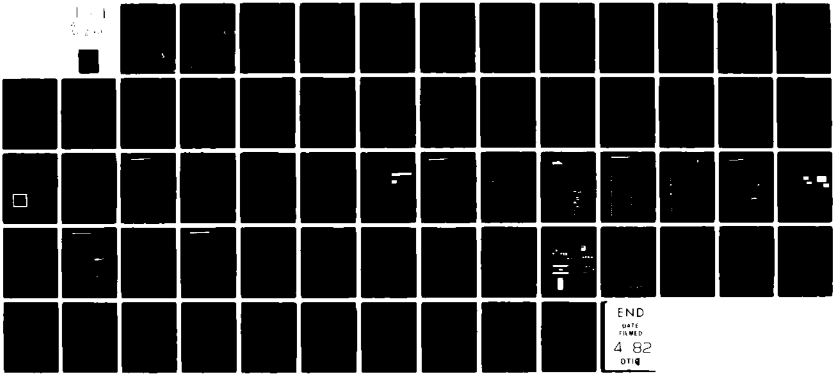
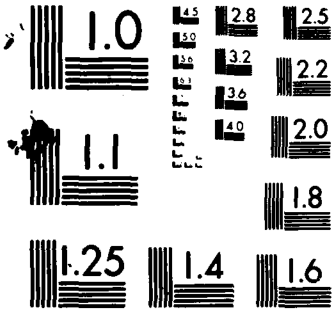
AD-A112 915 COMPUTER CORP OF AMERICA CAMBRIDGE MA F/G 5/8
DESIGN FOR A PROGRAM VISUALIZATION SYSTEM.(U)
JAN 81 C F HEROT, R CARLING, M FRIEDELL N00014-80-C-0683
UNCLASSIFIED CCA-81-04 NL



END
DATE
FILMED
4 82
DTIC


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

Design for a Program Visualization System

4

Technical Report
CCA-81-04
January 1, 1981

Christopher F. Herot
Richard Carling
Mark Friedell
David Kramlich

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)

Computer Corporation of America
675 Massachusetts Avenue
Cambridge, MA 02139

2a. REPORT SECURITY CLASSIFICATION

Unclassified

2b. GROUP

None

3. REPORT TITLE

"Design for a Program Visualization System"

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Final report

5. AUTHOR(S) (First name, middle initial, last name)

Christopher F. Herot, Richard Carling, Mark Friedell, David Kramlich

6. REPORT DATE

1 January 1981

7a. TOTAL NO. OF PAGES

7b. NO. OF REFS

8a. CONTRACT OR GRANT NO.

N00014-80-C-0683

b. PROJECT NO.

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

None

10. DISTRIBUTION STATEMENT

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES

None

12. SPONSORING MILITARY ACTIVITY

Defense Advanced Research Projects Agency/Defense Sciences Office Cybernetics Technology Division

13. ABSTRACT

This report presents the results of a study by Computer Corporation of America (CCA) to determine the feasibility of a Program Visualization (PV) system -- a system that uses computer graphics to provide lifecycle support for software development. The PV system will provide both static and dynamic (animated) views of systems to designers, programmers, and users. The goal is to facilitate their understanding of large (10**6 lines of code), complex software systems. Automated visualization of computer programs offers the opportunity to:

- Improve the productivity of the limited number of programmers who will be available to work on the increasingly large and complex software systems of the 1980's;
- Make it easier to express the mental images of the producers (writers) of programs, AND
- Facilitate communication of those mental images to the consumers (readers) of programs.

Unclassified
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Software engineering						
Software productivity						
Computer graphics						
Ada						
Programming Environments						

Unclassified
Security Classification

Design for a Program Visualization System

Christopher F. Herot
Richard Carling
Mark Friedell
David Kramlich

Technical Report
CCA-81-04

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-80-C-0683. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Department, the Office of Naval Research, or the U.S. Government.

CONTENTS

Page

1. INTRODUCTION 1

1.1 Motivation 1
1.2 The Role of Graphics 2
1.3 Approach 2
1.4 Outline of Report 3

2. A CONCEPTUAL FRAMEWORK FOR PROGRAM VISUALIZATION 3

2.1 System Requirements Diagrams 4
2.2 Program Function Diagrams 4
2.3 Program Structure Diagrams 4
2.4 Communication Protocol Diagrams 4
2.5 Composed and Typeset Program Text 4
2.6 Program Comments and Commentaries 5
2.7 Diagrams of Flow of Control 5
2.8 Diagrams of Structured Data 6
2.9 Diagrams of Persistent Data 6
2.10 Summary 6
2.11 Specifying Program Visualizations 6

3. USER INTERFACE 7

3.1 Requirements 7
3.1.1 The Visual Language 7
3.1.2 Integration 8
3.1.3 Defaults 9

3.2 User Interface Design Issues 9
3.2.1 Program Space 9
3.2.2 User Controls 9
3.2.2.1 View Specification 9
3.2.2.2 Entering Information About a Program 10
3.2.2.3 Interacting with a Program 11

4. IMPLEMENTATION APPROACH 11

4.1 The Visualizer 11
4.1.1 Interacting with the User 11
4.1.2 Monitoring Executing Code 12
4.1.3 Updating and Managing the Displays 12
4.1.3.1 Display Hardware Considerations 13

4.2 Visualization Descriptions 13
4.2.1 The Visualization Description Compiler/Linker 14
4.2.2 Picture Descriptions 16

4.3 The Visualization Description/Template Editor 18
4.3.1 Animation 19
4.3.2 Specifying Graphical Layout 19
4.3.3 Template Code 20

APPENDICES

A. PROGRAM VISUALIZATION EXAMPLES A-1

B. PROGRAM DESCRIPTION TECHNIQUES B-1

B.1 Flowcharts B-1
B.1.1 Analysis B-1

B.2 Nassi-Shneiderman Diagrams B-2
B.2.1 Analysis B-3
B.2.2 Programming Support System B-3

B.3 HIPO Charts B-4
B.3.1 Analysis B-4

B.4 GREENPRINTs B-5
B.4.1 Analysis B-5

B.5 PYGMALION B-8
B.5.1 Analysis B-8

B.6 The Mini-LOGO Animation System B-8
B.7 The Micro-PL/1 Animation System B-9
B.8 The SP/k Visualization System B-9
B.9 The Sorting Animation System B-9
B.10 CDEBUG B-10
B.11 Summary B-10

C. REFERENCES C-1

ILLUSTRATIONS

Page

4.1 Visualization Description Compilation Process 15
4.2 Picture Description Code - Line 1 17
4.3 Picture Description Code - Line 2 17
4.4 Picture Description Code - Line 3 17
4.5 Picture Description Code - Line 4 17
4.6 Picture Description Code - Line 5 17
4.7 Picture Description Code - Line 6 17
4.8 Picture Description Code - Line 7 18
4.9 Picture Description Code - Line 8 18
4.10 Picture Description Code - Line 9 18
4.11 Picture Description Code - Line 10 18
4.12 Picture Description Code - Line 11 19
4.13 Picture Description Code - Line 12 19
4.14 Picture Description Code - Line 13 19
4.15 Picture Description Code - Line 14 19
4.16 Picture Description Code - Line 15 20
4.17 Picture Description Code - Line 16 20
4.18 Picture Description Code - Line 17 20

B.1 Flowchart Symbols B-1
B.2 Process Symbol B-2
B.3 Decision Symbol B-2
B.4 Iteration Symbol B-2
B.5 BEGIN-END Symbol B-3
B.6 HIPO Components B-4
B.7 GREENPRINT Example B-5
B.8 GREENPRINT Objects B-6
B.9 Annotated GREENPRINT B-7

1. INTRODUCTION

This is the final report of a six-month study by Computer Corporation of America (CCA) to determine the feasibility of a Program Visualization (PV) system—a system that uses computer graphics to create and maintain computer software. Automated visualization of computer programs offers the opportunity to improve the productivity of the limited number of programmers who will be available to work on the increasingly large and complex software systems of the 1980s. The study focused on laying the groundwork for the development of a tool that could be replicated for use by programmers working on actual systems and applications.

The goal of PV is to facilitate the understanding of programs by people. To visualize means "to see or form a mental image of." A PV system would aid programmers in forming clear, correct mental images of the structure and function of programs. It would aid both the producers (writers) and the consumers (readers) of programs. The primary objective is to make it easier to express the mental images of the producers and to communicate those mental images to the consumers.

There is no hard and fast line dividing the writers of programs from its readers. As a program evolves, it passes through many representations and undergoes many transformations from representation to representation. A written description in some representation is read, visualized, and transformed into another representation. The reader may be a d'fferent individual than the writer, or he may be the same person returning to the program at a later time. The transformation may be a refinement into another, perhaps more detailed, language, or it may be another expression in the same language that attempts to correct an error in the previous version.

A PV system would be used by programmers charged with creating and maintaining large computer programs. The system would allow these programmers to manipulate graphical representations of programs. Such a facility would enhance their ability to *understand how the programs work, and to change* them and combine them with other programs.

It would provide a framework in which software productivity tools developed at different places could be combined in a coherent manner, making them more usable than they would be separately. These tools includ? editors and illustrators. Editors must be responsible for enforcing standards of style, consistency, clarity, and legibility in both the form and the content of the writings. Illustrators must be responsible for producing pictures, diagrams, charts, and movies that will further enhance the reader's comprehension of the intentions of the writer. A PV system could encourage (or even enforce) the use of these tools as an integral part of a system development effort. As such, it could provide a useful vehicle for their experimental evaluation.

1.1 Motivation

A program is a precise description of a process or system. Programming is the activity of expressing such descriptions. Machines execute these descriptions; that is, they obey their constituent instructions or commands to carry out a process or simulate a system.

Some programs are so simple and unimportant that they can be conceived, developed, used, and even thrown away in a single sitting at an interactive computer terminal. The art of "conversational programming" has been developed to facilitate such expression.

However, the bulk of programs upon which society relies are more complicated. They have been developed and refined by many individuals working over a period of many years. There is a growing body of programs that are sufficiently large and complex that they cannot be comprehended in their entirety by any one person.

This situation presents serious problems for both the creation and maintenance of programs. If the implementors of a large system are to ensure that it operates according to the specified requirements, they must be able to describe programs using techniques that are more powerful than reading individual lines of code.

The maintainers of a system have an even more serious problem. With the increase in the complexity of software projects—involving more programmers and longer life-cycles—normal turnover of personnel ensures that some of the original designers of a program no longer will be available by the time it is delivered to the user. Once the program is in use and maintenance is required, it is likely that the people called upon to do the maintenance will have had little to do with the original implementation.

It is imperative that techniques be developed that will allow such people to quickly understand the structure of a complex program and explore the interactions between its components. There must be a mechanism that preserves the original design goals and implementation concepts in a manner that makes them available to the maintainer and ensures their consistency with the program itself. In the absence of information like design goals, there must be a tool that allows the maintainer to probe the actual code. Such a tool would help the maintainer re-examine the original requirements, decide how they should be changed, locate the relevant code, and make the appropriate changes without introducing any untoward interactions with other parts of the system.

The techniques of PV can be useful in all the stages of a program's life cycle:

1. Listing the ***requirements*** that the program must satisfy.
2. Specifying the ***design*** of a software system, to meet the requirements.
3. Carrying out the ***coding*** of the system, following the plan of the design.
4. ***Debugging*** the code, to guarantee that it conforms to the design and fulfills the requirements.
5. ***Maintaining*** the system, to keep it functional despite changes in the requirements and the discovery of new bugs.
6. Helping the end-user ***use*** the program by showing how it operates and how it arrives at the results it presents.

The challenge of PV is to encompass these disparate phases of the programming process within a unified conceptual framework.

1.2 The Role of Graphics

Graphical representations have demonstrated their usefulness in a variety of endeavors as a means of illustrating complex relationships among components of a system. It would be inconceivable to build a ship, airplane, plant, factory, or piece of electronic equipment without the use of diagrams. These illustrations can capture essential features while suppressing extraneous detail. Often, they can be understood more readily than ordinary text.

While such illustrations find widespread use in computer programming, they are almost always manually generated. They have a tendency to become obsolete as the software they describe is implemented and changed. Also, the lack of tools for creating animated images restricts the ability to illustrate an essentially dynamic process such as a computer program.

Computer graphics offers the possibility of generating dynamic illustrations—illustrations of computer programs while they are running. These illustrations could be made to correspond to the most current version of a program, allowing a programmer to observe the actual operation of a system. By providing suitable tools for specifying the portions of a system to be illustrated and the manner in which they should be represented, interactions among various system components could be observed. If suitable design information were stored in the system, it could be used in the evaluation of running code to certify that a given module is performing its intended function. Furthermore, the graphical representations of programs could be used as a means of interacting with a programming environment. This would allow a programmer to manipulate and combine programs by changing their corresponding visualizations.

The idea of using computer generated images to visualize programs was developed in the earliest days of computer graphics [HAIBT], [STOCKHAM], and [KNOWLTON]. However, only recently have the requisite hardware and software advances been made which would allow such techniques to become cost-effective for a broad range of applications.

A PV system could capitalize on recent progress in the graphical representation of information and low-cost color graphics. The system that is envisioned will allow a person maintaining a complex software system to access many graphical representations. These include static descriptions such as module hierarchies and requirements specifications, and dynamic illustrations such as procedure activations and storage allocations. It will be possible to display several different representations of the same portion of a system (or the same representation of several different portions) simultaneously, through the use of multiple screens or multiple viewports on one screen. The level of detail presented in any given viewpoint can be varied to cover any point along the range from the entire system to discrete lines of code.

Special attention will be accorded to the means by which the various representations are specified. The envisioned system will avoid the necessity of modifying the program under observation by providing an external mechanism for selecting program constructs to be displayed and symbols to be employed in displaying them. A system of default representations, generated using knowledge about the particular programming language, will provide the user with an initial visualization. External descriptions can then be defined by the program author or maintainer to augment or replace these default representations.

The intention is to provide an environment for program creation and maintenance that makes the advantages of graphical representations available without placing an excessive burden on the people responsible for implementing and maintaining the programs. In this way, graphical program illustration can become a general-purpose tool applicable to a wide range of real world problems.

1.3 Approach

This report describes an envisioned PV tool that will aid the maintainers of large (10**6 lines of code), complex software systems. This tool is targeted primarily for use with programs written in ADA [ADA], the proposed standard DoD language. ADA is especially well suited to PV because it provides high-level, user-definable constructs which make much of the program design explicit in the code itself. Many other, less powerful languages require the use of comments or external documentation to explain how many seemingly unrelated programs and data structures fit together to form a coherent set of objects and actions.

This report addresses two issues involved in building a PV system:

1. The design of a visual language for describing programs and combining them, together with the processing, translation, and display routines necessary to create a visualization of a program.
2. The design of a system that uses such a visual language to examine and modify computer programs.

The various implementations will serve to explore a variety of techniques on a powerful, high-resolution, display environment, with an eye towards identifying a useful subset of techniques that could be implemented on a low-cost terminal costing in 1985 what an ordinary alphanumeric terminal costs today.

1.4 Outline of Report

The remainder of this report is organized into the following sections:

- Section 2 consists of an examination of the software process and suggestions for the type of outputs a PV system might produce.
- Section 3 is a description of how a user might interact with a PV system in the course of creating and maintaining a program.
- Section 4 is a presentation of a possible implementation of a PV system.
- Appendix A contains examples of some current and contemplated visualization techniques that could be used for illustrating programs and data structures.
- Appendix B contains the results of a survey of prior research in the areas of program illustration and graphical tools for software development.
- Appendix C is a list of references.

2. A CONCEPTUAL FRAMEWORK FOR PROGRAM VISUALIZATION

A framework for PV must categorize and classify those aspects of computer systems that can be visualized. The following is a first attempt at such a classification:

1. System requirements diagrams
2. Program function diagrams
3. Program structure diagrams
4. Communication protocol diagrams
5. Composed and typeset program text
6. Program comments and commentaries
7. Diagrams of flow of control
8. Diagrams of structured data
9. Diagrams of persistent data

Many of these visualizations can be either general or specific. General visualizations portray a program without referring to a specific activation. Specific visualizations are keyed to the execution of a program or some specific set of data. General visualizations can be either static or dynamic. Static visualizations portray a program at some instant of execution time or portray those aspects of a program that are invariant over some interval of time. Dynamic visualizations evolve and unfold under the control of the executing program.

For example, we can show an abstract representation of program data for all time, at some instant of time, or evolving through time. We can show how a program is organized into modules, which modules have activation records at a particular instant, or how modules are activated in the course of program execution.

These visualizations can be produced and used either singly or in combinations. Flow of control may be most meaningful if portrayed in relationship to program code or module structure. Input-output pairs may be displayed in relationship to the underlying data structures of individual program modules. Dynamic displays may be superimposed on static or general displays that provide the context within which the displays are interpreted.

In the following subsections, we describe each of the visualization classes in more detail. In the final subsection, we present a conceptual breakdown of visualizations into the separate aspects that must be specified.

2.1 System Requirements Diagrams

A computer program always exists as part of some larger system. Therefore, PV tools must assist in portraying the function and structure of that system. The tools should also aid in specifying the constraints imposed by the system on the program.

One very powerful method of describing system structure is the IDEF or SADT technique developed by SofTech [ROSS]. An IDEF model is a graphical representation of a system in terms of its subsystems and the data and control flow that link them together. This method deals quite naturally with the hierarchic nature of most systems. It provides a methodology for organizing the bookkeeping associated with large, complex system descriptions. A systems analysis in terms of an IDEF model can provide the foundation for automating some of the subsystems. The role of a program automating a subsystem can be understood in the context of the system within which it is to function.

To complete a requirements specification, the constraints on the program's design must be added. These constraints include execution speed, program size, user interface style, implementation vehicle, and cost. Further investigation is needed to determine the role of graphics in describing these specifications.

2.2 Program Function Diagrams

"What does the program do?" is usually the first question we ask about a program. In many cases, a program's function can be expressed as a mathematical function—a mapping from program input to program output. We can talk about the relationship of program input to program output in two different ways. We can attempt to characterize the general case; that is, the relationship of any input to any output. Alternately, we can simply enumerate a number of input-output pairs that are in some sense typical of the general case, and leave the rest to the inductive powers of the reader.

A statement of the program's function in the general case is a more powerful and useful description than an enumeration of sample behaviors. Yet it is an abstraction that is conventionally explained in terms of prose and mathematics rather than diagrams. It is very difficult to construct diagrams that portray the general case.

Sample behaviors, on the other hand, can be portrayed by pictures or diagrams of the output data produced from particular input data. One approach to the visualization of program function is to provide a "casebook" through which the user can browse. The user can induce a model of what the program is supposed to do by seeing what it actually does on a carefully selected set of sample inputs. The choice of these sample inputs significantly affects the utility of this technique. In many cases, certain values or classes of values are critical to understanding a function. For example, in understanding a factorial function, important values or classes are 0, 1, positive integers, negative integers, real numbers, and non-numerics.

2.3 Program Structure Diagrams

"How is the program organized?" is often the second question that we ask about a program. In many cases, a program's structure can be expressed in terms of a hierarchical collection of modules and constituent modules.

Structure diagrams and function diagrams should be used in tandem. At the top level, the task of the program should be associated with a corresponding structure definition. Each constituent module of the structure definition should have associated with it a function diagram portraying its task. This hierarchical decomposition of coordinated function and structure descriptions should be continued until the resulting modules are of an order of magnitude of one page of code.

It appears that the HIPO (Hierarchy plus Input-Process-Output) technique [STAY] consists of the coordinated use of program function and structure descriptions. It is necessary to investigate the HIPO methodology further and perhaps to enhance its use of diagrams and other visualization aids.

2.4 Communication Protocol Diagrams

Once it is known how a program is divided into its component parts, it is useful to know how those parts communicate. This is especially important when the program consists of many processes running on one or more processors. A visualization of the flow of data among modules can be displayed as part of another diagram. For instance, the program structure diagram can be overlaid with lines showing the data paths between modules. By using this technique dynamically, the actual flow of data can be monitored during execution.

A practical example of this approach is the System for Distributed Databases (SDD-1) [ROTHNIE *et al.*], a distributed database system at Computer Corporation of America (CCA). SDD-1 employs a color graphics terminal to show, in real time, the data transferred between sites on the ARPANET. The terminal monitor is useful as both a demonstration and debugging aid.

2.5 Composed and Typeset Program Text

The central activity in the visualization of programs has always been reading program code. While alternative graphical techniques are proposed here, there will be cases where code must be examined. This task can be made significantly easier than it is at present.

Since the Middle Ages, typographers and printers have developed tools and conventions for printing documents that can be read and understood easily. Since its inception, computer science has almost totally ignored these tools and conventions, making programs much harder to read than they need be.

Some relevant typographic tools that could be applied to the publishing of programs are:

1. Typographic hierarchies for distinguishing a program's constituent elements that belong to various syntactic or semantic categories. Typographic hierarchies are implemented by the consistent and controlled use of a variety of type fonts, type styles within a font (bold, condensed, italic), and point sizes.
2. A rich symbol repertoire employing a wider range of symbols and colors than is currently used.[1]
3. Skillful composition and layout of a program's constituent substructures. Layout conventions include the use of indentation, horizontal paragraphing, vertical paragraphing, pagination, footnotes, marginal notes, and page headers. The use of computer graphics also permits applying dynamic techniques—such as colored highlighting—over selected or active portions of program text.

By speaking of typesetting and composition, we do not mean that all documentation will be produced and used on the medium of paper. On the contrary, programs will be written and read using powerful, interactive graphics tools. These tools will be designed for manipulating structured, and possibly dynamic, text and pictograms. To enhance readability and comprehensibility, these tools must provide text quality that is far closer to the standard of today's printing industry than to the standard of today's computer terminal industry. High quality typesetting and printing on paper, primarily on a demand basis, also will be required to provide hard copy that is portable, malleable, and tangible.

2.6 Program Comments and Commentaries

Program comments, often known as internal documentation, are analogous to the critical annotations of conventional literary expression. Program commentaries, often known as external documentation, are analogous to prefaces, introductions, postscripts, and critical expository analyses. Both comments and commentaries are an important part of conventional programming discipline, yet they fall far short of attaining their ultimate potential. How can they be improved?

The greatest potential for improvement cannot be brought about by a technological "fix." The obstacle is the shortage of good writing skills among programmers and documentation specialists. The ***Commentary on the UNIX Operating System*** [LIONS] is a classic example of the value of well written system documentation.

A second significant problem area is documentation completeness. This area can be addressed by developing interactive computer systems to check that program documentation meets specified documentation standards and to prompt the writer to fill in what is missing.

A third problem is that documentation should be, but is not, an ongoing process. Although maintenance programmers sometimes will make slight enhancements to documentation to record their "bug fixes," documentation is not viewed as a continual process of enhancing and clarifying the meaning of the program. Logically, the original programmer is the person best able to explain what he meant. In practice, however, it is often someone else—who has discovered what is going on ***despite*** the obfuscation of available documentation—who is best able to explain it to others. Technical mechanisms and administrative procedures should enable and encourage him to do so.

Finally, we should not be concerned only with who adds what to the documentation at what time. We should also be concerned with how comments and commentaries may be related to the other visualization mechanisms in the way that is most helpful cognitively. This can be done with new kinds of interactive, multi-media, multi-sensory reading, browsing, and visualization systems.

2.7 Diagrams of Flow of Control

"What happens when the program executes?" is another question we ask about a program. We can further refine this question in two different ways. We can be interested in the order in which things happen or in the effect of program execution upon the underlying data. To address these concerns, diagrams of flow of control and diagrams of structured data are needed.

At the top levels of description, diagrams of flow of control may be dynamic structure diagrams. They will illustrate which modules are activated and in what order. They will illustrate how particular modules carrying out individual functions are linked together to achieve higher-level functions.

[1] Gutenberg had more than 300 symbols in his type case.

At a deeper level of description, we want to look inside particular modules and see how the code executes. We want to observe iterative loops, recursive procedure calls, and conditional ana case selection mechanisms. Flow charts, Nas‿i-Shneiderman Diagrams [NASSI and SHNEIDERMAN], and GREENPRINTS [BELADY, CAVANAGH and EVANGELISTI] are a beginning. However, they portray only the static structure and the potential flow of control rather than the actual flow of control during program execution.

2.8 Diagrams of Structured Data

"What happens when the program executes?" can also be answered in terms of the database upon which the program is computing. This database includes the program input when execution is initiated and the program output when it is terminated. The database also includes the variables that are central to the program's function, such as the data being sorted by a quicksort, and the variables that are incidental to the program's function, such as the artifacts of a particular piece of code or programming technique.

The visualization of structured data appears to be one of the most tractable and powerful of the approaches we have presented. Baecker's pilot film on sorting algorithms [BAECKER] and the work of Knowlton [KNOWLTON], Hopgood [HOPGOOD], and others have vividly demonstrated the power of this technique. The Spatial Data Management System (SDMS) [HEROT *et al.*] developed at CCA has demonstrated the feasibility of using graphics to access structured data.

The major difficulty in applying these methods results from the size and complexity of the databases of most interesting programs. It is for this reason that we have spoken of "diagrams of ***structured*** data." It is only through structuring the complexity that we are able to comprehend and master it. And we will not always be able to do this dynamically, for we will need to stop the program and look around, start it again, stop it and back up, and often change our point of view. We must be able to browse and explore both in space and in time.

Displays of data are in some ways the most fundamental unit of PV. One can argue that (1) program function diagrams expressed as input-output pairs are a special case of the display of program data, (2) program structure diagrams will provide insight only down to the level of module organization, and (3) flow of control is easily induced from observing changing data.

2.9 Diagrams of Persistent Data

An important category of structured data is that which remains in the computer system after the program has ceased execution, as in a database management system. Since the amount of such data is often several orders of magnitude larger than that contained in the memory of the computer, different techniques are required to visualize it. Furthermore, the user of a PV system often will be much more interested in the physical organization of persistent data if it resides on some storage medium that is not perfectly random-access, such as a disk. Fortunately, the database community has developed a rich set of symbols that can serve as a starting point in visualizing persistent data.

2.10 Summary

We have developed a framework for PV in terms of nine classes of methods for presenting information about a program graphically and often dynamically. These methods may be motivated and understood in terms of the stages of a program's life cycle presented in Section 1.1. To assist in describing the requirements of a program, we have system requirements diagrams. To assist in describing the design of a software system, we have program function diagrams and program structure diagrams (ideally, used together). To assist in describing the code of a program, we have composed and typeset program text, program comments and commentaries, diagrams of flow of control, and diagrams of structured data. To assist in the debugging, maintenance, and use of a system, all nine classes of diagrams can be employed.

Thus, a comprehensive and integrated approach to PV promises to have great impact on the entire process of software engineering. It will do so by contributing to the cost-effective production and maintenance of reliable software.

2.11 Specifying Program Visualizations

In this section, we describe mechanisms for specifying or requesting the production of a specific program visualization. Although these mechanisms may differ somewhat from class to class, we shall present a conceptual framework that we expect will be general enough to apply to the nine types of visualizations described above. In this description, the "visualizor" is the person who creates or specifies the semi-automatic production of a visualization.

A visualization can be defined by specifying six separate aspects:

1. The ***subject***. What aspect of the program's structure or behavior is to be visualized? The visualizor's first task is to specify what he wants to look at. He must:

- Indicate whether he wants to see the entire program or only a particular module.
- Specify the level of abstraction at which he wishes to view the subject he has selected.
- Select the variables or data structures he wants to make visible.
- Designate the control structures whose flow he wants to observe.

2. The ***symbolism***. What graphic representations are to be used in the visualization? The visualizor's second task is to specify how the subject of the visualization is to be portrayed. Is hierarchical module structure to be represented by a tree or by nested regions? Is program code to be represented literally or by an abstraction such as Nassi-Shneiderman diagrams or GREENPRINTs? Is a stack to be represented by a vertical array of boxes, a single colored box, or a dynamic bar?
3. The ***composition***. Where are the visualizations to appear in terms of the spatia! dimensions of the display medium? The visualizor's third task is to specify where the subject is to be portrayed. This is perhaps best approached through two independent decisions. The first involves choosing the location for each subject in page coordinates on a single, large display page. The second step involves mapping various windows of the display page onto various viewports on one or more display surfaces.
4. The ***event***. When are visualization snapshots or frames to be created in terms of program execution time? The visualizor's fourth task is to specify when the subject is to be portrayed. Should "snapshots" be taken at some instant, during some interval, or throughout program execution? Should the "camera" be running only when some predicate is satisfied? Should the camera turn on or off when some predicate becomes satisfied?
5. The ***dynamics***. How is screen time to relate to program execution time? The visualizor's fifth task, in the case of dynamic visualizations, is to specify how program execution time is to be portrayed in screen time. This may be done by establishing a mapping in advance of running the program or by putting the mapping under the visualizor's interactive control.
6. The ***context***. How is each new illustration to be related to previous illustrations of the same or of a different class? The visualizor's final task is to relate each new visualization to the context established by previous ones. This may be done by saving previous illustrations on a stack, juxtaposing or superimposing several of them, or displaying dynamic information on a static reference background. Thus, we might display flow of control as arrows over a representation of program code or module structure.

A PV system must provide a flexible mechanism for specifying the manner in which a visualization is displayed to the user. It must be possible to create these specifications in a way that enhances rather than disrupts the user's understanding of the structure of a complex system. For example, transformations from one level of detail to another or from one part of a program to another must be done in a way that makes clear how the new and old views are related.

3. USER INTERFACE

In this section, we describe a PV system from the user's viewpoint. The section is divided into separate discussions of user requirements and interface design issues.

3.1 Requirements

A PV system must offer the programmer the ability to increase his understanding of large programs. The system must satisfy this goal whether or not the programs operate correctly and whether or not the user is the author. Moreover, the system must not require a significant increase in the effort required to create a program or its visualization. To do this, three subgoals must be satisfied. The system must:

1. Employ a coherent visual language for program illustration.
2. Be well integrated into the software development process.
3. Provide usable visualizations with a minimum of work.

3.1.1 The Visual Language

The quality of the symbols used to represent program concepts is critically importance to the success of a PV system. These symbols should be part of a carefully designed visual language. The language should be both easy to learn and powerful enough to describe most program constructs without requiring arbitrary extension. The number of primitive symbols must be small enough to be recalled easily and varied enough to be distinguished easily.

These goals can be furthered through the use of the enhanced visual richness of an interactive, computer graphics environment. For example, a small number of symbols can be given various shades of meaning through variations in color, size, position, line weight, background, and changes over time.

At the same time, careful attention must be paid to pragmatic limitations of the user, such as:

- The number of parameters that can be differentiated.
- The number of objects that can be remembered in short and long term memory.

Also important are the limitations of the display medium, such as:

- Its spatial and color resolution.
- The number of objects that can be displayed effectively.
- How frequently objects can be changed.

These user and media limitations can be dealt with through the imposition of clear hierarchies. Such hierarchies will allow a large amount of information to be managed with a comparatively small number of objects.

All the above mentioned issues fall into the field of graphical design. They underscore the importance of drawing upon knowledge developed in that area. One subject of special interest is in the design of symbol systems. These range from specialized signs—such as those developed by the Department of Transportation for public transport—to extended universal symbol repertoires.

In these extended repertoires, a relatively small number of symbols can be extended with a consistent set of prefixes and suffixes to form an easily understood language. Such a language will draw upon the many symbols and display methodologies that already have been developed for various levels of software description. It will provide a framework within which the user can move among different levels and types of displays, while maintaining common symbols and points of reference. For instance, objects present in different displays will maintain as similar an appearance as possible or would bear some identifying characteristic. In addition, this visual language will provide a mechanism for adding motion to symbols that were previously static.

3.1.2 Integration

There are two main reasons why a PV system should be carefully integrated into the process of software development. First, any improvements in the area of initial requirements analysis or design will be reflected by fewer problems further on in the development process, when problems are more expensive to solve. Since many of the problems of building and maintaining large programs can be traced to errors or oversights in the initial areas, this is an important benefit.

A more important benefit is that much of the information manipulated in the early stages of system development would be of great assistance in the later stages. If this information were captured in machine-readable form, it could be used to generate more meaningful, graphical representations of a running program. This situation is especially common when low-level languages—such as assembler or Fortran—are used. In such languages, the constructs used by the designer may bear little surface resemblance to the actual code that is written.

For example, program constructs such as linked lists might be implemented as Fortran arrays. The programmer might explain the connection between the two in a comment statement or in an external, hard copy document. However, the production of a computer generated picture showing the data structure as a linked list will require some machine-readable indication that defines the mapping of the data to the picture. While this could be done by the person charged with fixing bugs in the system, it would be far better for the responsibility to be fixed with the original designer or implementor of the system.

Section 3.2.2.2 shows how this goal of linking early design data with actual coding can be achieved through the use of an intelligent program editor. Such an editor would provide macro facilities that encourage the input of structured information. The editor would offer program templates that provide the repetitive sections of code used in describing common constructs.

The input required by such a system will not be seen as an unreasonable burden on the programmer. This is because it replaces the need to create conventional documentation, usually considered to be a burdensome task. If the input of the required documentation yielded some immediate benefits, such as checking errors in the program and providing an easy means of taking notes, the task of persuading programmers to enter documentation might actually be eased. Information like the results of error-checking also could be of use in designing acceptance demonstrations and testing plans.

An alternative that may be attractive in some circumstances is to create a new job classification, program illustrator. This person would assist in visualizing novel or complicated constructs.

3.1.3 Defaults

A PV system must be usable with very large programs. Some of these programs may be written either without the aid of the integrated tools described above or with insufficient use of those tools. Accordingly, the system must have a way of generating reasonable default visualizations without any additional information.

The resulting displays often will be useful in and of themselves. This is especially true with a language such as ADA, which allows the use of high-level program constructs. When a more elaborate display is required, the default representation can serve as the basis for specifying more elaborate visualizations.

Thus, the user need not create a visualization from scratch. He can incrementally modify one provided by the system, selecting alternative symbols, portions of programs, and dynamics. By starting with a default visualization, such user input can be stated in terms of the graphical representation instead of a procedural language.

The default visualization mechanism will automatically analyze programs. Variables will be selected for display on the basis of frequency of reference and interdependencies of programs. For example, a variable used for communication between two processes could be displayed along with the high-level representation of those processes, with a line connecting the variable to the two processes. This facility will aid the user in forming hypotheses about the operation of the program and in selecting more detailed displays.

A facility for creating new symbols from existing primitives and/or composite symbols is required. This facility should be sufficiently powerful to eliminate the need for manipulating the graphics in the actual implementation language in all but the most unusual cases.

3.2 User Interface Design Issues

The preceding subsections have set forth the requirements that must be met by a PV system. Here, we show how such a system might appear to the user. The design of the user interface must be further developed as the research progresses.

3.2.1 Program Space

Creating and maintaining a large and complex program requires an easy way for the programmer to select among the various visualizations that can be employed. Some mechanism must be provided to manage the large set of modules and their various visualizations. In this way, the user can alter which portion is selected, how it is displayed, and at what level of abstraction it is viewed.

The solution proposed here is to allow the user to manage the visualizations by employing a space composed of a hierarchy of two-dimensional surfaces. The levels in this hierarchy will match the structure of the program under investigation. As one descends the spatial hierarchy, one views successively more detailed representations; for instance, increasingly lower-level modules. Within a branch of the hierarchy, one can move on any surface to select the modules or data structures to be displayed.

The visualizations at any position can be one of three types:

1. Requirements diagrams
2. Function diagrams
3. Data diagrams

They can be either formatted—as diagrams and typeset code—or unformatted—as comments and commentaries on the code. They can be either static or dynamic. In addition, requirements and function diagrams can include visualizations of control and data flow.

3.2.2 User Controls

There are three classes of actions that a user may perform:

1. Specifying views of a program
2. Entering information about a program
3. Interacting with a program

Since he may have to perform many of these activities in parallel and since it is often useful to be able to compare different parts of a program or different views of the same part, the system provides the ability to segment the one or more display screens into windows.

3.2.2.1 View Specification

When the user first starts up the system, one window presents a default representation showing an abstract view of the entire program. Any portion of this display may be selected for display at a more detailed level.[2] This process may be repeated until the

[2] The actual mechanism (continuous or discreet zooming, menu selection) and controls (joystick, touch-sensitive screen, tablet) would be determined as part of a research program.

actual code is displayed. For any particular location and level of detail, the user has a choice of viewing any of the nine types of diagrams described in Section 2.

It is intended that the user will seldom need to make direct reference to the actual source code to select visualizations of the program. Graphical representations of all programs, processes, and data are provided by the system. These can be manipulated to control the level of detail that is displayed or to provide alternative visualizations.

If the user needs an unusual display or wants to display some aspect of the program in a nonstandard way, he has two options. He can select additional graphical symbols to illustrate some part of the program. This will usually require entering more information about what the program is supposed to do. As such, it constitutes an addition to the general store of knowledge about the program.

For example, the programmer may know that several variables taken together describe one aspect of the program and thus should always be displayed together. This display might take the form of a predefined graphical symbol, such as a rectangle, that has its size and shape defined by the values of the variables.

A second option is available if there are no existing graphical symbols adequate to express the concept in question. A symbol creation program is provided which allows the user to define interactively a new symbol. To do this, the user combines graphical primitives and existing symbols; *e.g.*, combining two rectangle symbols to create a "nested rectangle."

3.2.2.2 Entering Information About a Program

Information about a program is entered into the system at many points in a program's life cycle. In the early stages of requirements analysis and design, the user may input IDEF-like diagrams describing the role the program is supposed to play and HIPO-type diagrams describing the structure of the design. Later, as the program is implemented, actual lines of source code are typed in or retrieved from a library. In the debugging and maintenance phases, the program will be modified. New insights will be gained (or old ones rediscovered) that should be incorporated into the repertoire of visualizations of the program.

At each of these stages, it would be useful to have a tool that eliminates as much of the repetitive part of the job as possible and that verifies the correctness of the input as it is entered. Nowhere is this need more apparent than in the entry of source code, if only because of its sheer volume. The graphical representation of programs offers the opportunity for a new way of building software systems. Rather than typing in lines of code, the user can combine pictorial representations of programs.

This section proposes a graphical program editor, integrated with intelligent text editors developed elsewhere, that would perform the following four services:

1. Providing templates of commonly used constructs
2. Interfacing to modular programming inventories
3. Checking consistency
4. Checking design rules

Templates provide the ability to input standard constructs with less effort and less opportunity for errors than when typing them in by hand. By hitting the appropriate function key and typing a few characters, the user can cause the system to provide a formatted, syntactic skeleton that contains the keywords, matched parentheses, and other punctuation marks of a given statement form. The template includes placeholders at each position where additional code is required. This approach was used in the Cornell Program Synthesizer for PL/CS [TEITELBAUM].

Just as templates aid in "programming in the small," the use of ***modular programming inventories*** provide the same advantages in "programming in the large." Such inventories provide the building blocks for performing the kinds of manipulations required in any large system. Data and control flow among modules are indicated graphically. There is automatic verification that formal parameters are of the appropriate type and number.

Regardless of whether code is entered through templates or by hand, the editor checks for ***consistency*** within the program in accordance with the rules of the language. It ensures that variables are declared exactly once, type conventions are obeyed, and statements are in legal syntax. Violations of these constraints produce a warning message on the screen.

The design of the warning mechanism must be considered very carefully to ensure that it is done in a manner that is least likely to be annoying but most likely to be heeded. For example, undeclared variables can be accumulated in a window on the screen, along with the system's best guess as to how the variable should be declared to agree with its context. At any point, the user can type in declarations, causing the variables to disappear from the window. Alternately, he can move his cursor to the window and specify the type there, implicitly declaring them. Or he may decide that the system has guessed correctly and therefore retain its decision.

Design rule checking will operate in a similar manner. The program will be checked against design principles defined as part of the ADA development effort.

3.2.2.3 Interacting with a Program

No debugger is complete without a method of interacting with a program. In a PV system, this activity might make use of two or more windows on the display screen. For programs designed to communicate with a terminal, one window simulates the screen of such a terminal. Characters typed by the user appear there and output from the program is directed there. Meanwhile, other windows can be used to examine the operation of the system as described above. Breakpoint locations can be indicated by pointing to the corresponding location on a program structure diagram or typeset listing.

Alternately, the programmer can use the same displays to specify events or locations that demark areas of interest. When these areas are active, the program is slowed down to a visible speed. At other times it runs at normal speed, unobserved but much faster. Data structures internal to the program can be modified by editing their visualization on the user's screen.

4. IMPLEMENTATION APPROACH

In the design described in this report, the graphical representation of programs and systems is produced by a visualizer run-time executive process. The visualizer is used in place of the standard UNIX command processor, or "shell." The visualizer provides a dynamic, graphical representation of the executing program or system, in addition to all the shell's standard executive functions. It is described in more detail in Section 4.1.

We have seen that there are various views of a system — system requirements diagrams, program function diagrams, program source code, data structures. These views can be selected interactively at "visualization time" through interactive devices. Such devices include touch-sensitive video monitor screens, a data tablet, and joysticks. The internal graphical definition of each view of a program is part of the ***visualization description*** of the program.

A visualization description provides the rules for producing the visualization of its associated program. Particular emphasis is placed on graphical semantics: what the various aspects of a program "mean" graphically. Basic visualization descriptions are automatically generated by a visualization description compiler.

A visualization description compiler is an extension to the standard C or ADA compiler. The visualization description compiler compiles and links C or ADA source code. It produces a visualizer-executable load file and a separate visualization description file. Visualization descriptions and the visualization description compiler are discussed in Sections 4.2 and 4.2.1.

The visualization descriptions automatically produced by the visualization description compiler will suffice in some applications. However, specialized graphical representations of programs often will be required. This is particularly true for program documentation. A highly interactive visualization description editor is provided for manual enhancement of visualization descriptions.

The visualization description editor provides a library of graphical representations for general program constructs at various levels of abstraction. These graphical representations are referred to as ***templates***. Templates are presented to the PV system user as simple graphical primitives that can be used in a program's visualization. Internally, a template is a visualizer-executable set of instructions that results in the production of the desired image. Templates are included in a visualization description through simple, graphical interaction with the visualization description editor. The editor also provides the means to create new templates. In Section 4.3, the visualization description editor is discussed more fully.

4.1 The Visualizer

The run-time visualizer executive of the PV system has three principal tasks:

1. Interact with the user to determine the aspects of the program that are of interest.
2. Monitor the executing program to determine its state.
3. Update and manage the graphics displays to present the user-requested visualization.

In this section, we discuss the issues involved in these tasks and suggest possible implementation strategies.

4.1.1 Interacting with the User

The visualizer is responsible for handling those user interactions that specify which aspects of the program under study are of interest. It is the visualizer's responsibility to transform these user requests for visualization into their internal representation. This representation is in the form of predicates based on program activities and data values — such things as module activation, procedure invocation, and data access. An initial selection of program aspects to visualize is provided at start-up time by the program's visualization description.

4.1.2 Monitoring Executing Code

As stated in the previous section, the program aspects of interest to the user are specified internally by sets of predicates. These predicates determine the conditions under which the user wants to be alerted to activity in the program. The system provides a set of default predicates that produce a display of all activity at the chosen level of detail.

Predicates may be set to detect procedure invocations, variable references (read or write), and violations of constraints on variables.[3] When a predicate is satisfied, an associated graphical action is performed. Hence, the visualizer must be able to detect when the executing code has satisfied a predicate. Ideally, interaction between the visualizer and the executing code would be kept to a minimum; this would keep response times low and execution speed reasonably high.

The implementation of the PV system discussed here would run on a DEC VAX-11/780 computer running the UNIX operating system [KERNIGHAN and MCILROY]. For a number of reasons, such as portability, it is desirable to implement the system in such a way that minimal modifications to the operating system are necessary. Fortunately, among the features of the UNIX operating system is the ability of one process to examine and modify the address space of another process. Using this feature, a parent process may examine variables in a child process, set breakpoints in the child's code, and suspend and resume its execution.

Given this capability, it is easy for a parent process to catch procedure invocations in its children. By setting a breakpoint at the entry point of the procedure, all invocations will be trapped. Examination of the program stack will reveal who the caller was. Thus, a predicate set to trigger on a particula: procedure's invocation of another procedure can be caught by setting a breakpoint at the called procedure's entry point. When the breakpoint is hit, the graphical action is executed if the calling process is the one specified in the predicate.

In general, predicates triggered by reference to a particular variable or by violation of a variable's value constraints are difficult to implement in an efficient manner. The executing code must be single-stepped: the visualizer must examine the variables after each instruction has executed. More efficiency can be achieved when predicates are set on variables local to a procedure or statement block. In this way, the range of the single-stepped code can be reduced to the scope of the current block.

Part of PV research would involve investigating the preanalysis of source code to make evaluation of variable reference predicates more efficient. One technique to be investigated involves noting the locations of machine instructions generated from source code that refers to the variable of interest by its symbolic name. Setting breakpoints at only the locations noted may provide a significant savings in processing time.

4.1.3 Updating and Managing the Displays

The visualizer must maintain the program displays in response to user visualization requests and changes in the program's state. The visualizer examines the visualization description to determine the graphical appearance of what is to be displayed at any given time. The visualization description describes the graphical representation of a program and its salient features; *e.g.*, input requirements, output produced, and key states during execution. The visualization description contains display directions for each level of detail that the user may wish to see. At the most detailed level, source code is displayed.

The visualization description of a program is tree-structured. At the top level, there may be a program structure diagram. More detailed views of each of the components in this diagram may exist in the hierarchy. The user may "zoom in" on a particular component to reveal a more detailed view (if it exists). Alternately, the user may indicate his interest in a particular component by touching its representation on the screen. This would have the effect of selecting the next most detailed view of that component.

The user may wish to create multiple windows in which particular views of components can be fixed. These windows would be updated as necessary while the code was executing, but would remain in place while the user manipulated other portions of the screen.

Animation of the program visualization is controlled by predicates that become true as program execution proceeds. Predicates are a natural way of directing the system's attention to a particular portion of the code. Predicates that are triggered by procedure invocations could cause both the calling and called code to be displayed.

Variable access and value constraint predicates direct the visualizer's choice of what to display. Their action is similar to that of procedure invocation predicates. Each predicate has associated with it a graphical action. The visualizer can deduce what to show based on this graphical action. The following example describes how this might work.

[3] In this section, the term "variable" indicates aggregate data such as structures, arrays, strings, and lists.

Assume the user selects a relatively abstract view of a program for display, such as a program structure diagram. Predicates have been set on a variable that has no representation at that level of abstraction. The user begins execution of the program. As the program executes, the abstract view is updated; for instance, to indicate which module is active. At some point, the predicates set on this variable become true. If the graphical action indicates that the representation of that variable must be updated, then the visualizer must switch to the appropriate view. The visualizer may try to preserve the current view by moving it in a scaled form to a window that is still displayed when the view changes. Later, should the user wish to return to the previous view, he need only touch the window containing that view.

4.1.3.1 Display Hardware Considerations

The choices of modes of interaction and display formats described above are based on implicit assumptions about the power and speed of the display system. The maintenance of multiple windows on a single screen requires that the display system contain a fast, powerful processor that can perform the necessary tasks itself. It would be possible to maintain multiple windows with a conventional frame buffer, but this would greatly increase both the computational and input/output loads on the host machine. There would be a consequent increase in response time—a significant factor in user acceptance of the system.

Several display systems are available that could provide sufficient processing power for the PV system. They have the added advantage that they support resolutions of up to 1024 by 1024 pixels. Such (relatively) high resolution systems are necessary to display large amounts of text and detailed diagrams.

The two most powerful display processors commercially available are the Ikonas RDS-3000 and the Ramtek RM 9400. A third system, the AED 512, lacks the high resolution of the other two, but is relatively inexpensive. All three systems allow the user to download the controlling microprocessors, giving him customized graphics primitives.

The Ikonas system is the most general and most powerful of the available systems. It can be configured with multiple processors that can be pipelined. Video format is readily switched, allowing it to produce NTSC-compatible video for videotaping. Vector write times are (claimed to be) 900 nanoseconds per pixel.

The Ramtek system contains one high-speed processor that can be user-programmed. It does support a large virtual picture size of 32K by 32K pixels. Any portion of this virtual picture can be mapped to the screen picture. Pictures can be stored as display lists that are executed; the results are stored in the frame buffer. Changing the display list and re-executing it allows fast modification of the picture. Pixel write times are on the order of 1.12 microseconds. The output video format cannot be reconfigured.

The AED system is a low-cost frame buffer system containing a user-programmable microprocessor. However, its 6502 microprocessor is relatively slow and has limited storage for display lists. Resolution is fixed at 512 by 512 pixels. Several of these in tandem with a computer controlled, video, special effects box could simulate multiple windows, but at added expense to the host machine.

One suggested course of action for a PV research project is to select one of the fast displays for use in evaluating the interaction techniques that would be developed. Selected techniques could then be implemented on the slower display. Every effort should be made to maintain a high level of utility despite the reduction in display capability. The hope is, however, that display systems of high capability will decrease in cost and become affordable for a wide range of applications.

4.2 Visualization Descriptions

Associated with each visualizer-executable program load file is a visualization description. The visualization description provides the visualizer with the database necessary to produce graphical representations of the program's execution. A visualization description has four main components:

1. Picture semantics
2. Graphical representation schema
3. Program semantics
4. Visualization predicate file

The ***picture semantics*** specify how the various aspects of the program should be represented graphically at multiple levels of abstraction. For example, an overview of a complex system might show process symbols connected by lines to depict inter-process communication channels. Inter-process communication might be shown as message packets moving through channels.

Zooming in for a more detailed view of a single process might show a hierarchical view of program modules, with the active module highlighted. Zooming to greater and greater detail will eventually resolve to source code level representation of the program or even into lower levels, such as machine instructions, micro-code, or hardware signals. Other views of the program will show data structures, flow of control diagrams, etc. The importance of picture semantics is in specifying the graphical appearance of these program representations.

Picture semantics contain "picture descriptions." Picture descriptions associate parts of programs with descriptions of the graphic symbolism used to represent them. In Section 2, we discussed program subjects and symbolism within a conceptual framework for specifying program visualizations. In Section 4.2.2, there is a detailed explanation of picture descriptions.

The spatial organization of graphical program representations is specified by the ***graphical representation schema***. The schema indicates which view replaces the current view as the user zooms in or out of a viewport. The schema also selects the appropriately related default views for otherwise inactive viewports. The graphical representation schema specifies the composition and context of the visualization, discussed in Section 2.11.

Unexpected computation in a program should attract the user's attention. ***Program semantics*** allow the visualizer to graphically indicate normal and abnormal operation of a program. In the context of this PV system, program semantics describe "what the program is supposed to do" in general terms of expected data values, code execution frequency, and process idle time. Special visualization directives are made available to the visualizer when an abnormal program condition is detected. The representation and processing of these kinds of program semantics are related to the maintenance of database integrity constraints. Common issues are discussed in [BERNSTEIN, BLAUSTEIN and CLARKE] and [HAMMER and SARIN].

Knowing what to show, rather than how to show it, is the purpose of the ***visualization predicate file***. The predicate file selects which aspects of the program will be displayed under what circumstances. The predicate file is read from the visualization description into the visualizer's address space when the program associated with the visualization description is executed.

The predicate file is continuously updated once it is read by the visualizer and the user begins to interact with the system. The initial form of the predicate file, as stored in the visualization description, serves to specify default views of the program when visualization is first initiated. The internal form of the visualization predicate file is a list of predicate-graphical action pairs. When a predicate is satisfied, the associated action is performed by the visualizer. A discussion of the efficient representation and processing of predicate files can be found in [WONG and EDELBERG].

4.2.1 The Visualization Description Compiler/Linker

A special visualization description compiler/linker is used instead of the standard C or ADA language processors to prepare programs for visualization. The visualization description compiler/linker produces a visualizer-executable program load file and an associated visualization description file.

This automatically produced visualization description describes as much of the program as can be safely inferred from scanning the program source code. The automatically produced visualization description is sufficient for some applications, such as watching lines of source code execute. However, the visualization description often must be manually annotated to include descriptions of complex or abstract program design concepts. The visualization description/template editor described in Section 4.3 is used to expand the visualization description in an interactive fashion.

The visualization description compiler is actually a system of four processes:

1. An extended C or ADA compiler
2. A visualization description generator
3. A binary linker
4. A visualization description linker

A schematic overview of the visualization description compilation process is shown in Figure 4.1.

The ***extended C or ADA compiler*** produces relocatable binary object files very similar to those produced by the standard compiler. The principal difference is the addition of object code to enable access to the process's address space by the visualizer run-time executive. This code is simply the addition of the "ptrace" system call, which allows a parent process to access the address space of its child process.

The ***visualization description generator*** automatically creates a basic visualization description from program source code. Conventional language processing techniques are used to scan and parse the program source. Visualization description production rules are used to map the program into a visualization description. A conservative approach to automatic visualization description production is taken; namely, only the program features that can be inferred from the program source with a very high degree of certainty are described.

The ***linking*** process involves two tasks:

1. Combining binary object files into a visualizer executable program load file.
2. Combining the partial visualization descriptions associated with each binary object file into a visualizer-interpretable, complete visualization description.

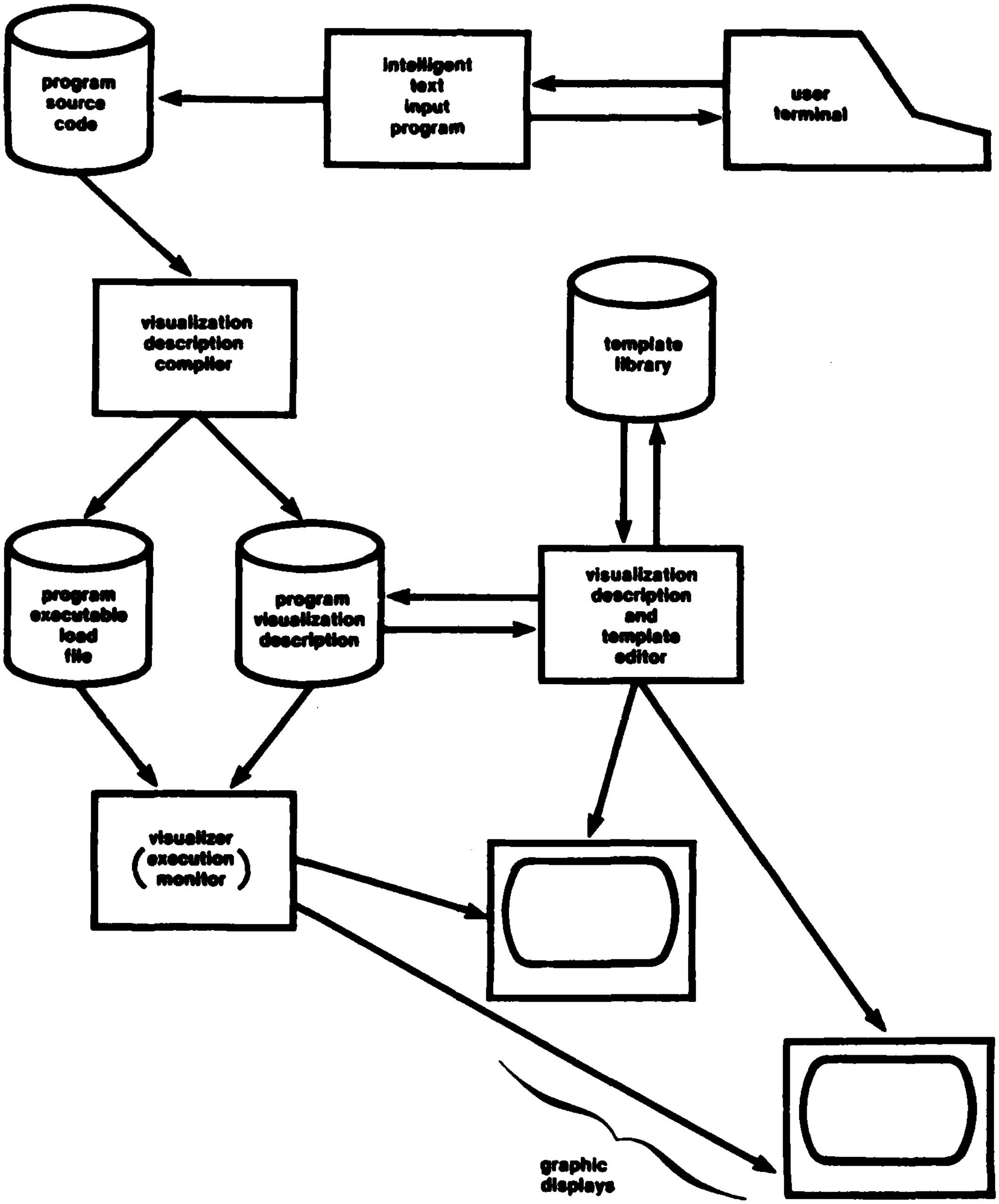

program source code
intelligent text input program
user terminal
visualization description compiler
template library
program executable load file
program visualization description
visualization description and template editor
visualizer (execution monitor)
graphic displays

Figure 4.1 Visualization Description Compilation Process

These two tasks are carried out by separate processes: the binary linker and the visualization description linker. The ***binary linker*** performs the functions of the standard UNIX linker. The ***visualization description linker*** combines partial visualization descriptions into a single, complete visualization description file. This visualization description file is then associated with the load file of the program it describes.

4.2.2 Picture Descriptions

All graphical symbols used to visualize a program are specified by the picture description component of the picture semantics. A different picture description is provided for each view of every ***program subject*** in the program. In PV terminology, a program subject is simply any "thing" or "aspect" to be visualized; for example, a data structure or a program module.

A picture description is actually a program that can be executed by the visualizer's ***picture generator***. The result of executing the picture description is the production of the desired graphic image.

The architecture of the picture generator is optimized foı the production of graphic images. In addition to a standard register file and data paths, the picture generator contains a ***picture workspace*** that is used to construct graphic images and a ***picture stack*** that allows subimages to be stored until they are combined into a complete picture.

Picture generator instructions are oriented toward the task of image production and make extensive use of the picture stack. Representative instructions follow. These instructions will be used in an example.

```
OP CODE   ARGUMENTS

templt    template_name  template_argument(s)
```

EXPLANATION

The templt instruction is similar to the subroutine call of a conventional processor. Template_name is the identifier of a self-contained code segment that takes its own arguments and produces an image. One might think of template_name as the name of a "picture type" procedure that executes primitive graphical instructions using vectors, pixels, etc.

```
OP CODE   ARGUMENTS

stack
```

EXPLANATION

The stack instruction puts the image in the picture workspace on the picture stack and then clears the picture workspace.

```
OP CODE   ARGUMENTS

combine
```

EXPLANATION

The combine instruction combines the image in the picture workspace with the image on top of the picture stack. The combined image becomes the new contents of the picture workspace. The top image is removed from the picture stack.

```
OP CODE   ARGUMENTS

frame
```

EXPLANATION

The frame instruction encloses the image in the picture workspace within a frame.

```
OP CODE   ARGUMENTS

halt      status
```

EXPLANATION

The halt instruction halts execution of the picture generator. The contents of the picture workspace and the value of status are available tn the visualizer executive.

Basic picture descriptions are automatically produced as part of the visualization description by the visualization description compiler. The algorithms for mapping C or ADA source code to picture descriptions are related to conventional language translation techniques. The following is an augmented production language specification for a portion of the visualization description compiler that produces picture descriptions for "structs" in the C programming language.

NOTE: Picture description code to be produced by a rule's semantic routine is indicated by the "→" symbol.

```
<structure> ::= {<elements>} <name> ;
            → templt struct_icon <name>
                 combine
                 frame
                 halt   OK_status

<elements> ::= <element> ;
                   → stack
          | <elements> <element> ;
                   → combine
                        stack
```

```
<element> ::= <character> | <integer>

<character> ::= char <name> [<integer>]
        → templt string_icon <name> <integer>

<integer> ::= int <name>
              → templt integer_icon <name>
```

As an example of picture description code generation, consider the following C language struct. This struct contains meteorological data pertaining to a city.

```
struct
        {
        char     cityname[32]    ;
        int      aveprecip       ;
        int      avetemp         ;
        int      lowtemp         ;
        int      hightemp        ;
        }
                 weatherdata     ;
```

The visualization description compiler defined by the above production rules would generate the following picture description code:

1. templt string_icon cityname 32
2. stack
3. templt integer_icon aveprecip
4. combine
5. stack
6. templt integer_icon avetemp
7. combine
8. stack
9. templt integer_icon lowtemp
10. combine
11. stack
12. templt integer_icon hightemp
13. combine
14. stack
15. templt struct_icon weatherdata
16. combine
17. frame
18. halt OK_status

The lines of code are numbered so that they can be referenced in Figures 4.2 through 4.18, which illustrate the operation of the picture generator in creating this graphical representation.

PICTURE WORKSPACE **PICTURE STACKS**

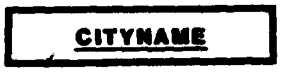


Figure 4.2 Picture Description Code—Line 1

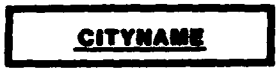


Figure 4.3 Picture Description Code—Line 2

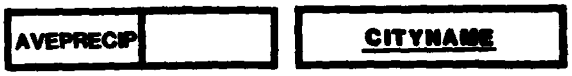


Figure 4.4 Picture Description Code—Line 3

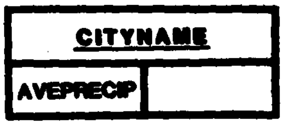


Figure 4.5 Picture Description Code—Line 4

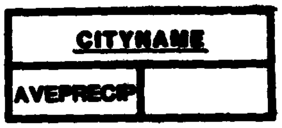


Figure 4.6 Picture Description Code—Line 5

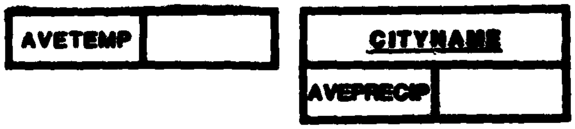


Figure 4.7 Picture Description Code—Line 6

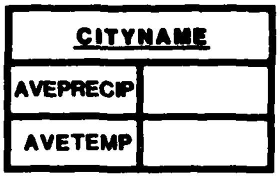


Figure 4.8 *Picture Description Code – Line 7*

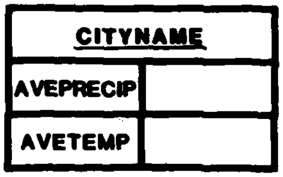


Figure 4.9 *Picture Description Code – Line 8*

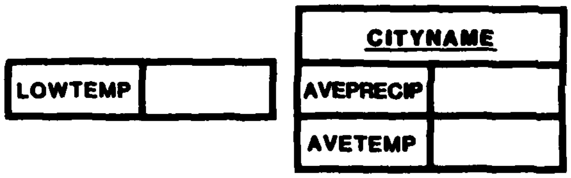


Figure 4.10 *Picture Description Code – Line 9*

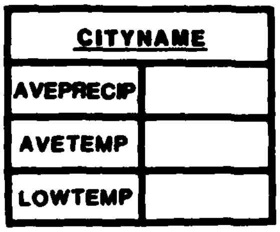


Figure 4.11 *Picture Description Code – Line 10*

4.3 The Visualization Description/Template Editor

Image production by the visualizer is based on rules embodied in the visualization description. A basic visualization for a program can be automatically generated by the visualization description compiler. However, that basic description often must be manually enhanced. The visualization description/template editor is the facility for performing manual enhancement.

As discussed in Section 4.2, the picture semantics section of the visualization description contains picture descriptions. A picture description defines the graphic appearance of one view of a program subject. Internally, picture descriptions are visualizer-executable programs that result in the production of the desired image. Picture descriptions instantiate and combine templates. A template is a type of procedure that produces a simple graphic image. Templates may accept arguments that determine their appearance.

The PV system user employs the visualization description/template editor for two tasks:

1. Creating special purpose templates.
2. Enhancing visualization descriptions by including templates in them.

These two tasks are performed through graphical interactions with the editor.

Templates tell the visualizer two things. First, they specify the appearance of a program subject during various stages in program execution. Second, they specify those variables or control structures in the program being visualized, upon which the template is based (*binding information*).

To make it easy for a user to create templates, the template editor provides an extensive set of painting commands for drawing templates. A set of commands is provided for binding the template that is drawn to variables in the program being visualized.

In addition, templates that the user creates can be catalogued for later retrieval and used as building blocks in the creation of new templates. This feature allows the user to create a set of templates for commonly used data structures. These can then act as template standards for the design of future templates.

It is important that the editor makes it easy to create simple templates and provides the capability to make any arbitrary template, no matter how complex. The catalogue of building block templates will provide a good foundation for making new templates easily. To further automate this activity, the editor uses its knowledge of how the building blocks bind to program variables to help automate the process of binding templates to programs.

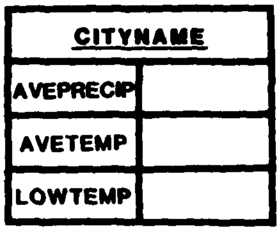


Figure 4.12 Picture Description Code—Line 11

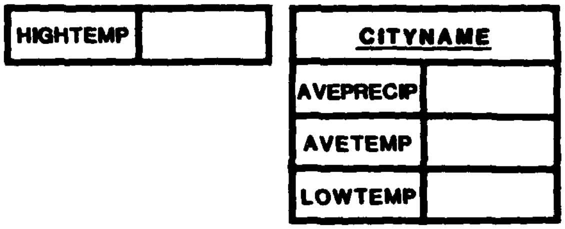


Figure 4.13 Picture Description Code—Line 12

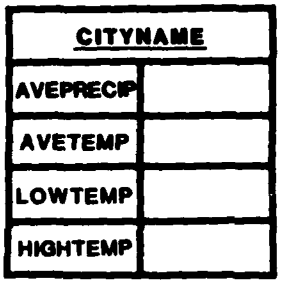


Figure 4.14 Picture Description Code—Line 13

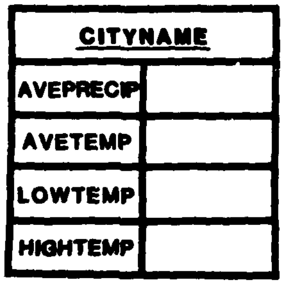


Figure 4.15 Picture Description Code—Line 14

For example, if a user wants to create a histogram chart, the editor will present the user with a set of various blank histogram layouts, from which the user will choose one. Then varying styles of elements to be placed on the chart will be presented. The user can arrange these as he pleases to design the template. Once the user has completed the graphical design of the template, he enters the binding mode. The editor uses its knowledge of the selected histogram elements to prompt the user for the variables each element is to be bound to. The editor highlights the appropriate element at each step.

4.3.1 Animation

Most of the templates the user will want to design will have some degree of animation associated with them. Two different types of animation are provided. The first type is the animation of a program variable going through some specified range. The animation is based on the value of the program variable. In this type of animation, it is necessary for the template designer to provide information about the range of acceptable values through which the template elements are to be viewed. Exception conditions also can be created with their own views. A typical example of this type of animation is the histogram chart. In this type of template, the size of each element is based upon the representative value of the program variable bound to it.

The second type of animation is not based on the range of a program variable, but on program states of activity. Animation is based on predicates associated with each template. The predicates are in terms of program states, program events, and data values.

To provide an easy way to create and edit template animation, an animation creation and viewing facility is designed into the editor. This facility lets a user manipulate the template elements to create animation. For example, the user can first create a background upon which the template activity will take place. The template elements can then be moved independently upon the background. Each change in the position of one or more of the template elements may be keyed to a program state.

4.3.2 Specifying Graphical Layout

The visualization description/template editor lets the user specify the initial graphical layout used by the visualizer. The user may also specify how the levels of detail are to be presented. In this way, the user controls the program visualization specification issues of composition and context discussed in Section 2.11.

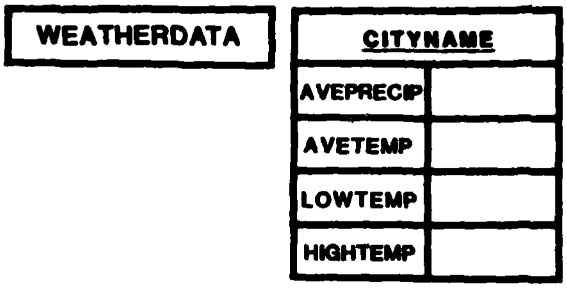


Figure 4.16 Picture Description Code – Line 15

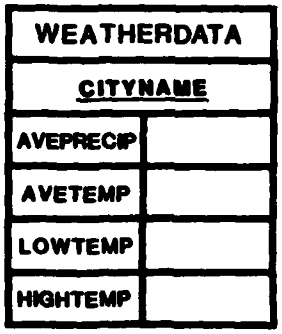


Figure 4.17 Picture Description Code – Line 16

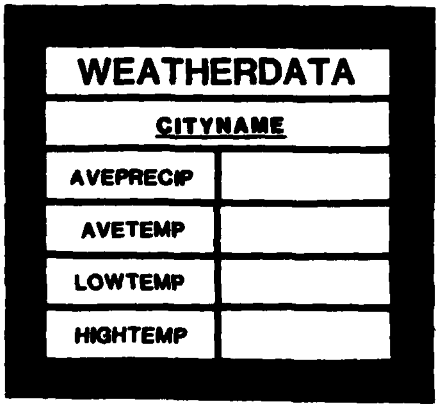


Figure 4.18 Picture Description Code – Line 17

The template editor also provides a mechanism for rearranging its own command menu layout. This feature lets the user substitute pictures or icons for commands usually represented by text.

4.3.3 Template Code

Interactive graphical specification of animation as discussed in Section 4.3.1 cannot fully exploit the visualization potential of a PV system. For example, suppose a user wants to create a visualization of an algorithm for converting a bit map representation of an image into a run-length encoded representation of the image. The algorithm to compress images into a run-length encoded form generates only two major variables. It is the relationship between these variables over time that is meaningful. All animation capabilities discussed so far, however, are based directly on the simple magnitude of program variables, program states, and program events. To visualize an algorithm in which complex relationships of program variables, states, and events are of principal interest, the user must be able to write explicit procedures that translate input variables into visualization directives. Template code provides that facility.

The user should not be required to learn a new language to write procedures in template code. Hence, the language used will be an extension of the language upon which PV is based. The extensions will be designed to provide an easy method of interfacing with both the graphics of the template and the program being visualized. One extension will allow the user's template code to reference variables in the program under study, through an external variable construct. Other extensions will provide a convenient method for controlling animation.

APPENDIX A

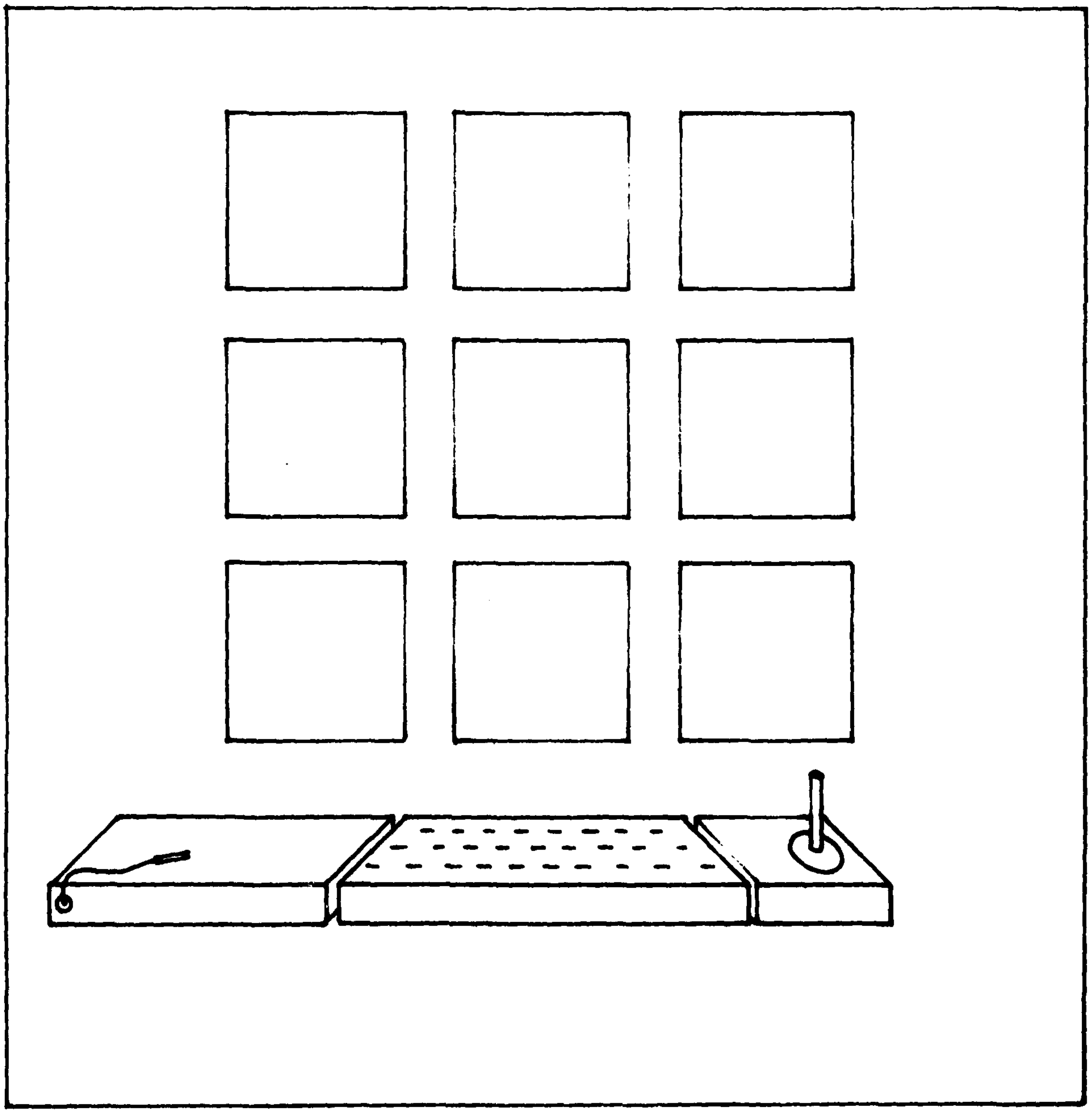
EXAMPLE PROGRAM VISUALIZATIONS

Graphic designers have an important role in the Program Visualization Project. They bring approaches and visualizations to the project that programmers and engineers do not. Part of the designer's task is to put him/her self in the context of a user or client; this facility makes the designer more sensitive to the needs and sensibilities of the end user. Designers also have experience in handling systems with many parameters relating to human functions; coordinating and making sense out of these various parameters are skills that designers can contribute to the project. This experience gives the designer effective tools for the use of rich visual heirarchies, which contribute enhanced semantic depth to the visualizations without adding confusion.

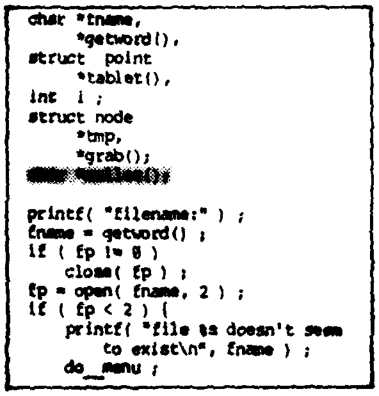
This section of the proposal is an attempt to show a few of the avenues that can be explored in the quest for worthwhile program visualizations. This set of directions is by no means exhaustive or complete; it is only a sampler of possibilities. Our concern at this point is not so much with the actual means of implementing these methods and techniques with particular hardware and software, but rather a specification of needs. Others have explored, classified and rated some of the existing program visualation techniques. A bibliography of some of these publications is included.

We have used existing C code for concrete examples of these possible techniques.

Overall approach:
A multiscreen workstation with various input devices: keyboard, joystick, knobs and dials, a digitizing pad. Six or nine screens allow for multiple displays and relationships between displays. For example, the middle screen could show the current program module (function, subroutine, etc.) being worked on, the screen to the left the module that calls it and the screen to the right the modules it calls. The top row of screens could show variable maps for each module (working, calling or called) while the bottom row shows status, command menus and other information (filenames, history, main memory requirements, CPU seconds used, etc.). Color could be used to link the same names from screen to screen. A similar approach has been explored at Xerox Palo Alto Research Center. The Smalltalk system of multiple overlapping windows based on the idea of sheets of paper on a desktop has been described by Teitelman [T1]. This technique of multi-windows on one screen has its advantages but could perhaps use an overall organization to prevent the "desk" from getting too cluttered.



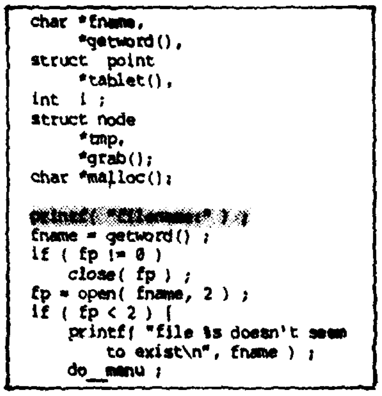
As the programmer needed to shift his/her attention up or down the working module, a reverse video bar could highlight the current line of code. Another possibility for the use of the reverse bar could be showing the line currently being executed as the programmer watches the program in the process of execution. The joystick could control the speed of execution so the user could step through the program as fast or slow as wanted.



```
char *fname,
     *getword(),
struct  point
     *tablet(),
int  i ;
struct node
     *tmp,
     *grab();
char *malloc();

printf( "filename:" ) ;
fname = getword() ;
if ( fp != 0 )
    close( fp ) ;
fp = open( fname, 2 ) ;
if ( fp < 2 ) {
    printf( "file %s doesn't seem
        to exist\n", fname ) ;
    do__menu ;
```

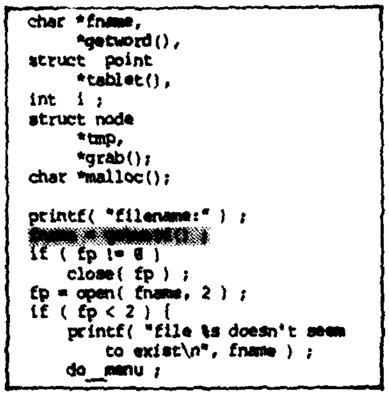
1



```
char *fname,
     *getword(),
struct  point
     *tablet(),
int  i ;
struct node
     *tmp,
     *grab();
char *malloc();

printf( "filename:" ) ;
fname = getword() ;
if ( fp != 0 )
    close( fp ) ;
fp = open( fname, 2 ) ;
if ( fp < 2 ) {
    printf( "file %s doesn't seem
        to exist\n", fname ) ;
    do__menu ;
```

2

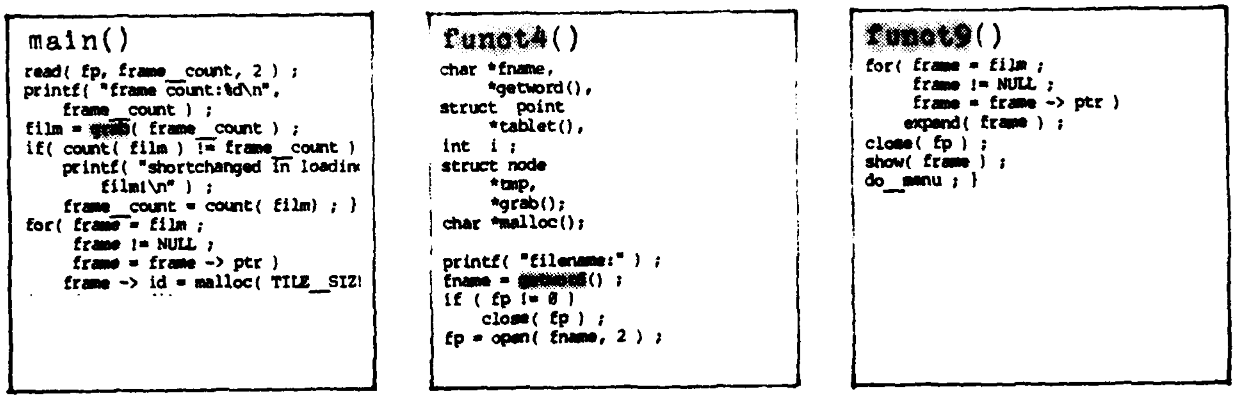


```
char *fname,
     *getword(),
struct  point
     *tablet(),
int  i ;
struct node
     *tmp,
     *grab();
char *malloc();

printf( "filename:" ) ;
fname = getword() ;
if ( fp != 0 )
    close( fp ) ;
fp = open( fname, 2 ) ;
if ( fp < 2 ) {
    printf( "file %s doesn't seem
        to exist\n", fname ) ;
    do__menu ;
```

3

If the programmer wants to shift to another module the screen would scroll or the modules could shift screens left-to-right or right-to-left. If the programmer got to the top level of code the information appearing next on the left screen would be a functional specification for that code; if at the bottom level, the next thing appearing on the right would be assembly or machine code. Continuing to the right, circuit paths would appear, executing the machine code.



1

main()

```
read( fp, frame__count, 2 ) ;
printf( "frame count:%d\n",
    frame__count ) ;
film = grab( frame__count ) ;
if( count( film ) != frame__count )
    printf( "shortchanged in loadin
        film!\n" ) ;
    frame__count = count( film) ; }
for( frame = film ;
     frame != NULL ;
     frame = frame -> ptr )
    frame -> id = malloc( TILE__SIZ
```

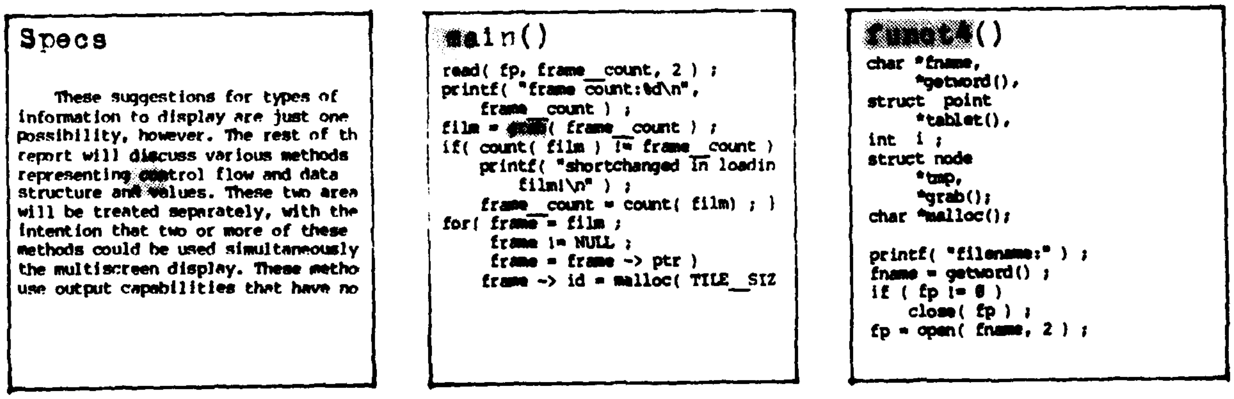
funct4()

```
char *fname,
     *getword(),
struct  point
     *tablet(),
int  i ;
struct node
     *tmp,
     *grab();
char *malloc();

printf( "filename:" ) ;
fname = getword() ;
if ( fp != 0 )
    close( fp ) ;
fp = open( fname, 2 ) ;
```

funct9()

```
for( frame = film ;
     frame != NULL ;
     frame = frame -> ptr )
   expand( frame ) ;
close( fp ) ;
show( frame ) ;
do__menu ; }
```



2

Specs

These suggestions for types of
information to display are just one
possibility, however. The rest of th
report will discuss various methods
representing control flow and data
structure and values. These two area
will be treated separately, with the
intention that two or more of these
methods could be used simultaneously
the multiscreen display. These metho
use output capabilities that have no

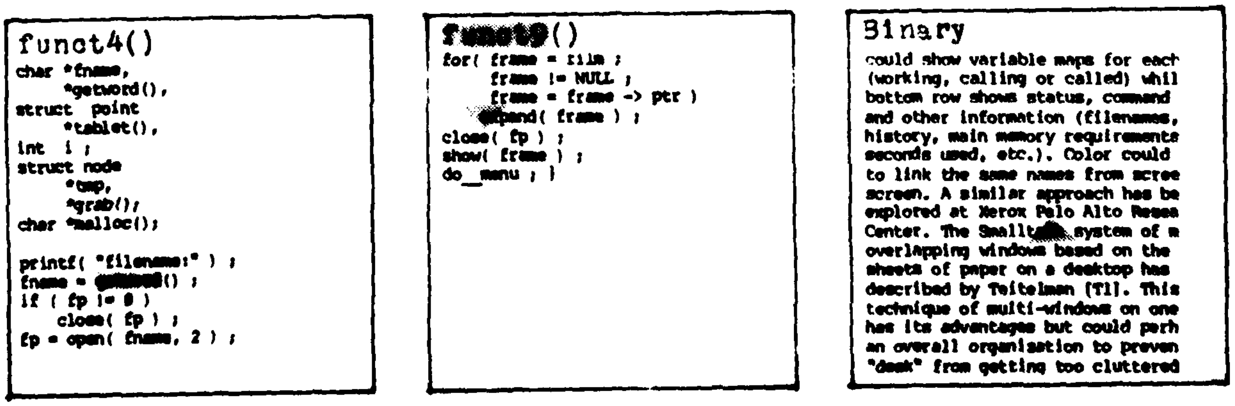
main()

```
read( fp, frame__count, 2 ) ;
printf( "frame count:%d\n",
    frame__count ) ;
film = grab( frame__count ) ;
if( count( film ) != frame__count )
    printf( "shortchanged in loadin
        film!\n" ) ;
    frame__count = count( film) ; }
for( frame = film ;
     frame != NULL ;
     frame = frame -> ptr )
    frame -> id = malloc( TILE__SIZ
```

funct4()

```
char *fname,
     *getword(),
struct  point
     *tablet(),
int  i ;
struct node
     *tmp,
     *grab();
char *malloc();

printf( "filename:" ) ;
fname = getword() ;
if ( fp != 0 )
    close( fp ) ;
fp = open( fname, 2 ) ;
```



3

funct4()

```
char *fname,
     *getword(),
struct  point
     *tablet(),
int  i ;
struct node
     *tmp,
     *grab();
char *malloc();

printf( "filename:" ) ;
fname = getword() ;
if ( fp != 0 )
    close( fp ) ;
fp = open( fname, 2 ) ;
```

funct9()

```
for( frame = film ;
     frame != NULL ;
     frame = frame -> ptr )
   expand( frame ) ;
close( fp ) ;
show( frame ) ;
do__menu ; }
```

Binary

could show variable maps for each
(working, calling or called) whil
bottom row shows status, command
and other information (filenames,
history, main memory requirements
seconds used, etc.). Color could
to link the same names from scree
screen. A similar approach has be
explored at Xerox Palo Alto Resea
Center. The Smalltalk system of m
overlapping windows based on the
sheets of paper on a desktop has
described by Teitelman [T1]. This
technique of multi-windows on one
has its advantages but could perh
an overall organisation to preven
"desk" from getting too cluttered

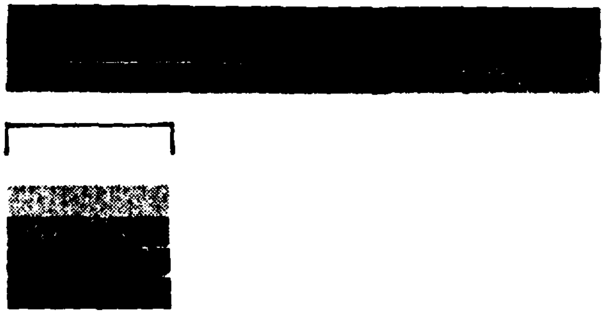
These suggestions for types of information to display are just one possibility, however. The rest of this report will discuss various methods of representing control flow and data structure and values. These two areas will be treated separately, with the intention that two or more of these methods could be used simultaneously in the multiscreen display. These methods use output capabilities that have not been well used (or used at all) up to this time. These capabilities include:

multicolors and grey levels
reverse video and blinking
size changes in typeface
bold and italic typefaces
different type fonts

For this report we have limited the range of these tools for perceptual as well as technical reasons. We will not necessarily use all of the possible variations of one tool, but show this list as a menu of clear and available variations.

3 colors plus black and white
6 grey levels including black and white
reverse video with blinking
3 sizes of type
type styles: bold roman u/lc
medium roman u/lc
medium italic u/lc
a single typeface

A study has shown that the range of color for letters is limited, but this can be expanded by using color bars with black or white letters on the bars.



6 point Times Roman
7 point Times Roman
8 point Times Roman
9 point Times Roman
10 point Times Roman
11 point Times Roman
12 point Times Roman
14 point Times Roman

Times Roman
Times Roman Italic
Times Roman Bold
Times Roman Bold Italic

Melior
Memphis
Optima
Palatino
Univers 55

Visualizing C code, possibility 1

Prettyprinting standards:

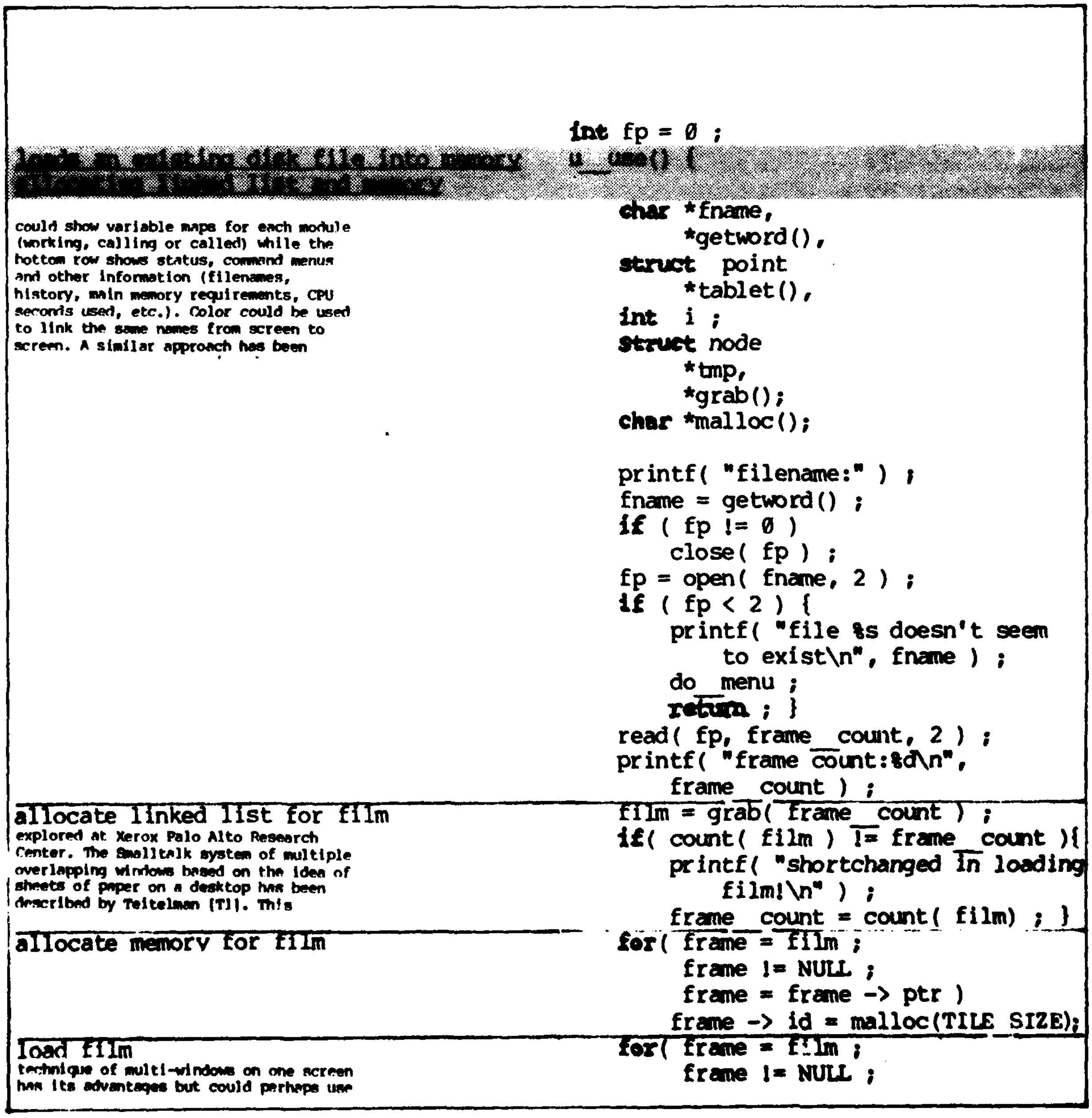
1. 4 character indents
2. no more (and possibly less) than one statement per line
3. "if", "for" and "while" conditions on their own lines, with "if", "for", "while" and "else" blocks on their own lines
4. single spaces (1 character) between all words and symbols
5. open and close brackets in same place each time, let the indent show the structure
6. left justify variable names in declarations and definitions

```
int fp = 0 ;
u_use() {
    char *fname,
         *getword(),
    struct  point
         *tablet(),
    int  i ;
    struct node
         *tmp,
         *grab();
    char *malloc();

    printf( "filename:" ) ;
    fname = getword() ;
    if ( fp != 0 )
        close( fp ) ;
    fp = open( fname, 2 ) ;
    if ( fp < 2 ) {
        printf( "file %s doesn't seem
            to exist\n", fname ) ;
        do_menu ;
        return ; }
    read( fp, frame_count, 2 ) ;
    printf( "frame count:%d\n",
        frame_count ) ;
    film = grab( frame_count ) ;
    if( count( film ) != frame_count ) {
        printf( "shortchanged in loading
            film!\n" ) ;
        frame_count = count( film) ; }
    for( frame = film ;
         frame != NULL ;
         frame = frame -> ptr )
        frame -> id = malloc( TILE_SIZE ) ;
    for( frame = film ;
         frame != NULL ;
         frame = frame -> ptr )
        expand( frame ) ;
    close( fp ) ;
    show( frame ) ;
    do_menu ; }
```

These prettyprinting standards could be used in a larger context. The major task of this context would be to separate and relate comments and code by putting each in its own column, 40 characters wide. The main title of the program could be in a larger size of type while each module (function) title is in a full screen-width reverse video bar, shown by the grey bar in the figure. Code and comments are linked by rules across the screen and have the same indentation structure. Code is in roman with bold keywords (if, else, while, for, int, char, etc.). The comments could be structured in different sizes to show levels of various types:

historical
authority
formality
anecdotal



loads an existing disk file into memory
allocating linked list and memory

could show variable maps for each module (working, calling or called) while the bottom row shows status, command menus and other information (filenames, history, main memory requirements, CPU seconds used, etc.). Color could be used to link the same names from screen to screen. A similar approach has been

```
int fp = 0 ;
u_use() {
    char *fname,
        *getword(),
    struct  point
        *tablet(),
    int  i ;
    struct node
        *tmp,
        *grab();
    char *malloc();

    printf( "filename:" ) ;
    fname = getword() ;
    if ( fp != 0 )
        close( fp ) ;
    fp = open( fname, 2 ) ;
    if ( fp < 2 ) {
        printf( "file %s doesn't seem
            to exist\n", fname ) ;
        do_menu ;
        return ; }
    read( fp, frame_count, 2 ) ;
    printf( "frame count:%d\n",
        frame_count ) ;
```

allocate linked list for film

explored at Xerox Palo Alto Research Center. The Smalltalk system of multiple overlapping windows based on the idea of sheets of paper on a desktop has been described by Teitelman [T1]. This

```
    film = grab( frame_count ) ;
    if( count( film ) != frame_count ){
        printf( "shortchanged in loading
            film!\n" ) ;
        frame_count = count( film) ; }
```

allocate memory for film

```
    for( frame = film ;
         frame != NULL ;
         frame = frame -> ptr )
        frame -> id = malloc(TILE_SIZE);
```

load film

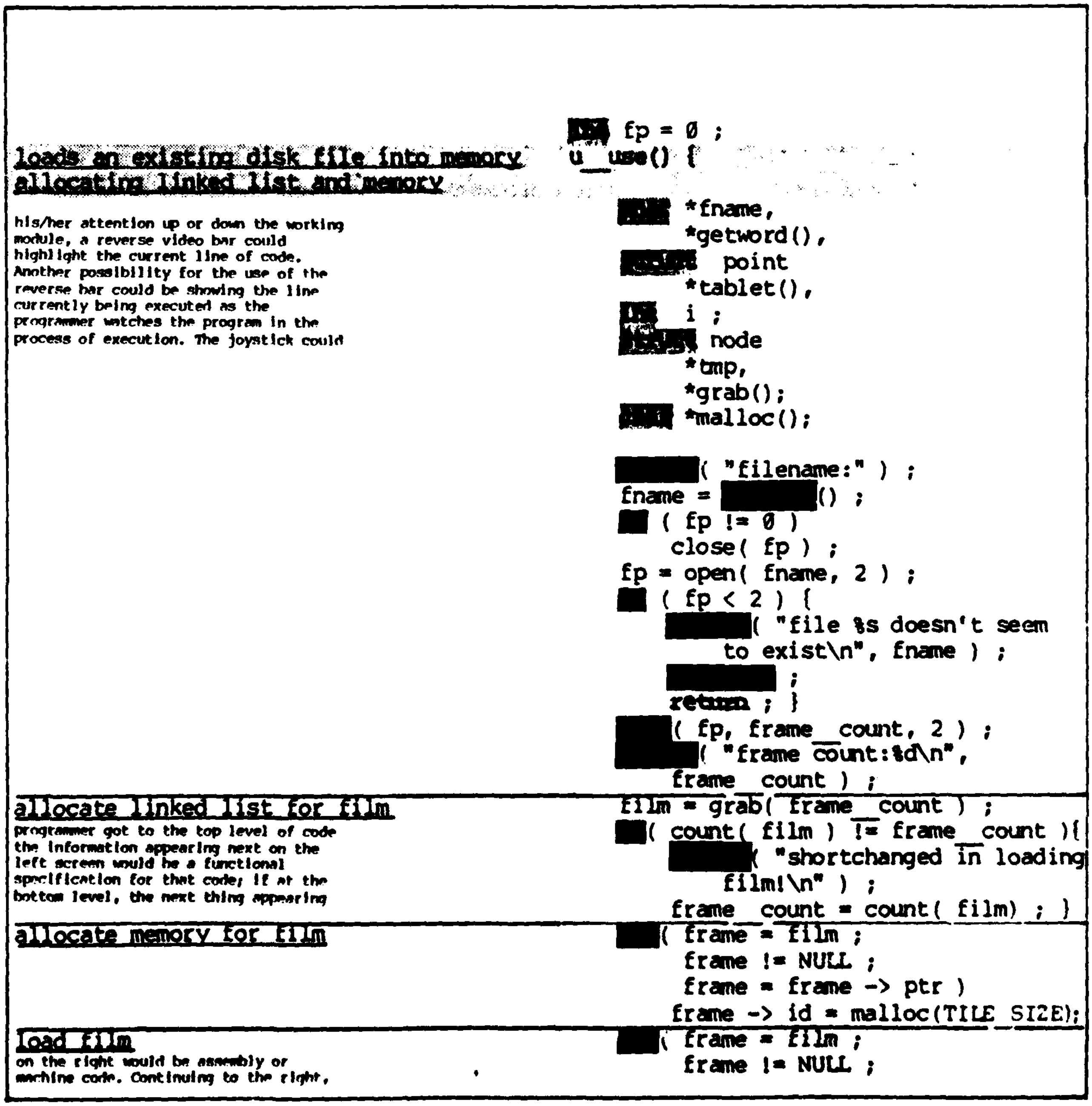
technique of multi-windows on one screen has its advantages but could perhaps use

```
    for( frame = film ;
         frame != NULL ;
```

Another variation on possibility 1 uses color bars to indicate type of command, for example:

etc.

Still another possibility could be to use color to differentiate between declarations and definitions, which look similar at first glance.



loads an existing disk file into memory allocating linked list and memory

his/her attention up or down the working module, a reverse video bar could highlight the current line of code. Another possibility for the use of the reverse bar could be showing the line currently being executed as the programmer watches the program in the process of execution. The joystick could

```
[redacted] fp = 0 ;
u_use() {

    [redacted] *fname,
         *getword(),
    [redacted] point
         *tablet(),
    [redacted] i ;
    [redacted] node
         *tmp,
         *grab();
    [redacted] *malloc();

    [redacted]( "filename:" ) ;
    fname = [redacted]() ;
    [redacted] ( fp != 0 )
       close( fp ) ;
    fp = open( fname, 2 ) ;
    [redacted] ( fp < 2 ) {
       [redacted]( "file %s doesn't seem
           to exist\n", fname ) ;
       [redacted] ;
       return ; }
    [redacted]( fp, frame__count, 2 ) ;
    [redacted]( "frame count:%d\n",
       frame__count ) ;
```

allocate linked list for film

programmer got to the top level of code the information appearing next on the left screen would be a functional specification for that code; if at the bottom level, the next thing appearing

```
    film = grab( frame__count ) ;
    [redacted]( count( film ) != frame__count ){
       [redacted]( "shortchanged in loading
           film!\n" ) ;
       frame__count = count( film) ; }
```

allocate memory for film

```
    [redacted]( frame = film ;
         frame != NULL ;
         frame = frame -> ptr )
       frame -> id = malloc(TILE_SIZE);
```

load film

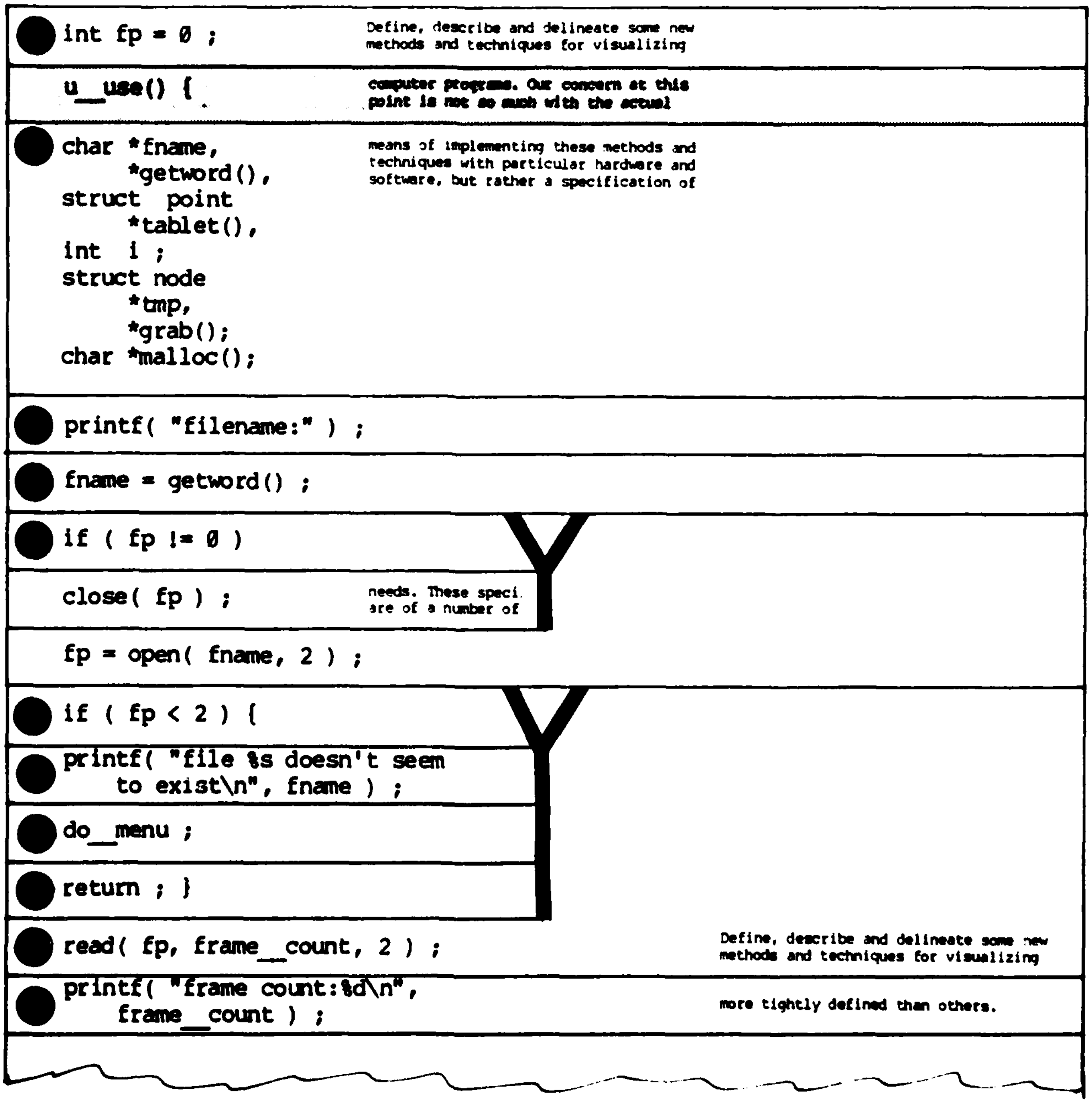
on the right would be assembly or machine code. Continuing to the right,

```
    [redacted]( frame = film ;
         frame != NULL ;
```

Visualizing C code, possibility 2
Modified Nassi-Schneidermann diagrams

N-S diagrams in their present form do not take advantage of the richness possible in graphic symbols and signs. Some enhancements have been suggested in an article by Frei [F2]. One possible avenue for exploration could involve keeping the basic form of a box made up of smaller boxes for individual commands, but with the following changes:

1) Use a heavier rule weight to show control structures; a 3-sided box for loops and a Y-shape for decisions. The loop signal could enclose all commands that are a part of the loop. In the case of the "for" loop, there is an initialization which only happens once, that is, not in the loop; but conceptually and code-wise is an integral part of the loop. The shape of the top part of the decision symbol could always stay the same, regardless of the shape of the box, so that it may be clearly and quickly recognized.



```
int fp = 0 ;
u_use() {
char *fname,
     *getword(),
struct  point
     *tablet(),
int  i ;
struct node
     *tmp,
     *grab();
char *malloc();
printf( "filename:" ) ;
fname = getword() ;
if ( fp != 0 )
close( fp ) ;
fp = open( fname, 2 ) ;
if ( fp < 2 ) {
printf( "file %s doesn't seem
    to exist\n", fname ) ;
do__menu ;
return ; }
read( fp, frame__count, 2 ) ;
printf( "frame count:%d\n",
    frame__count ) ;
```

Define, describe and delineate some new methods and techniques for visualizing
computer programs. Our concern at this point is not so much with the actual
means of implementing these methods and techniques with particular hardware and software, but rather a specification of
needs. These speci. are of a number of
Define, describe and delineate some new methods and techniques for visualizing
more tightly defined than others.

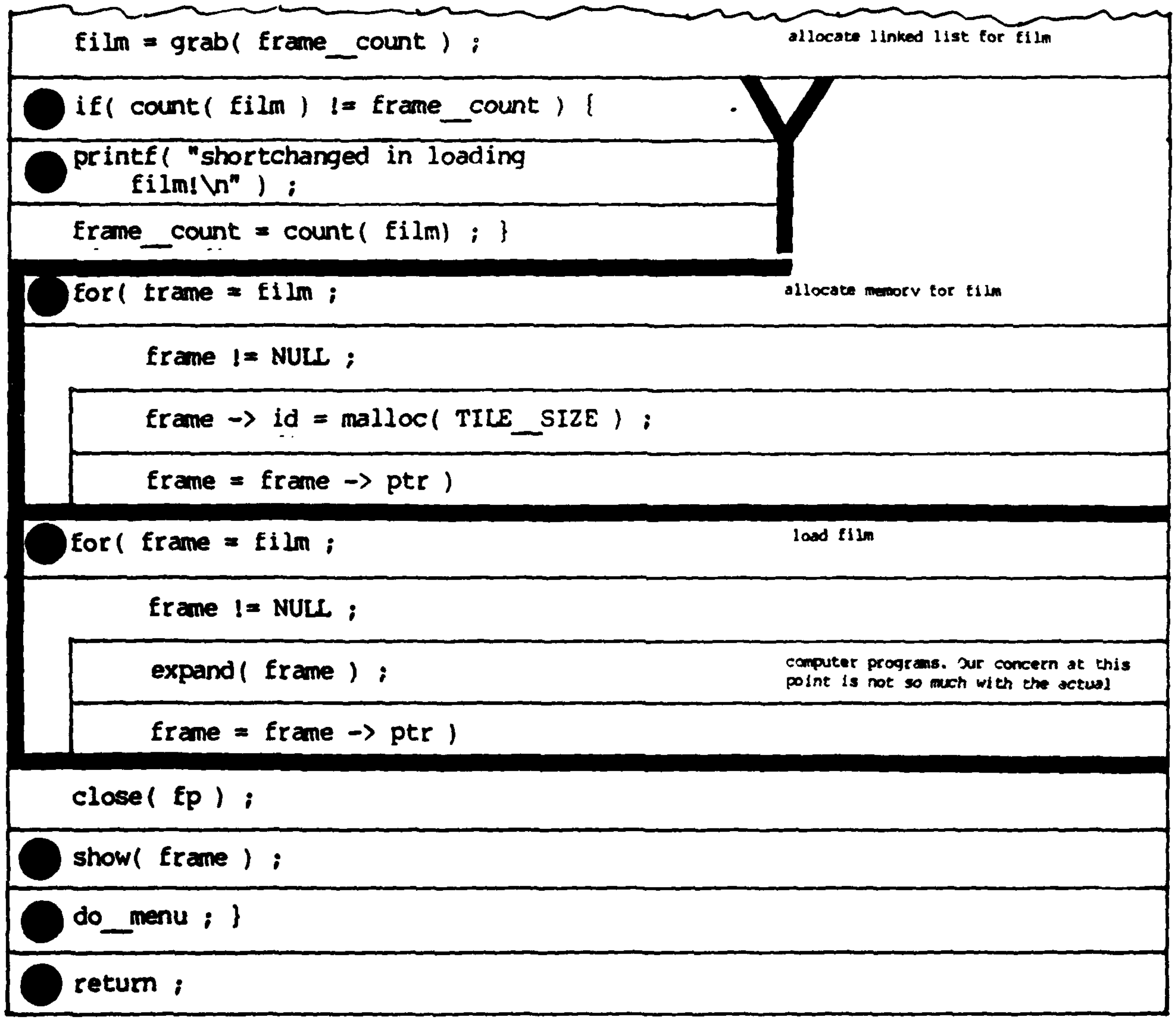
2) One problem with N-S diagrams is the space limitation, especially for comments. This could be easily taken care of by considering each comment space as a window with a scroll underneath. Other methods could be variations on the Smalltalk overlapping windows or the SDMS page-turning analogue [B2]. Some possible levels of meaning in the comments have been enumerated in the section on possiblity 1.

The boxes that contain commands and comments could an identifying symbol or color. For example:

● input/output
● control keywords
● declaration keywords

A grid could be established to determine position and size of type and symbols.

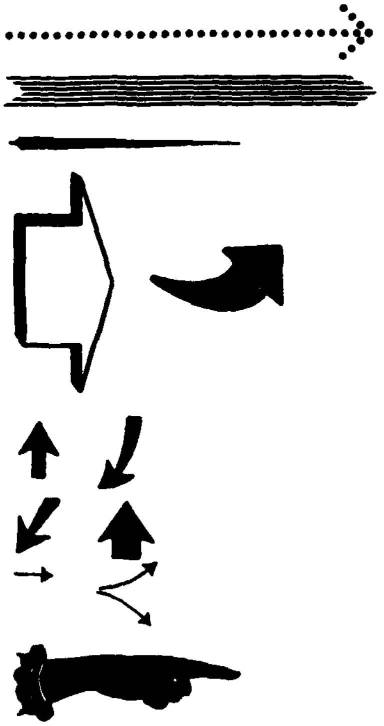
Note: It might be possible to build a compiler that compiled Modified N-S diagrams into machine code, reading the diagram along with the verbal part of the code.

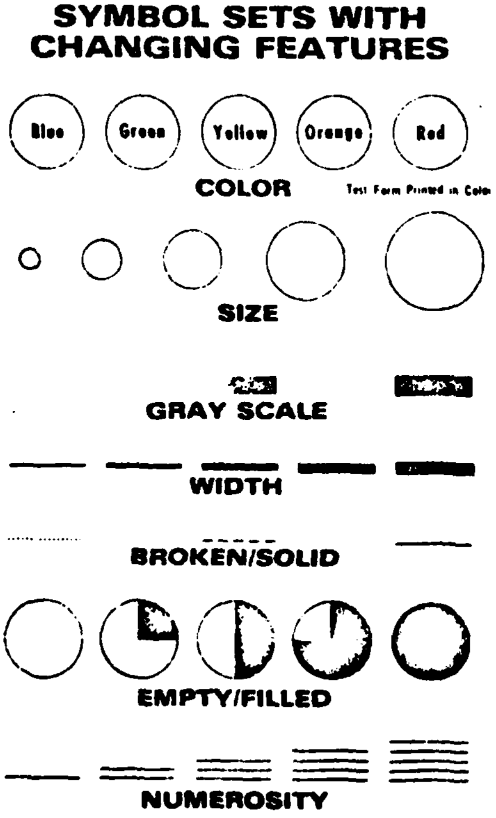

allocate linked list for film
allocate memorv for film
load film
computer programs. Our concern at this
point is not so much with the actual

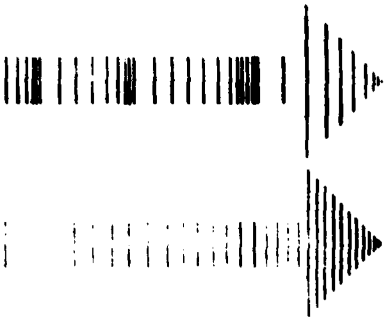
```
  film = grab( frame__count ) ;
● if( count( film ) != frame__count ) {
● printf( "shortchanged in loading
      film!\n" ) ;
  frame__count = count( film) ; }
● for( frame = film ;
        frame != NULL ;
        frame -> id = malloc( TILE__SIZE ) ;
        frame = frame -> ptr )
● for( frame = film ;
        frame != NULL ;
        expand( frame ) ;
        frame = frame -> ptr )
  close( fp ) ;
● show( frame ) ;
● do__menu ; }
● return ;
```

Visualizing C code, possibility 3
Modified Flow Charts

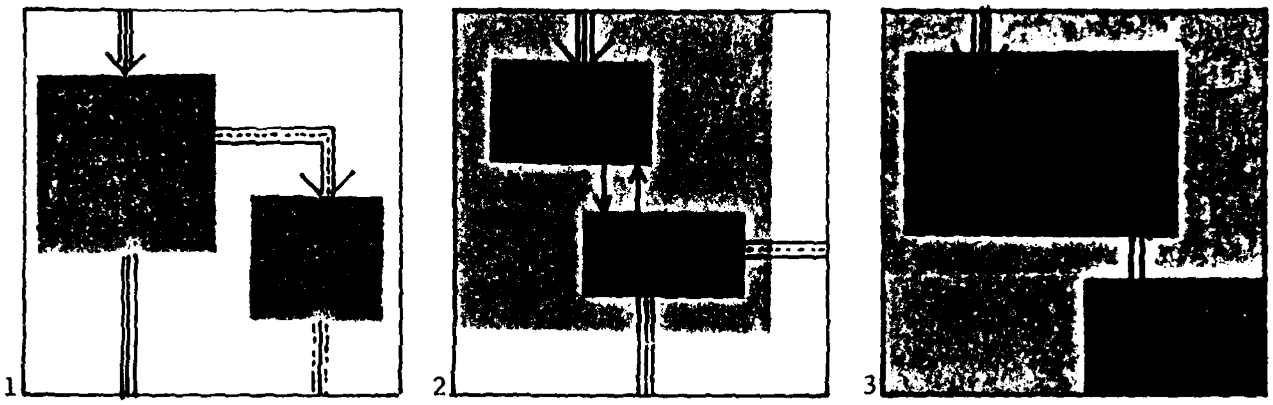
Flow charts, like N-S diagrams, do not take advantage of possible visual richness and variety. If the visual vocabulary is expanded to include different line weights and types, colors, symbols and even three dimensions, flow charts could contain more information with more relationships between pieces of information. A few suggestions of an expanded usual vocabulary are listed here. Animation is a powerful tool; for example, the speed of an event can sometimes be more informative than its shape or color.



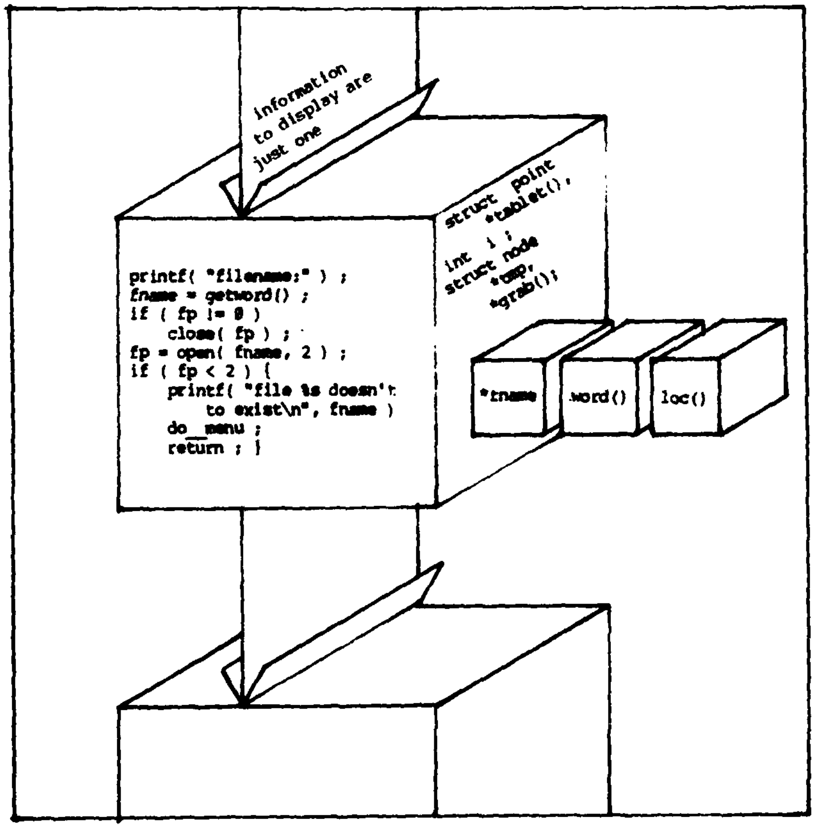




The SDMS concept [B2] of zooming in on an object to get a closer look at details could be applied to flowcharts. A user could get an overall view of a program, then use a joystick to zoom in on a particular node or cluster of nodes and have details appear, keeping no more than a certain amount of complexity on the screen at any one time. Automatic graphic adjustment could be made to convert symbolism from one conceptual level to another as the user zooms in and out.



One way of incorporating some of these elements into modified flowcharts is by using three instead of two dimensions, creating a flow construction. Code in some form would be on one face, comments on another, module history on another and other information on yet another (size, run-time, etc.). Background color of the node would indicate function, level of nesting or grouping. The programmer could turn the nodes and move through the three-dimensional flow construction. The construction could be enhanced by positioning related pieces of information around the node. The node's "gravity" and orientation would determine spacing and placement of the related information. Global information could be farther away from the node than local information. The arrows connecting nodes contain information passed to them. The user could literally "get a new view" of his/her program that had never been seen before. This could help in determining new relationships and connections between parts of the program.



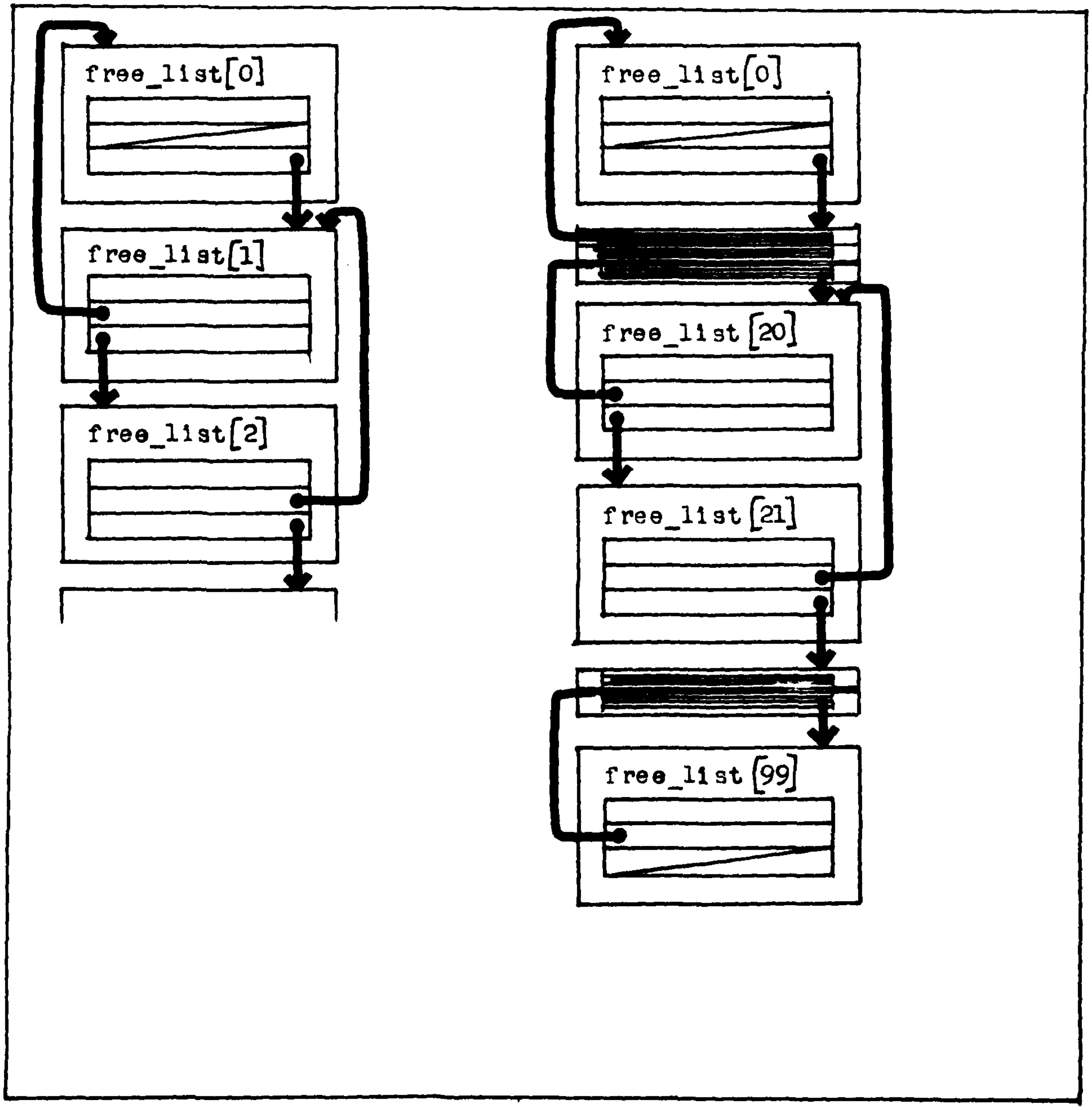
```
printf( "filename:" ) ;
fname = getword() ;
if ( fp != 0 )
    close( fp ) ;
fp = open( fname, 2 ) ;
if ( fp < 2 ) {
    printf( "file %s doesn't
        to exist\n", fname )
    do_menu ;
    return ; }
```

```
struct  point
        *tablet(),
int  i ;
struct node
        *tmp,
        *grab();
```

Visualizing C code, possibility 4
Data representation, variable maps and constructions

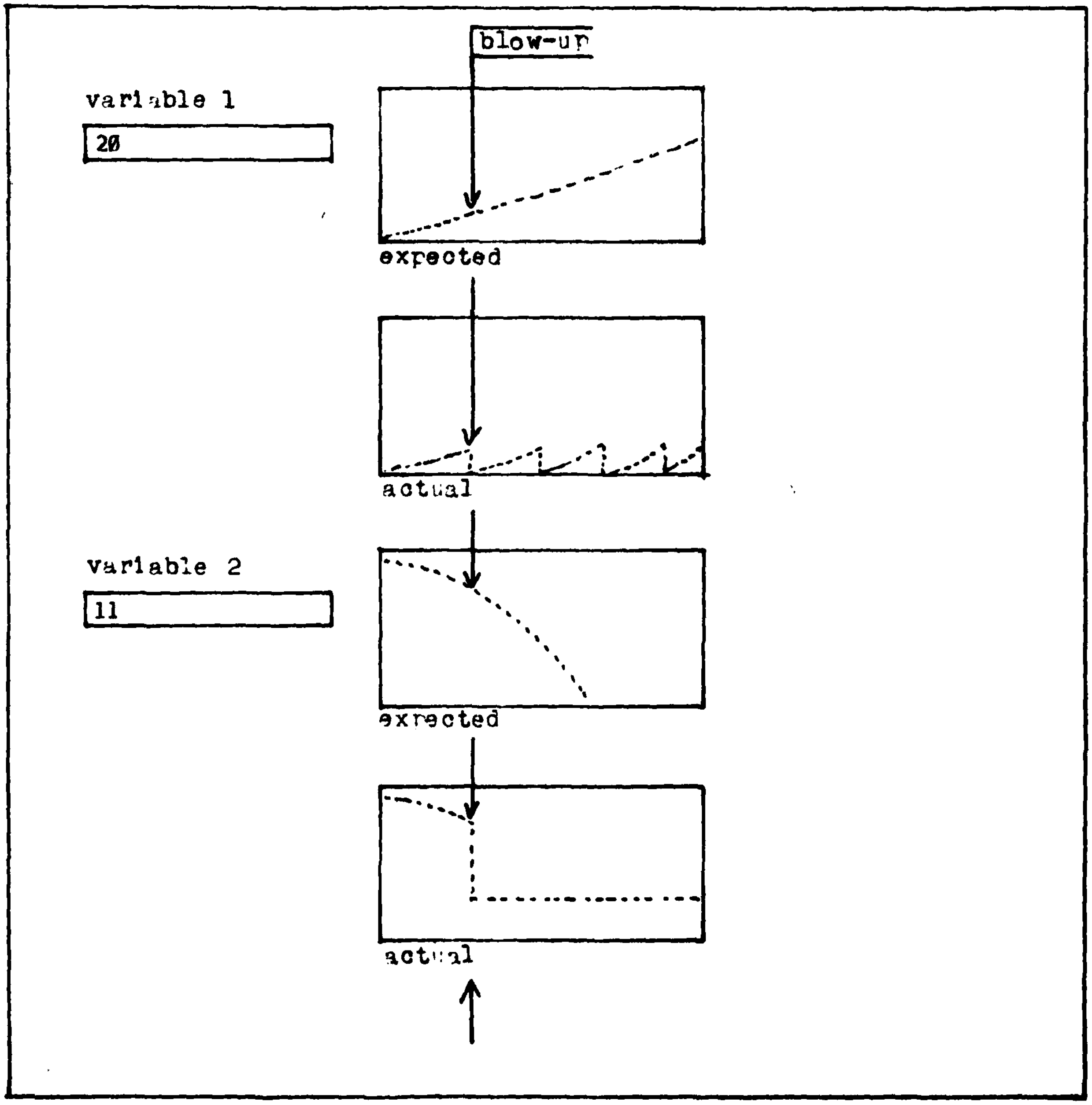
Programmers often need to see relationships between values of variables. A variable map coupled with a program display provides a complete picture of the relationship of the process of the program with the structure and content of the data. The standard method of showing data is with boxes labelled with a variable name. This can be enhanced with lineweights and styles, colors, grey values and animation. The methods of showing complex data representations, arrays of structures for instance, should clearly show that one piece of information is a part of another. Information about a variable that could be a part of the display includes:

name
type
value
space occupied
address



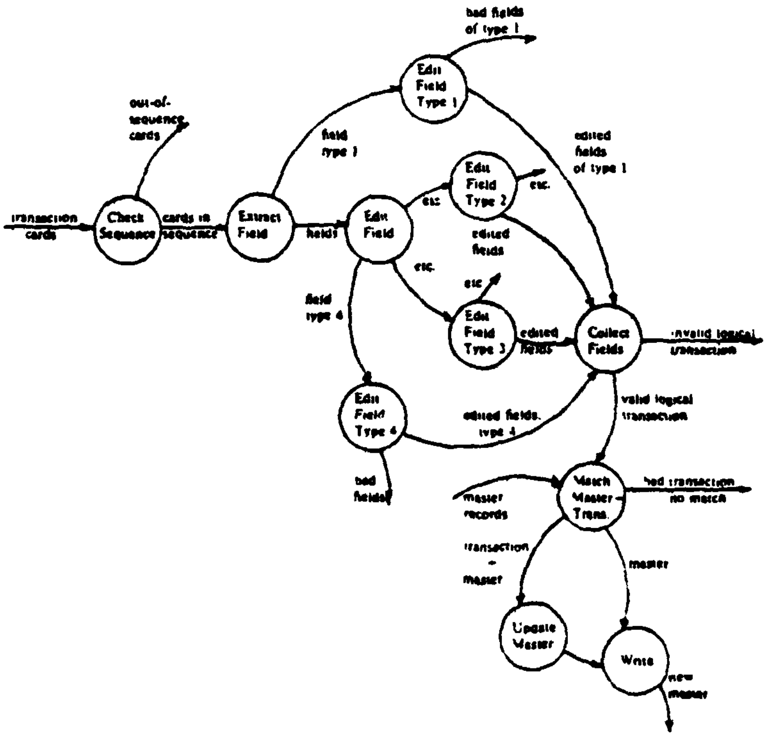
Each variable box could be considered as a window, with more information about the variable available by scrolling. Care should be taken to assure that the data representation (variable names an diagrams) be as clear as the data structure (the more abstract and general description).

Watching the values of variables change as a program runs is a valuable key to understanding the function and structure of the program, as well as providing a powerful debugging tool. The variable map could be animated, like a digital speedometer or gauge. It might be valuable, however, to include a graphical display of the change in the variable's value to compare expected results with actual results. Fitter [F1] mentions redundancy as a tool, that is, showing the same information in different forms simultaneously. Showing variable values alphanumerically and graphically is one application of this technique.

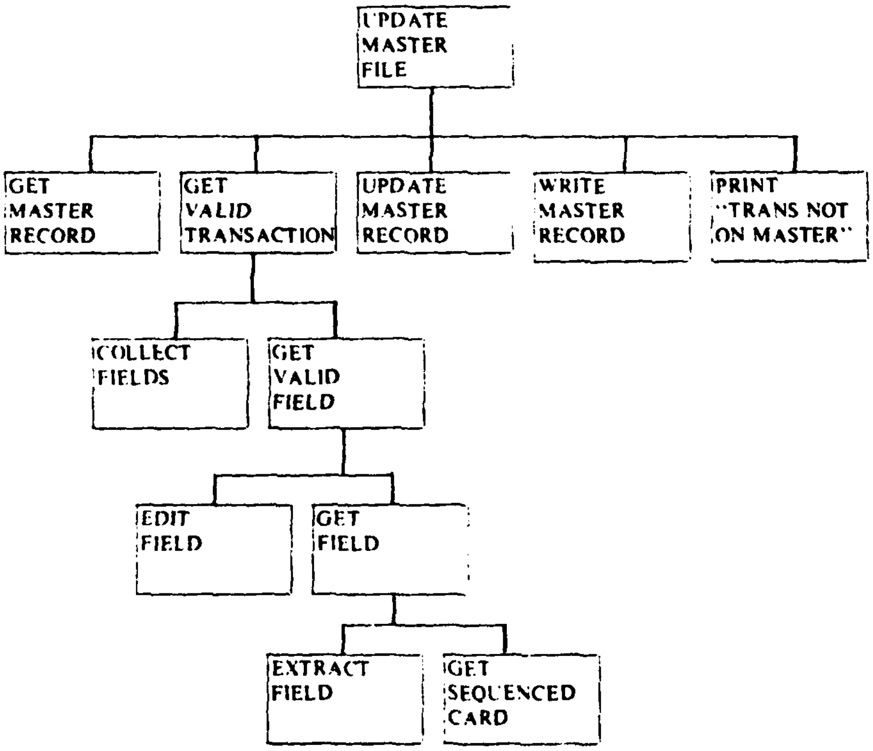


The final portion of this section consists of some images that have bearing on this problem of program visualization. These images come from many sources; we include them with notes on their appropriate features.

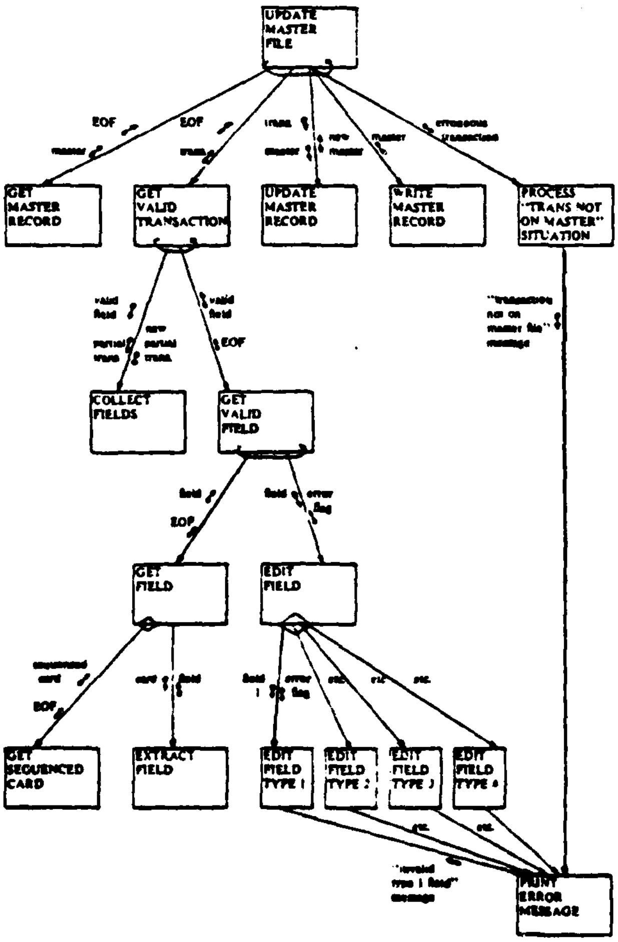
Typical data flow diagram



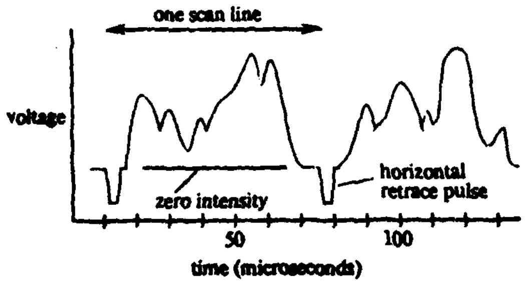
Typical HIPO hierarchy chart



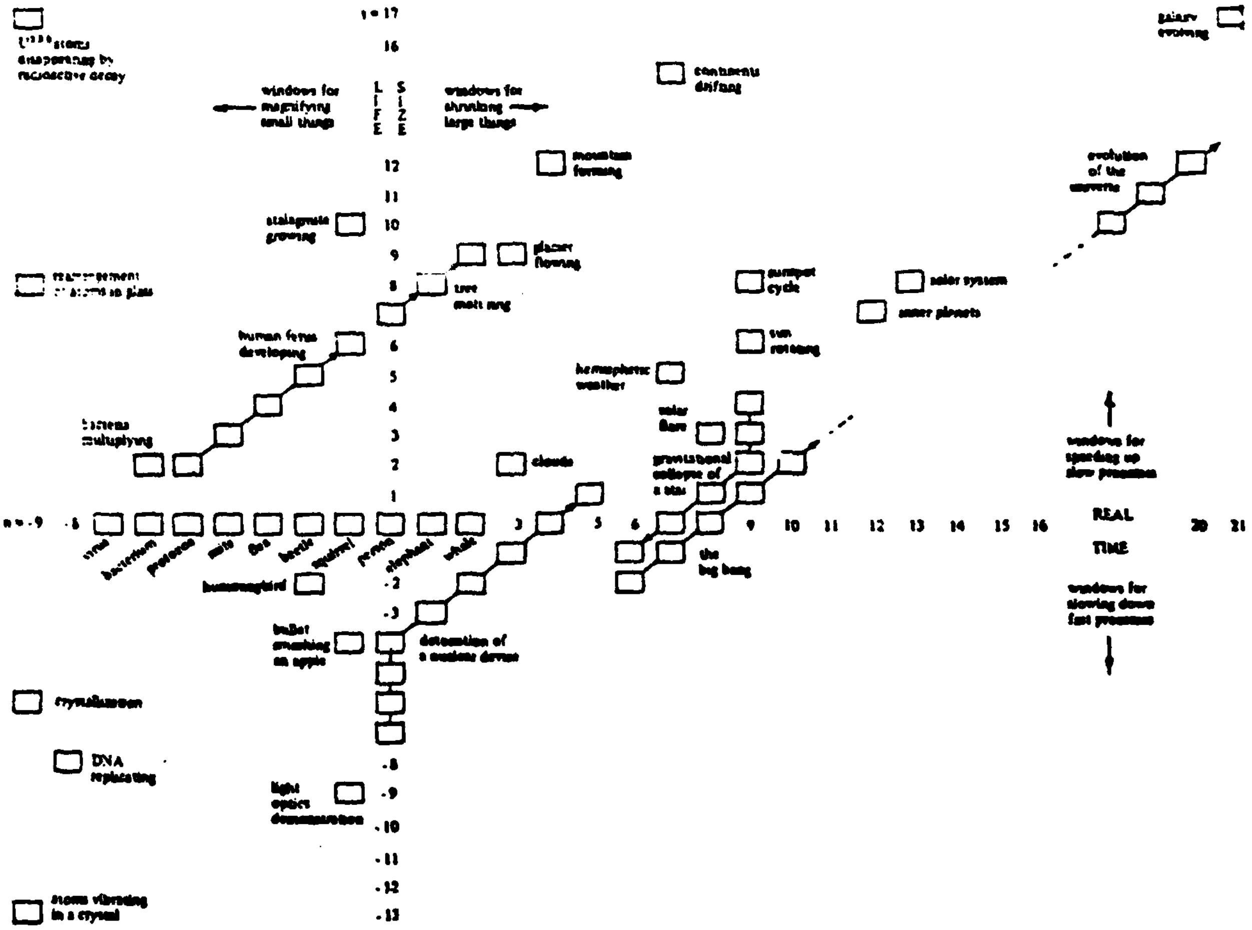
Typical structure chart



Scan line, seen usually as a variety of grey values, shown as a monoline graph

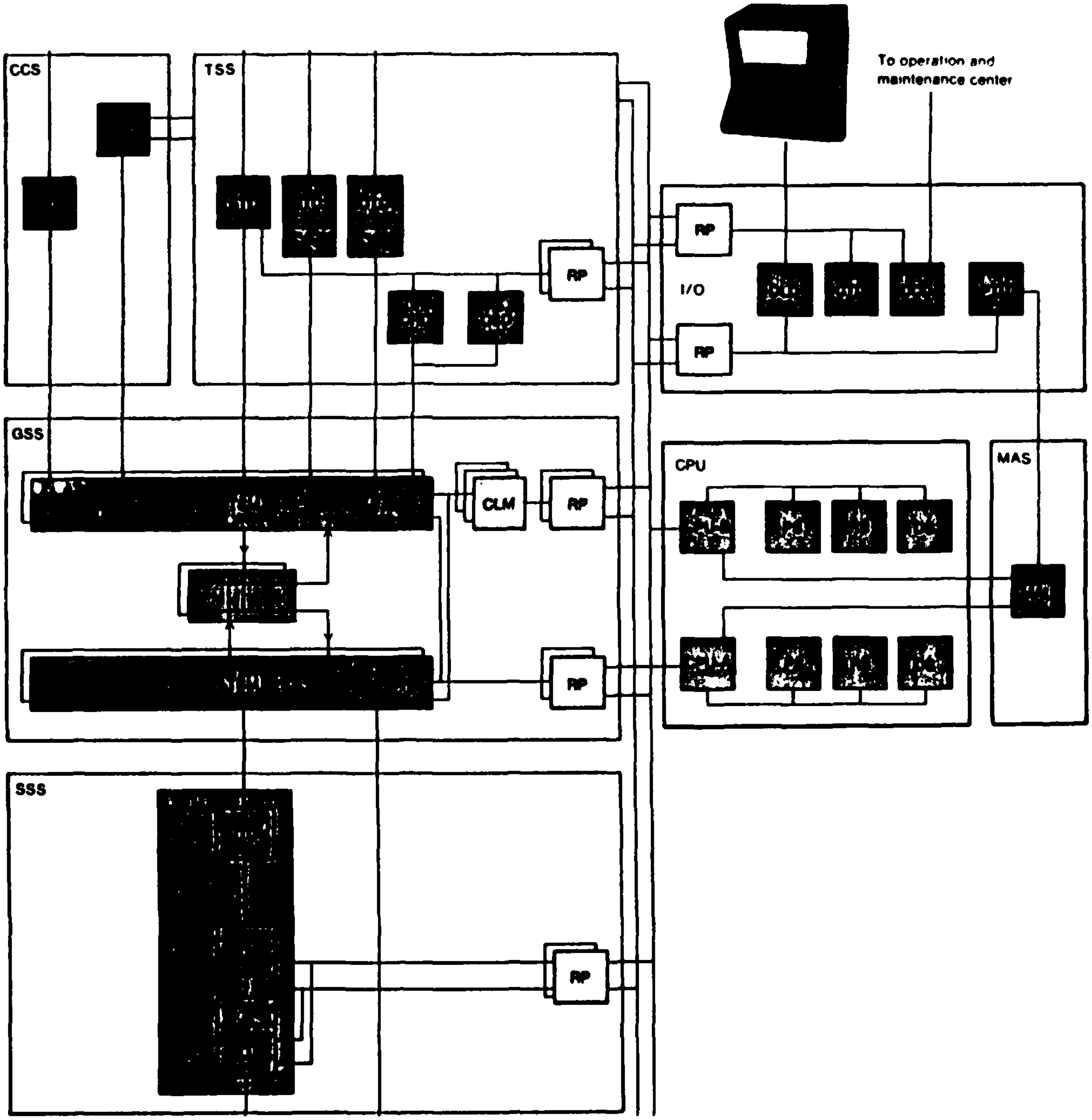


Graph with time as one axis and space as the other, with windows on events. This is an attempt to image a very large system; the universe [K1].

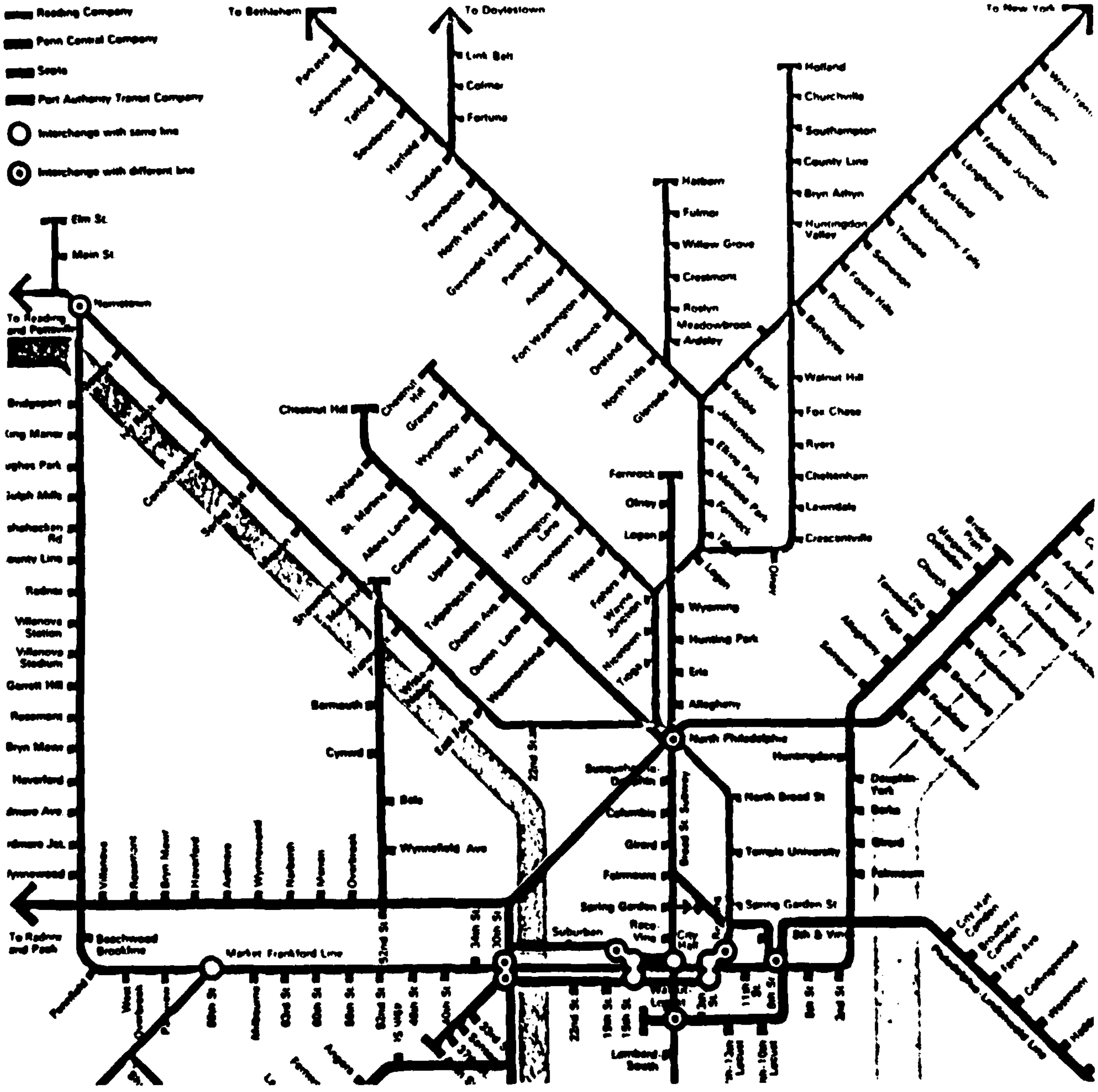


Proposed viewpoints (windows) for observing a number of objects and phenomena. Field of view is 2 X 10^nmetres wide for indicated n; time is speeded up by a factor of 10^t for indicated t.

Nodes and clusters of nodes, information grouping. Example of color being used to organize a complex network.



Information network abstracted and simplified for clarity



B. PROGRAM DESCRIPTION TECHNIQUES

Graphical representations for design and implementation have been known to be of great value in engineering and many other fields. But many fields have an advantage over programming in that they have what could be called a "natural" graphical representation. By "natural" it is meant that the items of interest already have a two-dimensional layout. For example, the formalism of schematic diagrams for electrical circuits just maintains the topology of electrical circuit connections without creating something new. Programming languages lack a "natural" graphical representation [FREI, WELLER and WILLIAMS].

Many techniques have been devised for graphically describing the structure of programs. They range in power of expression from the conventional flowchart to powerful graphical programming systems. They have varying degrees of appropriateness for today's programming style.

This appendix is a brief survey of some of the better known techniques for graphical program description. The salient features of each method are compared in an attempt to derive a minimal set of useful constructs. The description techniques surveyed are:

- Flowcharts
- Nassi-Shneiderman Diagrams
- HIPO charts
- GREENPRINTS
- PYGMALION
- Mini-LOGO animation system
- Micro-PL/1 animation system
- SP/k visualization system
- Sorting animation system
- CDEBUG graphical program debugger

B.1 Flowcharts

One of the earliest attempts at representing programs graphically is the flowchart. Flowcharts consist of a collection of simple and easy-to-understand symbols that represent primitive operations found in all programming languages. Figure B.1 shows the basic symbols of a flowchart. Not shown are symbols representing various storage media and input/output devices. Flow of control is indicated by lines connecting the flowchart symbols. Arrowheads indicate the direction of the flow.

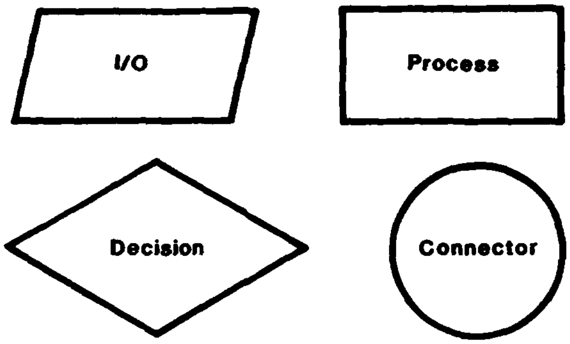


Figure B.1 Flowchart Symbols

As can be seen in Figure B.1, flowcharts lack symbols for explicitly representing loops, "blocks" of code as found in structured programming languages, and other high-level concepts. These must be assembled from the primitives found in the *flowchart repertoire*.

B.1.1 Analysis

Flowcharts are used to represent the flow of control in a program, a purpose for which they are adequately suited. Flowcharts were developed in the days of machine language programming. As a result, many of the constructs found in flowcharts are uniquely suited to machine and assembly language programs, but are inadequate for programming in "high-order languages." Thus, it is not always easy to express some programming language constructs using the basic flowchart symbols. For example, there are no symbols to represent looping or case statements. These must be constructed from collections of the basic symbols.

Because of their nature, flowcharts cannot enforce modular design in programs. No restrictions are placed on transfer of control. This can lead to flowcharts that are nearly illegible due to a dense forest of control lines.

Flowcharts of even relatively simple programs can grow to unmanageable sizes. Since most constructs in high-level languages require several symbols for their expression, flowcharts quickly become crowded. Off-page connectors allow the flowchart to be expanded to other sheets of paper, but result in a diagram that is difficult to comprehend in its entirety. Clever programmers have grown adept at filling every available square inch of a flowchart sheet in an attempt to avoid resorting to multiple sheets. These attempts, while economical of space, are very difficult to understand

Flowcharts satisfied a need that existed before the advent of high-level languages. Programs often were written with convoluted control flows to economize on precious resources. They are of interest from a historical perspective, but are lacking in the qualities necessary to expiess the "modern" style of structured programming.

B.2 Nassi-Shneideiman Diagrams

Nassi-Shneiderman diagrams (NSD) are an attempt to model computations using a control structure amenable to implementation in structured languages [NASSI and SHNEIDERMAN]. They feature simply ordered structures, each representing a complete thought. NSD prevent unrestricted transfers of control, a hallmark of structured programming.

There are four basic symbols that can be combined to form structures. Structures are labeled and are rectangular in shape. The basic symbols provide a basis for representing most operations, but the repertoire can be extended to improve the readability of diagrams that use the more advanced constructs found in many programming languages.

1. *The process symbol* (Figure B.2)—a rectangle—is used to represent assignment and input/output statements as well as procedure calls and returns.
2. The decision symbol (Figure B.3) represents the IF-THEN-ELSE construct found in most structured programming languages. The central triangle contains a Boolean expression. The left-hand and right-hand triangles contain T or F to represent the possible outcomes of evaluating the Boolean. The process symbols contain the sequence of operations to be performed depending on the value of the Boolean.
3. The iteration symbol (Figure B.4) represents looping statements, such as FOR and WHILE. The body of the iteration is a structure. The form of the iteration clearly shows the scope of the iteration. Iterations may be nested to any level.
4. The BEGIN-END symbol (Figure B.5) represents a block of code in the programming language. The scope of local definitions can be clearly seen with this construct. The body of the BEGIN-END is a structure.

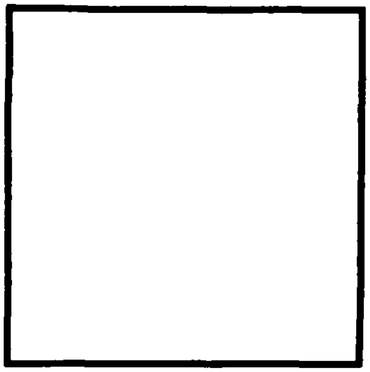


Figure B.2 *Process Symbol*

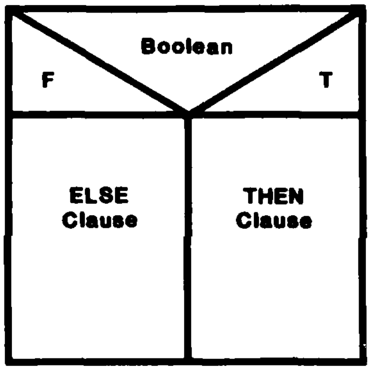


Figure B.3 *Decision Symbol*

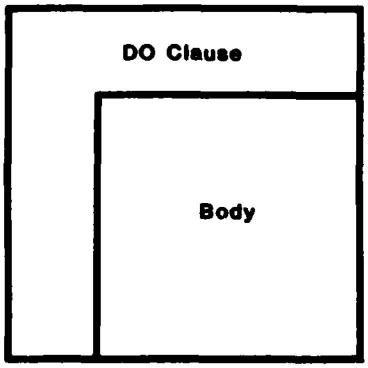


Figure B.4 *Iteration Symbol*

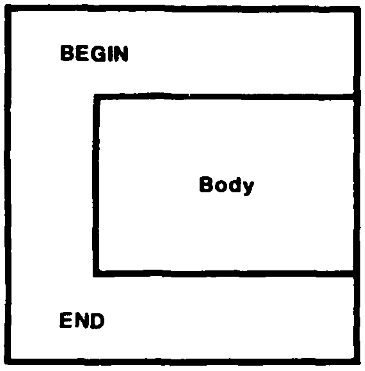


Figure B.5 BEGIN-END Symbol

B.2.1 Analysis

NSD are created from a flowchart language that has a control structure similar to that found in languages used for structured programming. Its creators claim the following advantages over conventional flowcharts:

1. The scope of iteration is well-defined and visible.
2. The scope of IF-THEN-ELSE clauses is well-defined and visible; moreover, the conditions on process boxes embedded within compound conditionals can be seen easily from the diagram.
3. The scope of local and global variables is immediately obvious.
4. Arbitrary transfers of control are impossible.
5. Complete thought structures can and should fit on no more that one page (*i.e.*, no off-page connectors).
6. Recursion has a trivial representation.

[NASSI and SHNEIDERMAN]

The use of NSD enforces the use of structured programming techniques. The absence of a way to represent GOTOs effectively precludes their use. Programmers accustomed to structured programming find no difficulty with its absence. Programmers accustomed to using GOTOs easily adapt to their absence; the graphical representation of the program aids in the process of adaptation.

As there are no off-page connectors, the programmer is forced to modularize the code. Large sections of code cannot be legibly inscribed in the structure described on a single sheet of paper. This forces programmers to write small, logically coherent modules.

The structure of Nassi-Shneiderman diagrams instantly reveals the structure of the code they represent. Blocks are clearly delimited. The range of loops can be seen clearly. "Parallel" blocks of code (as in the different branches of a conditional) can be discerned easily.

B.2.2 Programming Support System

Researchers at the IBM Research Division in San Jose have developed a system that interactively helps the programmer construct and execute NSD. Their goal is to build a programming system that will increase the quality of all phases of software production by:

1. Establishing charting techniques to specify programs in a way that clearly shows their structure and logic.
2. Using an interactive graphics system to draw and edit these charts.
3. Providing a preprocessor/compiler mechanism to translate charts into executable code.
4. Providing self-documentation as a by-product of the program development process.
5. Providing better, interactive diagnostics and program development aids than is currently the case.

[FREI, WELLER and WILLIAMS]

They have added data definition constructs to the symbol repertoire of the NSD as well as the ability to embed PL/1 statements in the symbols. Their Programming Support System (PSS) consists of these extensions to NSD and the following tools:

1. NSD editor
2. NSD interpreter
3. NSD preprocessor/compiler
4. Question-answering component
5. Utility routines

The user of PSS utilizes the NSD editc. to create NSD that contain data definitions and embedded PL/1 statements. Each NSD defines an executable module that may call other NSD and support routines. An NSD typically is no more than one display screenful ın size.

The NSD editor allows the user to point to any portion of an NSD and insert or delete symbols. When a new NSD is created, it is empty; the user fills in the structure with symbols from the NSD repertoire. PL/1 code, such as assignment statements and Boolean expressions, are embedded in the symbols. The system automatically adjusts the display when new symbols are inserted or deleted.

Once an NSD has been created, it may be compiled into machine-executable form or interpreted. Partially created NSD may be interpreted; when a portion of the NSD is encountered that has not been specified, the user may complete the specification. When an NSD is being interpreted, execution may be in a single-step mode. The user may also set breakpoints and examine and change variables. In this mode, the PSS acts as a powerful graphical debugger. Initial response to the Programming Support System has been favorable. The authors of PSS have found that specifying a program as a two-dimensional structure exhibits the meaning of a program more clearly and results in better coding, improved programming productivity and higher quality documentation thus reducing the time and effort (cost) for production and maintenance of software [FREI, WELLER and WILLIAMS].

The PSS system is not being actively developed at this time. However, it has been integrated into another system—TELL—for the design of hardware/software systems. In that system, NSD are being used to develop algorithms as a part of software design.

B.3 HIPO Charts

The HIPO (Hierarchy plus Input-Process-Output) technique ([STAY], [HIPO]) is a top-down design methodology for software systems. During the design phase of a software project, HIPO charts are used to successively refine the design until the basic components and their functions are elucidated.

The HIPO technique consists of two basic components (shown in Figure B.6). They are:

1. Hierarchy chart: shows how each function is divided into subfunctions.
2. Input-Process-Output charts: express each function in the hierarchy in terms of its input and output.

When developing the HIPO charts for a software project, it is essential that the hierarchy and input-process-output charts be developed concurrently. This creates a functional breakdown of the design.

The process of creating HIPO charts for a software project consists of two simple steps that are iterated until the design is completely specified. Those steps are:

1. Describe the function as a series of steps, in terms of their inputs and outputs.
2. Move to the next level of the hierarchy. If the steps in the input-process-output chart are not fully defined, create a new level in the hierarchy in which each step is a box.

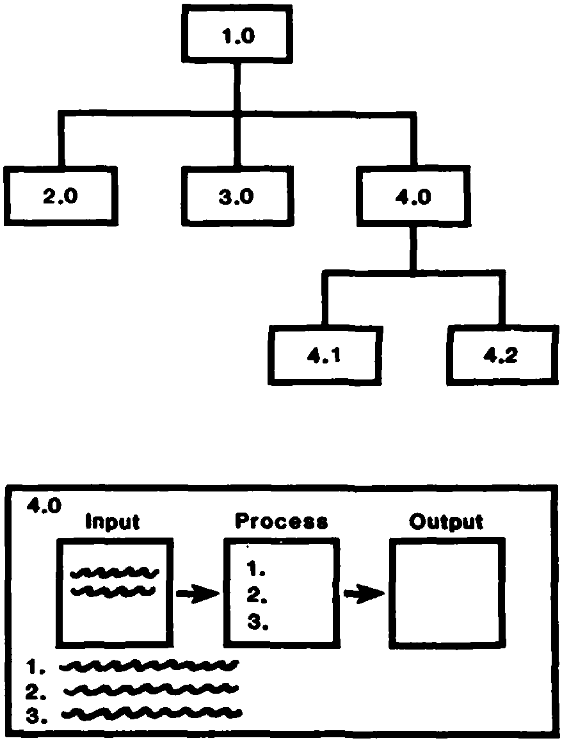


Figure B.6 HIPO Components

These steps are repeated until every function is fully defined. The bottom-most boxes in the hierarchy will probably contain structured English (pseudo-code) statements that describe their function.

B.3.1 Analysis

A set of HIPO charts created in the manner described above will contain a complete description of the components and their interfaces that will constitute the system being designed. Application of functional design techniques [STAY] will further result in a logical grouping of components into modules and processes.

HIPO charts serve a dual purpose. They aid in creating a rational and fully specified design for a software system that can be used by the implementors of the system. They also serve as final programming documentation for use by the maintainers when the project is complete.

HIPO charts provide a modular description of a software system. Functions are decomposed into subfunctions until basic primitives are defined. The interfaces to these functions are fully defined. HIPO charts can be thought of as maps of a software system; each level of the hierarchy provides a more detailed view of a part of the system.

HIPO charts do not descend to the level of code. Instead, they specify the function of the code and its interface to the outside world. From the program visualization viewpoint, they are useful as guidelines to the maintainer (debugger) for where to look in the code to find a particular function.

B.4 GREENPRINTs

GREENPRINTs [BELADY, CAVANAGH and EVANGELISTI] are intended to be to a programmer what blueprints are to an engineer. The GREENPRINT can convey to the programmer the text of a program as well as its control flow. The name GREENPRINT derives from initial implementations on CRTs with green phosphor.

GREENPRINTs consist of two types of objects: the block and the box. Programs are represented by means of objects connected and arranged over a two-dimensional virtual grid.

There are two types of blocks in GREENPRINTs: decision and loop blocks (see Figure B.7).[4] Decision blocks represent IF-THEN-ELSE and CASE statements. Loop blocks represent DO, FOR, and WHILE statements.

There are two types of boxes in GREENPRINTs: gate and processor boxes (see Figure B.7). Gate boxes are always embedded in either a decision block or a loop block. Processor boxes stand alone.

The GREENPRINT representation of a program is a tree where blocks and processor boxes are nodes with entry at the top and exit on the bottom or on the right. Gate boxes originate subtrees in the next column to the right (see Figure B.8).

A processor box can be thought of as containing a collection of code that is executed sequentially. A gate box contains a predicate that can control a decision block or a loop block. Source code may be displayed to the right of each box in a GREENPRINT. GREENPRINTs are configured so that there is only one box in each row, and that box is the right-most object in that row. This spatial arrangement allows the right contour of a GREENPRINT to mirror the left contour of the indented source code. Figure B.9 shows an example of an annotated GREENPRINT.

The decomposition of each block in a GREENPRINT is found in the column immediately to the right of the start of the block. Processor boxes are always in the right-most column of a row and, hence, are not obscured. Gate boxes, containing predicates, are immediately apparent by the right angle extending from them.

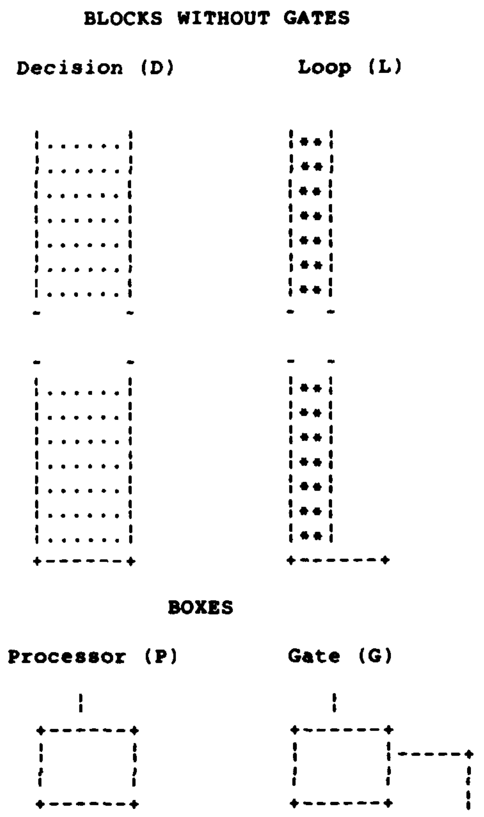


Figure B.7 GREENPRINT Example

B.4.1 Analysis

GREENPRINTs can be used in every phase of the development/maintenance process of software. An overview of the software system may be seen during the design phase by suppressing detail. During development, programming logic may be displayed. Program text can then be added to complete the specification. The resulting GREENPRINTs may be used by those maintaining the system.

GREENPRINTs were developed to serve two needs. The first is to create graphical representations of existing programs for use by the programs' maintainers. A program was written which parses the source code and creates the corresponding GREENPRINT representation.

The second need served by GREENPRINTs is that of a design tool. Program designers can start with schematic diagrams that indicate basic flow of control. As the design is refined, processor and gate boxes

[4] Note that all figures in this section are duplicated from [BELADY, CAVANAGH and EVANGELISTI].

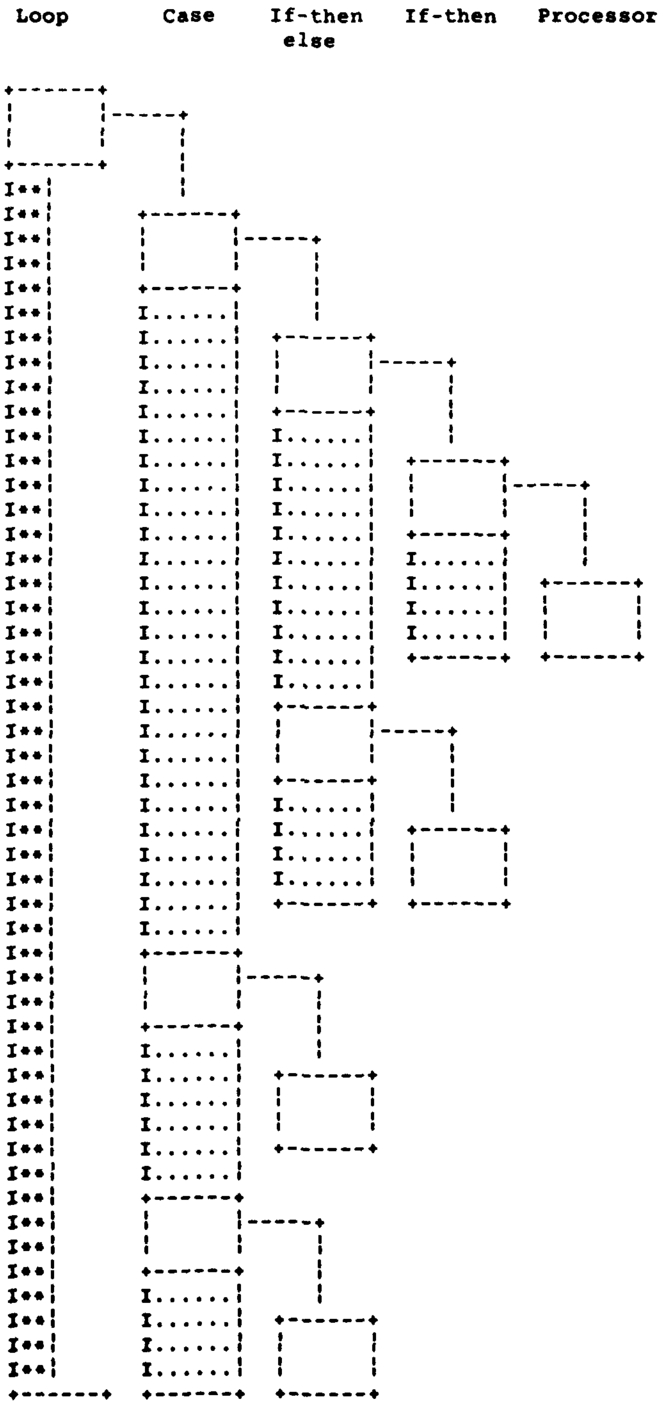
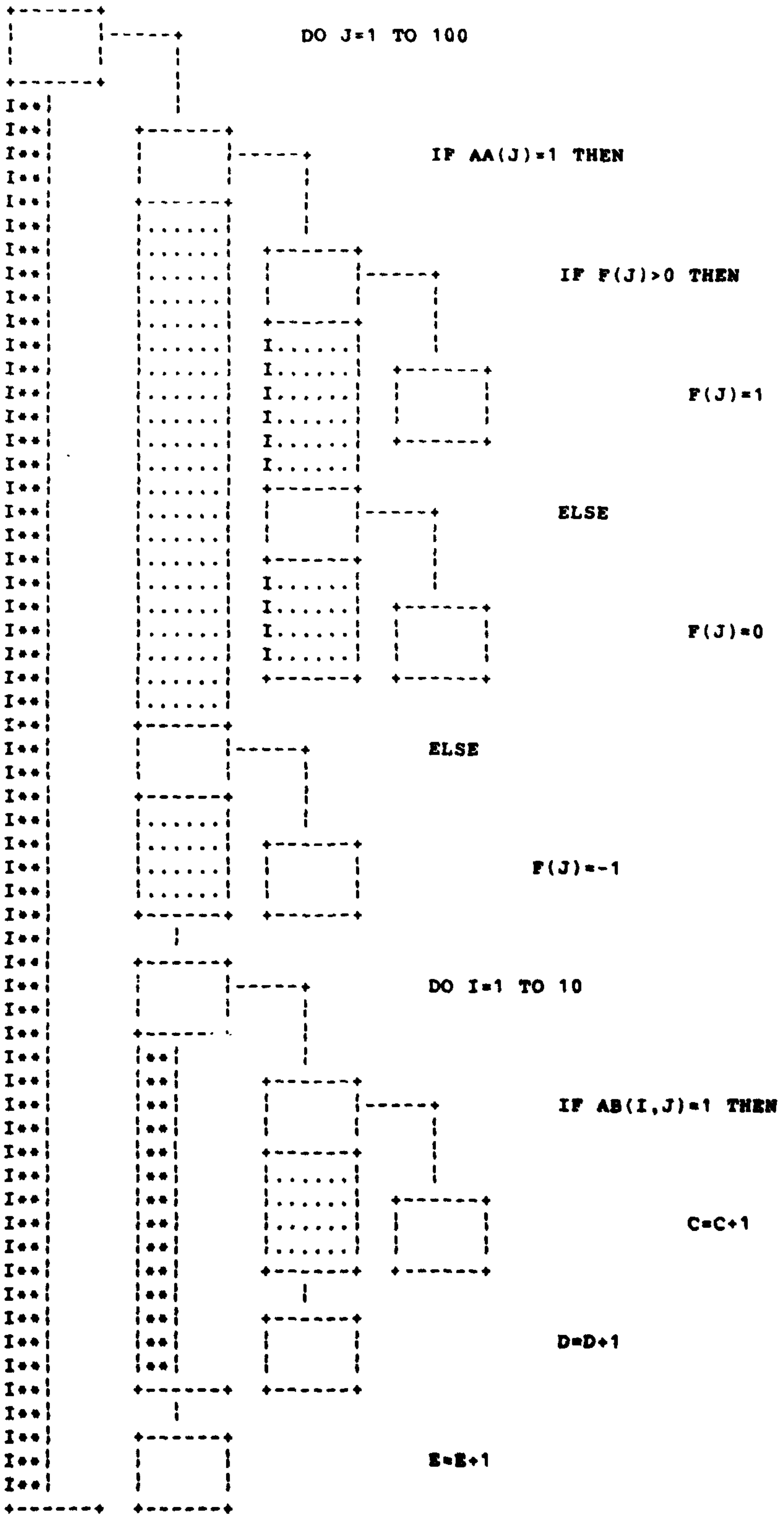


Figure B.8 GREENPRINT Objects



```
DO J=1 TO 100
IF AA(J)=1 THEN
IF F(J)>0 THEN
F(J)=1
ELSE
F(J)=0
ELSE
F(J)=-1
DO I=1 TO 10
IF AB(I,J)=1 THEN
C=C+1
D=D+1
E=E+1
```

Figure B.9 Annotated GREENPRINT

representing unspecified actions can be "filled in" with program text. The result is an executable program.

GREENPRINTs graphically show the nesting of a program and the decomposition of blocks into subcomponents. They represent an attempt to graphically portray the two-dimensional structure of structured programs in an intuitive way.

GREENPRINTs are being used internally within many labs at the IBM Thomas J. Watson Research Center. Their use is spreading to other labs as people become aware of their existence. The authors of GREENPRINT are exploring additional notations for representing constructs such as procedure calls and returns.

B.5 PYGMALION

PYGMALION [SMITH] is a two-dimensional visual programming system developed by David Canfield Smith at the Stanford Artificial Intelligence Laboratory. It differs from the other techniques described thus far because the user writes programs using only graphical constructs to represent operations and data. There is no notion of embedded program text in graphical representations of program control structures.

The user shows the system how to perform a computation by explicitly going through the steps. The machine may remember the sequence of steps that the user executes and reproduce them at a later time. In this respect, programming in PYGMALION is similar to programming a hand-held calculator.

The PYGMALION user employs a computer graphics display (raster or stroke), a pointing device, and a keyboard. The display shows a menu of predefined system functions and a large work area. The user manipulates icons that represent data and primitive operations to create graphical depictions of the intended operation and its operands. The user is free to create icons that are meaningful to him. Every operation the user performs has not only internal semantics, but also visual semantics. Thus, every operation affects the display.

Programming in PYGMALION consists of creating a sequence of display frames, the last of which contains the desired result. While the sequence is being created or replayed, the user may see the effect of each operation on the data and program state. Smith very emphatically states the distinction between PYGMALION and graphical programming languages:

> I want to emphasize that PYGMALION is not a graphical programming language in the traditional sense. Graphical programming languages have all attempted to find two-dimensional ways to tell programs what to do. This inherently involves the manipulation of formal representation of data. PYGMALION has no representation for telling a program anything; PYGMALION is an environment for doing computations. If the system happens to remember what is done, then a program is constructed as a side effect. But the goal of the programmer is to do a computation *once*. This is helpful for understanding in any case; a good way to understand a complicated algorithm in any language is to work through it with representative values. Instead of using the medium of paper or blackboard, the PYGMALION programmer uses the display screen [SMITH, p. 70].

B.5.1 *Analysis*

The main innovations of the PYGMALION system are:

1. A dynamic representation for programs—an emphasis on doing rather than telling.
2. An iconic representation for parameters and data structures requiring fewer translations from mental representations.
3. A "remembering" editor for icons.
4. Descriptions in terms of the concrete, which PYGMALION turns into the abstract.

Of these innovations, perhaps the most significant is the iconic representation of data and operations. By allowing the user to create representations that are meaningful to him and that require little effort to translate from mental representations, PYGMALION relieves the user from the distracting process of mapping mental constructs into programming constructs. It is too often the case in programming that the forest is lost for the trees. The programmer becomes so involved with the myriad details of implementation that he loses sight of the original intent.

PYGMALION is by no means a system for producing production versions of programs. In fact, it would be quite difficult to represent a large system in PYGMALION. Its importance lies in its ability to allow a programmer to visualize algorithms that will be used in the production version. PYGMALION allows the programmer to gain confidence in an algorithm by watching it in all stages of execution and under diverse circumstances.

B.6 The Mini-LOGO Animation System

The mini-LOGO animation system produces a dynamic display of flow of control in the context of a *display of program text*. The mini-LOGO animating interpreter accepts a LOGO command, a set of LOGO procedures, and a set of illustration specifications. It produces an animation sequence that depicts the execution of the command and the called procedures. It shows the execution of the initial command, followed by the execution of every statement in every called procedure in the order in which each is encountered.

All procedure statements are displayed surrounded by the code of that procedure.

The user specifies the subject by designating a LOGO command to be illustrated. The system gives the user no control over the composition, the events to be visualized, or the context, for they are "hard-wired." The user can exercise minimal control over the symbolism by specifying the visual transformations used to map one program "state" into another program state. The user has much control over the dynamics—the speed with which these transformations take place.

The mini-LOGO system is highly specialized towards the pedagogical display of small, recursive LOGO procedures. It is of little long-term use. It does suggest how hard it may be to develop cognitively meaningful displays that are effective for a wide variety of programming concepts.

B.7 The Micro-PL/1 Animation System

The micro-PL/1 animation system produces a dynamic display of program data. The micro-PL/1 animating compiler-interpreter accepts a program in a subset of PL/1, augmented with a set of pseudo-comments that appear at the beginning of the program text and at key locations within it. The pseudo-comments control the production of an animation sequence that shows how selected variables evolve over time.

The user specifies the subject by designating the variables that are to appear in the illustrations. Symbolism is specified by selecting parameterized icons from an image library. Composition is specified by the user in great detail through the use of the expression writing tools of the host language. Events are depicted whenever a selected variable changes value or whenever a pseudo-comment embedded in the code is "executed." The user has adequate control over dynamics. All illustrations are independent; no context is provided.

The micro-PL/1 system is useful pedagogically in dealing with small micro-PL/1 programs. It has three major conceptual weaknesses:

1. It produces illustrations of data only.
2. It forces the user to modify extensively the program that is to be illustrated.
3. It forces him to work with great precision and in great detail to produce even simple visualizations.

B.8 The SP/k Visualization System

The SP/k visualization system produces illustrations that consist of neatly formatted, automatically paragraphed program text interspersed with pictorial representations, or snapshots, of the program's data. The system consists of a preprocessor that reads an SP/k program and an "illustration specification." It expands the program with statements and new variables to keep track of the program's execution and to produce graphic output. Execution of the expanded program then produces the text and the interspersed snapshots. Each snapshot reflects the state of the data during a particular execution of the immediately preceding program code.

The user specifies the subject of the illustration by naming a ***flowgroup-execution***, which is one iteration of a particular loop or one execution of a procedure. He also names the variables that he wants to appear in the snapshot. Symbolism is primarily determined by the system, although the user can request particular labeling of arrays and subsets of arrays. The system determines the composition, selects the events automatically, and controls the juxtaposition of displayed text to displayed data. Because the implementation was done in a batch processing environment, there are no dynamics to control and no contextual possibilities other than those already described.

Yarwood's thesis, within which this system was developed, represents one of the most thoughtful investigations in the area of program visualization carried out to date. However, the concepts need to be extended to an interactive environment. Also, the approach must be applied to a far greater subset of modern language features than the fixed scalars and single-dimensional vectors of the original implementation.

B.9 The Sorting Animation System

The sorting animation system produces a dynamic display of data being generated by a fixed set of sorting algorithms. The system consists of a set of programs that accepts a file of unsorted data and certain run-time parameters. It then produces an animation sequence that shows the data being sorted.

The user specifies the subject of the illustration by providing the file of input data. The symbolism, the composition, and the events are "hard-wired," with the exception of a few minor details such as the choice of color. The user has a great deal of flexibility in the specification of dynamics. The system provides no capabilities for context.

This system consists of a small set of highly specialized programs written to make a 30 minute teaching film on a set of sorting algorithms. It has taught us a number of lessons about program visualization:

1. How effective symbolism depends upon the size of the subject, the scale of its image within the total composition, and the context within which the image is displayed.

2. How powerful and flexible the control over dynamics needs to be to produce meaningful animation sequences.
3. How much insight can be gained from simply viewing the data if the illustrations are designed carefully.

B.10 CDEBUG

The CDEBUG system produces a sequence of snapshots of program data and a tiny amount of flow of control in the context of a display of program text. The system consists of a run-time debugging environment that accepts a C program in source language, the object module produced by a suitably modified C compiler, and a debugging script. Execution of the program then produces a display of program text and a set of snapshots. Each snapshot reflects the state of the data during a particular execution of the immediately preceding program code.

The user specifies the subject of the illustration by naming the variables he wants to appear in the snapshot. Symbolism is determined by the system. It consists of a linear display of the contents of relevant machine locations expressed in a source language notation. Composition is also determined by the system. The user controls the events at which snapshots are produced by setting breakpoints. There is no capability for animation, so there are no dynamics. The user has some control over context. He can decide how much of the screen is to be allocated to program text and how much is to be used for program data, and he can review previous snapshots that have been saved on a stack.

Crossey's debugging system provides a foundation upon which one could base future experiments in program visualization. Her system provides a rich set of source language probes for interrogating the state of a C program. It remains for us to turn the output of these probes into meaningful visualizations.

B.11 Summary

The program description techniques surveyed in this document can be divided into two categories: those that describe programs by flow of control, and those that describe programs by flow of data. HIPO charts fall into the latter category, while conventional flowcharts, NSD, and GREENPRINTS fall into the former category. PYGMALION fits neatly in neither category, and cannot be considered a program description technique; it is a programming technique itself.

Of those techniques which describe the flow of control in programs, only NSD and GREENPRINTS present an accurate view of the structure of the program. Flowcharts do not reflect the structure of the program they are intended to model; they lack the appropriate constructs to do so.

The power of NSD and GREENPRINTS lie in their ability to graphically represent the structure of programs in a compact and obvious manner. A person examining one of these diagrams can easily discern the structure of the source code and quickly map the graphical constructs back into the source code. This is a crucial factor in the effectiveness of a program description technique.

Both NSD and GREENPRINTS are easy to learn. In fact, little effort is required once the basic repertoire of symbols is mastered. Also, they can be generated automatically from existing program text. This is important in the program visualization context; these techniques are adaptable to existing software. Furthermore, design systems can be built which use these techniques to create new software that satisfies high standards of program structure and documentation.

C. REFERENCES

[ADA]
SIGPLAN NOTICES: Preliminary ADA Reference Manual, 14, 6, Part A, Association for Computing Machinery, Inc., New York, June 1979.

[BAECKER]
Baecker, R, "Sorting Out Sorting," 16mm color videotape, sound, 25 minutes, *Dynamic Graphics Project*, Computer Systems Research Group, University of Toronto, Toronto, Ontario, 1981.

[BELADY, CAVANAGH and EVANGELISTI]
Belady, L.A., J.A. Cavanagh, and C.J. Evangelisti, "GREENPRINT: A Graphical Representation for Structured Programs," *IBM Research Report RC 7763*, July 1979.

[BERNSTEIN, BLAUSTEIN and CLARKE]
Bernstein, P.A., B.T. Blaustein, and E.M. Clark, "Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data," *Proceedings of the 1980 Conference on Very Large Data Bases*. Also available as Technical Report TR 05-80, Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts 02138.

[FREI, WELLER and WILLIAMS]
Frei, H.P., D.C. Weller, and R. Williams, "A Graphics-Based Programming-Support System," *Computer Graphics*, 12, 3, 43-49, August 1978.

[HAIBT]
Haibt, Lois M, "A Program to Draw Multilever Flow Charts," *Proceedings of the Western Joint Computer Conference*, San Francisco, California, March 1959.

[HAMMER and SARIN]
Hammer, Michael M. and S. Sarin, "Efficient Monitoring of Database Assertions," *Proceedings of the 1978 SIGMOD Conference on Management of Data.*

[HEROT *et al.*]
Herot, Christopher F., Richard T. Carling, Mark Friedell, David Kramlich, and J. Thompson, "Spatial Data Management System: Semi-Annual Technical Report," *Technical Report CCA-79-25*, Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139, 30 June 1979.

[HIPO]
HIPO – A Design Aid and Documentation Technique, IBM Corporation, Data Processing Division, White Plains, New York 10504.

[HOPGOOD]
Hopgood, F.R.A., "Computer Animation Used as a Tool in Teaching Computer Science," *Proceedings of the 1974 IFIP Congress*, Applications Volume, 889-892.

[KERNIGHAN and MCILROY]
Kernighan, B.W. and M.D. McIlroy, *Unix Programmer's Manual*, Bell Laboratories, Murray Hill, New Jersey.

[KNOWLTON]
Knowlton, K.C., *L6: Bell Telephone Laboratories Low-Level Linked List Language*, two black and white films, sound, Bell Telephone Laboratories, Murray Hill, New Jersey, 1966.

[LIONS]
Lions, J., *A Commentary on the Unix Operating System*, Department of Computer Science, University of New South Wales, 1977.

[NASSI and SHNEIDERMAN]
Nassi, I. and B. Shneiderman, "Flowchart Techniques for Structured Programming," *SIGPLAN Notices of the ACM*, 8, 8, 12-26, August 1973.

[ROSS]
Ross, D.T., "Structured Analysis (SA): A Language for Communicating Ideas," *Software Engineering* (SE-3, 1) IEEE Computer Society, Silver Spring, Maryland, 34, January 1977.

[ROTHNIE *et al.*]
Rothnie, James B. Jr., Philip A. Bernstein, Steven Fox, Nathan Goodman, Michael Hammer, Terry A. Landers, Christopher Reeve, David Shipman, and Eugene Wong, "SDD-1: A System for Distributed Databases," *Technical Report CCA-02-79* (revised), Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139, 1 January 1979.

[SMITH]
Smith, D.C., "PYGMALION: A Creative Programming Environment," *Stanford Artificial Intelligence Memo AIM-260*, June 1975.

[STAY]
Stay, J.F., "HIPO and Integrated Program Design," *IBM Systems Journal*, 15, 2, 143-154, 1978.

[STOCKHAM]
Stockham, T.G., "Some Methods of Graphical Debugging," *Proceedings of the IBM Scientific Computing Symposium on Man-Machine Communication*, Yorktown Heights, New York, 57-51, May 1965.

[TEITELBAUM]
Teitelbaum, R.T., "The Cornell Program Synthesizer: A Microcomputer Implementation of PL/CS," Department of Computer Science, Cornell University, Ithaca, New York.

[WONG and EDELBERG]
Wong, K.C. and M. Edelberg, "Interval Hierarchies and their Application to Predicate Files," *ACM Transactions of Database Systems*, 2, 3, 223-232, September 1977.

The Official Distribution List for the Technical, Annual, and Final Reports for Contract N00014-80-C-0683

Defense Documentation Center 12 copies
Cameron Station
Alexandria, VA 22314

Office of Naval Research
Arlington, VA 22217
Information Systems Program (437) 2 copies
Code 200 1 copy
Code 455 1 copy
Code 458 1 copy

Office of Naval Research 1 copy
Eastern/Central Regional Office
Bldg. 114 Section D
666 Summer Street
Boston, MA 02210

Office of Naval Research 1 copy
Branch Office, Chicago
536 South Clark Street
Chicago, IL 60605

Office of Naval Research 1 copy
Western Regional Office
1030 East Green Street
Pasadena, CA 91106

Naval Research Laboratory 6 copies
Technical Information Division
Code 2627
Washington, DC 20375

Dr. A. L. Slafkosky 1 copy
Scientific Advisor
Commandant of the Marine Corps (RD-1)
Washington, DC 20380

Naval Ocean Systems Center 1 copy
Advanced Software Technology Division
Code 5200
San Diego, CA 02152

Mr. E. H. Gleissner 1 copy
Naval Ship Research & Development Center
Computation and Mathematics Department
Bethesda, MD 20084

Capt. Grace M. Hopper (008) 1 copy
Naval Data Automation Command
Washington Navy Yard
Bldg. 166
Washington, DC 20374

Defense Advanced Research Project Agency 3 copies
Attn: Program Management/MIS
1400 Wilson Boulevard
Arlington, VA 22209

DATE
ILMED

8