transfer system was the first meaningful project on which FLOWTRAN was used, and was undertaken at the stage when only the most rudimentary functions (a simple editor and code generator) had been implemented. The file transfer system was designed to facilitate the transfer of files between two computers over an asynchronous communication line. The system provided

the necessary protocol for blocked data transfer with error detection and retransmission on block checksum error. The system also provided the facility to specify the name of the file on the receiving computer system from the transmitting computer, and to be able to return messages regarding errors made in the creation and naming of the files.

The file transfer system consisted of 30 subroutines, 22 of which were produced using FLOWTRAN. The average length of each subroutine was 40 executable FORTRAN statements and the project was completed in six weeks.

In spite of the limited facilities provided by FLOWTRAN at that stage, and in spite of the numerous bugs which had to be 'worked around', FLOWTRAN was regarded as beneficial by the user who commented

'FLOWTRAN forced a higher level of thinking, encouraging the development of the system by functional decomposition. Design, implementation and debugging were possible directly from the (hand drawn) NSDs, and the automatic generation of source code resulted in one less area in which errors could be made. FLOWTRAN lacked many of the facilities which are provided in the new FLOW environment, in particular the variable and interface checking and trace facilities, and the ability to easily modify the NSD being built. These features are considered important in making the environment a useful tool. In spite of the limitations, the use

of such a tool was still preferable to hand coding the programmes in FORTRAN.

'One limitation which was encountered, and which persists in the current FLOW environment, is that the physical size of the screen does not allow the entire content of a module to be displayed. In general, however, learning to use FLOWTRAN was very easy, especially if the user was familiar with NSDs, and the goal of a twofold productivity improvement was reached even with this primitive version.'

The success and enthusiasm inspired by this first system led to the decision to extend FLOWTRAN to include a hardcopy NSD generator (the function currently provided by FLOW-CHART), and some minor editing and code generation improvements. With this 'extended' environment, it was decided to undertake an ambitious medium sized project.

Laboratory system

The development of the laboratory system provided the first opportunity to use FLOWTRAN on a medium scale project (i.e. a project of size in excess of 6 man months). The specification of the system stated that the development was to be undertaken in FORTRAN IV. This requirement would obviously have made structured programming very difficult, if not impossible; for this reason, FLOWTRAN seemed an ideal tool.

From the outset, it was decided that FLOWTRAN was to be used exclusively for the creation of the targetted FORTRAN code, and the temptation of editing the target code with a text editor once it had been generated was to be avoided. This decision was made to ensure that flowcharts would always match the target code, since they were both generated from the same source. One consequence of this decision was that a way had to be found to 'fix' a source file (the prototype's version of the internal representation which was dumped and loaded from the disk) if it had been corrupted due to bugs in FLOWTRAN. (A common bug was that the pointers which defined the GNSD tree were often incorrect after a complex session of insertions and deletions). In retrospect, this trade-off was well worth the effort, although at the time it was felt to be a major drawback.

The system developed is currently in use at the haematology laboratory of the Groote Schuur Hospital, Cape Town. The functions performed by this system include the routing of blood specimens to the various benches within the laboratory where a variety of tests are performed, the collection and reporting of test results, and the provision of laboratory management information.

This system was implemented on a mini-computer, and many practical constraints which were imposed by the hardware had to be taken into account. As an example, the operating system of the mini-computer allowed a maximum of two independent programmes to operate concurrently (known as the 'background' and 'foreground'

programmes), although each of these programmes could consist of multiple concurrent tasks.

In the system developed, the foreground programme was used to control the on-line collection of data from three instruments, five CRT terminals and eleven special purpose terminals. This requirement and the lack of an un-pended I/O facility resulted in a design whereby each I/O device (instrument or terminal) was controlled by an independent task. This in turn made programming fairly complex, since, in order to provide record access and locking facilities for these multiple active tasks, a 'data base' task was required (the system having being implemented without the luxury of a data base management system). Thus the foreground programme had to contend with all of the problems associated with multiple concurrent intercommunicating tasks.

The background programme which runs in parallel with the 'online system' in the foreground was the next most difficult component. This programme is used to examine a multiple priority 'request queue' created by the foreground programme and a print-out of the appropriate requests for reports is produced.

The system currently consists of 15 programmes (including utilities), the most complex of these programmes (the 'foreground programme' described above) comprising 126 subroutines and utilising 31 overlays with overlays containing multiple subroutines. The overlay-code to resident-code ratio of this programme is approx-

imately 2:1. The following table gives some statistics concerning the system:

Lab System Development

Number of programmes in system.....	15	
Number of FORTRAN IV subroutines.....	178	
Number of machine code subroutines.....	8	
Total source code lines (approx.)	16000	
Average lines/flowchart.....	53	
Average FORTRAN IV lines generated/flowchart...	96	
lines entered manually.....	47	(49%)
comments generated.....	35	(36%)
control structures generated.....	14	(15%)
Systems analysis and functional specification...	3	man months
Programme design, coding and testing.....	7	man months
Lines/man month	2571	

The aspects of FLOWTRAN which are regarded as having contributed most towards the productivity attained are

- enforced structured programming
- the 'interface lookup facility' (in FLOWTRAN, the call arguments were displayed for references purposes whenever a subroutine was called, although no checking was performed).
- tracing (the prototype tracing was the equivalent of 'course tracing' described in chapter 8).
- the stub generation facility.

The use of FLOWTRAN on this project produced a number of surprises. The first surprise was encountered while entering programmes in NSD form at the terminal. It was noted that a large amount of concentration was needed to build GNSDs interactively, and that the process seemed very tedious. For example, it was found that the creation of each new GNSD construct was carefully considered as compared to the insertion a keyword such as 'IF' using a traditional text editor. This phenomenon can only be ascribed to the fact that the graphical representation of programmes makes a clear distinction between the control structure and the body of the programme, a distinction which is lost in the text of a traditional programme. The consequence of this distinction highlighted by the GNSD diagram is that the issuing of commands which markedly affect the picture (or in reality the flow of control of the programme) is not performed without the user first reflecting on its effect.

The second surprise was that an almost unbelievably small number of compilation errors (i.e. errors detected by the FORTRAN compiler) were encountered after each attempt at entering or updating the GNSDs, in spite of there being no syntax checker provided by the prototype. This fact was only brought home when a number of 'quick' programmes were written directly in FORTRAN on another system, where approximately one in ten insertion or modification attempts on a module resulted a compilation an error. The low error rate and the amount of concentration which the use of FLOWTRAN seemed to demand appear to be interrelated. That is to say, FLOWTRAN enforced care in the design and coding, and

this in turn lead to error free target code.

The third surprise was noted shortly after the start of the project after a period in which both the FORTRAN and flowchart listings were generated. It was soon realised that the flowchart listing was far easier to read than the FORTRAN code (with all its GO TOs), and that any modifications required could be easily sketched on this flowchart listing. Consequently the FORTRAN code was abandoned, and the flowchart listings were adopted as the working document for development. In effect, the target code became transparent. The practicality of a flowchart-based (or graphical) language was confirmed. At the same time, practical experience showed that a hardcopy printer is as essential a part of a programmer's workstation as the display terminal. Using flowchart listings instead of source code listing leads to a useful side effect, in that modular design was enforced, because non-modular flowcharts soon became very 'cluttered' and unintelligible.

The colleague who assisted in this project commented afterwards

'Coding a programme as a Nassi Shneiderman Diagram was found to have considerable advantages of the conventional method. The function key/menu based operations were found to be logical and easy to use. FLOWTRAN encouraged top-down programme design and stepwise refinement, and resulted in a reduction of compile time errors. In fact, thinking back, I do not recall having made any errors.'

'The major area of FLOWTRAN requiring improvement is the editor. It was sometimes easier to re-enter an entire programme rather than to modify an existing one.'

9.2 The FLOW environment

Since the complete FLOW environment has only recently been completed, very little experience has been gained in its application. One small scale project, a preventative maintenance planning system, was undertaken using the FLOW environment whilst it was still under development, and some parts of the FLOW environment, such as the environment manager (written in FORTRAN) and parts of the syntax checker (written in Pascal) have been generated using FLOWF4 and FLOWPAS.

The remarks so far has been favourable and the comments of a colleague (who developed the preventative maintenance system) sum up the current situation

'The FLOW environment as a principle:

- the best thing forcing you away from code generation towards programming while pretending to help you
- plus actually being a real helper
- looking back, a good programming teacher at the

same time

'On FLOW-EDIT:

- the first and foremost advantage is the automatic tracking of the relative call level and IF nesting; probably the naughtiest bug killer.

- hard after that is the call parameter support.

- an absolute time saver are the basic 'box' functions, with automatic 'frame' generation (even aside from the philosophy of the FLOW approach to problem solving).

'Some current problems:

- the 'inline' editor is only good enough when it looks at me and knows what to put in place. The line syntax checking is nice sometimes, but 'hackers' tend to object to 'no escape unless you comply' situations.

- definitely not a system for dumb terminals and I/O bound machines.'

10. Portability issues.

It is obviously advantageous for any software production environment to be able to operate under more than one specific hardware and/or software configuration, and perhaps more importantly, to allow development of systems targetted for more than one configuration. A number of factors affect the portability of the FLOW environment including the hardware on which the implementation is mounted, the particular dialect of the language in which the programmes are written, and the operating system used for the environment. The portability of the target code produced by the post-processors should also be considered.

The FLOW environment was designed bearing these portability factors in mind, and a general principle was adopted in order to enhance portability, namely that all facilities which could affect portability were to be localised either within separate procedures or, in the extreme case, within separate programmes. As an example, the environment manager (described in chapter 5) provides the operating system interface facilities and is not likely to be able to be ported to execute under another operating system. The environment manager is a small 'self-sufficient' programme, which communicates with the other programmes via the file system. On the other hand, the remainder of the programmes within the environment are written using standard Pascal (with exceptions noted below), and make no assumptions regarding the facilities of the operating system other than those supported

by Pascal.

10.1 Hardware.

Terminal Display Features

The characteristics of the CRT terminals used with the FLOW environment will have a bearing on the way in which GNSDs will be edited, and consequently affect the portability of FLOW-EDIT. Many of the terminals currently available provide features which may (indeed should) be used to good effect in producing a well engineered human interface. Access to all of these 'special' terminal features in FLOW-EDIT are achieved via procedure calls. Having a separate procedure for each terminal feature allows the similar features on different terminals to be supported by simply modifying the appropriate procedure. For example, the procedures **DIM** and **BRIGHT** are used to set the terminal display mode to either half or full intensity output. (FLOW-EDIT displays all user input in **BRIGHT**, and all system output in **DIM**). If a terminal supports this feature, the appropriate code can be written to make use of it. If the feature is not available, a dummy procedure may be used without affecting the operation of FLOW-EDIT. New terminal types are therefore supported by replacing terminal-specific procedures with new procedures which are functionally equivalent.

Of the various features provided by terminals, only one is essential

to the implementation of FLOW-EDIT namely, the facility for direct cursor positioning i.e. the ability to position the cursor programmatically at any co-ordinate within the display screen matrix. The resolution required for this crude form of graphics is very low (1 character by 1 character), and thus FLOW-EDIT will operate on both 'graphic' and 'non-graphic' terminals. Direct cursor positioning is required for the drawing of GNSDs, and it is also used for cursor positioning, allowing the user to identify places of interest on the screen. It should be noted that the user positions the cursor by entering text commands, which are then translated into appropriate screen addressing codes, and not by means of a cursor positioning device such as a cross hair cursor, 'mouse', light pen or bit-pad as found on some graphics terminals.

Various other features are desirable, but by no means essential, such as the full and half intensity display (mentioned above) and the ability to scroll the screen image i.e. the ability to roll the entire contents of the screen upward by one line when writing on the last line of the screen. Experience has shown that the (sequential) length, rather than the depth of nesting of a GNSD, is the factor limiting the full display on a terminal screen. This tendency away from deep nesting follows from the useful side effect which GNSDs impose on programmers in structuring systems, namely 'once a block is too heavily nested, it's about time that a procedure should be called.' In this regard, practical experience in the use of GNSDs has shown that they exhibit the properties of traditional programme

text, and, in spite of the many theories regarding elision for syntax directed editors [Wate82], the method adopted by the 'old fashioned' editors, namely scrolling, is both effective and desirable.

Terminal keyboard

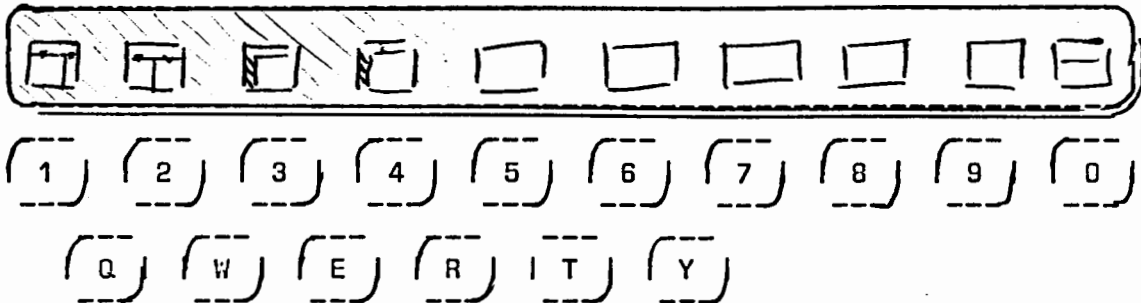
The keyboards of terminals currently available differ mainly in the number of special function keys provided, and in the way in which these keys operate. For example, some function keys generate a function key header character, followed by the function key identifier. Function keys have the advantage of providing unique keys for pre-selected actions, and definitely improve the user interface. The prototype version of FLOW-EDIT made extensive use of function keys in order to provide both key words and commands, and this was found to be a successful means of entering and editing GNSD. In order to implement the GNSD picture building commands using function keys in the current implementation, at least 10 such keys would be required. Since a great many terminals do not provide this minimum requirement, it was decided (in the interests of portability) to implement FLOW-EDIT without making use of any function keys at all. The command language was re-designed so that special keys are not required and the current implementation now requires only the minimum keyboard layout such as would be required in traditional programme editing for Pascal.

The concept of function keys was not abandoned however (for reasons described above), but rather an alternative measure was used to achieve the same effect. This was done by noting that (almost without exception) terminal keyboards provide the numerals along the top row of the keyboard. FLOW-DL was then defined such that any number entered at the beginning of a line is treated as a function key. This was possible because

- valid text lines within a 'sequence' block must be assignment statements or comments; assignment statements must begin with a variable name (which may not begin with a number), and comments must begin with the \$ comment indicator.

- valid entries within 'control' blocks (IF, WHILE etc) must be a predicates and function keys are not allowed as part of any predicate.

Thus any terminal input which begins with a numeral is treated as a 'function key' and used for GNSD picture building. Having the keys along the top row of the terminal permits a 'template' to be drawn with the appropriate symbols for the GNSD commands shown, and this template may be pasted on the terminal above the numeral keys for easy reference.

Keyboard with template

The keyboard/terminal hardware presents a second problem, namely the character set provided. FLOW-DL is defined using some 'special symbols' ([] \$ # etc.) which are not always found. In order to allow portability, these symbols have been defined as parameters within all of the FLOW programmes (FLOW-EDIT and the post-processors), and these symbols may be easily changed, if required, for use on a particular terminal.

Computer system interactiveness

The interactiveness of the host computer on which the environment is mounted (i.e. the ability of the computer to react to the depressing of terminal keys) has a marked impact on portability. Without formal classifications, three types of computer system can be identified, namely micros, minis and mainframes. The major distinction between these types of systems from the FLOW environment's point of view is the amount of interaction which is normally achieved. Typically, the implementation of a programme such as FLOW-EDIT on a micro computer (or single user) system

would allow each character to be processed immediately. Thus the editor programme executing on a micro could implement cursor movement and in-line correction by defining specific keys for the movement and editing commands. Mini computer systems (multi user), although also character oriented, would usually have entire lines read by the operating system (i.e. strings of characters followed by a terminator such as **newline** or **carriage return**), with some editing facilities (such as delete last character) being performed by the operating system. Complete lines would then be passed to the programme. Thus immediate response to keys is not possible; for example, the entry of a cursor movement command would require at least two keys, the command and the terminator, to be pressed. The terminals connected to mainframe computers usually allow the text to be entered and edited locally (i.e. the terminals usually have limited intelligence), and this text is transferred to the computer only when the **transmit** key is pressed. These differences in operation are perhaps the most difficult to overcome in porting interactive software, since the way in which the terminal operates will affect the actual design. This perhaps explains why portable interactive tools are not freely available.

Computer system address space

The micro/mini/mainframe implementation is also an issue affecting the allowable address space. Again micro and mini computer systems generally limit the address space of the application programmes to 64K bytes, whilst mainframes do not generally

impose practical limits on programme size. The architecture of the FLOW environment allows each component of the system to be a small independent programme, and these programmes have been implemented in less than 32K bytes of memory (without overlaying). Thus the address space requirement is not a major concern.

10.2 Software

Portable P4 Pascal with extensions

The FLOW environment was written using a portable P4 Pascal compiler. Although this compiler imposed many unnecessary restrictions (such as limiting text literals to a maximum of 16 characters), it did enforce a highly portable implementation. Two extensions to standard Pascal [Jens78] have been implemented in the version of the compiler used, namely the use of external procedures, and the ability to access records randomly. The extension which permits procedures to be compiled separately is merely a convenience for the developer, and it is a trivial matter to include the external procedures within the main body of each programme instead of the EXTERNAL definition.

The random access extension is, unfortunately, an essential part of the implementation of the FLOW environment programmes. According to the philosophy adopted (design for portability), the extended Pascal statements are used exclusively in a few special I/O procedures. Thus for example the only usage of

statements for random file reading from the source data base are to be found in the source read procedure defined as

```
PROCEDURE READSOURCE(REC:SOURCEREC;  
                    NUM:SRECNUM);
```

where REC is the record which is to be read, and NUM is the record number. Many implementations of READSOURCE are possible. For example, if a sufficiently large address space is available, READSOURCE could be implemented as a table look-up in place of a file access.

Data Base file handling System

The random file access facilities used in the current implementation of the FLOW environment are actually a poor substitute for some form of data base file handling, and if a data base system were available, its usage would be preferred. The programmes of the FLOW environment have in fact been written with a conceptual data base handler in mind, and if a data base file handling system (not necessarily a data base management system) is available, it could easily be accommodated.

The 'conceptual data base' forms one of three 'layers'. The first layer implements the main functions of the FLOW programmes, but does not perform any I/O. All I/O is performed by means of procedure calls from the first layer to a second layer. This second layer includes the conceptual data base access proced-

ures. Since the current implementation does not use a data base file system, all procedures in the second layer are required to simulate the data base operations by calling the third (primitive) layer. The third layer then performs the functionally equivalent (compiler specific or operating system specific) operations. All non-portable code occurs within this layer. As an example of this layered approach, the entry point record access procedure in (a second layer procedure) is defined as

```
PROCEDURE EPPACK(OPTION:DBOPT;  
                NAME:EPKEY;  
                REC:EPREC;  
                MSG:EPERMSG;  
                STATUS:EPSTATUS)
```

Where OPTION is the 'conceptual data base' operation namely, R for Read, W for write, or X for Delete. The name of the entry point is given in NAME, and the record containing all details pertaining to the entry point is given in REC. MSG is returned containing a message that can be displayed, and STATUS is a code which indicates an exception status or whether the procedure has been successfully completed

10.3 Software generated

An attempt has been made in all of the current post-processors to generate standard target code. Two aspects are considered here, namely language implementation differences, and the trace facility provided.

Language implementation differences

Where implementation differences have been identified, an attempt has been made to circumvent them. Thus, for example, the COBOL code generated for a GNSD WHILE construct is

```
IF NOT (VALUE = 100)
    PERFORM WHILE-LOOP-1 UNTIL VALUE = 100.
```

This code enforces a 'zero trip loop', as required by the GNSD, and will operate correctly irrespective of whether the code generated by the compiler executes WHILE-LOOP-1 once or not at all if VALUE = 100 on the first iteration. Many more implementation differences are likely to become apparent when the FLOW environment or the target code produced is actually implemented on different systems.

Trace facility

The trace facility provided by the FLOW environment (described in chapter 8) is fully portable, since the trace output is generated by high level (portable) source code inserted within the code generated by the post-processors. In this respect, the FLOW environment's trace facility differs from most of the other trace facilities available which are provided as an operating system or compiler function, or by means of a separate pre-processor e.g. [Scha80].

10.4 The 'non portability' issue

Although a high degree of portability of software generated by the FLOW post-processors is a desirable goal, portability should not be taken to the extreme, and implementation specific code should be allowed. In the current implementation of the FLOW environment, FLOW-DL has been extended to allow lines to be entered which will by-pass the syntax checking procedure of FLOW-EDIT. The post-processors will then output these lines without any translation (although automatic indentation is performed to maintain readability). Thus, should users require some special (and possibly non-portable) target code, they are not forced to modify the code generated. This has the advantage that, although some of the error detection mechanisms have been by-passed, the GNSD documentation will still contain the 'as implemented' code. The onus will then be upon the user (and the target language compiler) to validate the correctness of the line entered; this approach is considered preferable to the alternatives.

Bibliography

- [Jens78] Jensen K, Wirth N. **Pascal user manual and report** 2nd edition. Springer Verlag, New York 1978.
- [Scha80] Schach S.R. **A Portable Trace for the Pascal Heap.** Software Practice and Experience, Vol 10 pp. 421-426, 1980.
- [Wate82] Waters R C. **Program Editors Should not Abandon Text Orient Commands.** ACM SIGSOFT notices Vol 17 No 7 pp. 39-46, July 1983.

11. Conclusions and topics for future research.

11.1 Conclusions.

There has been a growing trend away from the use of flow-charts supported by a number of experimental studies [Shne77]. In the introduction, it was pointed out that the ability of a civil engineer to design a bridge and communicate this design by means of words alone would be seriously questioned. Yet, many software engineers do just that.

The FLOW environment has been created to allow pictures to play their rightful rôle in the production of software. That is not to say that the FLOW environment is a drastic departure from current programming environments, but merely that pictures are utilised where applicable. In most other respects, the FLOW environment was built along the lines of current software production environments, namely that a number of tools are integrated and a central data base is employed. The FLOW environment also embodies a number of methodologies such as structured programming and system development by functional decomposition and stepwise refinement.

The objective in developing the FLOW environment was to attain a reduction in development and maintenance effort by a factor of two in the production of medium to large scale software systems. Without formal measurements having been taken, a process which

CHAPTER 11. Conclusion and Future Research

would be both difficult and costly [Scha82] (since, ideally, two project teams working independently on the identical project for a period of six months or more should be monitored under controlled conditions), it has generally been accepted by all users that the objectives have been met, if not surpassed. A conclusion that can be drawn is therefore:

'A software production environment using GNSDs can result in a twofold improvement in programme development and maintenance effort'

The reason for stating that this 'a conclusion that can be drawn...', (as opposed to 'the conclusion is...') is that the above is an understatement of the real software production problem. The acuteness of this problem has been emphasised in the current work, and in a way, the above is an indictment of the software industry, stating, in effect, that major improvements can be brought about in the process of software production simply by using those tools and techniques which are currently well known. The FLOW environment is a very simple (possibly even crude) facility, but it makes use of ideas, many of which are plain old-fashioned good sense, and others which have appeared regularly in the literature in one form or another. A more realistic (albeit cynical) conclusion to the current work might therefore be

'The application of computers and current software production technology is in such an underdeveloped state that the use of a relatively simple tool, such as the FLOW environment,

CHAPTER 11. Conclusion and Future Research

may result in major improvements in the software production process'

So as not to present this conclusion as a criticism, it is possible to re-phrase it in the form of a challenge:

'The use of a relatively simple tool has resulted in major improvements in the software production process. This underlines the benefits that could be derived from the development of more sophisticated software production environments'

11.2 Topics for future research.

Having accepted the challenge, the first task should be to review the FLOW environment in order to find new directions. The current work followed two main streams

- The investigation of software production environments and their application to the software production process

- The development of graphical language constructs and extensions to conventional (written) languages which take into account the procedural nature of data structures.

Further research on programming environments.

Research into the software production process embodies a vast and highly controversial field. Symptomatic of the problem encountered in this field is the so-called 'software crisis' facing the computer industry, and the apparent inability of computer scientist, software engineers and computer practitioners (three distinct classes!) to resolve it. Software development support environments are (perhaps) part of the solution and should be given serious attention [Wass81].

Some aspects of software development using software production environments were investigated as part of this work, and many of these aspects have been described. At this stage, most of the facilities provided by the FLOW environment can and should be extended. It should be noted, for example, that the FLOW environment has been under development for almost three years with new improvements being constantly undertaken. Experience has shown that each new idea for computer assistance in software production has lead to the need for further facilities being defined.

Before discussing the theoretical aspects, some of the practical extensions to the FLOW environment which are planned or currently being implemented should be mentioned. Apart from providing some trimmings to the current environment (such as the implementation of the CASE construct) the short term extensions envisaged include

CHAPTER 11. Conclusion and Future Research

the development of post-processors for the FORTRAN 77 and C programming languages, and the development of a self contained programmer's workstation utilizing large (or multiple) display screens, and making use of alternate input devices such as voice recognition and light pen or touch sensitive screens. The development of a programmer's workstations are possible based on current commercially available hardware and require no new software technologies [Gutz81]. Another extension which is being planned is a programme verification aid which would operate in a similar manner to the trace package, and would output a 'test trail' (a list of all blocks of the NSD entered by the programme). This test trail would then be processed by a verification report programme which would give the frequencies of execution of each block presented in the form of GNSD listings (similar to that produced by FLOW-CHART). This information could be used to optimise the more frequently accessed blocks, and would also show which blocks have not been executed (in an attempt to test a programme exhaustively). It should also be fairly simple to generate automatically the predicates required to provide the additional test cases which will 'drive' the programme through untested blocks [Wass81].

As an introduction to the more theoretical aspects of the research into software production, it is interesting to attempt to place the current work in its proper perspective. The point to note is that the aspects of software production supported explicitly by the FLOW environment are only part of the total activities which constitute the software production process. This fact

CHAPTER 11. Conclusion and Future Research

can be seen from the diagram depicting the methodology of the software production process presented in chapter 4 (reproduced below). This lack of coverage of the process is typical of the shortcomings of present day software production environments [Wass82].

```

+-----+
| requirements-definition |
| top-level-system-design |
+-----+
| while software product is incomplete |
|/+-----+
|/! programme to the current level of refinement |
|/! define interfaces to next level of refinement |
|/! build intermediate system at current level of refinement
|/! test and debug system at current level of refinement
|/+-----+
|/! is emerging software product ok ? |
|/+-----\Y/-----\N/-----+
|/! document current level ! review |
|/! refine design ! backtrack |
++-----+
| produce user documentation |
+-----+

```

To recapitulate, the FLOW environment gives support to the most obvious of the activities defined in the methodology, namely

- system design
- programming
- building of intermediate system
- testing and debugging of system
- documentation

Those aspects for which support is lacking include

- requirements definition
- user documentation
- decision support

The first two aspects (requirement specifications and user documentation) would appear to be the next logical candidates for inclusion within an extension of the FLOW programming environment, with perhaps the user documentation being generated automatically from the requirements specification, in the same manner as the system documentation may be generated from the (extended) systems design.

The third aspect (decision support) requires in depth investigation and formalisation, and it is not clear how or when the computer could be of assistance. Some of the decisions shown in the diagram are

- 'is the software complete ?'
- 'is the emerging software product ok ?'
- what backtracking is required

Many more decisions are implied. It is desirable that a comprehensive programming environment assists in making these decisions. Note that providing decision support within the development process is not a problem specific to the programming environments, but rather a problem facing software producers in general; the programming environment context provides some formalisation

CHAPTER 11. Conclusion and Future Research

which should be of assistance in researching the problem, but the results of research on this topic would be widely applicable.

There are a number of other interesting aspects of the software production methodology given in the above diagram which could also be investigated. In particular, it should be noted that the programming and testing at the i -th level of refinement are completed before the design at the $i+1$ th level begins. i.e. the methodology is one of simultaneous top-down design with top-down programming and testing. (Programming here is assumed to include debugging and documentation). The problem with this stepwise refinement which should be investigated is determining the size of the increment on each iteration. That is to say, how far should design proceed before coding and testing are performed? Another related problem is deciding whether or not the iteration increment should remain constant throughout the development cycle, or should be some function of the iteration number, and if so, what should that function be. There is obviously an upper bound to the increment size function, namely completing the design before coding, and coding before testing. Experimentation varying the size of the increment could determine the optimum size of subsystems and frequency of programme 'builds' which would lead to the greatest productivity improvement (notwithstanding the problems of performing meaningful controlled experiments in software production on medium to large scale software projects as mentioned earlier).

CHAPTER 11. **Conclusion and Future Research**

A second point to note in the methodology presented in the above diagram is that, should the emerging product not be 'ok' (according to diagram), some sort of backtracking must be performed. (The backtracking activity allows for possible redesign at the current or previous levels of refinement, as well as the modification of the requirements specification). Modifications of the requirements appear to be inevitable, experience having shown that 'the customer usually knows what he wants only after he gets what he originally asked for'. In this sense, the top down approach expounded by this methodology allows a change of requirements to be made at an early stage in the development cycle, since the emerging software product can be used as a prototype or subset of the product being developed. Once again, the problem to be investigated is the amount by which 'i' is to be decremented. The size of 'i' is not defined within the review and backtracking operations and can only be optimized by experimentation. It should also be noted that process of backtracking should be undertaken in such a manner so as to ensure that the same path is not traversed on subsequent iterations of the process. This could be a problem in the development of very large scale systems where new people assigned to a job may easily make the same mistakes as their predecessors. A programming environment should provide the facility to keep track of the various 'builds' undertaken, and determine any repeated mistakes.

Thirdly, it should be noted from the methodology given in the diagram that, although no method is provided for the evaluation of the expression 'is software product ok?', one cannot substitute the expression with 'does the software product meet the requirements specification', since the latter assumes that the requirements specification is both complete and static [Swar82]. It is perhaps a major weakness of this and other software design methodologies that the evaluation of whether or not the emerging software is 'ok', is not mechanically computable. A first stage toward this evaluation is probably answering the question 'does the software meet the specification'. Software development methodologies (in all of their forms) obviously need a greater amount of formalization.

Lastly, it should be noted that the software production methodology given in the diagram may not terminate if the condition 'software product incomplete' remains true. This highlights a common problem in software management, where many software production processes observed in real world systems appear to be interminable. It would be highly desirable to find some method which would ensure that the development process stops. One possible suggestion in this regard (which could ensure termination) would be the a priori definition of the term 'complete', the term being defined as a function of elapsed time, cost, customer good-will etc. The selection of a 'completeness function' which is guaranteed to become true in finite time could also be used as the 'objective function' in the optimization of the process. Once again, it would be interesting to see the results in terms of software

productivity, in setting various termination conditions.

Future research on languages

The FLOW environment has proven the effectiveness of diagrammatic languages for definition of both procedures and data, and has introduced a number of 'new' language constructs.

In Chapter 3 the procedural nature of data structures were discussed, and some possible applications suggested. The use of procedural data structures is obviously a field for future study, especially since it shows that both the data oriented and procedure oriented schools of system design can be accommodated by the same language facility.

One suggested topic in the study of procedural data structures is determining the rôle which procedures would play in data oriented systems i.e. 'what exactly should the procedure be doing if the data structures are smart?'. It was suggested in chapter 3 that the data structures should be extended to give both a normal and an error return. There are obviously other possibilities.

Hybrid programming (i.e. data oriented programmes with procedural control) could also be investigated. Here the data structures would be implemented as 'procedures' to model the problem, and the traditional procedure would be used to control the flow

on exceptions. This type of programming would certainly be feasible. The hybrid approach suggests inter alia that the procedural part of the implementation should be viewed as a three-dimensional object with the dimensions being

- statement execution
- flow of control
- exception handling.

Two dimensional 'structured' representations (such as flowcharts) are no longer applicable to this concept, and a 'multi-layered' flowcharting technique would be necessary. A formal definition and implementation of hybrid programming would require a number of programming constructs to be defined, with three dimensional control of programme flow corresponding to a 'move to a new layer' as represented by the flowchart.

A second aspect of procedural data structures, which was mentioned but not expanded upon in the current work, is the issue of 'data types' and 'type conversion'. Type conversion functions similar to the FIX and FLOAT of FORTRAN would appear to be required to deal with problems in this regard. A requirement of type conversion functions to be researched is the ability to 'convert' entire data structures. Thus, for example, it should be possible to write programmes with statements of the form

printer { report(datafile)

where **printer** is a consume only data structure, such that
' printer { x ' results in x being printed on the
line printer

datafile is the data on mass storage from which the
report is to be generated. When datafile is used
as a producer, as above, the entire file is read.

report is a type conversion function that makes the
data file printable in a specified format.

Type conversions should also be investigated for their applicability
to defining programmes by step-wise refinement (or literally
designed by 'functional' decomposition). Functional languages
do currently exist, but these languages are not specifically
oriented to data type conversion. As a suggestion of what may
be possible, the function 'header' is defined. 'Header' convert
a report text data structure into a printed page data structure,
thus producing standard headers, page numbers etc. This function
could be used in the form

printer { header(report(datafile))

where 'printer', 'report' and 'datafile' are defined as in the
previous example.

One last observation regarding procedural data structures is that a large proportion of data structures are used in the form

c { p (where **c** is the consumer and **p** the producer)

The two procedures (for production and consumption) could easily be implemented as parallel processes which could in turn be implemented on multiple processors with a potential doubling of throughput (where the procedures **c** and **p** are non trivial). This is definitely a challenge for future hardware designers.

In summary, the FLOW environment has shown that improvements can be brought about in many aspects of the software production process. Extensions to the current system will undoubtedly lead to more efficient and effective programme production and maintenance, and, it is hoped, will also lead to the development of new tools capable of attaining that magical increase in productivity of 'an order of magnitude'.

Bibliography

- [Gutz81] Gutz S, Wasserman A I, Spier M J. **Personal Development Systems for the Professional Programmer.** IEEE COMPUTER Vol 14 No 4 pp. 45-53, April 1981.

- [Scha82] Schach S.R. **A Unified Theory for Software Production.** Software Practice and Experience, Vol 12 pp. 683-689, 1982.

- [Shne77] Shneiderman B. **Experimental Investigations of the Utility of Detailed Flowcharts in Programming.** Communications of the ACM, Vol 20 No 6 pp. 373-381, June 1977.

- [Swar82] Swartout W, Balzer R. **On the Inevitable Intertwining of Specifications and Implementation.** Communications of the ACM, Vol 25 No 7 pp. 438-440 July 1982.

- [Wass81] Wasserman A I. **Guest Editor's Introduction: Automated Development Environments.** IEEE COMPUTER (Special issue on programming environments) Vol 14 No 4 pp. 7-10, April 1981.

- [Wass82] Wasserman A I, Gutz S. **The Future of Programming.** Communications of the ACM, Vol 25 No 3 pp. 196-206 March 1982.

Bibliography

Included in this bibliography are all references made in the text of the thesis as well as other books and journal articles consulted during the course of the research.

- [Avak82] Avakan A. **The Design of an Integrated Support Software System.** ACM SIGPLAN Notices, Vol 17 No 6 pp. 308-317, June 1982.
- [Bake72] Baker F.T. **Chief programmer team management of production programming.** IBM Systems Journal, Vol 11 No. 1, 1972.
- [Bela80a] Belady L A, Evangelisti C.J, Power L.R. **Greenprint—a graphic representation of structured programmes.** IBM Systems Journal, Vol 19 No 4 pp. 542-553 Nov, 1980.
- [Bela80b] Belady L A. **Modifiability of large software systems.** IBM Research Report RC8525, August 1980.
- [Berg81] Berg G D, Gordon R D. **Tutorial:Software design strategies.** (2nd Ed) IEEE Computer Society Press, New York, 1981.
- [Boeh76] Boehm B W. **Software Engineering.** IEEE Transactions on Computers Vol 25 No 12 December, 1976.
- [Buxt80] Buxton J.N. **An informal bibliography on programming support environments** ACM SIGPLAN Notices, Vol 15 No 12 pp. 17-30, December 1980.
- [Broo80] Brooke J B, Duncan K D. **Experimental Studies of Flowchart Use at Different Stages of Program Debugging.** Ergonomics, Vol 23 No 11 pp. 1057-1091, 1980.
- [Cail82] Cailliau R. **How to Avoid Getting Schlonked by Pascal.** ACM SIGPLAN Notices, Vol 17 No 12 pp. 31-40, December, 1982.
- [Came82] Cameron J.R. **Two Pairs of Examples in the Jackson Approach to System Development.** Proceedings of 15th Annual Hawaii International Conference on System Sciences, 1982.
- [Chap74] Chapin N. **New format for flowcharts.** Software Practice and Experience, Vol 6 No 4. pp.341-357, 1974.

Bibliography

- [Chap82] Chapman D. **A Program Testing Assistant.** Communications of the ACM, Vol 25 No 9 pp. 625-634 September 1982.
- [DoD80] United States Department of Defence. **Requirements for ADA Programming Support Environments, Stoneman.** 1980.
- [Denn81] **Smart Editors - ACM President's Letter.** Communications of the ACM, Vol 24 No 8 pp. 491-493, August 1981
- [Deut81] Deutsch M S. **Software project verification and Validation.** IEEE COMPUTER, Vol 14 No 4 pp. 54-70, April 1981.
- [Dick78] Dickson H E, McGowan C L, Ross D T. **Software Design using SADT.** Structured Analysis and Design, Infotech Internation Limited, Volume 2 pp. 101-114, 1978.
- [Dist80] Distaso J R. **Software Management - A Survey of the Practice in 1980.** Proceedings of the IEEE, Vol 68 No 9 pp. 1103-1119, September 1980
- [Dijk68] Dijkstra E W. **The Go To Statement Considered Harmful.** Communications of the ACM, (Letters to the editor) Vol 11 No 3 pp. 147-148, March 1968.
- [Druf82] Druffel L E. **The Need for a Programming Discipline to Support the APSE.** ACM SIGSOFT Software Engineering Notes, Vol 7 No 34, July 1982.
- [Fitt79] Fitter M, Green T R G. **When do diagrams make good computer languages?** International Journal of Man-Machine Studies, Vol 11 pp. 235-261, 1979.
- [Fras81] Fraser C W. **Syntax-Directed Editing of General Data Structures.** ACM SIGPLAN Notices, Vol 16 No 6 pp. 17-21, June 1981.
- [Flec83] Fleckenstein W O. **Challenges in Software Development.** IEEE COMPUTER Vol 16 No 3 pp. 60-64, March 1983.
- [Glas82] Glass R L. **A Minimum Standard Software Toolset.** ACM SIGSOFT Software Engineering Notes, Vol 7 No 4 pp. 3-13, October 1982.
- [Gimp80] Gimpel J F. **Contour - a method of preparing structured flowcharts.** ACM SIGPLAN Notices, Vol 15 No 10 pp. 35-41, October 1980.
- [Good81] Good M. **Etude and the Folklore of User Interface Design.** ACM SIGPLAN Notices, Vol 16 No. 6, June 1981.

Bibliography

- [Gutz81] Gutz S, Wasserman A I, Spier M J. **Personal Development Systems for the Professional Programmer.** IEEE COMPUTER Vol 14 No 4 pp. 45-53, April 1981.
- [Heba78] Hebaiker P G, Zelles S N. **Tell - A system for graphically representing software designs.** Research Report RJ2351, IBM Research Laboratory, San Jose, September 1978.
- [Houg83] Houghton R C. **Software Development Tools, a Profile.** IEEE COMPUTER Vol 16 No 5 pp. 63-70, May 1983.
- [Howd81] Howden W. **Contemporary software development environments.** ACM SIGSOFT Software engineering notes, Vol 6 No 4 pp. 6-15, August 1981.
- [Huan80] Huang J C **Instrumenting Programs for Symbolic-Trace Generation.** IEEE COMPUTER Vol 13 No 12. pp 17-23, December 1980.
- [Ivie77] Ivie E L. **The Programmer's Workbench - A Machine for Software Development.** Communications of the ACM, Vol 20 No 10 pp. 746-753, October 1977.
- [Jack75] Jackson M A. **Principles of program Design.** Academic Press, 1975.
- [Jack76] Jackson M A. **Constructive Methods of Programme Design.** Lecture Notes in Computer Science, Vol 44, Springer Verlag Inc, New York, 1976.
- [Jack78] Jackson M A. **Information systems: Modelling, Sequencing and Transformations.** Preceedings of the Third International conference on Software Engineering, May 1978.
- [Jens78] Jensen K, Wirth N. **Pascal user manual and report** 2nd edition. Springer Verlag, New York 1978.
- [Jens79] Jensen R W, Tonies C C. **Software Engineering.** Prentice Hall, Englewood Cliffs, New Jersey, 1979.
- [John80] Johnson S C. **Language Development Tools on the UNIX System.** IEEE COMPUTER Vol 13 No 8 pp. 16-24, August 1980.
- [Kern78] Kernighan B W, Ritchie D M. **The C programming language.** Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1978.
- [Kern81] Kernighan B, Mashey J R. **The UNIX programming environment.** IEEE COMPUTER, Vol 14 No 4 pp. 12-22, April 1981.
- [Kim83] Kim K H. **A Look at Japan's Development of Software Engineering Technology.** IEEE COMPUTER Vol 16 No 5 pp. 26-37, May 1983.

Bibliography

- [Ledg80] Ledgard H.F. **COBOL Under Control**. Communications of the ACM, Vol 19 No 11 pp. 601-608 November 1976.
- [Lee80] Lee G, et al. **Fortran Programming Standards**. ACM SIGPLAN Notices, Vol 15 No 2 pp. 51-63, February 1980.
- [Lega80] Legard H.F. **A Human engineered variant of BNF**. ACM SIGPLAN Notices, Vol 15 No 10 pp. 57-62, October 1980.
- [Lern82] Lerner E J **Automating programming**. IEEE SPECTRUM Vol 19 No 8 pp. 34-38, August 1982.
- [Lewi76] Lewis P M, Rosenkrantz D J, Stearns R E **Compiler Design Theory**. Addison Wesley Publishing Company Inc. Phillipines 1976.
- [Mike81] Mikelson M. **Prettyprinting in an Interactive programming environment**. ACM SIGPLAN Notices, Vol 16 No 6 pp. 106-116, June 1981.
- [Morr81] Morris M.J., Schwartz M D. **The design of a Language directed editor for Block structured languages**. ACM SIGPLAN Notices, Vol 16 No 6 pp. 28-33, June 1981.
- [Nass73] Nassi I, Shneiderman B. **Flowchart Techniques for Structured Programs**. ACM SIGPLAN Notices, Vol 8 No. 8 pp. 12-26, August 1973.
- [Newm79] Newman W.M., Sproul R.F. **Principles of Interactive Computer Graphics**. [2nd Edition] McGraw-Hill, New York, 1979.
- [Ng78] Ng N. **A Graphical Editor for Programming Using Structured Charts**. Research Report RJ2344, IBM Research Laboratory, San Jose, September 1978.
- [Orr77] Orr K T. **Using Structured System Design**. Structured System Development, Yourdon Press, pp. 81-106, 1977.
- [Orr78] Orr K T. **Introducing Structured System Design**. Structured Analysis and Design, Infotech Internation Limited, Volume 2 pp. 285-305, 1978.
- [Oste81] Oseterweil L. **Software Environment Research: Directions for the Next Five Years**. IEEE COMPUTER, Vol 14 No 4 pp. 35-43, April 1981.
- [Pong80] Pong M.C. **A System for Programming with Interactive Graphic Support**. MPhil thesis, Univ Hong Kong 1980.
- [Raja82] Rajaraman M K. **A Charecterization of Software Design Tools**. ACM SIGSOFT Software Engineering Notes, Vol 7 No 4 pp. 14-17, October 1982.

Bibliography

- [Ritc74] Ritchie D M, Thompson K. **The UNIX Time Sharing System.** Communications of the ACM, Vol 17 No 7 pp. 365-375 July 1974.
- [Rohl75] Rohl J S **An Introduction to Compiler Writing.** American Elsevier Publishing Co. Inc, New York, 1975.
- [Ross77] Ross D T. **Sturctured Analysis (SA): A language for Communicating Ideas.** IEEE Transactions on Software Engineering, Vol 3 No 1 pp. 16-34, January 1977.
- [Roy76] Roy P, St Dennis R. **Linear Flowchart Generator for a Structured Language.** ACM SIGPLAN Notices, Vol 11 No. 11 pp. 58-64, November 1976.
- [Rubi83] Rubin L F. **Syntax-directed Pretty Printing - A Fisrt Step Towards a Syntax Directed Editor.** IEEE Transaction on Software Engineering Vol 9 No 2 pp. 119-127, March 1983.
- [Scha80] Schach S.R. **A Portable Trace for the Pascal Heap.** Software Practice and Experience, Vol 10 pp. 421-426, 1980.
- [Scha82] Schach S.R. **A Unified Theory for Software Production.** Software Practice and Experience, Vol 12 pp. 683-689, 1982.
- [Shap81] Shapiro E, et al. **Pases:a Programming Environment for PASCAL.** ACM SIGPLAN Notices, Vol 16 No 8 pp. 50-57, Aug 1981.
- [Shne77] Shneiderman B. **Experimental Investigations of the Utility of Deatiled Flowcharts in Programming.** Comm-unications of the ACM, Vol 20 No 6 pp. 373-381, June 1977.
- [Shne82] Shneiderman B. **Control Flow and Data Structure Document-ation: Two experiments.** Communications of the ACM, Vol 25 No 1 pp. 55-63, January 1982.
- [Snow78] Snowdon R A, Henderson P. **The TOPD System for Computer Aided System Development.** Stuctured Analysis and Design, Infotech Internation Limited, Volume 2 pp. 285-305, 1978.
- [Stan81] Standish T. **Advanced Development Support Systems.** Software engineering notes ACM SIGSOFT. Vol 6 No 4 Aug 1982.
- [Swar82] Swartout W, Balzer R. **On the Inevitable Intertwining of Specifications and Implementation.** Communications of the ACM, Vol 25 No 7 pp. 438-440 July 1982.

Bibliography

- [Teic77] Teichroew D, Hershey E A. **PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems.** IEEE Transactions on Software Engineering, Vol 3 No 1 pp. 41-48, January 1977.
- [Teit81a] Teitelman W, Masinter L. **The Interlisp Programming Environment.** IEEE COMPUTER, Vol 14 No 4 pp. 25-33, April 1981.
- [Teit81b] Teitelman T. **The Why and Wherefore of the CORNELL Program Synthesizer.** ACM SIGPLAN Notices, June 1981.
- [Teit81c] Teitelman T, Reps T. **The CORNELL program synthesizer.** Communications of the ACM, Vol 24 No 9 pp. 563-573, Sept 1981.
- [Thay80] Thayer R H, Pyster A, Wood R C. **The Challenge of Software Engineering Project Management.** IEEE COMPUTER Vol 13 No 8 pp. 51-61, August 1980.
- [VanL76] Van Leer P. **Top-down Development using a Program Design Language.** IBM Systems Journal, Vol 15 No 2 pp. 155-170, 1976.
- [Van083] Van Oost E M J C. **DUIF: A Data-Oriented Flowchart Environment.** ACM SIGPLAN Notices, Vol 18 No 2 pp. 69-75, February 1983.
- [Walk81] Walker J H. **The Document Editor: A Support Environment for Preparing Technical Documents.** ACM SIGPLAN Notices, Vol 16 No 6 pp. 44-55, June 1981.
- [Wasn80] Wasner N R. **A FORTRAN Preprocessor for the Large Program Environment.** ACM SIGPLAN Notices, Vol 15 No 10 pp. 92-103, December 1980.
- [Wass81] Wasserman A I. **Guest Editor's Introduction: Automated Development Environments.** IEEE COMPUTER (Special issue on programming environments) Vol 14 No 4 pp. 7-10, April 1981.
- [Wass82] Wasserman A I, Gutz S. **The Future of Programming.** Communications of the ACM, Vol 25 No 3 pp. 196-206 March 1982.
- [Wate82] Waters R C. **Program Editors Should not Abandon Text Orient Commands.** ACM SIGSOFT notices Vol 17 No 7 pp. 39-46, July 1983.
- [Wilc76] Wilcox T R, et al. **The Design of a Table Driven Interactive Diagnostic Programming System.** Communications of the ACM, Vol 19 No 11 pp. 609-616, November 1976

Bibliography

- [Wirt71] Wirth N. **Program development by Stepwise Refinement.** Communications of the ACM, Vol 14 No 4 pp. 221-227 April 1977.
- [Wood81] Wood S R. **Z-The 95% program editor.** ACM SIGPLAN Notices, Vol 16 No 6 pp. 1-7, June 1981.
- [Work83] Workman D A. **GRASP: A Software Development System using D-Charts.** Software proactice and experience Vol 13 pp. 17-32, 1983
- [Your75] Yourdon E. **Techniques for Program Structure and Design.** Prentice Hall, Engelwood Cliffs New Jersey, 1975.
- [Zelk81] Zelkowitz M. **High Level Language Programming Environments.** ACM SIGSOFT Software Engineering Notes, Vol 16 No 4 pp. 36-43, August 1981.

Appendix A

FLOW-DL Grammar

The following is the combined language and edit command LL(1) grammar for FLOW-DL. [NAME] denotes a single key named NAME, and \$ is used in place of 'epsilon'.

FLOW programme

Data definition

```
<data-def>    ->  [IF-THEN-ELSE]<variable><data-def>[END]
                <data-def>[END]
<data-def>    ->  [ITERATE]<variable> = <iconst> , <iconst>
                <data-def>[END]
<data-def>    ->  <variable> { <data-type><data-def>
<data-def>    ->  $
```

Data type

```
<data-type>   ->  I <integer>
<data-type>   ->  F <integer> . <integer>
<data-type>   ->  A <integer>
<data-type>   ->  L
```

Appendix A

Procedure Definition

<p-block>	->	[WHILE] <boolean-expr><p-block> [END]
<p-block>	->	[IF-THEN] <boolean-expr><p-block> [END]
<p-block>	->	[IF-THEN-ELSE] <boolean-expr><p-block> [END] <p-block> [END]
<p-block>	->	[ITERATE] <variable> = <variconst> , <variconst> <p-block> [END]
<p-block>	->	[PROC-CALL] <procname><proc-params><p-block>
<p-block>	->	[INPUT] <iochan> / <var-list> / <format> <p-block>
<p-block>	->	[OUTPUT] <iochan> / <var-list> / <format> <p-block>
<p-block>	->	<variable> { <expression><p-block>
<p-block>	->	\$

Variables

<var-list>	->	<variable><varl-2>
<varl-2>	->	, <varlist>
<varl-2>	->	\$
<variable>	->	[Alpha] <var-2>
<var-2>	->	[Alpha] <var-2>
<var-2>	->	[Numeric] <var-2>
<var-2>	->	[<subscriptlist>]
<var-2 >	->	\$

Subscript list

<sublist>	->	<subscript><subs-2>
<subscript>	->	<variconst>
<subscript>	->	[<expression>]
<subs-2>	->	, <sublist>
<subs2>	->	\$

Appendix A

Procedure Names

<procname> -> **[Alpha]**<proc-2>
<proc-2> -> **[Alpha]**<proc-2>
<proc-2> -> **[Numeric]**<proc-2>
<proc-2> -> \$

Procedure Parameters

<proc-parm> -> / <vclist>
<proc-parm> -> \$

Integer constants

<iconstant> -> **[Numeric]**<iconst-2>
<iconst-2> -> **[Numeric]**<iconst-2>
<iconst-2> -> \$

String constants

<sconstant> -> " <sconst-2> "
<sconst-2> -> **[Character]**<sconst-2>
<sconst-2> -> \$

Variable or integer constant list

<vk-list> -> <variconst><vkl-2>
<vkl-2> -> , <vklist>
<vkl-2> -> \$
<variconst> -> <variable>
<variconst> -> <iconst>

Appendix A

Real constants

<constant> -> <iconstant><fract>
<fract> -> . <fconstant>
<fract> -> \$
<fconstant> -> <iconstant>
<fconstant> -> \$

Variable or constant list

<vc-list> -> <varconst><vcl-2>
<vcl-2> -> , <vclist>
<vcl-2> -> \$
<varconst> -> <variable>
<varconst> -> <constant>
<varconst> -> <sconstant>

Boolean Expressions

<b-expr> -> <be-term><be-list>
<be-list> -> **AND** <be-term><be-list>
<be-list> -> **OR** <be-term><be-list>
<be-list> -> \$
<be-term> -> <be-primary><be-tlist>
<be-tlist> -> = <be-primary><be-tlist>
<be-tlist> -> <> <be-primary><be-tlist>
<be-tlist> -> <= <be-primary><be-tlist>
<be-tlist> -> >= <be-primary><be-tlist>
<be-tlist> -> > <be-primary><be-tlist>
<be-tlist> -> < <be-primary><be-tlist>
<be-tlist> -> \$

Appendix A

<be-primary> -> (<b-expr>)
<be-primary> -> <varconst>
<be-primary> -> NOT (<b-expr>)
<be-primary> -> NOT <varconst>
<be-primary> -> (<expression>)

Expressions

<expression> -> <term><e-list>
<expression> -> + <term><e-list>
<expression> -> - <term><e-list>
<e-list> -> + <term><e-list>
<e-list> -> - <term><e-list>
<e-list> -> \$
<term> -> <factor><t-list>
<t-list> -> * <factor><t-list>
<t-list> -> / <factor><t-list>
<t-list> -> \$
<factor> -> <primary><f-list>
<f-list> -> ^ <primary><f-list>
<f-list> -> \$
<primary> -> (<expression>)
<primary> -> varconst

Appendix A

Edit Commands

<edit-cmd>	->	[MOVE] <direction>
<edit-cmd>	->	[DISPLAY] <d-qualifier>
<edit-cmd>	->	[CLIP] <optional-num>
<edit-cmd>	->	[GLUE] <optional-num>
<edit-cmd>	->	[MAKE-SPACE] <optional-number>
<edit-cmd>	->	[EXIT]
<direction>	->	U <optional-num>
<direction>	->	D <optional-num>
<direction>	->	L <optional-num>
<direction>	->	R <optional-num>
<d-qualifier>	->	>
<d-qualifier>	->	<
<d-qualifier>	->	F <proc-qual>
<d-qualifier>	->	I <proc-qual>
<d-qualifier>	->	D <proc-qual>
<d-qualifier>	->	P <proc-qual>
<d-qualifier>	->	\$
<proc-qual>	->	/ <procname>
<proc-qual>	->	\$
<optional-num>	->	<iconst>
<optional-num>	->	\$

Appendix B

The example given in Chapter 4 was entered using FLOW-EDIT and documentation and source code produced by the post-processors.

B.1 Documentation produced by FLOW-CHART.

FLOW (REV 0.0)

NAME : QCPLOT

TYPE : MAIN ROUTINE

REFERENCES

CALLS: A00INI
 A10FIN
 Z07MNU

 CALLED BY: NONE

FUNCTIONAL SPEC

FUNCTIONAL DESCR

QCPLOT IS A LABORATORY QUALITY CONTROL SYSTEM
PLOT UTILITY. QC ON DIFFERENT ANALYTICAL METHODS
IS ACHIEVED BY ANALYSING SUBSTANCES WITH KNOWN
PARAMETERS. THE RESULTS OF THESE 'CONTROLS'
ARE PLOTTED ON A SHEWART CHART WITH THE CONTROL
LINES DRAWN FOR THE MEAN +- 2 STANDARD DEVIATIONS.
STATISTICS ARE PROVIDED BY THE MANUFACTURERS OF THE
CONTROL SUBSTANCES

INPUT DESCR

A MENU SELECTION IS ENTERED FROM THE KEYBOARD AS FOLLOWS
* 1 - PLOT A SINGLE CONTROL CHART
* 2 - PLOT ALL CHARTS
* 3 - EXIT

OUTPUT DESCR

ONE OR MORE SHEWART CHARTS ARE PRODUCED DEPENDING
ON WHICH OPTION IS SELECTED FROM THE MENU

Appendix B

B.2 FORTRAN-IV source code produced by FLOW-F4.

```

C*   PROGRAM:QCPLLOT

      INCLUDE "SYS.PD"
      INCLUDE "XXX.PD"

C   *****
C   *
C   *   PRODUCT REVISION 0.0   DATE:XX/XX/XX   BLAME -> J.D.   *
C   *
C   *****

C*
C*   FUNCTIONAL SPEC
C*   -----
C*
C*       FUNCTIONAL DESCR
C*       -----
C*
C*           QCPLLOT IS A LABORATORY QUALITY CONTROL SYSTEM
C*           PLOT UTILITY. QC ON DIFFERENT ANALYTICAL METHODS
C*           IS ACHEIVED BY ANALYSING SUBSTANCES WITH KNOWN
C*           PARAMETERS. THE RESULTS OF THESE 'CONTROLS'
C*           ARE PLOTTED ON A SHEWART CHART WITH THE CONTROL
C*           LINES DRAWN FOR THE MEAN +- 2 STANDARD DEVIATIONS.
C*           STATISITICS ARE PROVIDED BY THE MANUFACTURERS OF THE
C*           CONTROL SUBSTANCES
C*
C*
C*
C*       INPUT DESCR
C*       -----
C*
C*           A MENU SELECTION IS ENTERED FROM THE KEYBOARD AS FOLLOWS
C*           *           1 - PLOT A SINGLE CONTROL CHART
C*           *           2 - PLOT ALL CHARTS
C*           *           3 - EXIT
C*
C*
C*
C*       OUTPUT DESCR
C*       -----
C*
C*           ONE OR MORE SHEWART CHARTS ARE PRODUCED DEPENDING
C*           ON WHICH OPTION IS SELECTED FROM THE MENU
C*
C*
C*
C*
C*   DATA DEFN
C*   -----
C*
C*   $$$
C*   $ G L O B A L   D A T A
C*   $$$
C*
C*   INCLUDE "IONAMES"
C*   INCLUDE "QCDEFN"
C*   INCLUDE "QCRESULT"
C*
C*   $$$
C*   $ L O C A L   D A T A
C*   $$$
C*
C*   INTEGER CHOICE ;  I3
C*   INTEGER ERROR  ;  I1
C*   INTEGER NPLOT  ;  I2

```

Appendix B

```

C*   $$$
C*   $ INITIALISE
C*   $$$
C*   CALL A00INIT (ERROR)
C-   IF ERROR = 0
C-   --
C-   IF (.NOT.(ERROR .EQ. 0)) GOTO 5
C*   $$$
C*   $ PRESENT THE USER WITH THE SYSTEM IDENTIFICATION
C*   $ USING A GENERAL PURPOSE DISPLAY UTILITY
C*   $$$
C-   WHILE CHOICE <> 3
C-   -----
15   IF (.NOT.(CHOICE .NE. 3)) GOTO 20
C*   $$$
C*   $ DISPLAY MENU GIVING OPTIONS
C*   $           1 - PLOT SINGLE CONTROL CHART
C*   $           2 - PLOT ALL CONTROLS
C*   $           3 - EXIT
C*   $$$
C*   CALL Z07MNU (QCP,INCH,OUCH,CHOICE,2)
C-   IF CHOICE <= 2
C-   --
C-   IF (.NOT.(CHOICE .LE. 2)) GOTO 25
C-   IF CHOICE = 1
C-   --
C-   IF (.NOT.(CHOICE .EQ. 1)) GOTO 35
C*   $$$
C*   $ GET PLOT NUMBER
C*   $$$
C*   GOTO 40
C-   ELSE
C-   -----
35   CONTINUE
C-   ITERATE NPLOT = 1,MAXPLOTS
C-   -----
C*   DO 45 NPLOT = 1,MAXPLOTS
C*   $$$
C*   $ PRODUCE PLOT NUMBER 'NPLOT'
C*   $$$
45   CONTINUE
C-   END LOOP
C-   -----
C-   ENDIF
C-   -----
40   CONTINUE
C-   ENDIF
C-   -----
25   CONTINUE
C*   GOTO 15
20   CONTINUE
C-   ENDWHILE
C-   -----
C-   ENDIF
C-   -----
5   CONTINUE
C*   $$$
C*   $ FINALISE
C*   $$$
C*   CALL A10FIN
C*   E N D

```

Appendix B

B.3 Pascal source code produced by FLOW-PAS.

```

PROGRAM QCPLLOT (INPUT,OUTPUT);

    (*****
    (*
    (* PRODUCT REVISION 0.0 DATE:25/10/83 BLAME -> JD
    (*
    (*****

    (* FUNCTIONAL DESCR *)
    (* ----- *)
    (* QCPLLOT IS A LABORATORY QUALITY CONTROL SYSTEM *)
    (* PLOT UTILITY. QC ON DIFFERENT ANALYTICAL METHODS *)
    (* IS ACHIEVED BY ANALYSING SUBSTANCES WITH KNOWN *)
    (* PARAMETERS. THE RESULTS OF THESE 'CONTROLS' *)
    (* ARE PLOTTED ON A SHEWART CHART WITH THE CONTROL *)
    (* LINES DRAWN FOR THE MEAN +- 2 STANDARD DEVIATIONS. *)
    (* STATISTICS ARE PROVIDED BY THE MANUFACTURERS OF THE *)
    (* CONTROL SUBSTANCES *)
    (* *)

    (* INPUT DESCR *)
    (* ----- *)
    (* A MENU SELECTION IS ENTERED FROM THE KEYBOARD AS FOLLOWS *)
    (* * 1 - PLOT A SINGLE CONTROL CHART *)
    (* * 2 - PLOT ALL CHARTS *)
    (* * 3 - EXIT *)
    (* *)

    (* OUTPUT DESCR *)
    (* ----- *)
    (* ONE OR MORE SHEWART CHARTS ARE PRODUCED DEPENDING *)
    (* ON WHICH OPTION IS SELECTED FROM THE MENU *)
    (* *)

    (* DATA DEFN *)
    (* ----- *)

VAR

    (* $$ *)
    (* GLOBAL DATA *)
    (* $$ *)
    INCLUDE IONAMES;
    INCLUDE QCNAME;
    INCLUDE QCRESULT;

    (* $$ *)
    (* LOCAL DATA *)
    (* $$ *)

    CHOICE : INTEGER (* I3 *) ;
    ERROR : INTEGER (* I1 *) ;
    NPLOT : INTEGER (* I2 *) ;

    INCLUDE A00INI;
    INCLUDE A10FIN;
    INCLUDE Z07MNU;
    
```

Appendix B

```

(* PROCEDURE *)
(* ----- *)
BEGIN
(* $$$ *)
(* $ INITIALISE *)
(* $$$ *)
  A00INIT (ERROR);
  IF ERROR = 0 THEN
    BEGIN
(* $$$ *)
(* $ PRESENT THE USER WITH THE SYSTEM IDENTIFICATION *)
(* $ USING A GENERAL PURPOSE DISPLAY UTILITY *)
(* $$$ *)
      WHILE CHOICE <> 3 DO BEGIN
(* $$$ *)
(* $ DISPLAY MENU GIVING OPTIONS *)
(* $           1 - PLOT SINGLE CONTROL CHART *)
(* $           2 - PLOT ALL CONTROLS *)
(* $           3 - EXIT *)
(* $$$ *)
          Z07MNU (QCP,INCH,OUCH,CHOICE,2);
          IF CHOICE <= 2 THEN
            BEGIN
              IF CHOICE = 1 THEN
                BEGIN
(* $$$ *)
(* $ GET PLOT NUMBER *)
(* $$$ *)
                  END
                ELSE
                  BEGIN
(* $$$ *)
(* $ PRODUCE PLOT NUMBER 'NLOT' *)
(* $$$ *)
                      END; (*LOOP*)
                    END;
                  END;
                (*ENDIF*)
              END;
            (*ENDIF*)
          END;
        (*WHILE*)
      END;
    (*ENDIF*)
  (* $$$ *)
  (* $ FINALISE *)
  (* $$$ *)
  A10FIN ;
END. (*QCPLT *)

```

Appendix B

B.4 COBOL source code produced by FLOW-COB.

```

/*****
*
*   PRODUCT REVISION 0.0   DATE:XX/XX/XX   BLAME -> J.D.   *
*
*****/

IDENTIFICATION DIVISION.
*****

PROGRAM-ID. QCPLLOT.

AUTHOR. FLOWCOB.
DATE-WRITTEN. 01 10 1983.

*
*   FUNCTIONAL SPEC
*   -----
*
*   FUNCTIONAL DESCR
*   -----
*
*       QCPLLOT IS A LABORATORY QUALITY CONTROL SYSTEM
*       PLOT UTILITY. QC ON DIFFERENT ANALYTICAL METHODS
*       IS ACHEIVED BY ANALYSING SUBSTANCES WITH KNOWN
*       PARAMETERS. THE RESULTS OF THESE 'CONTROLS'
*       ARE PLOTTED ON A SHEWART CHART WITH THE CONTROL
*       LINES DRAWN FOR THE MEAN +- 2 STANDARD DEVIATIONS.
*       STATISITICS ARE PROVIDED BY THE MANUFACTURERS OF THE
*       CONTROL SUBSTANCES
*
*
*
*   INPUT DESCR
*   -----
*
*       A MENU SELECTION IS ENTERED FROM THE KEYBOARD AS FOLLOWS
*
*           1 - PLOT A SINGLE CONTROL CHART
*           2 - PLOT ALL CHARTS
*           3 - EXIT
*
*
*
*   OUTPUT DESCR
*   -----
*
*       ONE OR MORE SHEWART CHARTS ARE PRODUCED DEPENDING
*       ON WHICH OPTION IS SELECTED FROM THE MENU
*
*

ENVIRONMENT DIVISION.
*****

***
CONFIGURATION SECTION.
***
SOURCE-COMPUTER. NOVA.
DESTINATION-COMPUTER. NOVA.

***
INPUT-OUTPUT SECTION.
***
FILE-CONTROL.
COPY "IOSECTION".

```

Appendix B

DATA DIVISION.

FILE SECTION.

COPY "FILESECTION".

WORKING STORAGE SECTION.

* \$\$\$
* \$ G L O B A L D A T A
* \$\$\$

COPY "IONAMES".
COPY "QCDEFN".
COPY "QCRESULT".

* \$\$\$
* \$ L O C A L D A T A
* \$\$\$

01 CHOICE PIC 9(3) .
01 ERROR PIC 9(1) .
01 NPLOT PIC 9(2) .

Appendix B

PROCEDURE DIVISION.

* QCPLOT
* *****

```

*   $$$
*   $ INITIALISE
*   $$$
*   CALL A00INIT USING ERROR .
*   IF ERROR = 0
*       $$$
*       $ PRESENT THE USER WITH THE SYSTEM IDENTIFICATION
*       $ USING A GENERAL PURPOSE DISPLAY UTILITY
*       $$$
*       IF CHOICE NOT = 3
*           PERFORM WHILE-LOOP-5 UNTIL NOT (CHOICE NOT = 3)
*       ELSE
*           NEXT SENTENCE
*   END IF
*   ELSE
*       NEXT SENTENCE.
*   END IF
*   $$$
*   $ FINALISE
*   $$$
*   CALL A10FIN
*   STOP RUN.

```

* WHILE-LOOP-5.
* *****

```

*   $$$
*   $ DISPLAY MENU GIVING OPTIONS
*   $       1 - PLOT SINGLE CONTROL CHART
*   $       2 - PLOT ALL CONTROLS
*   $       3 - EXIT
*   $$$
*   CALL Z07MNU USING QCP,INCH,OUCH,CHOICE,2 .
*   IF CHOICE NOT > 2
*       IF CHOICE = 1
*           $$$
*           $ GET PLOT NUMBER
*           $$$
*       ELSE
*           PERFORM WHILE-LOOP-10 VARYING NPLOT FROM 1 BY 1 UNTIL NPLOT > MAXPLOTS
*       END IF
*   ELSE
*       NEXT SENTENCE.
*   END IF

```

* WHILE-LOOP-10.
* *****

```

*   $$$
*   $ PRODUCE PLOT NUMBER 'NPLOT'
*   $$$

```