THE UNIVERSITY OF MICHIGAN

COMPUTING CENTER

Final Report

CONCOMP: RESEARCH IN CONVERSATIONAL USE OF COMPUTERS

F. H. Westervelt
Project Director

ORA Project 07449

under contract with:

U. S. DEPARTMENT OF DEFENSE
ADVANCED RESEARCH PROJECTS AGENCY
CONTRACT NO. DA-49-083-OSA-3050
ARPA ORDER NO. 716
WASHINGTON, D.C.

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

December 1970

TABLE OF CONTENTS

ABSTRACT

This report describes the final research results of the CONCOMP Project: Research in the Conversational Use of Computers, which was funded from 1965/70. This research involved the design, development, and testing of computer programs for graphical input of problem statements and graphical output of results from a computer; the application of the techniques so developed to speech synthesis, systems design research, and research in the logic of computers; the study, design, implementation, and testing of systems for describing graphical operations within the format of procedure-oriented computer programming languages. All of this work was predicated on the availability of IBM 360/67 hardware and software. When TSS was unavailable, CONCOMP undertook two additional tasks: (1) development of the conversational aspects of an operating system for the central computing facilities to support effective man-machine interaction; (2) development of an effective hardware interface for the support of the remote terminal devices.

1. INTRODUCTION

The research project on the Conversational Use of Computers, known as CONCOMP, was initiated during 1965 to explore the wide spectrum of man-machine interactions implied by the project title. The research was to include explorations and developments in hardware and software, central computing facilities and remote-access terminals, languages, systems programs, and applications. The research anticipated the availability of the IBM System/360 Model 67 hardware with its unique (within IBM systems) bus organization and Dynamic Address Translation hardware and the IBM TSS (Time-Sharing System) operating system.

Happily, the IBM System/360 Model 67 hardware was delivered on time and has proven itself to be as versatile and reliable as had been anticipated. Unhappily, the IBM TTS operating system was not available on a timely basis and, when finally available, its performance was not adequate to effectively discharge the research objectives of CONCOMP. TSS is not a truly viable system for a broad spectrum of interactive terminals with rapid response even today, although TSS continues to improve steadily. In addition, there was not (and is not yet) available any standard IBM Terminal Controller able to accommodate a broad range of interactive terminals with widely varying characteristics and performance needed for effective conversational computation.

Thus, the CONCOMP project found itself engaged in at least two research tasks not anticipated prior to its initiation:

1. The development of the conversational aspects of an operating system

for the central computing facility needed to support any type of effective man-machine interaction.

2. The development of an effective hardware interface for the support of the remote terminal devices.

The first of these additional tasks, executed in cooperation with the staff of the Computing Center at The University of Michigan, resulted in the operating system known as MTS (Michigan Terminal System). MTS is now serving an active user community of more than 9000 active user "sign-on" accounts with a single unified system for both remote interactive use and for non-interactive batch processing. A fairly typical demand peak finds 50 to 60 active terminals together with 4 to 6 batch streams operating concurrently.

The remote terminals may range from the ultimate low-cost and simplicity of the push-button or Touch-Tone® telephone through the numerous low-speed alphanumeric terminals, both cathode-ray-tube and impact-printer types, and storage-tube graphic displays to the more versatile remote graphics terminals built around a local mini-computer. It should be observed that the facilities developed under CONCOMP for conversational computations research have been general enough to permit extension into many areas not encompassed by CONCOMP by the staff of the Computing Center and other units of the University.

For example, unusually flexible remote job entry facilities and the interaction of the Computing Center facility with remote mini-computers in a variety of laboratory digital/analog control systems have been implemented on the CONCOMP base.

The second of the additional tasks resulted in the Data Concentrator, the first of what have come to be common data communications control computers, directly interfaced to the System/360/67 multiplex channels. The early lessons learned in the development of the Data Concentrator communications protocol and general organization served as a part of the basic work upon which later networks of computers such as the ARPA network and the MERIT network have been built. It is important to observe that the flexibility of the Data Concentrator in responding to a dynamically changing environment of line speeds, code frames, code conversions, line controls, and device-support requirements is still unmatched in any commercially available equipment.

In addition, CONCOMP pursued research in numerous interrelated subjects. These efforts included languages, data structures, computer-aided design, executive systems for local graphics terminals, applications, and broadly generalized graphics support. These activities produced 33 technical reports and 36 project memoranda reporting the results of these efforts. The doctoral program of more than 10 students with the direct production of 4 doctoral dissertations also derived support from the CONCOMP research.

In the balance of this final report an attempt will be made to highlight some of the activities that comprised the totality known as CONCOMP. The true success of CONCOMP can best be measured in the steadily expanding utilization of the developed facilities by the entire University community at a rate exceeding 20% per year. It is a generally observed phenomenon of interactive computing facilities with suitable mechanisms for sharing of programs and data that such systems achieve a "critical mass" and then begin to sustain a self-generated internal growth of new capabilities and facilities. As a result of the cooperation between the Computing Center, CONCOMP, Systems Engineering Laboratory, Logic of Computers Group, and the entire user community of The University of Michigan, this phenomenon has again been experienced in an unusually broad environment of users. Finally, the facilities thus developed have been exported to more than 10 other major installations with at least 5 of these making regular use of the MTS capabilities on a regularly scheduled basis.

On these measures it is felt that the goals of CONCOMP to explore the nature of conversational computing on a broad basis and to extend the developed concepts into new areas of computer science have been accomplished successfully.

2. SUMMARY REPORTS OF AREAS OF SPECIAL INTEREST

2.1 MAD/I

The success of MAD (Michigan Algorithm Decoder) on the IBM 7000 series hardware resulted in its rather wide adoption including CTSS at Project MAC. This, in turn, led to the enquiry by CONCOMP into a new formulation, MAD/I for the IBM System/360 hardware. Since there were available in 1965 and 1966 several reasonably fast implementations of algebraically oriented compilers, it was decided that MAD/I should concentrate upon rather different facilities from those which served as a dominant guide to the design of 7090 MAD.

In particular, the questions of language extension and extensibility were selected to receive special attention. The result, reported in CONCOMP Project Technical Reports 7 and 32 and in CONCOMP Project Memoranda 1, 31, and 32, has achieved the desired flexibility. The MAD/I activity was conducted under Professors Arden and Galler with the services of many contributors including D. L. Mills, R. J. Srodawa, A. L. Springer, B. J. Bolas, C. F. Engle, and F. G. Swartz. MAD/I has served as the language for numerous user applications. As might be expected, only a few of these have exploited the full range of language extensibility available to the MAD/I programmer.

A concrete illustration of the power of the features provided in MAD/I can be found in Memorandum 32, Sample Definitional Facility in MAD/I, by R. Srodawa. Here the basic MAD/I language is extended, in an easily followed example, to provide special new data types and structures and new statements and operators intended to implement a simple graphical language.

For example, new data types 'POINT' and 'LINE' are added to the basic MAD/I data types. Each of these data types includes several different kinds of data values and pointers. The MAD/I definitional facilities allow the user to define these new kinds of variables and then deal with them either as a unified whole or as individual component parts as required in his problem solution.

In addition, the user defines new statements into MAD/I such as:

```
'CREATE POINT' POINT, X,Y

          or

'CREATE LINE' LINE,P1,P2

          or

'CONNECT' POINT 'TO' PICTURE
```

In each instance, the user employs the extensibility features of MAD/I to accomplish these extensions within the basic MAD/I facility.

The user, depending upon his need and sophistication, may deal with MAD/I extensions at several levels. MAD/I itself is defined in itself at a very basic level. If desired, a knowledgeable user is free to extend MAD/I himself at this same fundamental level. There are, however, several "higher" levels at which extensions may be accomplished more smoothly by less sophisticated users. The previous examples from Memorandum 32 are such extensions. Other intermediate levels are also illustrated in Memorandum 32 and Technical Report 32. One such is the 'ENTER ASSEMBLER CODE' feature described in detail in Technical Report 32.

This feature is significant because it illustrates rather clearly how two computer languages may be merged smoothly to provide the user with unusual power while maintaining the controls and services of the usual higher-level compilers. It is also significant because, while not reported in Technical Report 32, the addition of this feature to MAD/I using the extensibility features of MAD/I required less than one man-week of programming effort to accomplish and check out.

MAD/I has provided a valuable research vehicle for the exploration of language extensibility. While, in retrospect, the numerous applications generated by CONCOMP might have profited greatly from the earlier availability of the final version of MAD/I, the explorations have, in themselves, produced new insights into computer language generation that qualify the MAD/I activity as success.

2.2 DATA CONCENTRATOR

The complete lack of adequate terminal controller hardware available for the IBM System/360 hardware with the facilities needed to allow the attachment of diverse remote-access terminals needed for the exploration of conversational computations resulted in the design, development, fabrication, and systems implementation of a very general data communications control computer known as the Data Concentrator.

This work was accomplished by D. L. Mills with the assistance of many other co-workers including J. DiGiuseppe, W. S. Gerstenberger, and V. M. Powers. The effort included the design of both hardware and software. The hardware designs are influencing the construction of several contemporary machines for interconnecting major computers in the MERIT network. The software system, known as RAMP, has been incorporated in remote graphics facilities such as the DEC 338 in another portion of the CONCOMP activity and similar graphics facilities in at least two industrial applications, one in the Ford Motor Company and another in the Whirlpool Corporation. In addition, the RAMP system has formed the system nucleus for several laboratory control experiments in the physical sciences and in medicine as well as the control of an usually flexible language laboratory installation at The University of Michigan.

The design experience has been reported in CONCOMP Project Technical Reports 8, 19, 20 and CONCOMP Project Memoranda 5, 11, 13, 15, 16, 17, 19, 20, 24, and 34.

2.3 AUDIO RESPONSE

An outstanding example of the breadth of man-machine interfaces explored under CONCOMP is the standard push-button or Touch-Tone® telephone and the IBM 7772 Audio Response Unit. This work was executed almost entirely by D. Smith and is reported in CONCOMP Technical Report 27. It is important to report here some experience gained in two areas not covered in that report.

During the first months of 1970, the Touch-Tone® telephone and the digitally synthesized speech facilities developed to support the IBM 7772 were employed to train a totally blind student, Mr. Anthony Sutter, in the use of MTS and the FORTRAN IV language. Mr. Sutter had only recently lost his sight and did not yet read Braille when his training began. An ordinary push-button telephone equipped with an auxiliary loudspeaker was provided by the State of Michigan through Workmen's Compensation and served as Mr. Sutter's entire medium of communication with the computer on a conversational basis.

As a result of his interaction with MTS through the Audio Response Unit, several special features were developed. For example, a means of speech compression is now available under the control of the user to greatly accelerate the delivery of verbal output. To the casual listener, the rate of speech may become so rapid as to become almost unintelligible. To the experienced listener, however, the output is still readily understood, and to the blind programmer, the frustration of a slow "reader" is eliminated.

Other controls which permit the pronunciation (or suppression) of punctuation including "blanks," permits the blind programmer to deal with all of the intricacies of programming languages and formats. Finally, the interactive facilities developed under CONCOMP permit complete device-independence of the applications programs from the terminal device. In this way, a sighted programmer and the blind programmer may readily use the same applications.

The training sessions for Mr. Sutter were carried out on a twice-per-week basis for approximately three months. With no prior computer experience, Mr. Sutter was able to progress to the point that he could input, debug, and run FORTRAN problems given him at a Tuesday session before the Thursday meeting. It is also significant to note that the average hourly cost for his entire training period came to approximately $8.42/hour. This cost included the cost of speech synthesis on the central computing facility.

The second area to be reported here is the work by D. Smith in speech synthesis using the IBM 7772. By developing a special package to support this device in MTS, it was possible to extend the vocabulary of the 7772 greatly beyond the usual 1000 to 1500 words available from IBM. The extension process involves a dynamic scanning and decomposition of the alphanumeric strings supplied to the ARU device support routines by MTS. By recognizing and combining the sounds represented by the character strings, many "new" words are pronounceable without a serious degradation in speech intelligibility. It is difficult, if not impossible, to assess the size of the "new" vocabulary thus available, but it is at least an order of magnitude larger.

In addition, facilities for the user to "tailor-make" special new words in an interactive on-line mode of operation make some otherwise inaccessible applications feasible. One current effort in the area of the reporting of medical diagnoses to physicians through their office telephones has resulted in the artificial synthesis of such terms as: HYPERTROPHY, HYPOKALEMIA, and INFARCTION.

2.4 COMPUTER GRAPHICS AND COMPUTER-AIDED DESIGN

One of the important areas of the CONCOMP research dealt with computer graphics and the related topics in computer-aided design. The results encompassed the design of facilities at both the central computer and the remote graphics terminals. Three different remote terminals which included local mini-computers were employed. The least complex was the DEC 338 which included a PDP-8, 12-K words of local core, and a small local disk. The next in increasing logical power was the DEC 339 in the Systems Engineering Laboratory. This terminal incorporated the same display controller as in the 338 but utilized the PDP-9 computer for local processing. The most complex was the interconnected IBM 1800/PDP-7 with associated display controller located in the Logic of Computers Group. This combination of remote processing power made possible very significant amounts of local picture manipulation and problem solving without extensive intervention by the central computer.

The systems developed for the 338 were reported in CONCOMP Project Technical Report 33 and Memoranda 2, 3, 4, 6, 9, 23, 28, 29, 30, 33, 34, 35, and 36. The systems developed for the 339 were reported in CONCOMP Project Technical Reports 15, 23, and 24, and Memorandum 25. The systems for the 1800/PDP-7 were reported in CONCOMP Project Technical Reports 10, 11, 12, and 31, and Memorandum 7. These systems resulted in facilities readily usable by numerous applications.

In particular, the area of computer-aided design in several problem disciplines was made viable through these graphics facilities. In the area of mechanical systems design, the work of Professor Chace and his students requires special mention. The methods of treating generalized multi-freedom constrained mechanical systems have received special recognition nationally. This work was reported in CONCOMP Project Technical Reports 18 and 26. The doctoral dissertation by J. Allan reported in Technical Report 9 was another design system in the mechanical design area.

In the area of queuing systems analysis, the work of the Systems Engineering Laboratory was reported in Technical Reports 13, 14, 21, and 23. Systems Engineering Laboratory also developed some techniques for circuit design and simulations reported in Technical Report 24. These graphics programs included some unusually well-adapted methods of utilizing the light-pen for communicating the user's requirements to erase, move, create, connect, rotate, and branch while entering the topology and parametric values for the systems to be analyzed. The GENESES (GEneralized NEtwork SErvice System) developed by Systems Engineering Laboratory is reported in Appendix A of this report.

Another area of work in the simulation of cellular spaces was reported in the dissertation of R. F. Brender (Technical Report 25). A related area in the simulation of atrial fibrillation is reported in Appendix B of this report.

The special development of facilities for instruction in computer graphics by Herzog and Shadko reported in Technical Report 30, the versatile digital-plotting facilities developed by Fronczak reported in Technical Report 29, and the interactive graphical mathematics system developed for the IBM 2250 reported in Memorandum 27 are final examples of the range of topics treated. The facilities developed under CONCOMP have resulted in the introduction of well over 500 students to the use of interactive computer graphics methods and techniques.

2.5 DATA STRUCTURES

The ever-present problems of providing rapid, flexible access to complexly interrelated data resulted in several efforts in the area of data structures under CONCOMP. The doctoral dissertation of Mowshowitz reported in Technical Report 1, the work of Prof. Sibley, W. Ash, and others reported in Technical Reports 5 and 17 and Appendix C of this report, and the work of Childs reported in Technical Reports 3 and 6 and Appendix D of this report represent the scope of efforts expanded.

In addition, special topics such as the work by DiGiuseppe in pictorial data-compression, and Wolf, Julyk, Dingwall, and Goodrich in CAMA (Computer-Aided Mathematical Analysis) also dealt strongly with topics related to data structures.

In each of the above cases, the theoretically derived structures have been implemented and employed by numerous applications users in MTS. The flexibility and versatility represented deserve greater emphasis than is possible in this brief summary. Accordingly, Appendices C and D provide greater detail for this important area.

3. CONCLUSIONS

The CONCOMP effort began on The University of Michigan campus at a critical time in the emerging development of effective interactive man-machine conversational computing. As a direct result of the efforts of this project, a highly effective system of conversational computing employing a wide spectrum of interactive terminals has come into daily heavy use by an active University community of more than 9000 users.

In addition, the results have gone far beyond the boundaries of The University of Michigan to systems now operational literally around the world. Installations in France, Great Britain, Canada, Australia, and New Zealand, as well as other installations in the United States are now in possession of facilities developed from the efforts of the Computing Center and the CONCOMP Project. The resulting system continues to remain one of the most effective implementations available for IBM System/360 hardware.

Considering the wide range of research topics explored in depth by numerous faculty and student researchers, it is easy to understand why the full impact of the CONCOMP Project will be realized only as time permits the fruition of the concepts germinated by these efforts.

4. PUBLICATIONS OF THE CONCOMP PROJECT

TECHNICAL REPORTS

1. Mowshowitz, A., Entropy and the Complexity of Graphs, Technical Report 1, August 1967, 120 pp. (doctoral dissertation).

2. Sibley, E. H., The Engineering Assistant: Design of a Symbol Manipulation System, Technical Report 2, August 1967, 31 pp.

3. Childs, D. L., Description of a Set-Theoretic Data Structure, Technical Report 3, March 1968, 27 pp.

4. Pinkerton, T. B., Program Behavior and Control in Virtual Storage Computer Systems, Technical Report 4, April 1968, 160 pp. (doctoral dissertation).

5. Ash, W., and Sibley, E. H., TRAMP: A Relational Memory with an Associative Base, Technical Report 5, May 1968, 80 pp.

6. Childs, D. L., Feasibility of a Set-Theoretic Data Structure: A General Structure Based on a Reconstituted Definition of a Relation, Technical Report 6, August 1968, 40 pp.

7. Mills, D. L., The Syntatic Structure of MAD/I, Technical Report 7, June 1968, 91 pp.

8. Mills, D. L., The Data Concentrator, Technical Report 8, May 1968, 113 pp.

9. Allan, J., Man-Computer Synergism for Decision-making in the System Design Process, Technical Report 9, June 1968, 194 pp. (doctoral dissertation).

10. Frantz, D. R., Brender, R. F., and Foy, J. L., Jr., LOCOSS: A Multiprogramming Monitor for the DEC PDP-7, Technical Report 10, October 1968, 122 pp.

11. Brender, R. F., and Foy, J. L., Jr., and Schunior, T. W., Specialized System Software for Interacting DEC PDP-7 and IBM 1800 Computers, Technical Report 11, December 1968, 95 pp.

12. Brender, R. F., and Foy, J. L., Jr., Flexible High-Speed Interface between IBM 1800 and DEC PDP-7 Computers, Technical Report 12, October 1968, 21 pp.

13. Wallace, V. L., and Irani, K. B., *Network Models for the Design of Stochastic Service Systems*, Technical Report 13 (also Systems Engineering Laboratory Technical Report 30), November 1968, 55 pp.

14. Wallace, V. L., and Irani, K. B., *A System for the Solution of Simple Stochastic Networks*, Technical Report 14, April 1969, 137 pp.; also published as Systems Engineering Laboratory Technical Report 31.

15. Jackson, J. H., *An Executive System for a DEC 339 Computer Display Terminal*, Technical Report 15, March 1969, 70 pp.; also published as Systems Engineering Laboratory Report SEL-32-T.

16. DiGiuseppe, J., *A Survey of Pictorial Data-Compression Techniques*, Technical Report 16, March 1969, 60 pp.

17. Ash, W. L., *A Compiler for an Associative Object Machine*, Technical Report 17, May 1969, 55 pp.

18. Chace, M. A., *A Network Variational Basis for Generalized Computer Representation of Multi-Freedom, Constrained, Mechanical Systems*, Technical Report 18, May 1969, 38 pp.

19. Mills, D. L., *Multiprogramming in a Small-Systems Environment*, Technical Report 19, May 1969, 41 pp.

20. Mills, D. L., *Topics in Computer Communication Systems*, Technical Report 20, May 1969, 106 pp.

21. Wallace, V. L., *On the Representation of Markovian Systems by Network Models*, Technical Report 21, December 1969, 112 pp.; also published as Systems Engineering Technical Report 42.

22. Randall, S., Uppal, N., McClain, G., and Blinn, J., *Implementation of the Queue Analyzer System (QAS) on the IBM 360/67*, May 1970, 152 pp.

23. Jackson, J. H., *SELMA: A Conversational System for the Graphical Specification of Markovian Queueing Networks*, October 1969, 76 pp.

24. Blinn, J., *Systems Engineering Laboratory Circuit-Drawing Program*, Technical Report 24, July 1970, 24 pp.; also published as Systems Engineering Laboratory Technical Report 47.

25. Brender, R. F., *A Programming System for the Simulation of Cellular Spaces*, Technical Report 25, January 1970, 160 pp. (doctoral dissertation).

26. Chace, M. A., and Korybalski, M. E., Computer Graphics in the Dynamic Analysis of Mechanical Networks, Technical Report 26, February 1970, 50 pp.

27. Smith, D., The Audio Response Units User's Guide, Technical Report 27, July 1970.

28. DiGiuseppe, J. L., Gerstenberger, W. S., and Mills, D. L., Data Concentrator User's Guide, Technical Report 28, April 1970, 88 pp.

29. Fronczak, E., Digital Plotting System, Technical Report 29, August 1970, 45 pp.

30. Shadko, F., and Herzog, B., DRAWL 70: A Computer Graphics Language, Technical Report 30, August 1970, 145 pp.

31. Brender, R. F., et al., DEC PDP-7/IBM 1800 High-Speed Interface, Technical Report 31, August 1970, 135 pp.

32. Bolas, B., Springer, A., and Srodawa, R., The MAD/I Manual, Technical Report 32, August 1970, 198 pp.

33. Wagman, Richard, The DF (Display File) Routines User's Guide, Technical Report 33, December 1970, 8 pp. + appendices.

MEMORANDA

1. Galler, B., MAD/I: Preliminary Draft, November 1966, 20 pp.

2. Lundstrom, S., Comparative Evaluation of Digital Equipment Corporation's 340 and 330 Display Controls, June 1966, 17 pp.

3. Lundstrom, S., DEC 338 Light Pen Sense Indicator, November 1966, 7 pp.

4. Lundstrom, S., PDP8-103A Dataphone Interface, November 1966, 7 pp.

5. Mills, D., RAMP: A PDP-8 Multiprogramming System for Real-Time Device Control, May 1967, 24 pp.

6. Frantz, D., PDP-8/338 Executive System, June 1967, 35 pp.

7. Brender, R., Use of DDT with "Interrupts On" Programs, July 1967, 9 pp.

8. Lundstrom, S., and Callan, D., PDP-8 Simulator, July 1967, 27 pp.

9. Lundstrom, S., Engineering Design Report: PDP-8/CR01B Card Reader Interface, Memorandum 9, August 1967, 8 pp.

10. Lundstrom, S., Engineering Design Report: PDP-7/Modified 338 Display Interface, Memorandum 10, August 1967, 23 pp.

11. Mills, D., I/O Extensions to RAMP, Memorandum 11, October 1967, 32 pp.

12. Powers, V. M., PDP-8 Assembler, Memorandum 12, November 1967, 13 pp.

13. Mills, D., System 360 Interface Engineering Report, Memorandum 13, March 1968, 166 pp.

14. Burkhalter, K., DEXEMBLER, Memorandum 14, February 1968, 13 pp.

15. Wood, D. E., A 201A Data Communication Adaptor for the PDP-8, Memorandum 15, February 1968, 134 pp.

16. Burkhalter, K., PDP-8 to 103A Dataphone and/or Online Teletype Interface, Memorandum 16, April 1968, 20 pp.

17. Mills, D. L., and Powers, V. M., PDP-8 Program Relocation: Concepts and Facilities, Memorandum 17, February 1968, 22 pp.

18. Pinkerton, T. B., The MTS Data Collection Facility, Memorandum 18, June 1968, 13 pp.

19. Burkhalter, D. E., A Cyclic Check Computer for Error Detection, Memorandum 19, May 1968, 33 pp.

20. Powers, V. M., Mills, D. L., and Laurance, N., An Assembly Language System for DEC Minicomputers, Memorandum 20, May 1969, 64 pp.

21. Powers, V. M., Portaterm Software, Memorandum 21, March 1969, 20 pp.

22. McCreery, D. R., The Electrowriter as a Computer I/O Device, Memorandum 22, March 1969, 40 pp.

23. Cocanower, A. B., The DF Routines User's Guide, Memorandum 23, May 1969, 5 pp. + appendices.

24. Mills, D. L., RAMP Architecture in a Utility Calculator System, Memorandum 24, May 1969, 24 pp.

25. Jackson, J. H., and Blinn, J. F., Modifications to the SEL Executive System, Memorandum 25, February 1970, 8 pp.

26. Guskin, J., and Dingwall, T., The Discrete, Logical Design, Simulator System, Memorandum 26, August 1970.

27. MOMS: Michigan's Own Mathematical System, R. Taylor, ed., Memorandum 27, March 1970, 190 pp.

28. Goodrich, Mrs. S. D., CAMA: Define-Problem Command, Memorandum 28, June 1970, 31 pp.

29. Julyk, L., Wolf, L. W., The CAMA Data Structure, Memorandum 29, August 1970, 100 pp.

30. Julyk, L., The CAMA Operating System, Memorandum 30, August 1970, 100 pp.

31. Springer, A., Defaults and Block Structure in the MAD/I Language, Memorandum 31, August 1970, 45 pp.

32. Srodawa, R., Sample Definitional Facility in MAD/I, Memorandum 32, August 1970, 65 pp.

33. Wolf, L. W., CAMA (Computer-aided Mathematical Analysis): A General Description, Memorandum 33, August 1970, 30 pp.

34. Gerstenberger, W.S., and Taylor, R. W., Graphics RAMP User's Manual, Memorandum 34, August 1970, 50 pp.

35. Dingwall, T., Julyk, L., and Wolf, L. W., The CAMA Macro Processor, Memorandum 35, August 1970, 31 pp.

36. Dingwall, T. J., Julyk, L. J., and Wolf, L. W., The CAMA Interpreter, Memorandum 36, August 1970, 18 pp.

APPENDIX A

GENESES (GEneral NEtwork SErvice System)

TABLE OF CONTENTS

GENESES

(GEneral NEtwork SErvice System)

1.0 Introduction

The proliferation of programs which facilitate solution of substantial technical problems through the conversational use of computers is inevitable. Each class of problems, and each approach to their solution, is likely to call for a specific graphical programming system with specially tailored inputs, outputs, procedures, and data structures. Evidence of this truth is easily found. The communication between man and computer must be both personal and specific if the conversation is to be effective. Thus, the proliferation of languages and systems is a natural byproduct of the very same factors which make the conversational mode useful.

In the case of the System Engineer, the problem classes which could benefit from designing conversational programming systems are especially numerous because the System Engineer must inevitably deal with problems involving large numbers of entities and open, poorly defined questions. In his case, however, a very high proportion of these problem classes are described by him graphically by means of network models: electrical networks, flow-graphs, control-system block diagrams, queueing networks, switching networks, decision trees, computer programs, etc. On the other hand, the things he does with these networks are also diverse: reliability analysis, optimization, performance analysis, experimentation with adaptive controls, simulation, etc.

The piecemeal programming of each of these classes of problems and approaches to their solution using conventional techniques and software aids is extremely burdensome, and must be avoided. What is needed are new general purpose service systems which make the task of solving such problems as easy as possible for the qualified practitioner of the art who is not also a sophisticated programmer.

The result of the research undertaken here is twofold. First, a framework (both theoretical and practical) within which various network models may be described graphically and analyzed, in a conversational manner, was developed. Second, two systems - one for analyzing queueing networks (stochastic service networks) and the other for analyzing electrical networks - were implemented in order to illustrate the practicality and the generality of the approach taken.

Certain philosophic points of view have governed our work.

First, it is assumed that when substantial technical work is involved, one must highly value speed of response and efficiency. The limits of computer capability will frequently be approached, and all unnecessary delay or problem restriction will seriously interfere with the effectiveness of cooperation between the user and the computer.

In both of the systems implemented a user interacts directly with a satellite computer-display combination (a DEC 339) to describe topographically the network in question and to obtain displays of pertinent results. This computer-display combination in turn interacts with a central computer (an IBM 360/67) which contains a mathematical description of the network and which computes results from this mathematical description. This configuration provides a large, fast computer for the involved computation of network analysis or solution, permits rapid response to a user's demands through use of a small satellite computer, and requires only a relatively narrow bandwidth line joining the two facilities.

Second, the data structure of individual technical entities, particularly when they are relations or functions, will often need to be specifically tailored to known characteristics of the function. Sometimes a function is representable by an algorithm, sometimes by a matrix or regular array, sometimes by rings or lists.

Finally, the individual user must be able to use a very specialized and individualized notation in both picture language and control language. To be forced to use an unfamiliar notation for a familiar problem, or to use a powerful and general notation for a very specialized subclass of problems will again seriously impede cooperation between man and computer.

The following sections outline the achievements of this research. Section 2.0 discusses the philosophy and implementation of the executive system for the remote terminal. The decisions made here were instrumental in creating an atmosphere conducive to effective man-machine communication for interactive design problems. Two such problems were considered and fully implemented, and are described in subsequential sections. Section 3.0 discusses the factors considered in designing the system for the analysis of queueing networks. Sections 3.1 and 3.2 describe the software systems developed for the analysis of these networks for the remote terminal and the central computer, respectively. Certain factors considered in the design of a system for the analysis of electrical networks and some details of the remote terminal software system are described in Section 4.0 and 4.1, respectively. Finally, documentation efforts are listed in Section 5.0.

2.0 Remote Terminal Programming Considerations

In order to use the DEC 339 as a terminal to the IBM 360/67, a decision about the division of labor between the two machines had to be made. Generally, there were two alternatives:

(1) The DEC 339 could be programmed to accept display data from the IBM 360/67 and to encode inputs from

the user to be sent back to this machine. All application programs would then be resident in the IBM 360/67.

(2) Application programs could be written so that they reside partly in the DEC 339 and partly in the IBM 360/67.

The first of these alternatives would provide for easy preparation of application programs, since programming in assembly language would be required only initially to interface the two machines. All application programs could then be written in high-level languages for the IBM 360/67. However, the limited data link between the two machines would degrade response time if this scheme were used. Consequently, the second alternative was selected, with the DEC 339 performing all display-related functions and the IBM 360/67 performing only the large-scale computations of analysis and solution. Display data is not transmitted between the two machines during normal operation, and response time suffers less from data link limitation.

The price that was paid for this type of operation is that the DEC 339 has to be programmed in assembly language for each application. However, attempting to reprogram all input/output devices for each application is impractical. Consequently, a small operating system was needed to handle input/output devices on this machine. Furthermore, since this machine is dedicated to the tasks of communicating with the user and with the IBM 360/67, much CPU time is available in the terminal. Some of this CPU time can be used to establish a multiprogramming environment in the terminal to permit more efficient use of input/output devices.

An operating system, called the "SEL Executive System" [3], was written for the DEC 339 both to establish a multiprogramming environment and to support input/output devices. In order to take advantage of the sophistication of the display control, provision for maintaining highly structured display files is included in the system. (Since any modification of these display files requires synchronization with display control activity, their maintenance is intimately associated with input/output activity and is therefore properly a system function.) The form of these display files was chosen to utilize practically every display control instruction and is well suited to the complete representation of network diagram topologies. It is also applicable to many other two-dimensional display applications.

3.0 Queueing Networks [1], [2], [4]

It is frequently desirable to design stochastic service systems which cannot be adequately analyzed by normal queueing theoretic models. Such systems consist, in the most usual instances, of numerous waiting lines (or "queues"), servers, and controlling or directing stations which determine the discipline of task flow through the system. These systems are often realistic representations, for high-traffic design purposes, of behavior in diverse fields such as plant management, telephone switching, air traffic control, electronic warfare, logistics, and computer system specification and control.

Improved techniques for the design of such systems are greatly needed. For example, a design problem of this type which has not yet been adequately resolved is at the heart of a current crisis in the development of executive systems for demand-paged, multiprogrammed, time-shared computer systems. Problems of this type are also found frequently in the course of selecting the equipment and executive control strategies which permit a modern large-scale computer system to serve a specific environment. These problems have, in the past, required painstaking study and could not be answered generally enough to be treated routinely. Increasingly, as the "computer utility" concept gains acceptance, these "queueing" problems will assume a more and more dominant position as the source of computer system inefficiency.

Similar needs for improved techniques are found everywhere that high-traffic problems occur. This is the natural result of the trend of every technology toward ever larger and more complex systems, with ever greater levels of traffic. New techniques should permit a more routine design of individual systems and strategies for specific environments.

One of the major hopes for significant improvement of design capabilities using queueing models lies in the so-called "conversational" computer techniques, whereby the calculating power of a computer can be closely coupled to the creative power of a design engineer. If a designer can freely pose alternative models to a computer and get immediate evaluation of various performance criteria, he may then generate enough insight (via cut-and-try procedures) to guess a near-optimal design for a system far too large or tightly interrelated to be treated by conventional optimization techniques.

However, the use of conventional techniques for the analysis and design of stochastic service systems requires development of a problem-oriented programming system which is specifically tailored to the demands of conversation. The input language must be terse, the calculations must be fast and reproducible, the variety of models available must be broad, and the results should be in a graphic, insight-provoking form.

The most promising approach to developing such a system is to use graphic displays for the man-computer communication, and to use recursive numerical methods applied to Markovian models for the analysis. Graphical input to the computer in the form of queueing network diagrams provides an ideally compact medium for description of problems, while an output in the form of graphs provides a suitably insight-provoking form for results.

The recursive techniques provide fast calculations with excellent accuracy (hence reproducibility) of the calculated results. They also can be applied to a wide variety of models. This variety of models is considerably broader than is feasible with closed form analysis, but not as general as the slower, less accurate simulation methods. The detail available in numerically solvable models is also midway between that available using closed form analysis and simulation methods.

There are three basic operations involved in a programming system to serve these goals. They are:

1) the servicing of the graphic operations,
2) the translation of the diagram to the form required for (Markov chain) input to the solution system,
3) the solution of the Markov chain.

The second of these is a process which, to our knowledge, has not been previously attempted, and has no obvious solution.

3.1 Queue Analyzer System

A graphical, problem-oriented system has been developed by means of which solutions to simple stochastic networks can be obtained rapidly enough to facilitate conversational use. The problem descriptions are constructed in network diagram form on a remote DEC 339 graphic console under a software system called SELMA (Systems Engineering Laboratory's Markovian Analyzer). Simultaneously, information concerning the construction is sent via dataphone to a time-shared IBM 360/67, which prepares the solutions. The stochastic networks which can be solved are restricted to systems which can be modeled by a continuous-time finite Markov chain, and are solved by numerical solution of the Kolmogorov equilibrium equations.

The central 360/67 programs and data structures which accept graphical descriptions of Markovian queueing networks and which return solutions to these networks to the remote system according to requested specifications are referred to as the Queue Analyzer System (QAS). Because the data required by the analysis program is in a very different form from that describing a network (which is in terms of blocks and connections), a translator, called the network compiler, is required. After the network has been compiled, QAS analyzes the resulting structure for the equilibrium state probabilities and finally it calculates from these probabilities the specific results requested by the console user.

QAS consists of five sets of routines: supervisory and support routines, generation phase routines, compilation phase routines, result phase routines, and documentation phase routines. The supervisory and support routines include: (1) those routines which interact with SELMA to interpret commands received from the DEC 339 and dispatch them to the proper phase routine, to request information from the DEC 339, etc. and (2) those routines which are used in general by all the various phases to manage free core for the various data structures to perform set manipulations on certain data structure, to provide error checking and debugging facilities, etc. The generation phase routines create the data structures representing the stochastic service networks which are described by commands received from SELMA. The compilation phase routines are divided into two groups. The first of these groups operates upon the data structures created by the generation phase routines and reduces (i.e., compiles) these structures to a form more suitable for use by the second group. The second of these groups, operating on the outcome of the compilation, computes a vector of the steady state probabilities for all the states which the stochastic network may assume. This group also creates a data structure representing the multi-dimensional cartesian product state space for the model, along with information which allows mapping from this state space into a linear index for referencing the steady state probability vector. The result phase routines compute and format requested results for display as graphs or printed tables by the remote system, SELMA. Finally routines in the documentation phase save and retrieve QAS data structures and corresponding SELMA display structures for partially or completely solved networks.

The details of these phase programs and the operations performed by them are documented in the report, "An Implementation of the Queue Analyzer System (QAS) on the IBM 360/67" [5].

In order to reduce some of the QAS system programming problems to manageable portions, certain limitations of capability were accepted. These limitations are chiefly ones which limit the meaning which can be assigned to network symbols, and the manner in which the symbols can be related. The current system can treat networks consisting wholly of queues, exponential servers, infinite sources, infinite sinks, random branches and merges, and priority branches and merges. While many other symbols can also be treated in this system, treating all meaningful symbols is by no means a simple task. However, considerable thought was given during the implementation of this system to making the definition of new symbol a relatively easy task, and networks which can be treated in the current system constitute a significant class of models having considerable variety and power.

The current system is rather limited in the kinds of results that can be displayed. Depending upon the user request, the probability density function for the entire model, or for a single element can be computed and displayed as graphs or printed tables. Additional subroutines to compute expected values, marginal probability functions, etc. could easily be implemented. However, a universally acceptable set of results satisfying the majority of users is difficult, if not impossible, to define. Perhaps a facility whereby a user supplied routine is used to compute results would be the most viable alternative. An interesting possibility would be a post-processing system which, using the QAS documentation file, would generate via the Calcomp plotting system a hard copy of the network diagram and plots of useful results.

One of the most crucial improvements which this system requires is a provision which permits compilation of networks with element parameters treated as variables. With such a provision, the operation of compilation will be expanded into three operations: compilation, evaluation, and calculation. The first operation compiles the network in terms of algebraic expressions of model parameters. The second operation evaluates the algebraic expressions after the parameter values are supplied. Then, when a user is exploring a single model with many values of the parameters, the compilation is performed only once, and the evaluation and calculations are performed for each set of parameter values. Currently, changing a parameter values necessitates a recompilation.

The organization of the QAS system is imposed by the calling system, SELMA, which in turn is ultimately subject to the desires of a user. Such an organization imposes a minimum constraint upon the thought sequence of the user. The implementation of QAS and SELMA has demonstrated the feasibility and the usefulness of such a programming system for the conversational design of stochastic service systems using a graphical display for both specifying the stochastic network and evaluating it. The advantages of this approach in terms of speed, precision, and ease of design have been demonstrated.

3.2 SELMA

In order to provide an interactive means of analyzing Markovian queueing networks, SELMA (Systems Engineering Laboratory's Markovian Analyzer) [6] was developed for the DEC 339 graphic terminal. This program accepts a network diagram from the user and transmits the information which is required to solve the network via dataphone to QAS (Queue Analyzer System) on the IBM 360/67. QAS then analyzes the network and transmits back to SELMA the results of solution. SELMA then either displays these results graphically or types tables of the results, depending on the user's specification.

In designing the SELMA/QAS system, the major hardware limitation was the bandwidth (2000 baud) of the 201A dataphone which connects the DEC 339 and the IBM 360/67. If reasonable response times were to be achieved, the information which is transmitted between the two machines had to be minimized. Since the DEC 339 has considerable computing capability, it was decided to perform all graphical operations at the terminal so that picture information does not have to be transmitted in either direction between the terminal and the central computer. Hence, the only information which is sent from the terminal to the central computer is that which is required to describe the topology of the network and parameter values, and the only information which is returned from the central computer is a set of values to be plotted or typed.

In order to avoid transmitting a large amount of information to the central computer just after the user requests solution, changes in the topology of the network and in parameter values are transmitted to QAS as they occur. In this way, the entire network is described to QAS at the time that the user requests solution, and the only delay in obtaining the solution is that which is produced by solving the network and returning the results. However, one graphical input from the user can produce considerable information to be transmitted to QAS. If changes in the network were to be transmitted to QAS as they occurred, the transmission of information to describe the effect of one input could delay response to subsequent inputs. To avoid this problem, a multiprogramming environment is needed in the terminal to allow SELMA to transmit to QAS information which results from one input while responding to subsequent user inputs. In order to provide this multiprogramming environment, an executive system [3] was developed for the DEC 339.

Not only was an attempt made to minimize the dataphone traffic between the two computers, but an attempt was also made to make communication between the user and the terminal as natural to the user as possible. In particular, a technique for interpreting motions of the light pen has been incorporated into SELMA. Generally, whenever the user refers to some part of the diagram with the light pen, a program is scheduled to interpret subsequent motion of the light pen in order to determine what action is to be taken by the program. For example, the user deletes an element symbol by aiming the light pen at it and then stroking vertically across it. He moves the symbol by aiming the light pen at it and then moving away from it. All other operations which are involved in modifying the diagram, with the exception of creating symbols and assigning parameter values, are initiated by recognizing similar motions of the light pen. In this way, the relatively awkward use of push buttons and light buttons is avoided for most operations.

4.0 Electrical Networks

In order to illustrate the generality of the techniques employed in developing a system for the analysis of queueing networks, we wanted to implement a second network analysis system. Since the central computer software required for the actual analysis of a network is highly dependent upon the type of the network (i.e., since it is not easily generalized), we decided to implement a system for which the analysis software already existed.

Already in use in the Systems Engineering Laboratory was a program called CIRAN, which analyzes electronic circuit networks and generates frequency response and time response tables (suitable for plotting). This, then, suggested the very attractive proposal of altering SELMA to draw electronic networks (instead of queueing networks), communicate the network topology to CIRAN, (instead of QAS), and plot the resultant tables as graphs. The result of this effort is SELCIR; Systems Engineering Laboratory Circuit-Drawing Program [7].

4.1 SELCIR

The main design criterion in SELCIR, as in SELMA, was to make the drawing process as quick and natural as possible. Thus, the basic drawing procedure, utilizing thresholding to choose between graphical operations, was adopted with little modification. However, the differences between the semantic interpretations of the two types of networks necessitated some new programming.

The first obvious change was to allow electronic network elements in SELCIR to be rotated, unlike those in SELMA. For simplicity and economy of storage, the elements rotate in 90^0 increments only. This action was originally triggered by a light button but later a new threshold pattern was devised to react to circular motions of the light pen around the element to be rotated. A similar pattern was later installed in SELMA to distinguish between random and priority branches.

Connection lines in SELCIR represent a much more general case than those in SELMA, a fact that eventually required a fairly sophisticated program for connection manipulation in order to operate satisfactorily. The first difficulty encountered came from the fact that elements can have either vertical or horizontal orientation. This necessitated allowing connection lines to connect to elements in both horizontal and vertical directions. In the case of SELMA all elements are horizontal and all connection lines terminate with a horizontal line. Termination of connection lines is triggered by a light pen hit on the end of an element. Since the aperature of the light pen is large, this light pen hit can occur when the endpoint of a connection line in the process of being drawn is still half an inch away from the element. The program must then draw in the rest of the connection line to terminate at the element, making decisions as to what horizontal and/or vertical segments to insert. Initial attempts to do this with a fairly simple algorithm continually resulted in the programs adjusting lines into aesthetically undesirable forms. The scheme finally used has to examine the orientations of the elements involved and the existence of other connection lines at the connection points. While somewhat more involved, this scheme consistently produces good looking connections.

Another complication in the handling of connection lines arose for the delete operation. In SELMA, the semantic constraints on a connection force it to be treated as an indivisible entity; if any electronic circuit simply represents a wire, it seems reasonable to be able to delete only one branch at a time. When a branch is deleted the structure is also scanned to separate elements which are no longer connected and to merge connection lines which no longer branch. This necessitates a somewhat more sophisticated data structure to treat each branch line as a separate entity. The end result is a very pleasing connection operation which allows the user to draw connections and rub-out parts of them much as if he was drawing lines on paper.

The final added feature in SELCIR is in the results request for the analysis program. CIRAN can produce many more different types of output than QAS, requiring a more elaborate results frame. The scheme used allows the user to specify the graph he wishes by choosing the x and y axis labels and their modifiers from light buttons. A set of interlocks prevents the choice of unavailable combinations of entries. The axis labels are set up so that the user can request a graph, make changes to the circuit to which the graph applies, and request another graph without respecifying the coordinate axes labels.

The programs SELMA and SELCIR have diverged somewhat in data structure, but these differences reflect differences in the network interpretation and the analysis program.

5.0 Documentation

A number of reports and a 16-mm film have been produced to document the results of this research. A bibliographical listing of these items is given here.

1. Wallace, V. L., and K. B. Irani, Network Models for the Conversational Design of Stochastic Service Systems, Technical Report 30, Systems Engineering Laboratory, Department of Electrical Engineering, University of Michigan, Ann Arbor, 1968; also Technical Report 13, Concomp Project, Computing Center, University of Michigan, Ann Arbor, 1968.

2. Irani, K. B., and V. L. Wallace, A System for the Solution of Simple Stochastic Networks, Systems Engineering Laboratory Technical Report 31, Concomp Project Report 14, 1969.

3. Jackson, James H., An Executive System for a DEC 339 Computer Display Terminal, Systems Engineering Laboratory Technical Report 32, Concomp Project Report 15, 1968.

4. Wallace, V. L., On the Representation of Markovian Systems by Network Models, Systems Engineering Laboratory Tcchnical Report 42, Concomp Project Report 21, 1969.

5. Randall, L. S., et al.,An Implementation of the Queue Analyzer System (QAS) on the IBM 360/67, Systems Engineering Laboratory Technical Report 43, Concomp Project Report 22, 1969.

6. Jackson, James H., SELMA: A Conversational System for the Graphical Specification of Markovian Queueing Networks, Systems Engineering Laboratory Report 45, Concomp Project Report 24, 1969.

7. Blinn, James, Systems Engineering Laboratory Circuit Drawing Program, Systems Engineering Laboratory Report 47, Concomp Project Report 24, 1969.

8. Irani, K. B., et al., SELMA/QAS, 16-mm movie, Systems Engineering Laboratory, Department of Electrical Engineering, University of Michigan, Ann Arbor, 1970.

APPENDIX B

SIMULATION OF ATRIAL FIBRILLATION

Larry K. Flanigan
Henry H. Swain
John Foy

Fibrillation is a condition in which the electrical activity of cardiac muscle is uncoordinated, giving an appearance of multiple wavelets moving in random patterns. Since the muscle electrical activity is responsible for the contraction of the muscle, this loss of electrical synchrony results in an uncoordinated muscular contraction which fails to move blood effectively through the organism. In the case of ventricular fibrillation, this loss of effective pumping is fatal; for atrial fibrillation, however, normal gravity blood flow from the atria to the ventricles is sufficient to maintain normal activities. The conditions which lead to the initiation and termination of fibrillation have been studied for some time, as have the properties of the heart muscle which presumably allow it to support such unsynchronized activity once initiated. Despite these many studies, however, the exact causes of fibrillation at the cellular level, and the cellular-level mechanisms for sustaining fibrillation, are still highly speculative. The basic problem is that fibrillation is a function of a network of cells (i.e., a tissue), not of an individual cell; hence, the underlying mechanisms are related not only to individual cell behavior but also to the interconnection patterns of cells and to the manner in which cells affect one another. To investigate this problem, a cellular model of cardiac muscle has been devised, and a program to simulate this model has been written to run under MTS on the System/360.

The cellular model is based on a set of mathematical-logical equations which provide a description of a cardiac cell, together with a set of rules which provide the local and global network structure and neighborhood relations. This model was originally developed to simulate A-V node tissues (1) but has since been adapted to atrial tissues. The simulation program is written in FORTRAN IV, together with a few System/360 Assembler Language routines, and it has been devised to run in a time-sharing system in which a user is in direct communication with the program at all times. To this end much of the decision-making was not coded, but was left to the terminal user to be done in an on-going interaction with the simulator. To aid in this control process, an internal command language has been devised which is interrupt-driven from the user's terminal. Among the more important functions provided are the following:

1. Complete control over several forms of output and its frequency, including various data generation mechanisms for later batch processing;
2. Dynamic control over cell types and network structure;
3. A save/restore capability allowing multiple simulation runs from a given network status;
4. Dynamic control over input stimulation patterns to the network.

Several other less important, but convenient, functions have also been included in the command language.

The current model uses a two-dimensional network of approximately 500 cells in the simulation; a maximum network of 625 cells may be obtained with the current program. The use of a two-dimensional network, rather than one of three-dimensions, saves computer time and is not unreasonable in view of the thinness of atrial walls. In order for the user to make decisions during a simulation run he must have available a great deal of information, such that he may know at each instant the current network status. Since hard copy output is too slow for this purpose, the simulator uses the DEC 338 display to present a "picture" of the network. This, together with certain statistics produced on an attached teletype, allows the user to "see" the network status at all times. The display is a two-dimensional representation of the model network, showing for each cell its current state; this display is accomplished through the DF Routines (2) provided in MTS. As a simulation proceeds, the cell state changes give an effective picture, via the display, of activity waves moving through the model network. Through the internal command language the display may be turned on and off, the frequency of displays may be varied, and the specific cell states to be displayed may be controlled. Currently, all interaction through the display via light pen interrupts will be added. While the use of the display is at a rather primitive level in this work, as far as programming techniques are concerned, it is an absolute essential in the simulator to provide meaningful information in a reasonable format at a rate which allows economic interaction. In addition, the display output is immediately understood by physiologists without their having to interpret pages of data or statistical results of experiments.

Initial tests on the model were highly satisfactory and reproduced, in part, the results obtained by Moe (3) using a simpler model. Currently, the model is being used in a series of tests designed to investigate critical fibrillation frequencies as reported by Swain and Valley (4). The early tests in this series substantiate the existence in the model of such frequencies. Ultimately, it is hoped that information may be obtained as to the important cellular parameters which are critically involved in initiating and terminating fibrillation. Once these parameters are identified, a series of tests will be devised to test various hypotheses concerned with the use of certain drugs to prevent or control fibrillation, and to determine how in the model such drugs produce their known effects. If successful, these model tests should produce hypotheses about the manner in which cellular behavior supports fibrillatory behavior in cardiac tissue.

REFERENCES

1. Flanigan, L. K. and Swain, H. H., "Computer Simulation of A-V Nodal Conduction," The University of Michigan Medical Center Journal, Vol. 33, No. 4, 1967, pp. 234-241.

2. Cocanower, A. B., The DF Routines User's Guide, Memorandum 23, CONCOMP Project, Computing Center, University of Michigan, Ann Arbor, May 1969.

3. Moe, G. K., Rheinboldt, W. C., and Abildskov, J. A., "A Computer Model of Atrial Fibrillation," Am. Heart Journal, 67:200, 1964.

4. Swain, H., and Valley, S. L., "Critical Fibrillation Frequency," The University of Michigan Medical Center Journal, 1970, in press.

APPENDIX C

MAN-COMPUTER GRAPHICAL SYSTEMS

William Ash
Robert W. Taylor
Edgar H. Sibley

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This report summarizes a research effort which explored advanced techniques in man-computer graphical systems. The work was done under the CONCOMP Project starting in late 1966. Personnel involved, at various times, were Dr. E. H. Sibley, W. L. Ash, D. G. Gordon, Captain R. McDonald, and R. W. Taylor. Major portions of the results have been published elsewhere (1-6). This report serves the dual purposes of providing more detail than the above references, and of providing a coherent framework in which to present the results.

The work started with the development of the TRAMP system, which operates under the Michigan Terminal System, MTS. TRAMP is a simulated associative memory system with a deductive capability. It is embedded in the UMIST* language, which has served as an elegant host. This effort was undertaken as the first step in the research because of our convictions that associative memory schemes would be vital in any advanced graphic system. Our results and the presence of associative memory systems at other installations, e.g., Lincoln Laboratories, give evidence regarding the validity of this assumption.

TRAMP has proven useful in other applications besides the graphical one reported here. It forms the main data storage and retrieval mechanism for

*UMIST is a local dialect of the TRAC T-64 language, and was implemented at The University of Michigan with the cooperation of Mr. C. N. Mooers, creator of the TRAC T-64 language.

the CONCOMP accounting program which was used for all budgeting and accounting on the Project. Also, TRAMP was used extensively in a doctoral dissertation on automated engineering design (7). Finally, TRAMP will serve as the basis for further work in a new approach to question-answering systems.

This report concentrates on aspects of generalized computer graphics and presents thoughts about the state of the art in this area. Section 2 presents a taxonomy of generality in existing computer graphics systems. This taxomony serves as background for questions and conclusions in later sections. Section 3 compares and contrasts ring structures and associative structures. Ring structures are widely used in the computer graphics community, yet there has been little questioning of their appropriateness or their shortcomings. This section attempts to do that and to point out some advantages of the associative approach. Section 4 reviews TRAMP in enough detail for the reader to get a flavor of the language and its capabilities. Section 5 documents the TRAMP/RAMP graphic system, which served as the primary vehicle for our graphics research, and comments about this approach to graphics research. Finally, Section 6 presents a detailed example of the kind of exercise which can be carried out in the TRAMP/RAMP system. Also included are comments about the problems which these exercises raise and some suggestions about further work along these lines.

2. A TAXONOMY OF PROBLEM-SOLVING GENERALITY IN GRAPHIC SYSTEMS*

2.1 INTRODUCTION

This appendix presents a framework through which various graphics systems can be compared and contrasted. As is true of any classification, it is of interest to the extent that it clearly delineates important features of the various classes. If features other than those treated by the taxonomy are of more concern, then other means of comparison must be found. But in the areas where the taxonomy applies, it can provide a measure of order and structure where none existed before. It can thus be of aid in visualizing the scope of certain existing graphic systems, and also point out the difficult and unsolved areas of the field, which should be approached carefully when new systems are being designed.

The primary dimension under consideration in this taxonomy is capability for problem-solving over a range of problems. Thus system A will rank higher on the scale than System B if A has the potential for solving a wider range of problems than B.

Certainly there exist other dimensions on which systems could be compared. Sophistication of approach to a given problem is one possible measure. Others which could be of interest at various times are efficiency, modularity, transportability, extensibility, and device-independence. It would certainly aid the graphics community to have complete classifications along all of these

*This appendix is based, in part, on a presentation given by EHS at the 1970 SJCC.

dimensions, for then one could study the more advanced systems along whichever dimensions were of interest. Moreover, new systems could hopefully incorporate the best features of all dimensions. Such an extensive classification scheme has been undertaken in the data base system area (8), and a similar effort in graphics would undoubtedly aid the user population in understanding the field. But such work is well beyond the scope of a single author and is certainly not being attempted here. Rather, it is felt that the generality dimension can serve as a starting point for other classifications, and also can serve immediate purposes with respect to estimating the problems likely to be encountered in undertaking a new system.

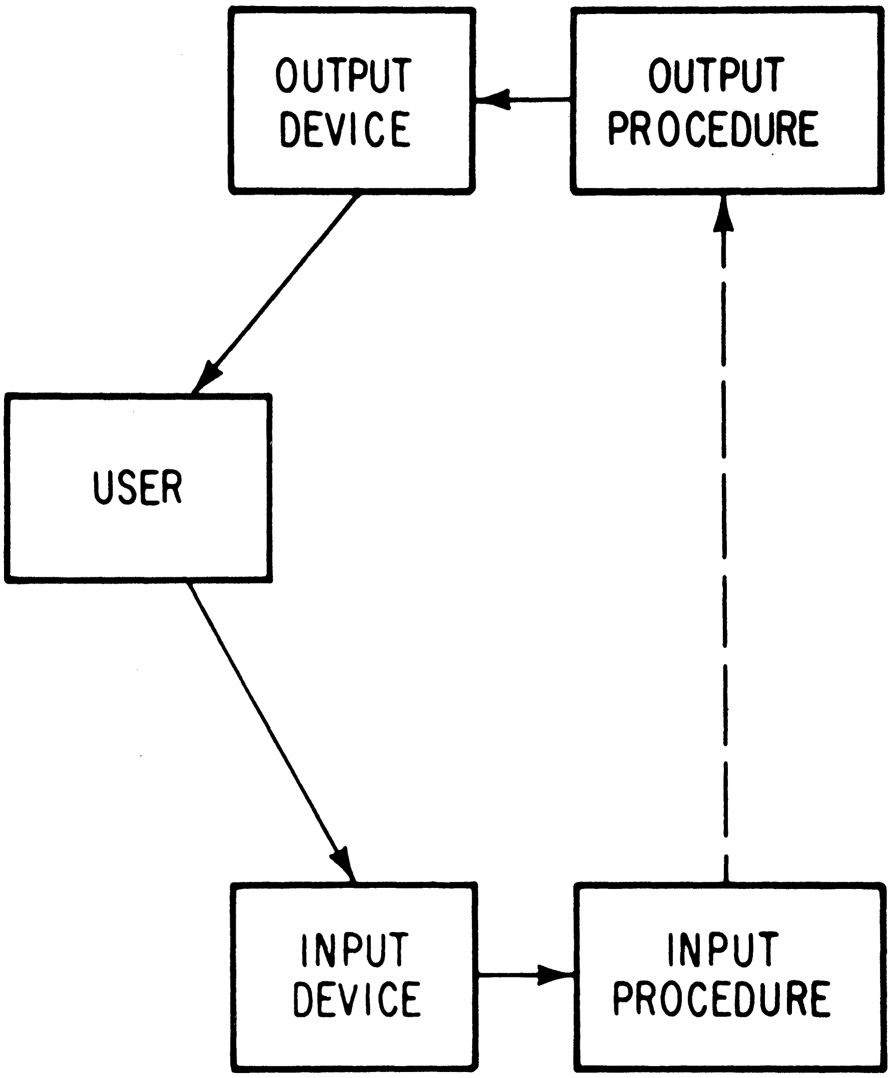
It is also worth noting that although the taxonomy will be presented in terms of discrete levels, it is more realistic to assume that this dimension is continuous. Thus the classifications presented here do not necessarily have clearly distinguishable boundaries. The examples are provided in an attempt to make the taxonomy understandable. The inclusion of two or more systems in the same class does not imply precisely equal generality.

2.2 THE TAXONOMY

This section presents the taxonomy. The reader should note that, for the most part, the capabilities of a system at level 1 encompass those of all lower (numeric) levels. Thus a subset relationship is implied.

The first level of problem-solving generality is illustrated in Figure C-1. It is designated as "level 0 problem solving" because systems of this sort solve almost no problems at all. Only the most trivial kinds of

LEVEL 'O' PROBLEM SOLVING



Examples: Tabulating Machines
α-scopes
Storage Tube Displays

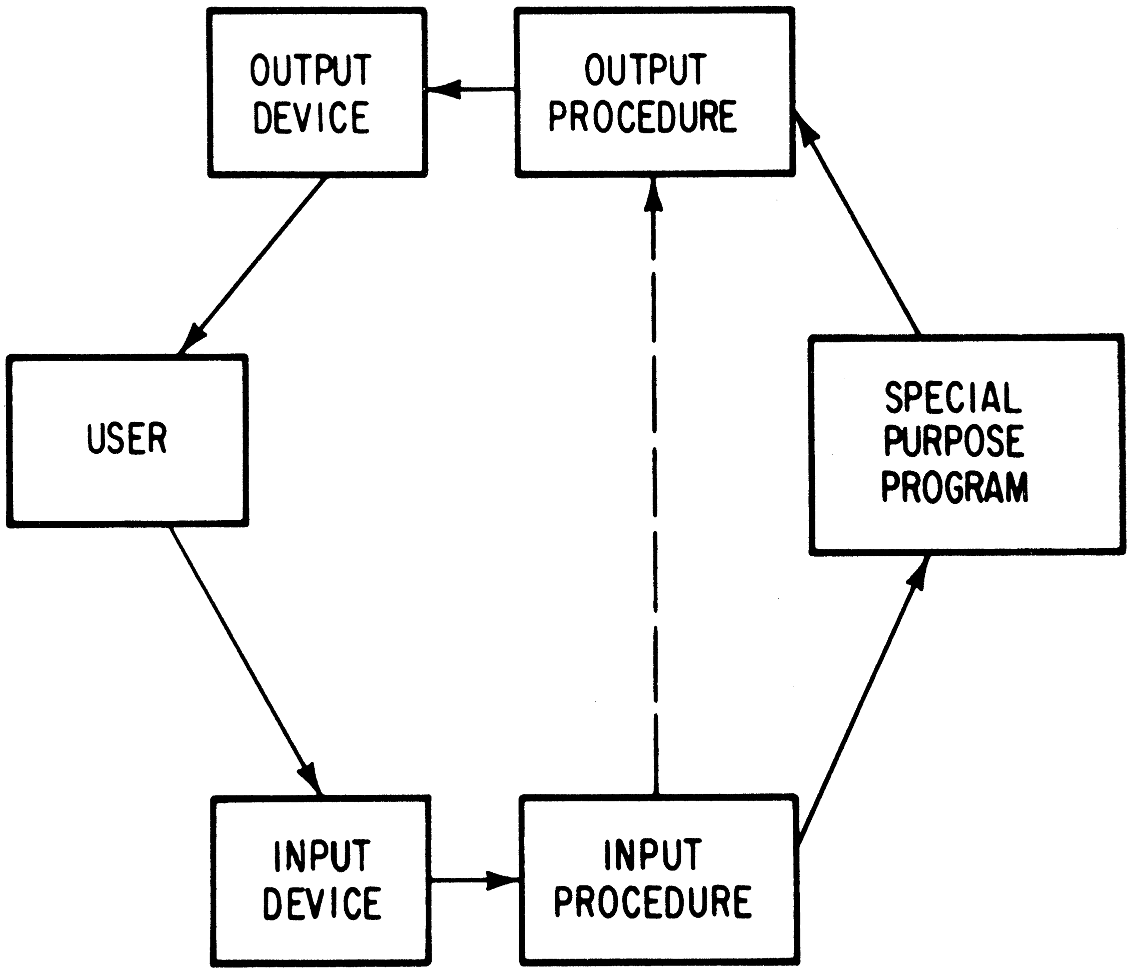
Figure C-1

processing are within the capabilities of a level 0 system.

The system, of course, involves a user who generates data which is entered through some sort of input device. The device is supported by a device support routine, which will be called the input procedure. Control then passes to the device support routine for the output device—the output procedure—then to the output device and the user. Note that nothing in the way of a general-purpose computation facility is implied in the passage from the input to the output procedure. If these procedures reside in a powerful computer system, this power is not used. The only transformations allowed are those of format conversion and rearrangement. More typically, however, the input and output procedures will reside in a processor of limited power. The examples are drawn from this environment. Tabulating machines, alphanumeric terminals, and storage-tube displays functioning as stand-alone units are all examples of this level of processing.

The obvious next step is to augment the level 0 capabilities with a program running on a computer. This is shown in Figure C-2. The restriction is that the program serves a relatively special purpose. By this we mean that while the program may be especially good at solving the problem for which it was designed, it is of no use in solving problems in a different area. Certainly the vast majority of programs in existence today are of this special-purpose variety. Two examples, which have received publicity in the graphics community (and elsewhere), are the Electronic Circuit Analysis Program (ECAP) and the Numerically Controlled Tool Programs, which produce paper tapes to drive automated milling machines. To the extent that a problem is well

LEVEL '1' PROBLEM SOLVING



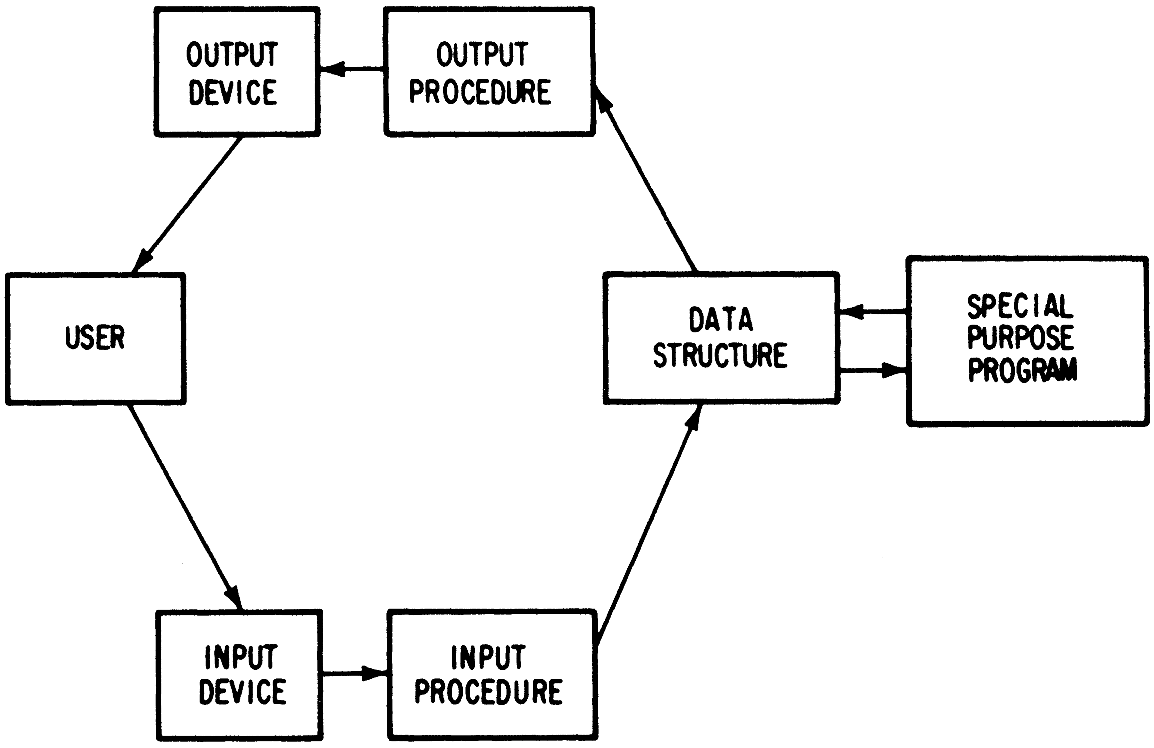
Examples: ECAP
N/C Tool Programs

Figure C-2

defined, well understood, and relatively stable over time, that problem is a candidate for solution via a special purpose program. Economic considerations will probably be the final determinant in the construction of such a program.

The next level of problem solving generality, illustrated in Figure C-3, is a natural progression from level 1 problem solving capabilities. Here, a family of special purpose programs is involved in the problem solution. These programs communicate with each other through a common data structure. When one observes that the family of special purpose programs might be running parallel on a time-sharing system, each possibly serving a different user, the power of this level begins to be revealed. Level 2 problem solving is clearly more general than level 1—the class of solvable problems will in some sense be the union of the set of problems solved by the family of participating programs. The price paid for this generality is embedded in the data base sharing facility. Because each participating program must adhere to the structure and conventions of the data structure provided, it is a practical certainty that some programs will be forced to use a structure and/or algorithm which is not the most efficient for that single problem. Overhead costs will also result from the fact that the volume of data maintained in the data base will be the union of that data necessary to solve the problems attacked by the family of programs. It is unlikely that a given user will need all of these data and/or programs all of the time. Yet it will be hard to predict the requirements of the family of programs over the entire user population. Thus, unless techniques for dynamic restructuring of the

LEVEL '2' PROBLEM SOLVING



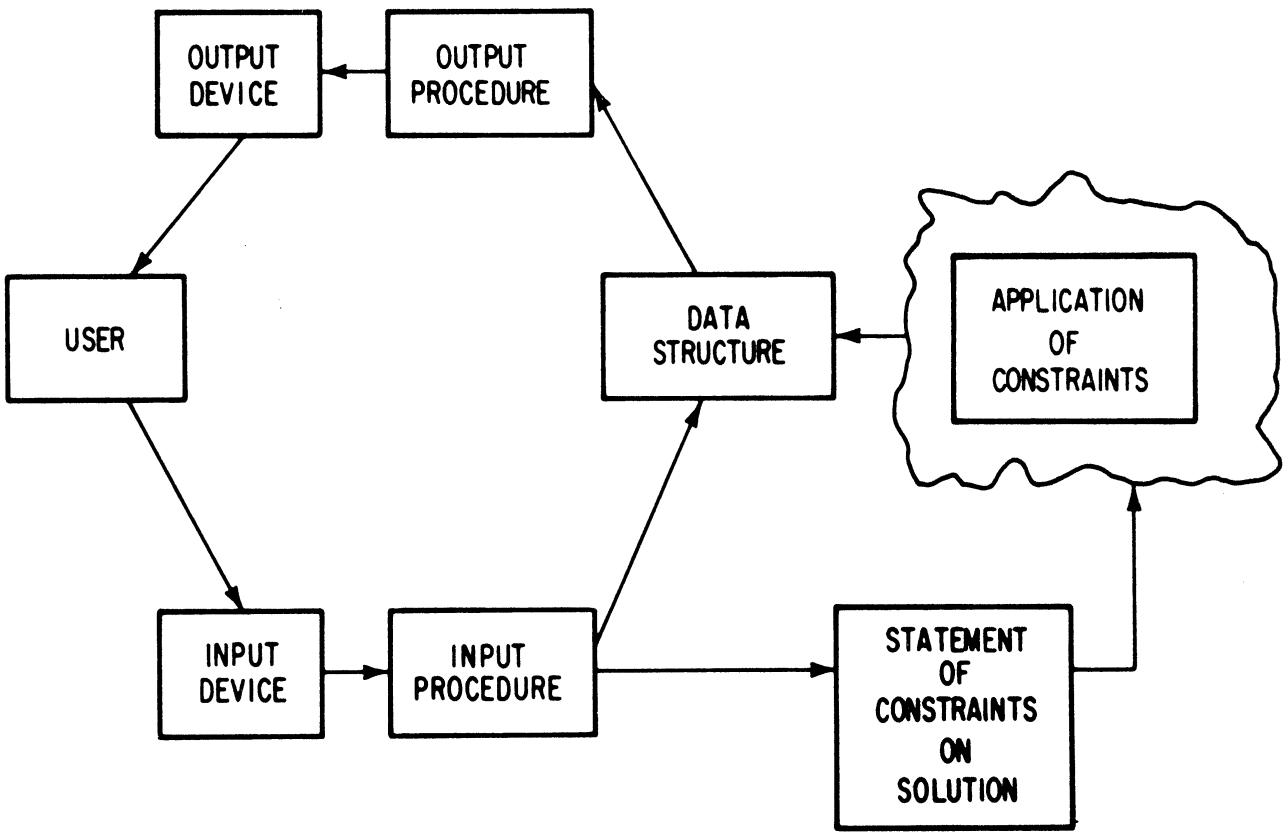
Examples: Data Base Management
Graphics-in-Industry

Figure C-3

data base can be developed, the various users will be forced to share the overhead costs. Finally, it is worth noting that as the family of programs grows over time, it is likely that each new program will be more and more difficult to incorporate into the total system. This results from the fact that various flexibilities will have been precluded by previous decisions regarding other programs in the family. This further points out the need for data base restructuring capabilities. The examples shown suggest that these problems are being attacked, both by the graphics community and the data base management community. Close cooperation between these two groups will be a necessity for the next few years.

Level 3 problem solving (Figure C-4) represents the beginning of a different approach to general problem solving systems. Instead of providing a family of programs, each of which solves a specific user problem, the system provides less specific capabilities which form the basis of a solution for a number of problems. The user's job then becomes one of specifying to the system, along with the data, the proper combination of these basic elements. The more sophisticated of these systems also provide capabilities by which the system itself can discover the proper combinations. Probably the best known example of this approach is the SKETCHPAD system developed by Ivan Sutherland. Here the basic capabilities, aside from those of drawing objects on the screen, were various geometric "constraints"—make parallel, fix length, etc.—along with a solution procedure which carried out constraint commands from the user. The basic capabilities were sufficiently general so that Coons' five problems could all be solved by SKETCHPAD. The SKETCHPAD

LEVEL '3' PROBLEM SOLVING



Examples: SKETCHPAD

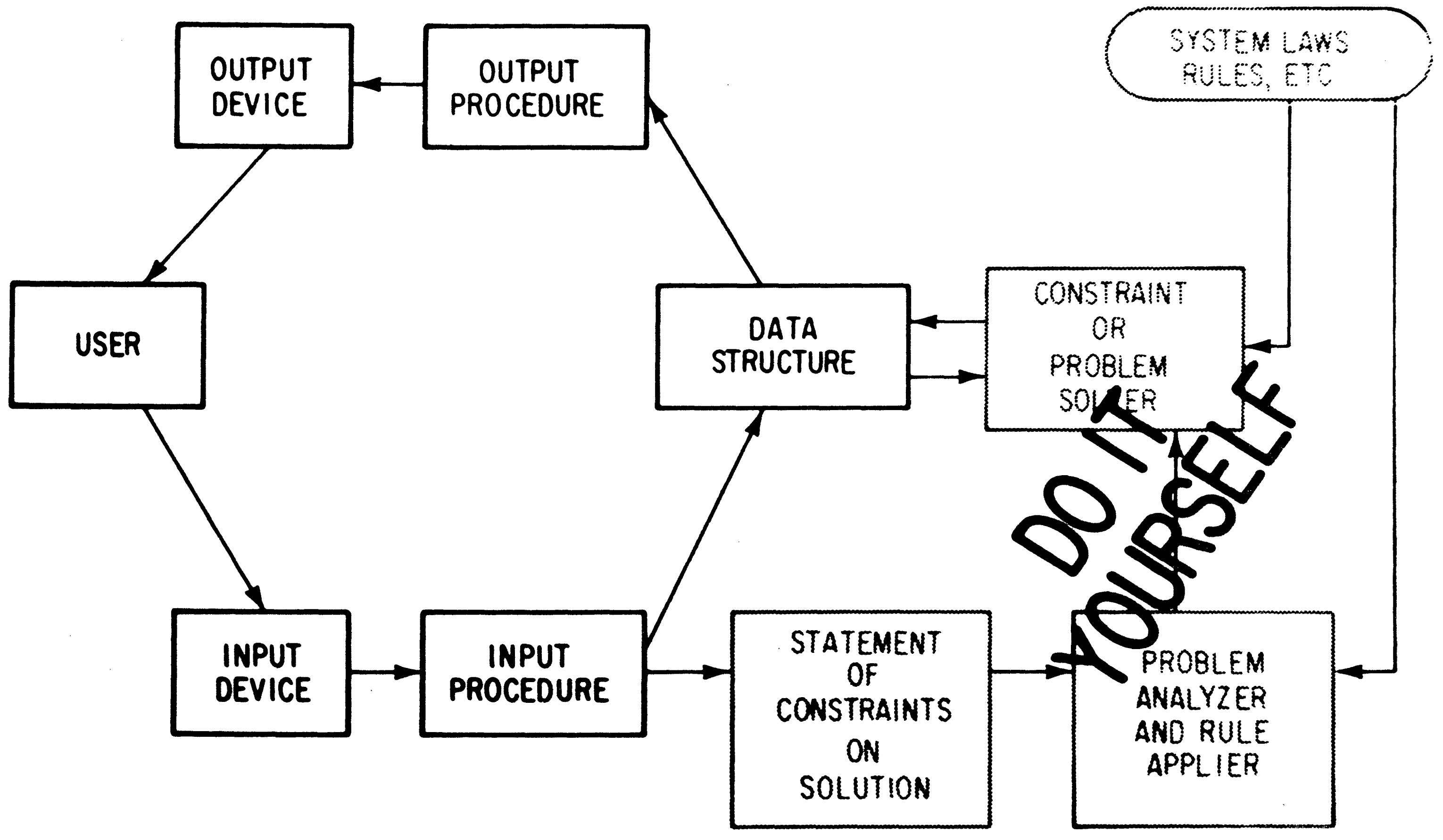
Figure C-4

data structure, in contradistinction to those on level 2, did not have to be altered upon addition of a new capability in the problem solving family. However, this new capability did have the restriction that it be compatible with the overall solution procedure, which was general enough for a certain class of problems, but not totally general. (For further discussion of SKETCHPAD see Reference 5.)

Thus in a further move toward generality, an attempt must be made to generalize the solution procedures as well as the basic operators, be they geometric (as were Sutherland's for the most part) or not. Before discussing an approach to the total problem we wish to mention an intermediate approach. This is designated level 4 problem solving (Figure C-5). The principal example is the AED system (14). It might be argued that AED is a system for building systems and thus is a meta-system for any of the levels discussed here. This is certainly true and, when viewed in this way, AED does not fit into the subset-superset progression we have been developing. On the other hand, AED contains a number of generalized packages and techniques which are oriented toward the generalized approach of level 3 and beyond. Its capabilities for generality thus make it worth mentioning in this context. However, because in practice a user must expend substantial effort in the production of modules which comprise the problem solving machinery, AED cannot be considered as completely generalized.

What then is an approach to a very generalized level-5 system (Figure C-6)? As can be seen, it has all the capabilities for user interaction in a drawing mode as has the lower-level system.

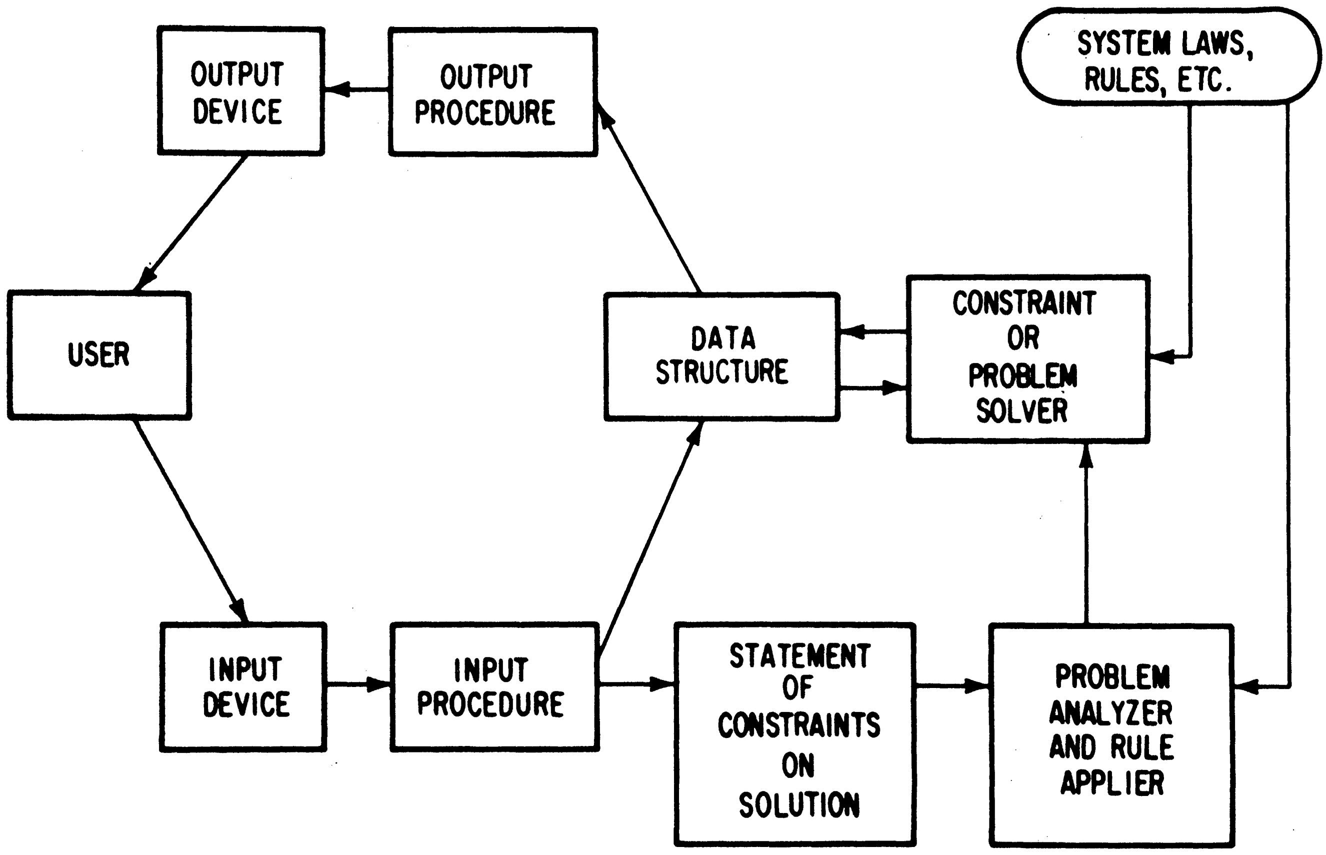
LEVEL '4' PROBLEM SOLVING



Example: A.E.D.

Figure C-5

LEVEL '5' PROBLEM SOLVING



Examples: ?

Figure C-6

In addition, it has problem solving machinery of a generalized sort. In other words, the problem solving machinery is in the same spirit as that of SKETCHPAD. But in addition, the System Laws and Rules are specifiable arbitrarily by the user. Thus the problem solving machinery must be able to cope with a variety of situations involving arbitrary picture semantics. It is our belief that such a system would, for all practical purposes, be indistinguishable from a sophisticated artificial intelligence system. Our thoughts as to how one might approach the building of such a system are outlined in Reference 5. Briefly, we view the system as having a generalized parsing facility in order to analyze user statements of the problem. We also see the necessity for a Problem Analyzer and Rule Applier which restricts the domain of possible environments depending on prior constraints and other environmental statements. Finally, we see the necessity for a generalized problem solving mechanism which takes the restricted environment and attempts to find a problem solution, which is also defined by the Problem Analyzer, within the environment. Notice that in contradistinction to the AED approach, the user need only specify the applicable System Laws and Rules. There would be no necessity, in a truly general system, for laborious construction of problem solving packages.

The taxonomy does not depend strictly on the existence of these three separately defined boxes. We would be willing to call any system which deals with arbitrary picture semantics a level-5 system. We do not know of any such system.

2.3 CONCLUSIONS AND COMMENTS

Having presented the taxonomy, it is appropriate to draw several conclusions from it and make some comments about it. Such conclusions are, to a certain extent, independent of the taxonomy itself.

First, because level 5 is considered the most general type of problem solving system, the authors do not wish to imply, and it is probably a mistake to infer, that state-of-the-art (graphic) systems should attempt to problem solve at this level. The authors believe that the construction of such a system is beyond the state-of-the-art. Thus, it is not likely that any useful results will be forthcoming from the generalized approach for many years.

It is also worth noting that level 5 problem solving will, in its most advanced state, always be grappling with problems that are not well understood. As soon as a problem is well understood, it is likely that a more specialized approach to the problem solution, possibly using the tools of level 4, will be the more practical way of proceeding. In other words, the extraneous generality can be discarded in the interests of, for example, efficiency, once the problem is well understood. Because the generality is discarded, the system which deals with the problem will not be a level 5 system.

The authors feel that research in generalized graphic systems is currently at two levels. Most of the practically oriented research is at level 2, i.e., producing systems in which dissimilar programs (possibly in rather different programming languages) can access a common data base. The problems in this area should not be underestimated. Many basic computer science questions concerning the nature of data and file structures and interfaces with an operating system form the heart of this problem. On the other hand, the resolution of these questions will yield immediate benefits, both in this type of system and elsewhere.

There is also some existing research at level 3. The on-line mathematical systems tend to provide a set of routines for solving a fairly wide class of problems in mathematics. Results are frequently presented graphically. Often input statements, which are usually from a keyboard, may refer to results on the screen via some pointing device. Mathematical systems form an excellent area for such generalized facilities. The range of applicable problems is wide, yet the field is obviously well formalized and oriented toward algorithmic treatments. Intensive study of such systems may well yield the most significant advances on this level of problem solving.

Another comment is: To what extent is this taxonomy independent of graphics? There are several aspects to this question. In many ways, the taxonomy does not depend on graphical communication at all. Problem solving which delivers answers from input data need not, in a sense, be concerned with such incidentals as input and output formats and presentation methods. On the other hand, graphics is a very natural means of communication for man, and it has striking potential when coupled with a sophisticated problem solver. If the machine could ever be a level 5 problem solver, then remarkable results would ensue. But a sophisticated problem solver on a lower level can also benefit markedly when graphical I/O is incorporated. The recognition of this fact has caused a certain muddling of the two areas. Sophisticated problem solving (often in the area of computer-aided design) and graphics are treated as a relatively inseparable pair. There is nothing inherently wrong with this coupling, but it is important not to confuse the problem-solving-oriented parts of such systems with the graphical I/O portions. To do so is to complicate two already difficult areas. On the other hand, capabilities in problem solving affect the graphic communication aspects and vice versa. Although the two areas are separable, the interrelationships must be recognized and incorporated into the design.

Finally, although this paper has concentrated on classifying levels of generality in problem solving and has tended to ignore I/O features and techniques, there is a need for detailed description and classification of the various I/O techniques used in graphics systems. Such a classification, to be rigorous, would probably have to draw heavily on statistical human factors data, much of which is nonexistent (but see Reference 12). Nevertheless work along these lines would be of great value.

3. CONSEQUENCES AND IMPLICATIONS OF AN ASSOCIATIVE MEMORY DATA STRUCTURE

It has been widely recognized for several years that the data structure plays a key role in graphic systems. By data structure we mean that portion of the system which serves as the chief source of operands for other portions, and which also contains representations of relations between operands and any rules governing the processing of those operands. The data structure gives meaning to the shapes created on the CRT. Its existence is the difference between "intelligent geometry" and "unintelligent geometry," to quote one author (13). Notice that nothing has been said in this definition about how the various operands, relations, and rules are represented in computer storage. Methods of doing this will be called "storage structure," and clearly data structure needs to be mapped to "storage structure" before processing begins. This mapping will not be discussed here. Rather, emphasis will be placed mostly on the data structure aspects of graphics processing.

Sutherland (9) was the first to show that complex ring data structures were sufficient to handle the meanings of pictures for the various constraint problems which his system could solve. Later work on the CORAL ring structures (14) can be considered a refinement of Sutherland's original approach. Others have followed the ring structure approach (15), and it is probably fair to say that rings, with nodes having various internal structures, are the most common method of implementing graphics systems data structures at the present time.

In the interest of clarity and precision, the following (complicated) definition of a ring data structure is offered.

Definition 1. A <u>ring</u> <u>data</u> <u>structure</u> is an octuple

$< N, R, D, \delta, \rho, \tau, \eta, \beta >$

where

(i) N is a finite set of Nodes

(ii) R is a finite set of Rings

(iii) D is a finite set of data types

(iv) δ is a mapping

$\delta : N \rightarrow 2^D - \emptyset$

such that δ is a total function

(v) ρ is a mapping

$\rho : N \rightarrow 2^R - \emptyset$

such that ρ is a total function

(vi) τ is a mapping

$\tau : N \rightarrow 2^R - \emptyset$

such that (i) τ is a total function

(ii) $\forall < a, b > \varepsilon t, < c, d > \varepsilon \rho$

$a = e => b \subseteq d$

(vii) η is a mapping

$\eta : N \rightarrow N$

the detailed restrictions of which will be described below.

(viii) β is a mapping

$\beta : R \rightarrow 2^N - \emptyset$

such that β is a total function

Thus every node has associated with it a non-null set of data types which are said to be the components of that node.* We also make the restriction that $N \cap D = \emptyset$ so that an allowable data type of a node cannot be a node. Each node is <u>allowed to be on</u> some set of rings, which is specified by the ρ mapping. Each node <u>is on</u> a set of rings specified by the τ mapping. We further see that

(i) each node is on at least one ring (from iv);

(ii) each ring contains at least one node, which will be called its <u>header</u> (from viii);

(iii) the set of rings that a node is on is a subset of the set of rings that that node is allowed to be on (vi.ii). This subset, thus τ, probably changes with time.

Finally, the mapping η specified which nodes are <u>directly connected</u>. (This too is subject to change over time.)

The restriction on η is as follows

(i) For all $n \in N$, there exists a natural number k such that

$n \leq \eta^k(n)$

and for all i, $1 \leq i \leq \lambda$ $\exists\, m_i \in N$ and a ring $r \in R$ such that

1. $m_i = \eta^i(n)$
2. n is on r $\Rightarrow m_i$ is on r
3. $n \cup \{m_i\}$ = Range (β/r), the range of β is restricted to r as the domain element.

*In practice, the components would be data values of these data types, but we need not be concerned with that here since this aspect of the processing will not be of great concern. δ is included merely to point out that the nodes themselves have further structure.

Which, said in words, requires that it be possible to start a node n on a ring r and, by composing the η mapping, reach all members of the ring r with a new member of r being reached at every step. Further, one must arrive back at the original node n after k steps.

The typical way of realizing this definition (with perhaps minor alterations) is to define blocks of storage which can contain both the data components of a node and places for indicating the set of rings which that node is on. Further, it is often the case that the storage allocated will be large enough to hold the set of allowable rings for that node. The techniques of this realization are common, and will not be discussed here, except to note that the space used for indicating ring membership typically contains one or more pointers to other members of the ring.

A final comment regarding rings is: Why does the ring data structure exist? Certainly the complexity of the definition indicates that it might be a hard data structure to use. The reasons are, of course, largely historical. Ring storage structures were shown to be sufficient to deal with Storage Structure problems. Thus one way of proceeding was to find a data structure language which could be mapped to a ring storage structure. This was done (15, 16), but the resulting languages have tended to reflect most of the associated storage structure complexity.

Various people have questioned the universal applicability of a ring organization.* Their objections fall into several categories:

*It should be noted that the matrix methods for curve and surface generation are not necessarily different organizations. Typically, a node in one of these ring structures will contain a matrix, which is subsequently used for line/surface generation.

(1) Ring data structures are difficult to use. The languages are typically very procedural and require a thorough knowledge of the accompanying storage structure. Thus data structure manipulation is at best tedious. Worse than that, the complex manipulations necessary lead to a variety of errors and thus raise the cost of graphic systems. Finally, since the storage structure of various nodes in a ring data structure is not uniform, the accessing language is more difficult that necessary, both to learn and to implement.

(2) The meaning of the various fields of a ring element is usually bound at compile time. As a practical matter, the structure of the ring elements tends to be bound much earlier than this— structures designed early in the life of a system tend not to be changed very often, and only the existence of rather sophisticated compilers allows any change at all. Reasons why one might want the ability to delay binding later than compile time will be given below.

(3) The ratio of pointer to non-pointer data tends to be excessive. While this multitude of pointers usually allows quick access to a variety of data (a necessity if the graphics system is to be interactive), it is also true that the deletion and/or insertion of nodes into an existing structure can cause severe overhead costs. Idealists might hope that there would be a technique for avoiding these high costs with only a small penalty in increased accessing

time. Dreamers on a smaller scale might hope that there would be a way of choosing from a range of values in an access/overhead tradeoff.

These objections led to an investigation of an associative structure as a possible alternative for the data structure of a graphic system. By associative structure we mean software simulation of an associative memory, where data is accessed by value rather than be name. Detailed investigations into this approach have been reported elsewhere (1, 2, 17-19). A summary of salient features will be presented here for completeness.

Once again we offer a definition

Definition 2: An associative data structure is a quintuple

$$< A, O, V, \alpha, \beta >$$

where

(i) A is a set of Attributes

(ii) O is a set of Objects

(iii) V is a set of Values

(iv) α is a mapping

$$\alpha : O \rightarrow A$$

(v) β is a mapping

$$\beta : O \rightarrow V$$

Thus we have a set of objects which have attributes and values associated with them. Notice that there are no restrictions on α and β. Thus they need not be totally defined over all objects, nor need they be functions.

Since there are only Attributes, Objects, and Values, any accessing of such a data structure can use only elements of these sets. This has led designers of associative data structure access languages to think in terms of content addressability and "associative processors."

The essential feature of an associative processor is that it has, in the conventional sense, no explicit addresses or access paths. Reference to storage is made by specifying all or any part of an associative cell, and all cells which match this field(s) are referenced. The conventional computer store may be thought of as a special (degenerate) case of an associative memory, in that the association is between the physical address and its contents. However, reference can be made only by specifying the address—one cannot ask directly for all cells which are zero! The true associative memory is accessed by specifying any of the N participants in the association.

The following example demonstrates why an associative processor can effectively be employed as an application of content addressability. Suppose we wish to know the phone number of Clark Kent. It is simple to look it up in the local phone book. It is, however, quite a different matter to find out whose number is 764-6148 (using the same directory). An associative processor would find both tasks equal. In this example, the "association" is between a subscriber's name and his phone number. In translating this to a two-place relation, "phone number of" could be the relation, and using the < R,x,y > format we would say: < Phone number of > < Clark Kent > is < KR 9-8765 >. This is a type of associativity wherein we may now directly reference this triple by any of its content-addressable components or combination thereof. If we use only the first component, phone number, in a search, what will be referenced is the entire book. If we specify two components: phone number and 764-6148, then we are referencing directly all associations containing those two components, viz., the associations containing the name(s) of the person(s) having the phone number 764-6148.

The general strategy used to effect the simulation of an associative processor and an approximation to content addressability is that of hash-coding. For those unfamiliar with the term, hash-coding is simply a technique whereby an arithmetic transformation is applied to an external name to generate an internal address. Hash-coding by itself provides a restricted but significant approximation to content addressability, but hashing alone does not provide any kind of associativity and there is always the problem of "collision," i.e., when two distinct names hash to the same internal address: $X \neq Y$; $H(X) = H(Y)$. Hashing partitions the space of names into equivalence classes. Hopefully, each class has only one element, but two or more names may be equivalent under this partition.*

Feldman (17) was the first to use the terminology.

$$A\,(O) = V$$

$$<\underline{A}\text{ttribute}> \text{ of } <\underline{O}\text{bject}> \text{ equals } <\underline{V}\text{alue}>$$

and other systems (1, 18) have adopted it. Thus the <u>Associative Triple</u> is <A,O,V>. Each of the three components is a non-empty set. To the data structure this is an ordered triple, but no interpretation or meaning is

*Even restricting names to four characters of the English alphabet, a one-to-one transformation would require a table with 456,976 entries to guarantee no collisions.

attached to the ordering, and all three are treated equally, giving none a priority. By appropriately designating the three components as being constant or variable, we can ask eight "questions" of the data structure. Again using Feldman's notation, with a slight re-ordering, they are

F0	A (O) = V
F1	A (O) = x
F2	A (x) = V
F3	A (x) = y
F4	x (O) = V
F5	x (O) = y
F6	x (y) = V
F7	x (y) = z

where [A,O,V] represent constants, and [x,y,z] are variables. Question F7 is not a question at all but a request for a dump of the associative memory. Question F0 simply asks: "Does A (O) = V?" and the answer in, for example, TRAMP, is a kind of truth value. In the case where A, O, and V are all singletons, the truth value is a straightforward 1 or 0 denoting whether or not the specified association can be verified by the data. The interpretation is slightly ambiguous, however, when one or more of the three sets has cardinality greater than one. To illustrate, assuming that the association

ENDS (L1) = p1;p2

has been stored, these five questions have the following truth values:

(1)	ENDS (L1) = P3	0
(2)	ENDS (L1) = P1	1
(3)	ENDS (L1) = P1;P3	?

Questions 1 and 2 are clearly false and true, respectively, but question 3 is partially false. An interpretation, which seems natural, and the one adopted by TRAMP (Section 4), gives the truth values as shown, namely:

> if ALL associations implied by the question are resident in memory, or derivable therefrom, the value is "1"
>
> if none, the value is "0"
>
> if some, but not all, the value returned is "?"

Questions F1-F6 simply ask the system to "fill in the blank(s)," i.e., to replace the variable with the set that is the answer to the question. For example, Question F1 asks for the set of all Vs that A (O) equals. Question F3 asks for the sets of all Os and Vs that have a first component "A." Because of the recursive nature of many systems, questions F1-F6 may be nested in any way, to any desired depth. One may ask: "How many fingers on a hand?"; "What figures are pointed to by the arrows in Window Q?"; "How old are the fathers of the wives of Mary's brothers?"; or any questions composed in any way compatible with the stored data, nested to any level.

It is interesting to explore how well the objections to ring structures are overcome by an associative approach. The first objection, that ring data structures are difficult to access, was based on the procedurality of the access languages and the detailed storage structure knowledge necessary to use them. On both of these counts, the authors contend that the associative structure largely solves the problems. By specifying the properties that the accessed data must have, the programmer is in a sense only stating requirements, with no notion of how to go about fulfilling those requirements. These requirements are the minimal amount of information which any data retrieval system must know. In this way, the programmer is freed from tedious considerations of how to get the data. The programmer proceeds with fewer errors of the data retrieval kind. The code that is written is more oriented toward the intended application. Moreover, at no point is storage structure a consideration.

The second objection was that the meaning of ring structure elements and their sub-elements was bound at compile time at the latest, and tended to remain as it was originally designed throughout the course of the program's life. To show how an associative system answers this objection, some background is necessary.

The separation of data structure and storage structure has been recognized as a valid concept since about 1964, when D'Imperio proposed it (20). Since that time, people in the area of large, shared, data-base systems have accepted the concept, and systems have appeared which separate the two concepts (8). Unlike the data-base community, the graphics community has not clearly made the separation of these two concepts. When explaining the data structure in his program, a member of the computer graphics community will spend most of his time detailing the various pointer and datum fields which make up his storage structure. One must usually infer the data structure from such a presentation.

The entanglement of these two areas has a number of undesirable consequences. First, communication between workers is difficult. Data structure concepts get lost in a morass of implementational details. Second, even a person who is thoroughly familiar with an implementation may not be able to see patterns inherent in the data structure. The ability to recognize these patterns might suggest a rather different storage structure—one that was more efficient and/or more easily extensible and/or less prone to errors, etc. The final, and perhaps in the long term, most undesirable consequence is the fact that as the system grows to encompass a family of users sharing common parts of the data, it becomes impossible to serve their needs without requiring absolute adherence to a standard storage structure and ring accessing language. Such adherence may make their job awkward or impossible, since it is likely that the structure of their problems is not understood by the storage structure designer, whose decisions directly influence the access language.

Returning to objection two—early binding of ring structures—it is apparent that to achieve a capability for delayed binding of any structure, data structure considerations must be separated from storage structure considerations. Such is the case because a programmer, even in an interpretive system, will be writing in a source language which is not dynamic in the time-frame of seconds. Thus one cannot hope that a user will dynamically change his view of the data he is processing. One can hope, however, that a system could dynamically change the storage structure used to implement that data structure. But to be able to change the storage structure, it must be independent of the source language data structure.

Two questions thus remain. First, is it desirable to be able to dynamically change storage structures? Second, how are associative structures a step toward this goal and thus better than rings with respect to binding time?

In answer to the first question, two examples of desirable properties of a graphics system are offered. It has often been stated that if one were designing, say, an amplifier using a graphic system, it would be desirable to be able to examine the electrical properties of the design, then the thermal properties. One would hope that it would not be necessary to carry the overhead of thermal information when doing electrical problems, and vice-versa. Clearly both kinds of properties are represented in the storage structure of the total problem (which is some place in secondary storage.) Thus we are talking about changing a storage structure when the loading process is underway.* A similar need arises when a number of users are sharing a large data base—one, say, with a FORTRAN program (for engineering analysis) and another with a COBOL program (for cost and inventory analysis). Since the storage structures expected by these two languages are different (consider FORTRAN's column major order for arrays), the ability to change storage structures is once again called for. Naturally, aspects of both examples could be combined. The point is that both of these examples have been postulated as being in the future of graphics systems. Their existence depends on such transformations.

Finally, to what extent do associative memory systems deal with these problems? Certainly they are not solved. The transformation of storage structure to storage structure remains very much a research problem. There is also the problem of how to parse data structure statements so that they can

*Dynamic loaders open up several possibilities, which will not be discussed here.

be interpreted in terms of an existing (target) storage structure. But associative methods have separated many aspects of data structure and accessing from their implementation. Thus one can begin to think about how to change storage structures and process intermediate stages without the complications arising from a lack of separation. The correspondence of ring data structures and ring storage structures makes these changes more difficult, if not impossible.

The third objection raised against ring structures is their typically high ratio of pointer to non-pointer data. Obviously this need not necessarily be the case. Since the number of pointers in a block reflects the number of relationships (rings) in which the block directly participates, one could postulate various ring structures where only a few relationships were of interest, hence the ratio of pointer to non-pointer data could be kept low. In current computer graphic systems, however, it is a fact that many relationships are defined for a given element. Thus space for many relationships is allocated in a typical block (whether or not it is used). In many cases the ratio of pointer to non-pointer data is at least 2 to 1. It should also be noted, as has been implied above, that with a ring structure the set of allowable relationships in which a node participates tends to be fixed at compile time. There are ways of avoiding this difficulty,* of course, but the methods for doing it are typically counter to the ability to access

*Methods such as getting another block and associating it via pointers to the original block. This may be done arbitrarily many times; thus the objection can be overcome via this mechanism.

with a high-level access language, since such languages depend on allocated fields for various relationships.

Another consequence of the multitude of pointers in graphics ring structures is the extensive "bookkeeping" necessary when certain relationships are no longer valid. The deletion of a point, for instance, can cause extensive processing if the rule holds that the deletion of a point deletes all lines which end on that point (and perhaps even all other entities which depend on the deleted lines, etc.).

Turning now to the advantages of associative structures, we will see that they can offer various tradeoffs that are unavailable with ring structures. Methods which can differ with associative data structures are not without their own disadvantages, but the designer is at least given some ability to pick his tradeoffs according to his problems. It should be emphasized that all of the features to be described are not implemented in all associative systems. The discussion below is more concerned with possibilities for associative structures.

What then are some of the ways that associative structures can save pointers and yet retain the same capabilities for retrieval as ring structures?

The first advantage of associative structures is the comparative ease with which one can exploit the traditional space-time tradeoff. Suppose that the "connected to" relation were of interest. With a ring structure, one would typically have a ring such that given an element, one would "run around the ring" to find the set of all elements which were connected to the given element. With associative structures, there are a number of possibilities.

Given an element X, one could store

```
CONNECTEDTOL(X)  =  Y
```

for each element Y that is so connected. This represents the equivalent of the ring approach. But one could route in the TRAMP system (1,2) something like*

```
CONNECTEDTOL(X,Y)  =  TYPE(X).EQ.'LINE'.A. TYPE(Y).EQ.'LINE'
                      .A.(ENDPOINTS(X)∩ENDPOINTS(Y)
                      .N."NULL".A.X.NE.Y
                      CONNECTEDTOL IS SYMMETRIC
```

which would find the set of all lines connected to a given line X. Similar definitions could be written for other kinds of entities. The points are several:

(1) At no point is storage used to represent the connectedness of the various graphical entities. All that is stored is the definition of how to find the required set.

(2) The price paid for this space saving is extra time in retrieval. Several probes of the associative memory must be made, and the resulting sets intersected, etc. However, it should be noted that the definition can be compiled when defined (6) so that the retrieval time is not excessive.

(3) Changes in the set of lines connected to X do not cause extensive bookkeeping. The definition CONNECTEDTOL still holds. Subsequent

* The examples are presented in a simplified syntax in the interest of clarity. The true syntax will become evident in Section 4.

invocations of the definition will cause the added (deleted) lines to be included (excluded) because their inclusion in the endpoint relation will have been adjusted.

Thus it is clear that one can trade the space for storing relations and the time necessary for keeping them valid against time necessary to perform several probes of the associative memory and to carry out the resulting set operations. Furthermore, various degrees of this tradeoff are achievable by storing more results explicitly, thus enabling more concise definitions. For example, if the following relation were stored explicitly:

```
for each X, Y, ∋

     ENDPOINTS(X) ∩ ENDPOINTS(Y).NE.'NULL'

store

     SHAREPOINT(X)  =  Y
```

then the connected to definition becomes

```
     CONNECTEDTOL(X,Y)  =  TYPE(X) .EQ. 'LINE' .A.
                           TYPE(Y) .EQ. 'LINE'
                           .A. (SHAREPOINT(X,Y))
                           CONNECTEDTOL IS SYMMETRIC
```

One could even speculate about the possibility of a system which adaptively adjusted to a point on this space-time tradeoff by choosing one of a set of equivalent definitions based on past performance. While this may seem far-fetched, its mere possibility accents the flexibility of this approach.

On a more practical level, it should be noted that the storage structures used to implement associative systems can be designed to deal directly with the paging problems of virtual memory systems (19). Ring-oriented systems have a great deal more trouble with this problem.

Finally, it should be noted that although associative systems typically store several representations of relations (1, 17), the cost of doing this in a paged system is not excessive, since on a given query only one of the representations will be used (paged in). Moreover, although associative systems must pre-allocate storage for relations, just as rings must, the storage will be used for the particular relations which arise, not for potential relations. Thus effective use of pre-allocated storage can be made, because the relations can grow more dynamically.

4. TRAMP

TRAMP (Timeshared Relational Associative Memory Program) is two packages of functions: the first—the data structure—may be used to enter, retrieve, and generally manipulate an associative data structure; the second—the relational memory—places an artificial structure on the "associative triples," viz., the relational structure. The relational package allows logical inference to be performed on the data within the associative structure. Specifically, rules may be entered; these will be followed by TRAMP, effectively expanding a "minimal" set of data to a workably large set; the number of associations that must be explicitly stored is thereby drastically reduced. For example, by defining the relation "HUSBAND OF" to be the converse of "WIFE OF," the user need only store marital relations in one direction, while effectively having them available in both directions. More detailed examples and the rules for using the relational package appear later.

These machine-coded functional packages are presently embedded in the UMIST interpreter on the IBM/360 model 67. Although this existing union has proved most fruitful, the data structure is totally independent of the interpreter and actually relies on it only for I/O. The relational package is also independent, except that it relies on the type of recursion that the interpreter provides. The relational package is totally dependent on the associative data structure.

4.1 THE ASSOCIATIVE PACKAGE

The associative package is closely patterned after the ideas presented in Section 3. We will thus present in this section a short description of how these ideas are implemented in TRAMP. Details may be found elsewhere (1,2). For those totally unfamiliar with the TRAC language, for this paper it is necessary only to know the syntax of a function call. The sharp sign (#) signals the start of a function call, with the call itself enclosed in an immediately following pair of parentheses. The arguments are separated by commas, and the first argument is the name of the function. #(sub,ARG) is therefore analogous to the FORTRAN: CALL SUB(ARG). One of the minor additions of UMIST is to allow implicit calls of functions, i.e., when the normal call might be #(cl, FUNC) in TRAC, the UMIST call may be either the same, or else #(FUNC).

The name of the storage function is dr and the syntax of the call is #(dr,A,O,V,). None of the three arguments to dr may be an empty set. Each point in the cartesian product of the three sets is stored using hash-coding techniques, i.e., each element of each set is grouped with each pair of elements of the other two sets, and the resulting triple is stored. Thus a single call on dr stores as many associations as the product of the cardinalities of the three sets.

The primary retrieval function has the name rl. The syntax of the function call is identical to that of dr except for variable specification. A variable in TRAMP is denoted by enclosing a name, possibly null, within asterisks (*). Thus, #(rl,A,O,V) has no variables and asks whether A (O) = V; #(rl,A,O,*X*) asks: what does A (O) equal? Place the answer in X. When the variable is not given, the answer is returned into the calling string.

rl generates the union of the answer sets. That is, the question: #(rl,ENDS,L1;L2,**) has two answer sets: the ENDS of L1 and the ENDS of L2. rl simply forms the union of however many sets there might be, however int is a function which generates the intersection of the several answer sets. Thus, #(int,ENDS,L1;L2,**) generates the set of all end-points common to L1 and L2. #(rl,ENDS,L1;L2,**), on the other hand, would generate the set of all points at the ends of either L1 or L2.

Throughout this article, the UMIST delimiter of arguments is the comma; the element delimiter for TRAMP cannot be the same, we therefore use the semi-colon.

4.2 LOGICAL INFERENCE PACKAGE

The associative memory accomplishes a kind of content addressability by using two quick hashes to address data, and the access time is essentially independent of the size of storage.* But as discussed in Section 3, many associations will be implied by a single associative sentence. This poses two real problems:

1. The user must make sure that all associations that apply are actually inserted into the structure. This is extremely tedious and prone to error and omissions.

2. Explicit storage results in gross inefficiency.

*As the size of storage increases, there are more collisions, but they are quickly resolved, and do not cause a significant delay. Even in extreme pathological cases, they involve only relatively minor list searches.

To alleviate this, TRAMP provides the facility to define, in a characteristic way, what other associations may be derived from a given association. This permits all of the information that might be contained in a single association or sequence of associations to be utilized instead of having to enter the same information redundantly in each of the several ways that it might be referenced. The name of the function which makes the definition is ddr. The syntax of the function call is: #(ddr,(R = EXP)), where R is the relation ("A" component) to be defined, and "EXP" is a logical expression which is the definition.

Before presenting examples of the use of ddr, two relational operators must be defined:

The first is converse, denoted in TRAMP by ".CON." Converse simply inverts the order of the two relational arguments*:

$$R(x,y) \leftrightarrow .\mathrm{CON.}\ R(y,x)$$

Thus "CHILD OF" is the converse of "PARENT OF"; "WIFE OF" is the converse of "HUSBAND OF"; "SPOUSE" is its own converse; any symmetric relation is its own converse.

Relative Product: The relative product or composition to two relations is commonly denoted by R_1/R_2, and this is the notation used by TRAMP.

*The relational notation used by TRAMP is derived from the format "R,x,y" by enclosing the relational arguments in parentheses. This is a slight distortion of the associative notation: A(O) = V, but the order is preserved: R(x,y) means that R(x) = y.

$$\forall x \forall y \exists^{z}\ [(R_1/R_2)(x,y) \longleftrightarrow R_1(x,z) \wedge R_2(z,y)]$$

Less rigorously, but more specifically,

#(ddr,(R_3 = R_1/R_2))

would tell TRAMP that $R_3(x,y)$ if a "z" can be found such that $R_1(x,z)$ and $R_2(z,y)$.

Besides these two relational operators, three logical operators are available: .A. (conjunction); .V. (disjunction); .N. (negation). Finally, there are six equality operators: .EQ.; .NE.; .GE.; .LE.; .GT.; .LT., with obvious meanings.

Examples of TRAMP relational definitions are:

```
#(ddr, (BIGGER  =  BIGGER / BIGGER)) Bigger is transitive

#(ddr, (BIGGER(A,B)  =  BIGGER(A,Q) .A. BIGGER(Q,B)))
                        exact same definition using
                        expanded format—specifying
                        dummy arguments.

#(ddr, (SIB  =  BRO .V. SIS .V. .CON.SIB))
                        a sibling is a brother or a sister
                        and it is symmetric.

#(ddr,(HUSBAND  = .CON.WIFE)) Husband is the converse of Wife.

#(ddr,(BIGGER  =  LARGER)  Bigger and Larger are synonomous.

#(ddr,(BRO(CAIN,ABEL)  =  SIB(CAIN,ABEL) .A. SEX(ABLE,"MALE")))
                         a brother is a male sibling.  Note that
                         constants are denoted by enclosing them
                         within double quotes.

#(ddr,(MALE(X)  =  SEX(X,"MALE")))  defined the unary relation MALE

#(ddr,(BRO(X,Y)  =  FATHER(X,Z) .A. FATHER(Y,Z) .A. MALE (Y)

                    .A. X.NE.Y))
                    a brother is a male offspring of the same
                    father, other than oneself.
```

```
#(ddr,(STEPMOTHER  =  FATHER / SPOUSE .A. .N.MOTHER))
                      a stepmother is the spouse of the father
                      who is not the mother.

#(ddr,(NEPHEW  =  SIBLING / SON))   a nephew is the composition
                  of a sibling and son.

#(ddr,(UNCLE  =  .CON.(SIBLING/SON))) in a male world, uncle is
                 the converse of nephew and may be defined as
                 the converse of the definition of nephew.

#(ddr,(UNCLE  =  °CON.NEPHEW))   or simply as the converse of nephew.
```

4.3 IMPLEMENTATION OF INFERENCE

The purpose of the inference mechanism is to allow the user to define under what conditions an implied association may be derived from data explicitly in memory. This is accomplished by generating where necessary (where defined) a more complex retrieval call from a simple one. Specifically, if the following definition had been entered:

```
#(rl,STEPMOTHER,JOHN,**)
```

which asks for the stepmother of John, would be expanded by the system to be the following:

```
#(rcom,#(rl,SPOUSE,#(rl,FATHER,JOHN,**),**),#(rl,MOTHER,JOHN,**))
#(rl,STEPMOTHER,JOHN,**)
```

The exact call generated would be slightly different, but that is a technicality, irrelevant at this point. The final retrieval call in the sequence generated asks if the desired association was entered explicitly. It is always assumed that a relation that has been given a definition may also appear explicitly. The rest of the expanded call will find the answer if it is present implicitly. This expanded call is then returned to the UMIST processor, which in turn makes the actual calls to the data structure. The importance of this is that relations need be expanded only one level at a time, with the UMIST recursion automatically taking care of the possibility that any relation is defined in terms of more complex relations, etc. (This is the major difference between the call as it actually would be generated, and as it appears above—the above, taken literally, would specify an infinite recursion!) Thus the inference compiler generates TRAMP procedures--they operate only within the TRAMP language—not at a lower, machine level. The definition, entered by #(ddr), specifies what information the procedure is to derive and what rules may be used to derive it; the compiler accordingly constructs such a procedure; and the interpreter (TRAMP inference interpreter--rather than UMIST) expands the procedure at retrieval time, filling in information specific to the call.

At retrieval time, a retrieval "preprocessor" looks to see if the "relation" ("A" component) has been given a definition. If not, the preprocessor exits and retrieval proceeds as described earlier. If the name is found to have been defined, then the "interpreter" is called in to interpret the program generated by the compiler at the time it was defined. This program tells the interpreter what TRAMP function calls are to be made, and what the function arguments are to be.

It should be noted that the compiler actually puts out two programs: one which, given x of R(x,y), builds a chain to generate y; the other builds the appropriate chain in the opposite direction, from y to x. Thus question F1: #(rl,A,O,**) generates a different sequence of function calls than F2: #(rl,A,**,V). It may not be immediately obvious why this is necessary, but, in general, the two programs will be quite different. This is always the case for composition. Still, the compiler would only have to output one program, and the interpreter could decide how to interpret it. Since the compiler will usually be called only once or twice for each relation, or certainly fewer times than the interpreter, it is most efficient to let the compiler do as much of the work as possible.

The compiler is prepared to handle definitions which are circular in the sense that a relation is defined in terms of itself. That is, symmetric and transitive relations are perfectly acceptable. However, the sequence:

```
#(ddr,(PPP  =  QQQ .V. ...))  #(ddr,(QQQ = PPP .V. ...))
```

is valid because of its circularity. Were the compiler to attempt to generate code for that sequence, the code would specify an infinite recursion. This situation is checked for and flagged if detected.

4.3.1 Algorithm for Parsing TRAMP Relational Language

The parser assumes the sentence is in disjunctive normal form; if not, a first pass must put it into that form. If the sentence is abbreviated, it must be expanded by the first pass. This is accomplished by making a complete scan of the sentence, counting and assigning to each term a nesting depth, as determined by slashes and intervening operators. A second scan is then made using that information to insert dummy relational arguments.

Phase II is the phase that does the actual parsing. It takes as input a sentence fully expanded in disjunctive normal form. Each conjunct (the dual of a clause in standard notation) is processed independently. The output for each conjunct is code representing a directed graph, or network, where each individual variable is a vertex and each relation is a directed line. For purposes of discussion only we will henceforth assume that the sentence being parsed is of the form

$$R(x, y) = \ldots$$

i.e., R is the relation being defined and x and y are its dummy relational arguments. The network to be constructed will always have x as its source and y as the sink. (We are ignoring a second pass that is made that reverses x and y; the two passes are of course identical and we will concern ourselves only with the first.)

The actual parsing is a matter of properly directing the edges in the network (it is obviously trivial to construct the undirected graph). The rules used are the following, applied by picking each source (x or any spurious source) and the single sink, y, and tracing out paths:

1. If degree = 1 (spurious source), line is directed out (away from vertex).
2. If degree = 2, one line goes in and one goes out.
3. If degree ≥ 3, terminate trace of this path.
4. If x, all lines go out.*

---

*This is not quite accurate. For R(x,y) = A(x,y) .A. B(z,x) a line would go into x, but this causes no problems and is interpreted in Figure B.

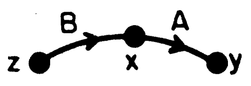


Figure A

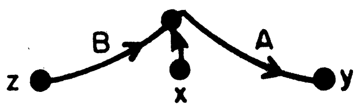


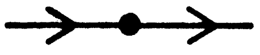
Figure B

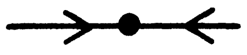
5. If y, all lines go in.

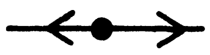
6. Any path that forms a <u>bridge</u> is mirrored by an identical path in the opposite direction, i.e., forms a cycle. A <u>bridge</u> is defined to be a path that connects two disjoint paths from V_1 to V_2, where neither V_1 nor V_2 lies on the bridge.

7. Any lines remaining undirected may be directed arbitrarily, constrained by two conditions: No cycles may be introduced (only valid cycles are those arising from rule 6); each vertex must lie on a directed path to the sink.

Phase III takes the network constructed by Phase II as its input and outputs a TRAMP procedure. The interpretation of the network is extremely simple. The only vertices that can have degree one are sources (or the sink), i.e., the data representing that vertex is the input to the object program. All other vertices will have degree ≥ 2 and the edges, taken pairwise, can only have one of the three following configurations shown with the corresponding interpretation:


COMPOSITION


CONJUNCTION


DISJUNCTION

Example:

A definition in disjunctive normal form is (1 conjunct)

$$R(x,y) = A(x,a).A.\ D(a,y).A.\ C(a,b).A.\ E(b,y).A.\ B(x,b)$$

i.e., x stands in relation R to y if

(1) $y \in$ Range $(D(A(x,-),-)$ and

(2) $y \in$ Range$(E(B(x,-),-)$ and

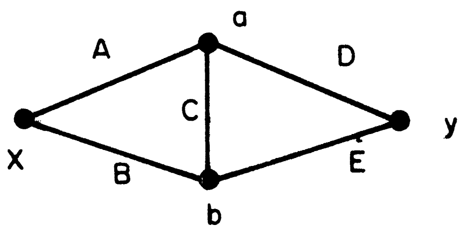
if a {q: A(x,q) holds} and if

b = {q': B(x,q') holds} then

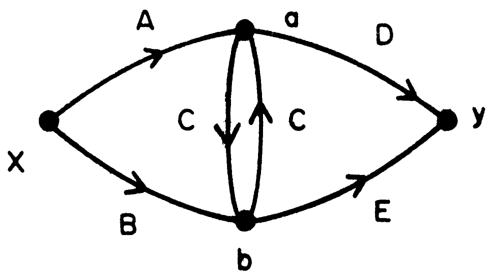
V_q ε Range (A), q' ε Range (B)

C(q,q') and C(q',q) both hold.

The input to the parser is thus



The output from the parser is



Notice that the parser had to discover the necessity for an extra line because of the structure of the network (the context of the statement). It turns out that the parser of the language of relations is highly context-sensitive. Thus the parsing of this language represents an advance in parsing techniques. For further details, see (6).

5. THE TRAMP/RAMP GRAPHICS SYSTEM

This section describes an experimental graphic system built using TRAMP as the primary vehicle for storage and retrieval of data. The display of information is under control of the RAMP system, which controls the DEC 338 display. This section gives an overall flavor of the process as well as a description of some of the ways TRAMP is used. Section 5.1 describes the various subroutines which reside at the 360. Section 5.2 describes the various display macros accepted by the RAMP system. Section 5.3 presents a few comments about this experimental system.

The 360 is used for storage, retrieval, and manipulation of the description of the graphical structure as well as communication of display commands to the smaller display computer. Essentially, the information is stored in a hierarchical structure built up in TRAMP, whereas the communication between the large and small computers is under the command of UMIST, which also resides within the large computer. This intercomputer communication is kept as small as possible because of the low channel capacity of the data lines.

At each of its transmissions, the small computer sends either the coordinates of a tracking-cross position or the reference name of an entry in the small computer's display file. The large computer interprets this message in terms of what it was expecting to receive and sends back the necessary position, line, character string, or command (as described in a previous section).

In addition to sending commands to the small computer, the large computer stores picture information so that it may either alter pictures (upon command), or perform computations on picture structure, or redisplay pictures at a later time.

The quiescent state of the large computer is "procedure check," which corresponds to pointing mode in the small computer. In this state, the critical items on the display are the list of operations which can be performed, the menu. Because each item in the menu is a character string at the display tube, it has a unique name, a four-digit number corresponding to its location in the data file, If the light-pen is pointed at an item, for example, the word POINT, the display file location corresponding to this entity is transmitted over the data line to the large computer, which can determine the procedure corresponding to this name and execute this procedure. This is possible because at the time that the name item was created, its display file location was associated with the corresponding procedure, i.e., the association was defined by executing: #(DR,PROC,#(XIN),#(TEST), where #(XIN) is an implicit call on an input string buffer. The call is replaced by the last input string, and #(TEST) is replaced by the value of TEST, in this case the string POINT, which was previously written as the procedure which defines a point. The input string, stored in XIN, is the display file location corresponding to the character string POINT. Let us assume this location is 0027; then we have #(DR, PROC,0027,POINT).

Now, whenever the light-pen is in pointing mode, and POINT is picked, the name 0027 will be transmitted over the data line. The larger computer is in the procedure check state, which means it is waiting to read an input string, viz: #(#RL, PROC, #(RS),**)). If POINT is picked, this becomes first #(#(RL,PROC,#(RS,**)), then #(POINT), which then initiates a string of commands to define points.

In addition to PROC(EDURES), there are several other attribute names of paramount importance in the large computer; among these are CLAS(S), COOR(DINATES), characters (CHRS), and HIST(ORY).

For example, to store the description of a <u>point</u>, P1, which is in Picture VI, the following set of TRAMP statements is executed:

```
#(DR,CLAS,P1,V1,)
#(DR,CLAS,POINT,P1)
#(DR,COOR,P1,0370:0426)
#(DR,HIST,0041,P1:VI).
```

This involves defining P1 in the class of V1; POINT in the class of P1; the coordinates of P1 as 370/426; and P1's display file entry (41) is then given the history of the dendrite* of P1.

The heart of the display file generator in the large computer is a recursive routine which climbs down this structure from the top going down the left-most dendrite until it finds a terminal element, or the end of the string. It communicates any displayable element to the small computer, deletes it from the string (but not from the memory), and climbs back up the structure until it finds the next non-terminal element. It then repeats the previous steps. When the whole structure has been examined, and all the terminal elements communicated to the display, the larger computer sends a pointing mode command and enters the quiescent state.

*A dendrite is a unique path down from the topmost element of the hierarchy.

As the routine goes down the structure it saves the path (dendrite) in order to know the HISTORY of each entity transmitted to the display. This history is used by many routines, for example, to change or delete items in the associative memory as the graphical entities that they represent are moved or deleted from the display.

5.1 THE TRAMP GRAPHICS PROGRAM AT THE 360

This portion of the TRAMP/RAMP system is a UMIST program. Its major parts are described in detail here. This detailed documentation is included for completeness; the casual reader may wish to skim much of this section.

The program has two modes—picture mode and construct mode.

PICTURE MODE

In picture mode, the name of the picture and a menu of operations appear on the screen, along with any elements already in the picture. Some of the items which may appear in the menu are ADD TO MENU, NEW PICTURE, POINTS, LINES, BLANK LINES, CHARACTERS, INSTANCE, BLANK, UNBLANK, DELETE, CONSTRUCT MODE, and ADD FUNCTION.

The user picks the item he wants from the menu by pointing at it with the light pen and working the light-pen shutter.

Whenever the user picks an item from the menu, the display computer sends a 4-digit number to the graphics program in the larger computer. This number is the position of the menu item in the display file of the smaller display computer. This number uniquely identifies the menu item. When the menu was originally displayed, this number was associated by the larger computer with a procedure which corresponds to the menu item. Therefore the number is used to refer both to the menu item and to the corresponding procedure.

Procedure ADD TO MENU

When this item is picked up by the light pen a 4-digit number is sent to the larger computer. The larger computer uses this number to call the corresponding procedure, ATM.

ATM asks the user to type in on the teletype both the character string which will be the menu name of this procedure, and the name of the procedure internal to the larger computer program, by printing:

```
ENTRY IN ... MENU?;PROGRAM?
```

For example, if the user wanted to put in a new function called DUMP he could, for example, give it the name

```
HAVE A LOOK;DUMP
```

and the character string HAVE A LOOK would appear at the end of the menu and its display file position, 4-digit number, would be associated with the function, DUMP.

This process will be repeated until the user enters a null—types only a prime—on the teletype.

Actually, it would be possible just to start with ADD TO MENU and build up a menu each time, instead of having other predefined items in the menu.

Procedure NEW PICTURE

Picking NEW PICTURE calls procedure VEW (view), which sends a command to the display computer to start again [$A^cA^cA^c$SN] and prints out on the teletype:

NAME?

If the user does not name the picture, he gives it a null name by typing a prime,'. The program generates a unique name by incrementing a picture number counter until it finds an unused name in the series V1, V2, V3, ... , V#, ... If the user gives the picture a name, the program will display this name and the menu (including any additional menu items associated with this name) and call a procedure DFG (display file generator, see below) which will display anything already in this picture.

Procedure POINTS

Picking POINTS in draw mode calls procedure PINT (point) which sends a message to the display computer [A^{c}A^{c}A^{c}LL], which tells it to display a tracking cross for the light pen to position. When the light pen releases the tracking cross,the coordinates of the release point along with a list of the display file names, 4-digit numbers,and other things (lines, points, etc.) which the light pen "saw" at this position on the screen are sent to the program. If the list of other things seen is empty the program will return the coordinates to the display [A^{c}A^{c}A^{c}PT XXXX YYYY]. The display will use this information to display a point at these coordinates, XXXX YYYY, and return a name, a 4-digit number, to the program. If, on the other hand, the list is not empty the program calls GRAVITY (see below). If gravity is *off* the list is ignored and the coordinates are sent. If gravity is *on* the list will be examined to see if the coordinates should be changed slightly before being sent in order to correspond with a neighboring point, line, or intersection point of two lines (arcs and other curves have not been implemented although there is no reason they could not be added).

While the display is adding the point to its display file, the program is storing references to this new point, giving it a unique point number (by incrementing a point counter) and associating the coordinates and the current picture name with this point number. When the 4-digit "name" is sent back from the display, the program associates this with the point name (P#) and picture name to keep a current history of this point for reference.

This procedure will repeat until the user pushes the END OF FILE push button on the display, sending a 4-digit character string, 7777, which the computer interprets as a signal to end the procedure.

Procedure LINES

Picking LINES in draw mode calls procedure LIN (line) which sends a message [$A^cA^cA^c$LL] to the display, as with point, telling it to display a tracking cross. When the light pen releases the tracking cross the display sends tracking cross current coordinates and the list of other entities seen (within the light pen aperture) to the program. If this list is non-empty the program calls gravity (as in procedure POINT above) and sends the display command [$A^cA^cA^c$LP], light pen tracking cross with rubber-banding.

Now when the user moves the tracking cross with his light pen a line will stretch from the center of the tracking cross back to the last position at which the tracking cross was released. When the tracking cross is again released the display will again send coordinates and a list of things seen by the light pen in the immediate neighborhood of these coordinates. If the list is non-empty Gravity will again be called. Then the program will send begin coordinates and end coordinates for the line, [$A^cA^cA^c$LE XXX1 YYY1 XXX2 YYY2] to the display. The program then associates a unique name for this line (gotten by use of a line counter) with the current picture and with names for its beginning and ending coordinates (coordinate names are prefixing it with "B" or "E" for "begin" and "end." The actual X-Y coordinates are associated with their respective names.

The program then receives a 4-digit display file name for the line and associates this with the line, L#, for a history of this line. Next the program sends another [$A^cA^cA^c$LP] to the display, saving the end point of the last line for the beginning of a new one and repeats. The program continues this process until the END OF FILE button is pressed.

Procedure BLANK LINES

This procedure is just like LINES above except that $A^cA^cA^c$NL XXX1 YYY1 XXX2 YYY2 is sent to the display instead of $A^cA^cA^c$LE XXX1 YYY1 XXX2 YYY2 and the line although present in the display file is not illuminated on the screen. This procedure may be used to tie a figure together, move the beam to a starting place, etc. Of course, the information that this is a blank line also is stored in the program.

Procedure CHARACTERS

Picking this menu item calls procedure CHR which sends a light pen tracking command [$A^cA^cA^c$LT] to the display. The display returns coordinates where the tracking cross is released and the message CHARACTERS? prints out on the teletype ending the line with a carriage return. The coordinates of the tracking cross and the string are then sent to the display following a character command, $A^cA^cA^c$CH XXXX YYYY CCCC...C. The display then writes these character strings starting with the coordinates sent. The program associates a new character number with this string and with its coordinates and picture name. Next the program reads the 4-digit name returned by the display and associates that with the history of this string, C#: PICTURENAME. The tracking cross command is then sent again to the display and the process repeats until the user hits the END OF FILE button.

Procedures BLANK AND DELETE

Picking BLANK calls a delete-blank routine with an argument of "B" for blank (DBL,B). This routine sends a message [$A^cA^cA^c$LL] to the display which instructs it to send back a list of things which the light pen "sees" when its shutter is operated. The program takes this list of items and checks to see if the list contains the name of a menu item. If the list does contain the name of a menu item, a message is printed out on the teletype asking for confirmation, otherwise the list of entities to be blanked is immediately sent back to the display with a blank command [$A^cA^cA^c$BL List]. In addition the list of entities is associated with a sequentially derived blank number so that the program has a record of what has been blanked. This process repeats until the user pushes the END OF FILE button on the display.

Picking DELETE causes nearly the same sequence of events to take place as in the BLANK command except that the routine is called with the letter "D" for delete (DBL,D). The program now sends a DELETE command [$A^c A^c A^c$DL list] to the display and removes the associations (references) to these entities from program storage if they are user-defined entities.

Procedure UNBLANK

Picking procedure UNBLANK will cause the last set of blanked entities to be restored by sending an unblank command [$A^c A^c A^c$UB List]. The blank number is decremented by one so that the next time UNBLANK is called the next previous set of entities blanked will be restored.

Procedure INSTANCE

Picking INSTANCE calls a routine ADP which causes NAME? to print out on the teletype. The user then types in the name of the picture. This name is checked with the list of picture names and added if it is not one of the existing names. Next the picture name is associated with the current picture name as a picture-in-a-picture and the display file generator, DFG below, routine is called to add the instance to the current picture.

At the present time there are no facilities to transform an instance in any way when adding it to another picture, nor is the added picture checked to make sure it does not in some way contain the current picture in some way which would lead to an infinite regression—the barbershop mirror problem. Transformations—scaling, rotation, etc.—and checking facilities could be easily added.

Procedures GRAVITY ON and GRAVITY OFF

Gravity is normally on in draw mode (and in construct mode) so these two items can be left out of the menu unless the user wants to change the normal convention. The user picks ADD TO MENU and inputs GRAVITY ON; (((SET, GV,GVX))) and GRAVITY OFF; (((SET,GV,GVXX))) where SET, GV, GVX, and GVXX are identifiers used in the program,and the parentheses are a UMIST requirement.

When gravity is on it will be assumed that points, ends of lines, etc. defined in the immediate neighborhood (within the light pen aperture) of previously defined entities are meant to lie on those previously defined entities. Therefore,the coordinates of the new entities will be changed to lie on such points, lines, or intersection points.

When gravity is off, display coordinates are used as is.

Procedure CONSTRUCT MODE

Picking CONSTRUCT MODE causes the display file to be cleared with a start again message [$A^cA^cA^c$SN] from the program and a construct menu to appear, replacing the draw menu.

Construct mode is described below.

Procedure ADD CONSTRUCTION

Constructions are written in files by the program while in construct mode. These constructions may be used in draw mode but must be read from the files and added to the menu. ADD CONSTRUCTION is a special procedure which, given the name of the construction and the name of the file in which it has been defined, will read (load) the construction routine and add it to the menu.

Picking ADD CONSTRUCTION will cause the message "NAME FILE WHERE CONSTRUCTION IS STORED" to print out on the teletype. The user replies with the file name. Then the message "CONSTRUCTION NAME IS?" prints out. Again the user replies on the teletype with the name requested.

If the program finds the named construction, the name will appear in the menu.

CONSTRUCT MODE

The user can define graphical constructs using a small set of "primitive" constructs and user-defined constructs. This is analogous to a system of mathematical axioms and theorems derived from those axioms being used to develop further theorems.

The system to be described is just a beginning, the primitives here certainly would not be sufficient to develop many interesting theorems (constructions), however the present system is enough to indicate the feasibility and potential of this approach.

In construct mode, a menu again appears on the screen. The construction menu at present contains the following: NAME OUTPUT FILE, POINT, LINE, INTERSECT LINES, OUTPUT LINE, END CONSTRUCTION, and PICTURE MODE.

Procedure NAME OUTPUT FILE

Picking this procedure calls two routines C%NS and CONSN%ME. The routine C%NS types a message, "CONSTRUCTION FILE NAME?," on the teletype. The user answers on the teletype with the name of a line file* (sequential files cannot

*MTS supports two types of files—sequential files and line files. Line files can be accessed by line number; sequential files must be accessed sequentially.

be used because the program utilizes line numbering). If the user inputs a null, ', the program gives the output file a default name of "-TEMP" which will create a temporary file of this name.

Next the routine CONSN%ME is called which prints out the teletype message, "NAME NEW CONSTRUCTION!". The user must put in a name here, otherwise the name of the construction will not be printed in the output file. If the user inputs a null, ', here he must pick NAME OUTPUT FILE again or escape from construct mode (currently by picking PICTURE MODE).

Naming a construction does several things: it causes the name of the construction to be temporarily stored in the associative memory, and it causes a call to the construction to be stored in the output file.

Procedure POINT in construct mode

Picking POINT calls a procedure P%NT which works like procedure PINT above (see picture mode) except that it also writes a call for a point in the construction output file. Essentially we are writing an interpretive routine for a graphical construction and storing it to be read in later and used in picture or construct mode just as the current definitions for point or line can be used.

Procedure LINE in construct mode

Picking procedure LINE calls L%NE which operates like LIN above except that like P%NT it prints a call for an input line in the current construction file.

Procedure INTERSECT LINES

This procedure prints "LINE" on the teletype and then sends [A^{c}A^{c}A^{c}LL] to the display. The user responds on the display by picking a line with his light pen. The display sends a list of the entities in the aperture of the light pen. This list is then examined by the program until a line name is found. If no line name is found in the list the program prints "WHOOPS!" on the teletype and asks again for the line. After the first line has been found, the program asks for the second intersecting line.

The program computes the intersection point of the two lines and sends it to the display [A^{c}A^{c}A^{c}PT XXXX YYYY]. The program then associates a name and coordinates, etc. with the point (as with any point) and in addition prints a call to the routine to intersect lines in the construction file. It also associates the line name with its data file name to keep a history.

Procedure OUTPUT LINE

Procedure OUTPUT LINE is different than procedure LINE in that it expects as input the data file names of two points from the display rather than their coordinates. It sends [A^{c}A^{c}A^{c}LL] to the display and expects back a light pen list containing a point name. If it receives no point name it prompts the user. When it has gotten two points it sends the display a [A^{c}A^{c}A^{c}LE XXX1 YYY1 XXX2 YYY2] command. Then it writes a routine to output a line in the construction file and saves references to the line coordinates and a history of the line using the name the display sends back.

Procedure END CONSTRUCTION

When procedure END CONSTRUCTION is picked it closes up the construction file by in effect declaring all inputs to the construction to be variables (for details, see Section 6).

Procedure PICTURE MODE

Picking PICTURE MODE clears the display and returns to picture mode. The mode menu is again displayed.

Procedure Display File Generator (DFG)

Display file generator is a routine to climb up and down the data structure, a semilattice structure, and display what it finds there.

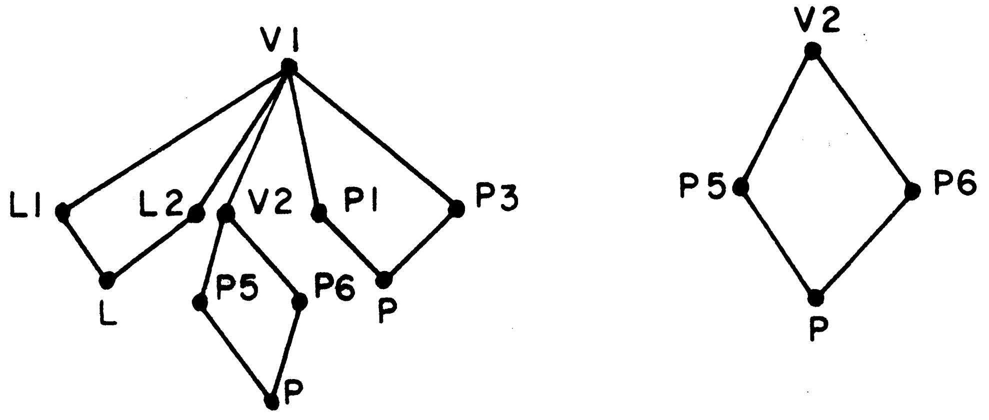


Figure C-7. Path of display file generator through a semilattice data structure.

In Figure C-7 above we have a simple picture V2 which contains just two points, P5 and P6. Picture V1 is slightly more complex, containing lines, L1 and L2, points P1 and P3, and an instance of picture V2.

DFG starts from the top of the semilattice and goes down the left-hand dendrite until it finds an end symbol in the above, the letter P standing alone. It then goes back up one level on the dendrite and finds the coordinates of the line or point by querying the memory. The appropriate command is sent to the display $A^{c}A^{c}A^{c}$LE XXX1 YYY1 XXX2 YYY2 or $A^{c}A^{c}A^{c}$PT XXX1 YYY1 depending upon whether L or P was found at the end of the dendrite. The display sends back a 4-digit display file entry name which the program associates as the history of the point or line as the case might be for the example of Figure C-7.

5.2 THE GRAPHIC MACROS

The macro generator approach was undertaken at the large computer because of the associative memory capabilities of TRAMP. The fact that we took such an approach there led us to consider a similar approach at the small computer. To this end, we defined a set of graphic macros which we felt would be sufficient for our experiment, though they certainly would be incomplete in a system having a full range of capabilities.

The set of graphic macros defined can be divided into several parts. These are the basic drawing commands, the display file editing commands, and the figure commands.

The five basic commands used in the construction of graphic entities are: POSITION (PN), POINT (PT), LINE (LE), NON-INTENSIFIED LINE (NL), and CHARACTER STRING (CH).

LINE and NON-INTENSIFIED LINE take four decimal numbers, in the range 0 to 1023, and an optional display file name as arguments. (There are 1024 points in the basic 338 display, but this is an easily adjustable parameter at the command language level.) If a display file name is supplied the new definition replaces the old definition of that display file. Clearly, the only difference between LF and NL is the beam status. POSITION and POINT are similar commands; both position the beam to the coordinates X,Y given in the arguments. POINT intensifies the beam once positioning is accomplished. CHARACTER STRING takes a position and a variable number of characters from the ASCII set, and displays this string starting at the given position.

One of the design decisions that even this limited subset of commands raises is: Should the commands refer to absolute locations on the screen at all times, or should there be some facility for relative or incremental coordinates? An example of a relative positioning command would be "RP 10, — 3" which would position the beam to a point 10 raster units to the right and three raster units below the current beam position. Relative commands for lines could be similarly designed. Although, with a little computation, we have some facility for relative behavior,* it is clear that we took the more absolute approach. A complete set of primitives would probably have facilities for both types of commands.

Editing of drawings can be carried out by using the commands: DELETE (DL), BLANK (BL), and UNBLANK (UB). Each of these commands takes a display file location as an argument, and either permanently deletes the item from the 338 display file, temporarily blanks the item without removing it, or unblanks a previously blanked item. A sketch may also be repositioned using the command MOVE (MV), which will translate the named item to a new position on the screen.

*Relative position is simply a non-intensified line from the current position to the desired position; the distance must be computed before the command is given.

Two other commands are provided which allow one to manipulate drawings. These are REPLOT (RP) and COPY (CP). The first of these plots a previously displayed entity at the current beam position. Internally, this means that another subroutine jump (push jump) to a previously existing section of code is generated and another so-called instance will appear. Instances thus have the property that if one changes, all change. COPY, on the other hand, generates a copy of the previously existing code in addition to creating a subroutine jump to this copy of the code. Thus a copy can be changed without modifying the entity from which it originally came. Both facilities were included in our command set so that we might gain some insight about the situations when each method of creating new entities is useful.

The next step is to allow a user to build up a library of drawings, each with a name, which he can call up for use in other drawings. To this end, we have defined the commands DEFINE FIGURE (DF), END FIGURE (EF), PLOT FIGURE (PF), MODIFY FIGURE (MF), and KILL FIGURE (KF).

DEFINE FIGURE places the system in a special mode whereby all succeeding actions with the light-pen are made relative to a starting point, and are placed in a special block until the END FIGURE command is given. At this time, the figure definition <u>in macro form</u> is stored on the disk under the BCD name given by the user. When a user wishes to include a previously defined figure in a new drawing, he issues a PLOT FIGURE command, which plots the named figure at the current beam position. Because of the subroutine feature in the 338, only one plot of a figure need be kept in core, no matter how many times the figure is used.

Figures, like any displayed entity, are subject to the editing commands DELETE, BLANK, and UNBLANK, as well as to MOVE. To remove or change the definition of a figure, however, the commands KILL FIGURE and MODIFY FIGURE are provided. MODIFY FIGURE brings a previously defined figure into core, displays it, and places the system in DEFINE FIGURE mode so that DELETE, MOVE, etc. will permanently change the definition of a figure from the appropriate tables and disk files.

The principal means by which a user communicates the above commands to the system is through use of the light-pen. Thus the two final commands of this admittedly incomplete set are POINTING MODE (PM), which places the light-pen in a mode whereby the name of an item pointed at will be sent to the 360, and LIGHT PEN TRACK MODE (LT), which causes a tracking symbol to appear and transmits to the 360 the X,Y coordinates at which the user finally loses tracking.

There exist several other macros which are more specialized. These include: Light pen track and return a List of all entities seen when tracking is lost (LL), Light pen track with rubber banding (LP), enable Pointing mode and return a list of all entities seen (PL), and track the Grafacon and return a point when the tip switch is pressed (GN). LP is, of course, useful in helping a user to see what line he will end up drawing. LL is useful in performing "gravity" calculations, i.e., the redefinition of the end point of a line (or some other entity) because of its proximity to other entities. The actual gravity calculation is carried out at the 360, but information about proximity should come from the terminal in order to avoid excessive calculations involving "nearness." PL is useful in a variety of situations where potential ambiguities are being resolved. For instance, when a point is defined at the intersection of two lines, then a person pointing at such an intersection would be pointing at three entities. PL will return a list of items and thus give information which can aid in the resolution. PM, in contrast, returns only the first entity seen plus an X,Y position. This is more useful than might be supposed, as will be indicated later.

Other, more mundane, commands are: Start Again (SN), which clears the screen and resets various system pointers; and Names (NS), which is useful in simulating inter-computer communications and thus helps in debugging.

5.3 COMMENTS ON THE TRAMP/RAMP SYSTEM

Our purpose in building the TRAMP/RAMP system was to provide a vehicle through which research questions in graphics could be approached. Note that the emphasis was definitely placed on research-type questions. Efficiency and/or speed of operation were sacrificed where necessary in order to provide a flexible tool. Thus, for example, an interpretive system was chosen as a host for the associative memory package. We feel that in this respect, our decisions are not greatly different from those of researchers who use LISP as their primary vehicle. To have been restricted by questions of efficiency would definitely have slowed our progress. Research in graphic software systems is difficult enough without having to grapple with a restrictive tool.

With this aim in mind, some comments are appropriate. It is probably fair to say that the 338 graphic console was the weakest link in our system from the point of view of flexibility. Our approach there was to define and implement a set of macros which were graphic in nature. With this set we hoped to be able to investigate questions of the division of labor between the small and large computers. More important, it was hoped that a macro approach would allow the definition of a number of figures and figure combinations whose optional parameters could be much more flexible than the traditional scale and rotation parameters current in more conventional graphic systems. Thus, for instance, one could specify round-headed or square-headed bolts as a variable in the macro call sent from the big computer, rather than having to specify a number of different figures. It is then easy to postulate a user calling up a library of general purpose figures which he can specialize for his needs, through the use of appropriate parameters. The definition of these display macros could also be done graphically though the conditions necessary to allow this might be quite stringent.

Our experiments along these lines were partially successful but were greatly hampered by the limited instruction set of the PDP-8. The code to implement this facility, when combined with the code for communication and display file manipulation, etc., quickly exhausted our 338 capabilities. Thus macro-expansion became the job of the 360, and no division of labor results were obtained. We still feel that the approach of macro definition of display files is a valid one. A larger terminal facility would undoubtedly aid in further investigation along these lines.

Moving to the large computer, it obviously became quite feasible to do extensive macro manipulations. The example of the graphic generation of graphic procedures, presented in Section 6, is one example of the generality available on the big machine. It is also instructive to note the concise procedures which result from the associative memory/macro generator approach. We offer the following example:

Find the set of all points joined by lines to a given point.

This may seem relatively trivial, since it requires only a knowledge of the end-points of the lines, a selection of those lines which end on the given point, and continuation by asking for all points now joined to the new set of end-points. There are two problems with this solution. The first is that we are potentially in an infinite recursion, where we must exclude all points previously found from our next search. The second is that the picture does not necessarily use the same name for points occupying the same space. As an example, in constructing the letter Y we draw two lines in a continuous fashion through the center point, and then draw the third line to intersect these two at the center point. Unless special software has been produced to check for this, there is no reason why the data structure should describe the third line as terminating at the same point <u>name</u> as that of the first two-line intersection.

This is really another example of the synonym in keyword searches, or common node points of graph theory. The problem occurs because, although the coordinates of the end-points may be the same, they have been defined at different times with different external or internal names. Figure C-8 is in three parts; it fully describes the process of finding all points. Part A is a description in a normal language; Part B is a solution in meta-language using the associative language of an earlier section; Part C is a set of TRAMP language statements describing the same process. It is important to note that the transformation from one language to the next is relatively simple, because of the similarity between the original language statements, the meta-language, and the TRAMP statements.

	-A-	-B-	-C-
1.	Name Program POICON Read Input	Function POICON Read into P1 ANSWER: PANS TEMPORARY: NEWP (holds current points generated)	# (DS, POICON) (# (DS,P1, # (RS)) # (DS,PANS,) # (DS,NEWP, # (P1)) # (TWO)))
2.	Find all Synonyms of all new points, re- move all previously found points (relative complement), find union of new points with answer set, and with latest new points to give newest points.	COOR(NEWP)=X,(FIND X) COOR(Y)=X, (FIND Y) RELCOM(Y,PANS) Z UNION(ZPANS) PANS UNION(Z,NEWP) NEWP	# (DS,TWO) # (RL,COOR, # (NEWP), *X*) # (RL,COOR,*Y*) # (X)) # (RCOM, # (Y) # (PANS), Z) # (UN, # (Z); #(PANS), PANS) # (UN,#(Z); #(NEWP), NEWP) # (THREE)))
3.	Find all lines from new point s, then ends of these lines. Remove all previous- ly found points. Union the new points to the answer set. If no new points found, exit.	END(X)=NEWP,FIND X END(X)=NEWP,FIND NEWP RELCOM(NEWP,PANS)→NEWP If NEWP is null, exit else go to step 2	# (DS,THREE,(# (RL,END,*X*, #(NEWP)) # (RCOM,#(NEWP),#(PANS),NEWP) # (UN,#(NEWP); #(PANS),(PANS) # (EQ,#(NEWP),,#(PANS)) (#(TWO)))))
	NORMAL LANGUAGE	AN ASSOCIATIVE LANGUAGE	TRAMP PROGRAM

Figure C-8. Point connectivity: Definition in an associative structure.

6. GRAPHIC LANGUAGE CONSTRUCTION OF GRAPHIC PROCEDURES

This describes an exercise in the construction of graphic procedures—procedures which operate on pictures—where the procedures are defined using a graphic language—one where the nouns and verbs are specified by actions at a graphic console. We undertook this exercise for a number of reasons. First, it seemed that such an exercise might demonstrate the flexibility and generality of the TRAMP/RAMP system. We feel the experiment accomplished this aim; we suspect that most graphic systems would have difficulty replicating this exercise. Further, the flexibility of TRAMP/RAMP allowed us to complete the exercise in reasonable time. Flexibility cannot easily be demonstrated on paper, but it is fair, we think, to offer this exercise as an indication of our claim.

The second reason we carried out this exercise was a wish to start investigating the much heralded "graphic language of the future." The existing languages are either oriented toward passive graphics, i.e., output pictures (21,22) or toward the definition of a control sequence based on the arrival of inputs (23). While such languages will certainly continue to exist, they are a far cry from the sort of high level man-machine communication that was postulated several years ago. Such high level communication can only happen when man could (1) build graphic procedure upon graphic procedure, (2) define each graphic procedure quickly and easily, making full use of the two-dimensionality of his input medium.

Finally, we suspect that such a graphical language, when investigated in depth, would yield a number of problems not unlike those encountered in the artificial intelligence field. We wanted to clarify our suspicions, making the problems precise wherever possible.

The rest of this section will therefore further describe our work on this exercise. We have reported the work in more detail elsewhere (4).

6.1 THE PROBLEM IN DETAIL

We define a graphic procedure as an algorithm, program, or function which takes graphic entities (i.e., POINTS, LINES, etc.) and produces outputs. As an example, we might have a graphical procedure which will find the perpendicular bisector of a given line. The algorithm has, as input, the line which is to be bisected; the output will be the required line displayed on the screen. This type of graphic procedure may be termed a "construction." Another type of graphic procedure could be one which checks whether two lines are parallel. This could be termed "checking a constraint." In this, the input to the algorithm is the pair of lines, and the result is a truth value (i.e., T or F). Finally, we have the type of graphic procedure which makes two lines parallel, either by moving one or both in some prescribed fashion. This can be termed "imposing a constraint."

Thus we will consider three types of "graphic procedures":

(a) Constructing

(b) Constraint checking

(c) Constraint imposing

Now the definition of any one of these graphic procedures could be done in a slightly augmented procedural language (such as ALGOL). However, it must be remembered that we are trying to state these procedures by using a graphic language. Thus the description of the procedure (akin to writing the program) is done by motions of a light pen on a screen, or pen on a tablet. These motions are neither the act of physical writing (using a character recognizer) nor defining a procedure by drawing its flow chart. In fact the motions are very similar to those a user carries out when he is executing a procedure using drafting equipment.

Unfortunately, when we explain such a procedure to another human being, many of our statements are heuristic in nature. But in this explanation, there may be dialog between the participants. In "explaining" or defining the procedure to a machine, we now have two possible modes of operation, analogous to the above.

Thus, one of the first questions which arose was whether to use heuristics or man-machine interaction as a basis for defining these procedures. We have all heard the arguments for one approach or the other. The interactionists strike terribly meaningful blows on the shoulders of the artificial intelligentsia, who continue unconcernedly to work towards that nirvana of no-man's land. In looking at this problem, _we_ came to the conclusion that a little bit of both was necessary, since neither seems sufficient.

Consider a heuristic program which can deduce a relationship from a given set of data and previously deduced relationships. This program may sit stewing over a picture trying to find out what can be inferred from it, when in fact, there is not enough information to produce any meaningful inference at all; the picture may not yet be complete enough to contain relevant inferences.

But the man-machine program <u>needs</u> the man present, and could ask embarrassingly silly questions, before "agreeing" with the user or his deductions.

If we consider a heuristic program which recognizes complex objects, we must have previously defined triangles, quadrilaterals, and other polygons as well as right angles, parallel lines, and the usual drafting "functions." Now suppose we use these definitions to describe a special type of roof truss; then we start drawing, and make the heuristic program follow our progress. Ultimately, the program must have found several hundred triangles, polygons, parallel lines, right-angled figures, etc., before it determines that the whole <u>entity</u> is this special type of roof truss. The fact may, however, be buried in the great volume of other relationships available. This paragraph is, of course, intended to point out the problems, not to damn heuristics.

Now consider man-machine interaction, when constructing a procedure graphically. Let us assume that a construct function exists, and can be run in the manner described:

The user looks at the CRT, and sees a picture and a MENU of allowable operations one of which is <u>CONSTRUCT</u>. He points at this with the light pen, and the question "NAME OF CONSTRUCTION?" appears on the screen. He types in "COPY ANGLE," and the machine responds "DRAW INPUTS." By pointing at <u>LINE</u> in the menu, and making appropriate light pen motions, he defines the three lines (1,2, and 3 of Figure C-9a), and point 4 on line 3. It is important to note that this particular construction has two implied constraints on the inputs (i.e., when this procedure is executed):

(i) lines 1 and 2 are not parallel

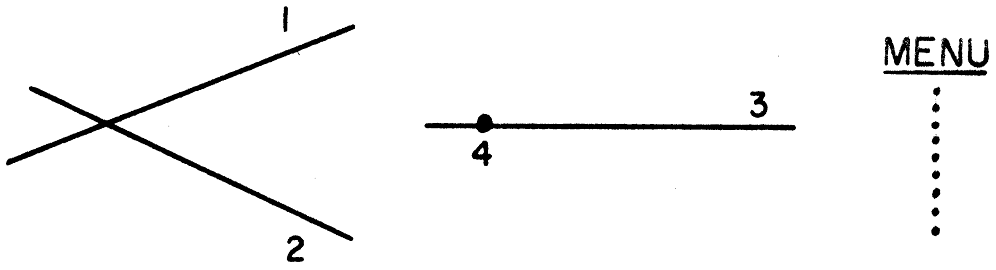


Figure C-9a. Construct inputs.

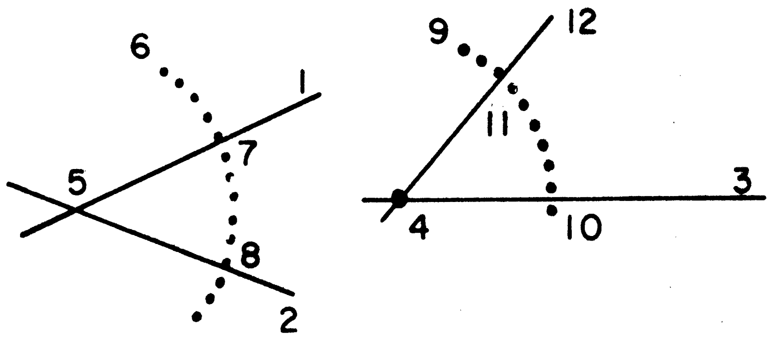


Figure C-9b. Construction phase.

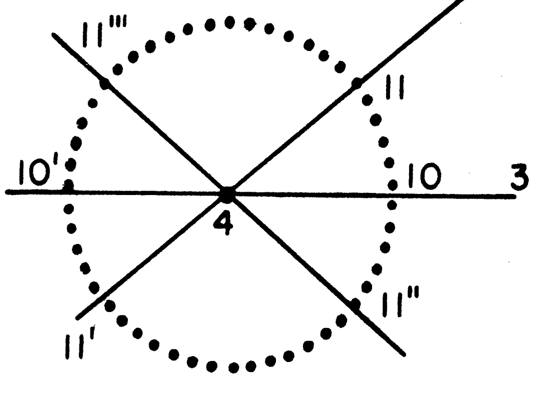


Figure C-9c. Four output possibilities.

(ii) line 3 contains point 4

There is also, of course, the constraint that the input objects must be three lines and one point presented in the proper order. It is possible to imply constraints by default conditions, e.g., the default of not stating that two lines are parallel would be that they must not be parallel. This would probably be more confusing than stating constraints explicitly.

Now the user starts on the "construction" phase; he defines point 5 as the intersection of lines 1 and 2, and takes an arbitrary arc 6 which intersects lines 1 and 2 in points 7 and 8 (see Figure C-9b). It should be noted that we now have two more potential problems:

(iii) what is an "arbitrary" arc? Is it always the same size?

(iv) the circle, of course, intersects each line in two places. How did we decide which of these two intersections to use?

Finally, the user strikes an arc with the same radius, centered at point 4. This intersects line 3 at point 10. He then measures a length, equal to that between points 7 and 8, along the arc from point 10 to define point 11. By drawing line 12 through points 4 and 11, he obtains the required line (the angle between lines 12 and 3 is the same as the angle between lines 1 and 2). This has, however, introduced a problem similar to (iv) above, because there is more than one angle that could be generated; in fact, there are four (as shown in Figure C-9c), although two of them are redundant.

Finally, the user must specify that line 12 is the output from the procedure; this raises another question:

(v) Is an output temporary or absolute? i.e., is the result always to be displayed, or merely passed on to be used as an argument for another procedure in a similar fashion to that of a nested procedure.

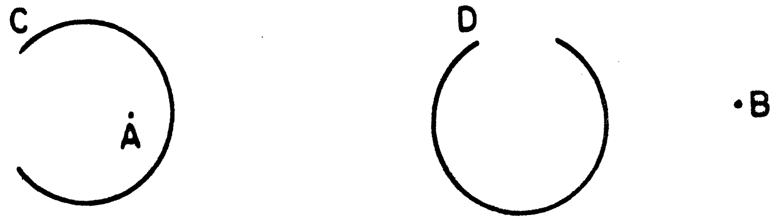
Before continuing, we must look at the five problems unearthed in the example:

The first two are associated with the problem of explicit and implicit constraints. Presumably, by specifying his inputs, the user implies that these inputs have certain properties, e.g., that they are POINTS, LINES, CIRCLES, etc. He may also wish to impose certain constraints which are not easy to state, e.g., that two circles have a common tangent. We must define these constraints explicitly; one way to do this is to "write" a procedure— using the same graphical CONSTRUCT techniques, except that the result of executing this new porcedure is not an output entity but the TRUTH function. The execution of this procedure is then the constraint checking procedure described earlier.

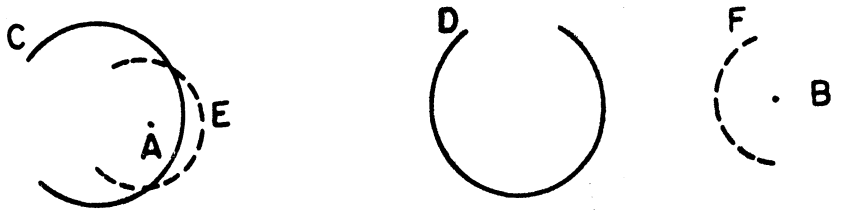
The problem of arbitrary values is not yet solved, though these are certain possible techniques. First, the user could be asked to give the size or angle, etc. that he wishes to use. This is probably the worst way to solve the problem, because the used may not know what is 'best,' in fact he may even be confused by the request, since he may have no idea why the arc must be drawn. More is said on this later. A second method is to pick the value arbitrarily, e.g., a length may be one half the screen size. This obviously has one disadvantage, it may not be appropriate, for the command in the textbook may be "draw an arc to intersect the second circle," and the arbitrary (half screen) arc may completely miss. A third method is to start with some given value, such as in the second method, but modify this to suit the situation. This may be termed "hunt-and-peck" and is heuristic. This approach is probably the most satisfactory of the three, except that it is difficult to talk about several arcs, all arbitrary, which might have to be _copied_ (i.e., the same size duplicated). One can also imagine the case where a second circle (copied as having the same radius as the first) may also have to pass through a third circle, thereby imposing a change on the first circle (for further explanation, see Figure C-10). This situation may seem contrived, but could occur for systems where the problem was _over_ constrained.

The fourth problem can be tackled in several ways. The first method is by making an exact definition of sequence and angle. This could involve statements that an angle goes from the _first_ line to the _second_, is always acute, and that all arc lengths measured along a circle are in the anticlockwise direction. With these definitional constraints, there is only one choice of intersection in this example. Unfortunately, however, when we consider the intersection of a circle with a line, we often need both points. Therefore, an additional statement might be made—"this is the solution to the procedure if only _one_ point is required by the next procedure." The solution of this fourth problem is therefore either careful design involving careful description of meanings, or else a heuristic approach, maybe involving a learning process on the part of the procedure.

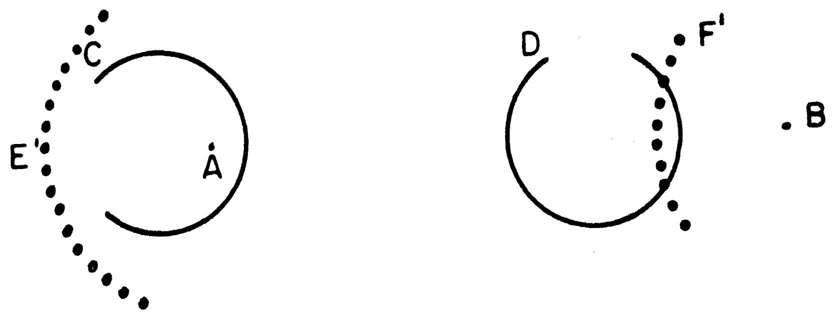
The final problem is associated with the use of the output, but must be resolved by the user rather than by the procedure. The user, on requesting



Problem: Draw an arbitrary circle, center A to intersect circle C, then with the same radius, draw circle center B to intersect D



After first attempt, E intersects satisfactorily, but F does not. Modify radius to suit second constraint



Now circle E does not intersect C, but the solution is bounded:

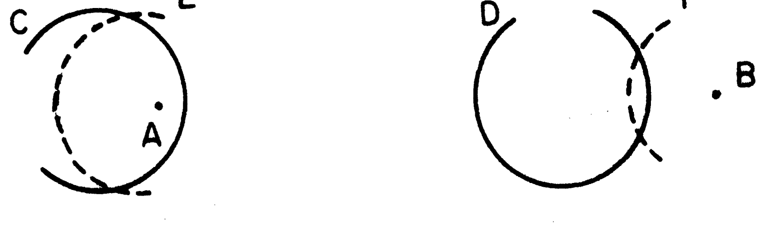


Figure C-10. Arbitrary circles and imposing constraints.

execution of a construct procedure, could be given an answer in a "temporary mode" (e.g., as a flashing object on the screen). He might then either use it as input to another procedure (in which case it is treated as a temporary value, and disappears after use in the next construct) or else made permanent. This can be achieved by defining the output the attribute of permanency, and also displaying the object by placing it in a permanent display file.

Suppose we try, however, to use the "man-machine interaction" technique, and have procedure COPY ANGLE ask a question (at the end) which says "There are two answers, which do you want?" and display both (possibly flashing) so that the user can choose which he needs. The user wishes to define a new procedure (using COPY ANGLE) to produce a PARALLEL LINE construction (see Figure C-11).

The construction requires an "arbitrary" line 3 which passes through point 2 and intersects line 1. Then by using the "COPY ANGLE" routine, line 4 is produced so that it and line 3 include the same angle as that between lines 3 and 1. At the last stage, the user is asked "there are two answers, which do you want?" (the required answer is 4, but another answer is 4').

Another problem occurs when this new PARALLEL LINE procedure is used by someone other than the definer. This unsuspecting user merely asks for a parallel line—and is startled by the question (generated by COPY ANGLE, which he may not appreciate exists), "There are two answers, which do you want?"!

Now that the principal problems have been discussed using an example, it is necessary to consider the structure of the final procedure. Presumably,

Figure C-11a. Inputs for "parallel line."

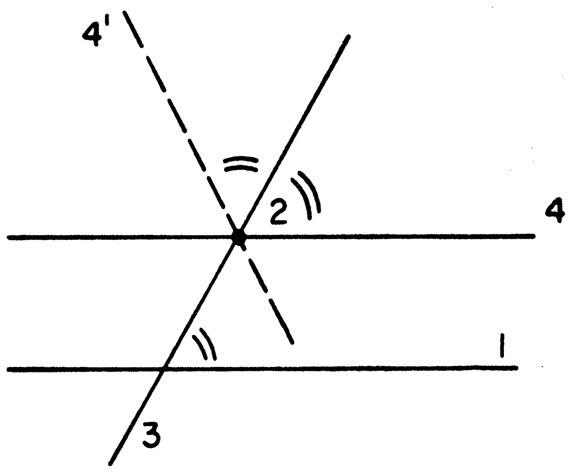


Figure C-11b. The construction "parallel line."

the procedure must be either compiled (into a subroutine) or grouped into a set of interpretable statements (in the form of a function). This involves definitions of a program name, the inputs, any validation criteria, the procedural statements, and finally the outputs. It is possible, therefore, to have the following formats:

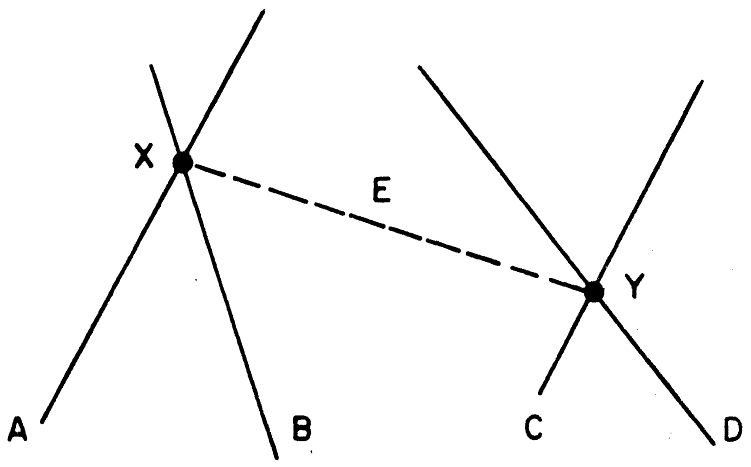
```
NAME                NAME
INPUTS              TRANSFER ITEMS(INPUTS & OUTPUTS)
VALIDATION          VALIDATION
PROCEDURE           PROCEDURE
OUTPUTS
```

For example, the procedure could be a quasi-FORTRAN program to perform a procedure shown in Figure C-12:

```
       SUBROUTINE   INTLINE (A,B,C,D,E)
       IF TYPE((A, 'LINE').AND.
     1 TYPE (B, 'LINE').AND.
     2 TYPE (C, 'LINE').AND.
     3 TYPE (D, 'LINE'))
     4 GO TO 10
 5     RETURN F
10     IF(INT(A,B,X))
     1 GO TO 20
       GO TO 5
20     IF(INT(C,D,Y))
     1 GO TO 30
       GO TO 5
30     LINE (X,Y,E)
       RETURN T
       END
```

In this somewhat "traditional" approach, the assumption is that all arguments are passed by name,* that the three subroutines called are isomorphic, all being called by name, in a transfer vector and returning the result through

*In fact, the usual descriptive known is not the item name but its synonym in the form of a display file entry. This is, therefore, a second level of indirection.



Given lines A,B,C,D

Intersect the first two to give Point X
Intersect the last two to give Point Y
Construct line E from X to Y, output E

Figure C-12. The intline procedure.

the transfer vector with a logical value of the routine passed as the result of the "function," so that it can be tested in an IF clause. The routine TYPE is intended to determine the type of item (e.g., its "pictorial" or other class, such as POINT, LINE, FIGURE, NUMBER (INTEGER, FLOAT, ETC), ALPHA, etc.) and compare it with the character string which is its second argument, returning T for a match or F for none. The routine INT will take the lines in pairs (A,B) and (C,D) and find their intersection points. If this is outside the limits of reality (e.g., it is far off the screen limits or if the lines are parallel) it returns F, otherwise it returns T and the coordinates of the Point are returned as X. The routine LINE takes two points as its first arguments and defines the line between them as the third argument. In this procedure, it is not necessary to check for errors (i.e., whether the result of this procedure is "true"). Since it has no possible error condition (X and Y must be points in order to arrive at this part of the program), presumably the next operation is to compile this "program" for future use.

The same example as a TRAMP procedure would be:

```
    #(ds, INTLINE,(
    #(eq, #(TYPE,A), LINE,
     (#(eq,# (TYPE,B),LINE,
     (#(eq,#(TYPE,C), LINE,
      (#(eq,#(TYPE,D), LINE,
       (#(eq,#(INT,A,B,X),T,
        (#(eq,#(INT,C,D,Y),T,
         (#(LINE,X,Y,E)),F)),F)),F)),F)),F)),F)))
#(SS,INTLINE,A,B,C,D,E)
```

The satisfaction of constraints in these two examples is relatively unsophisticated, since it merely involves calling special-purpose routines which test for the required "existence" condition that the (user-) specified inputs are of the correct type, so that we shall not try to intersect a LINE with a POINT or FIGURE. There are generally, however, much more complicated constraints (often implied in textbook definitions) such as:

The two lines are parallel

The three lines form a <u>triangle</u> (which implies that we have a <u>closed</u> 3-sided figure), etc.

Finally, the CONSTRAINT CHECK procedure will be very similar to the CONSTRUCT procedure, except that the graphic procedure generator must not put out a result, but instead check to see whether the temporary result of the "CONSTRUCT" is equal to that of the final "input." This may be accomplished by writing essentially the same program as CONSTRUCT, but enclosing it in an "IF" statement to give the form:

Condition ≡ (IF, result of construct (equals) constraint, give True, otherwise give False).

Which then itself is called outside the construct as (IF condition (constraints), go to procedure, print error and call procedure recursively which asks for inputs again.

6.2 A SIMPLE EXAMPLE

In order to illustrate the operation of the CONSTRUCT procedure, which, although relatively unsophisticated to the user, is quite complicated, a simple example of its use will be described. The construction, illustrated in Figure C-13 is to "draw a line between a given point and the point of intersection of two given lines." Obviously, this is an extremely simple operation, containing none of the ambiguities described in the section above.

As Figure C-13 is constructed the figure not only appears on the screen but a generalized procedure corresponding to it is stored in FILEX. Once THING has been defined it can be used to repeat the sequence of displaying an output line, given two intersecting input lines and an input point.

On initiating the CONSTRUCT procedure, the user is asked for the "Output File Name?," where the graphic procedure (generated by CONSTRUCT and the user's actions) is to be stored. Figure C-14 shows the contents of the output file.

Then he is asked to "Name New Construction," he replies "THING." Next he is asked to "Define Inputs!," which he does by drawing them in the way he would if he were merely using the graphic system to draw pictures; thus as far as the user is concerned, he is drawing a picture, but the CONSTRUCT procedure is also generating code.

The procedure #(LINENAME) will check to see if the data file name returned by the display corresponds to a line (at the time when the construction THING is actually used). The procedure THING++ will be segmented on...,2005,... in order to generalize the construction, i.e., in order to pass inputs into the graphic procedure.

Once a construction has been defined, the file may be read, by the TRAMP procedures, and thus this new program is available to the user. The construction name is added to the "picture mode" menu for later use.

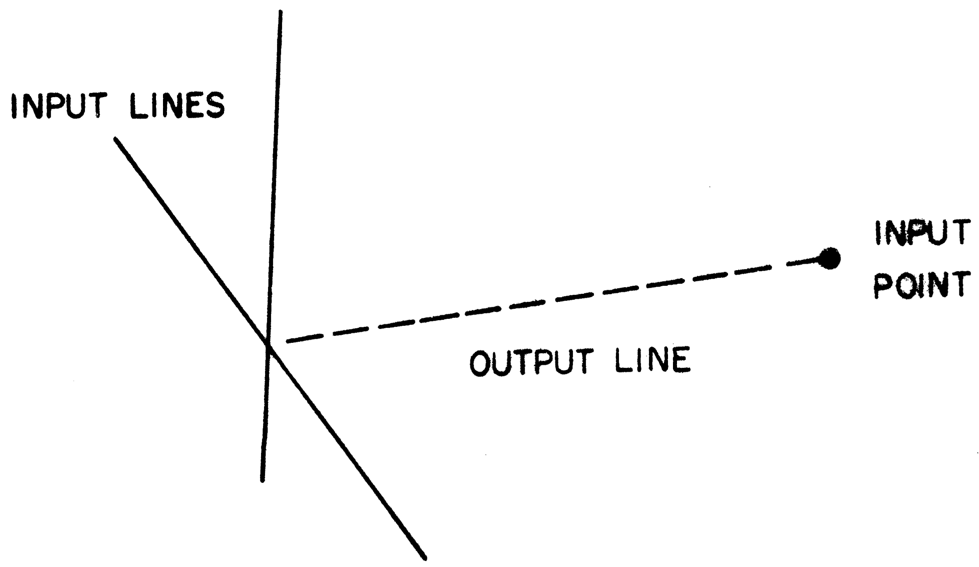


Figure C-13. Construct "thing."

```
1       #(DS,THING,(&
1.001   #(PS,GIVE INPUTS!)#(THING++&
1.002   ,#(LINENAME)&
1.003   ,#(LINENAME)&
1.004   ,#(POINTNAME)&
2       #(DS,THING++,(&
2.001   #(DS,L1,2005)&
2.002   #(DS,L2,2006)&
2.003   #(DS,P3,#(INTERSECT++,##(L1),##(L2)))&
2.004   #(DS,P4,2008)&
2.005   #(OUTPUTLINE,##(P4),##(P3))&
2.006   )...
2.007   #(SS,THING++,2005,2006,2008)
```

Figure C-14. List of contents of output file.

Note that, at execution, #(LINENAME) will replace a data file name with a line name, for example, line name L0913 might replace data file name 2065.

The sequence would then be

#(THING) which does the following

#(PS,GIVE INPUTS!) #(THING++,#(LINENAME),#(LINENAME),#(POINTNAME))

which prints on the teletype GIVE INPUTS!

the user then selects a line, an intersecting line, and a point on the display. After the 360 names corresponding to the 338 data file names have been evaluated by #(LINENAME) and #(POINTNAME), the procedure calls THING++ with its arguments:

#(THING++,L1403,L913,P613)

Which become, within procedure THING++

#(DS,L1,L1403)

#(DS,L2,L913)

#(DS,P4,P613)

etc.

6.3 FURTHER INVESTIGATION IN THIS AREA

If work in this area were to continue, various comments about it could be made. First, constraint checking on inputs must certainly be much more sophisticated. It has been suggested above that this involves "writing" (perhaps graphically) a number of constraint check procedures. While this is certainly true, it should be noted that the problem becomes more manageable when one can take advantage of properties of graphical entities such as symmetry, transitivity, etc. Thus one would like to be able to write basic routines and have recursive, symmetrical, and other sorts of considerations handled by a separate definition.

Moreover, it is important that wide ranging use be made of existing graphic procedures. This necessitates the ability to combine them via logical statements of combination. TRAMP should be an excellent vehicle for doing both of these things.

Second, it has been stated above that the problem of an "arbitrary" arc (or an "arbitrary" anything) has not been solved except in specialized ways. While this is true, it is the sort of problem which might be amenable to a theorem-proving approach, since the requirements can be "axiomized."

Finally, throughout this discussion has been implicit an assumption about "facts." We have assumed that facts like theorems or relationships between elements are easily retrievable and can easily be built upon. This assumption is largely the result of working with the TRAMP system, under which the assumption holds elegantly. Probably the most important result of this effort has been in demonstrating the necessity for an associative memory approach when working with experimental graphic systems.

REFERENCES

1. Ash, W. L., and Sibley, E. H., TRAMP: A Relational Memory with an Associative Base, Technical Report 5, Concomp Project, University of Michigan, Ann Arbor, June 1967.

2. Ash, W. L., and Sibley, E. H., "Tramp: An Interpretive Associative Processor with Deductive Capabilities," Proc. 1968 A.C.M. Nat'l Conf., pp. 143-156.

3. Sibley, E. H., Taylor, R. W., and Gordon, D.G., "Graphical Systems Communication: An Associative Memory Approach," Proc. 1968 FJCC, pp. 545-555.

4. Sibley, E. H., "The Use of a Graphic Language to Generate Graphic Procedures," Pertinent Concepts in Computer Graphics, Proceedings of the Second Illinois Conference, April 1969.

5. Sibley, E. H., Taylor, R. W., and Ash, W. L., "The Case for a Generalized Graphic Problem Solver," Proc. 1970 SJCC, pp. 11-17.

6. Ash, W. L., A Compiler for an Associative Object Machine, Technical Report 17, Concomp Project, University of Michigan, Ann Arbor, May 1969.

7. Allen, John J., Man-Computer Synergism for Decision Making in the System Design Process, Technical Report 9, Concomp Project, University of Michigan, Ann Arbor, June 1968.

8. __________, "A Survey of Generalized Data Base Management Systems," CODASYL Systems Committee, May 1969.

9. Sutherland, I. E., SKETCHPAD: A Man-Machine Graphical Communication System, Technical Report 296, Lincoln Laboratory, January 1963.

10. Coons, S. A., "An Outline of the Requirements for a Computer-Aided Design System," Proceedings 1963 SJCC, pp. 299-304.

11. Ross, D. T., "The AED Approach to Generalized Computer-Aided Design," Proc. 1967 ACM Nat'l Conf., pp. 367-386.

12. English, W. K., Englebart, D. C., and Berman, M. L., "Display-Selection Techniques for Text Manipulation," IEEE Transactions on Human Factors in Electronics, Vol. HFE-8, 1967, pp. 5-15.

13. Moffett, T. J., "On-line," Modern Data, September 1969.

14. Roberts, L. G., "Graphical Communication and Control Languages," Second Congress on Information Systems Sciences, pp. 211-217.

15. Gray, J. C., "Compound Data Structure for Computer-Aided Design; A Survey," Proc. A.C.M. Nat'l Meeting, 1967, pp. 355-365.

16. Dodd, G. G., "APL—A Language for Associative Data Handling in PL/I," Proc. AFIPS FJCC, Nov. 1966, pp. 677-684.

17. Feldman, J. A., "Aspects of Associative Processing," Lincoln Laboratories, Lexington, Mass., April 1965.

18. Rovner, P. D., and Feldman, J. A., "The LEAP Language and Data Structure," Proc. IFIPS Congress, 1968.

19. Rovner, P. D., "An Investigation into Paging a Software-Simulated Associative Memory System," University of California, Berkeley, Doc. 40.10.90, Jan. 1966.

20. D'Imperio, M. E., "Data Structures and Their Representation in Storage," Annual Review in Automatic Programming, New York, Pergammon Press, 1968.

21. Herzog, B., "Computer Graphics for Designers," Emerging Concepts in Computer Graphics, W. A. Benjamin, Inc., 1968, pp. 189-230.

22. Duffin, John David, A Language for Line Drawing, Technical Report 20, Dept. of Computer Science, University of Toronto, May 1970.

23. Newman, W. M., "A System for Interactive Graphical Programming," Proc. 1968 SJCC, pp. 47-54.

APPENDIX D

DEVELOPMENT OF A SET-THEORETIC DATA STRUCTURE

Basis for Machine-Independent Information Management Systems

D. L. Childs

TABLE OF CONTENTS

A SET-THEORETIC DATA STRUCTURE (STDS)

1. Purpose: To provide a storage structure representation of arbitrarily related data allowing the data to be stored as SETS and then retrieved using SET OPERATIONS.

2. Consequences:

   a. Flexibility, generality, and scope allowed by set theory.

   b. Pointer-free representation of data.

   c. Minimal storage requirements.

   d. Quick access of data.

   e. Easy modification of data.

   f. Ideal for paging environment.

   g. No restrictions on questions as long as they are formulated using sets and set operations.

ABSTRACT

A search for commonality of purpose in existing data structures showed the basic concern of all data structures to be: INFORMATION versus MACHINE REPRESENTATION. Since all information is machine independent before being forced into a machine representation, an effort was made to separate those properties of data structures which were machine independent, the INFORMATION ENVIRONMENT, from those properties which were machine dependent, the MACHINE ENVIRONMENT, and then develop a data structure which allows isolation and separate control of these properties. This approach yielded the following definition: A DATA STRUCTURE IS AN ISOMORPHISM BETWEEN A MACHINE ENVIRONMENT AND AN INFORMATION ENVIRONMENT PRESERVING THE PROPERTIES OF EACH. The particular type of data structure depends on the structure of the information environment. When the information environment is represented by set theory, the isomorphism is a SET-THEORETIC DATA STRUCTURE.

KEY WORDS AND PHRASES: information environment, machine environment, data structure, set theory, Set-Theoretic Data Structure, transformation operations, set operations, machine independence

At the beginning of the CONCOMP project, Dr. Franklin H. Westervelt initiated an investigation into data structures. It was his view that present efforts in data structure development were too machine-oriented and would, inherently, be unable to handle future needs. It was his contention that users of huge data bases of the future should not be burdened with the intricacies of complicated data representation -- rather, data representations should be transparent to the user. In fact, the user should not even be aware of the type of storage media or even the type of machine being used. Dr. Westervelt had often lectured on the need for a general data structure that was not machine oriented but one that was information oriented, where design emphasis was placed on information content instead of machine representation. For example, in a graphics problem concerning diagrams constructed from lines and points, the positions of lines and points relative to one another are invarient no matter what machine is used or how the information is stored. The idea behind this investigation was to try to separate those properties of data structures which were machine independent from those properties which were machine dependent, and then develop a data structure which allowed isolation and separate control of these respective properties. Historically it seems that most data structures have been developed for a particular problem on a particular machine and then generalized. It seemed that the best approach would be to start with a general means for expressing any problem and then worry about expressing the general representation on a particular machine. This was the approach taken by David L. Childs which eventually resulted in the development of the concept of a Set-Theoretic Data Structure.

A search for commonality of purpose in existing data structures showed the basic concern of all data structures was: INFORMATION versus REPRESENTATION. Other complexities involving the amount of data, kinds of questions to be answered, speed with which they were answered, updating requirements, all depend on the information contained in the data and the representation of this information in the machine. These two areas can be characterized separately and independently, and therefore should be approached separately and independently instead of being blended together. However, this does not seem to be a widely accepted practice in developing data structures.

In order to pursue this approach a distinction must be drawn between a MACHINE ENVIRONMENT and an INFORMATION ENVIRONMENT. The first may partially be characterized by: addresses, codes, memories, machine hardware, computer software, programming costs, data locations, sorts, storage medium, storage size, retrieval speed, cpu-seconds, pointers . . . in short: the empirical world of the computer. This seems to be the most comfortable area to work in for those involved with the development of data structures. However, it only covers half the problem. The second area is not so easily characterized, which may partially account for its not being isolated. This is the area of 'pure information', data relationships, questions, answers, information extraction . . . in short, the abstract world of information. Here rests the heart of the data structure problem, for in order to treat the information environment and the machine environment as functionally separate entities, their respective structures must be precisely stated. The structure of the machine environment is generally accepted, but a separate structure for the information environment is not. Therefore, the information environment is usually couched in terms of the machine environment, thus abolishing any hope of functional separability. The problem then is to devise or find a suitable structure for the information environment, such that the resulting information environment operations will insure the same result, independent of <u>any</u> particular machine-representation of such information. If the information environment operations themselves are defined in terms of a particular machine-representation, then they are intrinsically dependent on that representation. (This, unfortunately, characterizes many current data structures.) The means for isolating information environment operations from the machine-representations of data is not necessarily obvious. The difficulty arises in trying to insure 'generality' and 'consistency.' 'Generality' in that the operations must not be restricted to a particular information configuration, but must be applicable for any possible information configuration; and 'consistency' in that, independent of the information configuration, the operations must yield unique and well-defined results. This amounts to the adoption or development of an abstract mathematical model for the information environment. (Set theory was eventually chosen as such a model.) Assuming that such a model can be found and that the machine environment and the information environment are functioi ally separate and completely independent structures, the function of a general data structure would be to connect them in such a way as to preserve their integrity. Therefore, the view is proposed that: ANY DATA STRUCTURE IS ACTUALLY AN ISOMORPHISM BETWEEN A MACHINE ENVIRONMENT AND AN INFORMATION ENVIRONM PRESERVING THE FUNCTIONAL ASPECTS OF EACH. The particular type of data struc ture would depend on the structure of the information environment. The schema in Figure 1 represents the relationship of the 'user,' the information environment, the data structure, and the machine environment. It will be argued subsequently

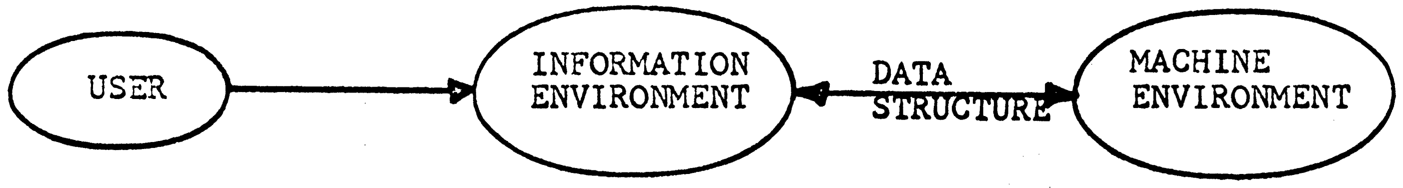


FIGURE 1

that this definition allows a data structure to enjoy a degree of machine independence. However, it may not be immediately obvious that such a definition implies any operational improvement in the use of the resulting data structure; but since any particular machine representation has inherent properties associated with it (initial programming cost, storage allocation, updating costs, retrieval speed, etc.), some machine representations are better for some purposes than others. Given a selection of representations, which one, or ones, should be used with a particular data base; since, in general, no one representation is best or even adequate for all retrieval needs? The answer depends on which storage and retrieval requirements best meet the information environment conditions (which may even change during interrogation of the data base). If the different machine representations could be transformed from one to another (at a 'minimal' cost) without affecting the form of the retrieval program, then any retrieval requirement could be facilitated by the selection from machine representations available. The control of these operations should be accessible to the user, but since these operations do not affect the information environment, only the machine environment, they should be distinct and separate from the operations of the information environment. Figure 2 represents this schematically. The box labeled ME-Controls allows for any operation which affects only the machine environment, which in turn can not affect the information environment since they are functionally separate.

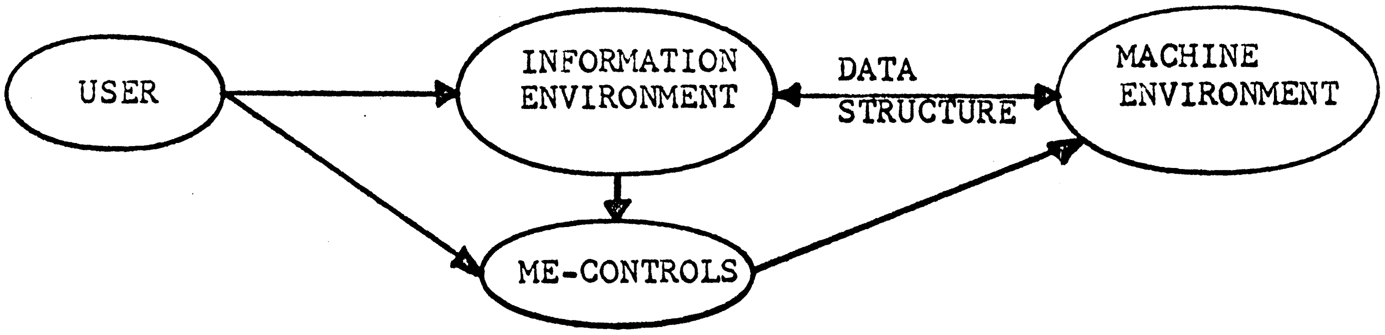


FIGURE 2

Therefore, these operations do not belong as part of a data structure, (though the previous definition of a data structure does not preclude the use of these operations in some other context). The transformation operations, or MODE operations, are in this catagory and every transformation must have some cost associated with it in the form of storage and cpu-time used. From the user's view the mode operations are independent of the information environment being used, they are to allow transformations with a cost savings compared to the resulting retrieval and storage costs. The most expensive transformation cost would result if each transformation had to be programmed when the need arose, which is the current situation with many data structures. With MODE operations, however, any transformation can be made between existing machine representations subject only to the gained or lost retrieval speed versus the increased or decreased storage requirements. It is important to remember that the information is an invariant under these transformations, only the machine-representation of the information is changed. With MODE operations any user has the following economic flexibility: During extended periods of non-use the data can be compacted and stored at the lowest possible storage rate. When the need arose for interrogation, modification, updating, or analysis the data could be put into an expanded storage form, thus acquiring the most economical retrieval costs and fastest response time.

The concept of a Set-Theoretic Data Structure (STDS) evolved over a period of three years. The initial direction leading to its development was inspired by the efforts of Timothy E. Johnson and his work, and by the work of Jerome A. Feldman, on Associative Data Structures. A most significant feature of the ADS was the introduction of the functional notation $A(o) = v$. This notation is strictly a part of the information environment and can have an arbitrary implementation in the machine environment. The principle drawback, however, with any functional notation is the prerequisite for the arguments to be single valued. This lack of generality at the outset may be an undue restriction. Since a more comprehensive notation does exist, allowing a collection of values for the arguemtns, it seemed worth exploring. The notation $A[O] = V$ may not seem to connote any more information or generality than did $A(o) = v$. However, this notation represents the IMAGE operation in set theory, where 'O' and 'V' are sets of values instead of being restricted to single values. Ambiguities possible with a functional notation are no longer a problem with the image operation: if 'S' represents 'square root' then what is the value of v in $S(4) = v$? Here two values for 'v' are correct: +2 and -2. The ambiguity disappears using the image notation $S[\{4\}] = \{+2, -2\}$. This may further be demonstrated: by the '719-th root of 13,' by the 'children of x,' or by the 'points reachable from y.' Besides increasing the generality of $A(o) = v$ to $A[O] = V$, the introduction of 'sets' allows the potential of utilizing the powerful operations of set theory (which are not defined algorithmically, thus allowing easy adaption to machine independence).
Crispin Gray and Charles Lang considered set theory in their 1967 paper on ASP. They were also quick to point out the inherent difficulty of attempting to implement set theory on a digital computer: a set by definition is an <u>unordered</u> collection of objects while any machine representation forces an <u>order</u> on the elements. For every set with n elements there exist n! different orderings. Which of these should be used for a storage representation, can a canonical ordering always be found, or does it make any difference? Much of the investigation into data structures was directed toward resolving these questions. It does make a difference! One particular ordering is preferred and it can always be found, (thus allowing a canonical machine representation for any arbitrary set). During the investigation

it became ever increasingly evident that set theory -- if it could be applied to a computer -- would be an ideal structure for the information environment. The feasibility of a set-theoretic approach was described in Concomp Technical Report 6 and was also presented in Edinbourgh, Scotland, at the IFIP Congress 68, under the rather cumbersome title: "Feasibility of a Set-Theoretic Data Structure: A general structure based on a reconstituted definition of relation." The result allows the separation of the information environment from the machine environment, by letting set theory represent the information environment. It is now the function of the data structure to tie the two back together again, using the earlier definition of data structure as an isomorphism.

DEFINITION: A Set-Theoretic Data Structure -- STDS

> A STDS is an isomorphism between a machine environment and set theory preserving a given universe U under a collection of set operations S.

This conforms to the schema of Figure 1 with the information environment represented by set theory. One immediate consequence of this definition is that from the user's view (the set theory side) it appears to be a machine independent data structure. Some people may argue that a machine independent data structure is a contradiction in terms since any data structure must be concerned with addresses, searches, pointers, word length, and other machine dependent characteristics. By such a definition, of course, a machine independent data structure is a logical, and hence technical, impossibility. Whether or not there is any agreement to calling an isomorphism a data structure in some classical sense is irrelevant. The relevant issue is the separation and definition of those aspects of "data structure usage" that do or do not depend on the particulars of a given machine. It would seem that any data structure could properly be called machine independent if a user could access <u>any</u> type of data represented in <u>any</u> way desired without needing to know what machine was being used. The only clue to the machine might be the storage sizes available and the retrieval times experienced. To justify that STDS is a machine independent data structure in this sense, it may help to examine why other data structures are not machine independent. Most fail immediately since they require a fixed data representation and seldom allow spontaneous construction of questions. The data is forced into these fixed structures which are already geared to specific types of questions. <u>All</u> data is machine independent <u>before</u> it is forced into a particular machine. What happens to this information independence, why is it lost? It is only lost in the sense that it can no longer be accessed directly, it has to be embraced with "handles" or "hooks" or pointers or labels or whatever, which in turn are able to be accessed. The result is that all data is accessed indirectly through arbitrary appendages. Unfortunately these appendages always seem to possess machine properties which now render the data accessability machine dependent. The reason for all this indirection seems to stem from the view that all information processing must be algorithmic in nature.

A command like "FIND" must invariably reduce to an algorithmic language or procedural representation, which on the surface seems quite reasonable since any machine implementation of anything results in an algorithim. There, however, is the flaw! What is necessary for the information environment is a structure emboding operations that are <u>not</u> defined procedurally, but whose implementation, of course, will be. Since the operations do <u>not</u> dictate a procedure, <u>any</u> procedure giving the correct result is legitimate. Set theory is such a structure. Given two sets 'A' and 'B', and any set operation '*' then A*B = C is completely defined for all 'x' if it can be determined if 'x' is an element of 'C' <u>just</u> by determining the truth or falsity of a statement concerning the membership of 'x' in 'A' and in 'B'. In other words, in set theory only the result is defined, <u>not</u> how to obtain the result. Therefore, if this aspect of set theory can be preserved and implemented on a computer, then all operations would be independent of how the sets were represented in the machine and only dependent on the information content of data represented. <u>Any</u> procedure for executing the set operation would work as long as the information content of 'C' was correct. 'C' could even be represented in the machine a different way every time. Clearly there would be representations that would be more desirable than others for certain operations. Since the information content of any set is the same no matter what the machine representation is, a set can have its representation changed without effecting the result of <u>any</u> set operation, (the only effect would be on execution time and storage allocation).

A STDS, therefore, allows a user to see only set theory while operating on a machine environment. However, a user may wish to exercise control over the machine representation. The MODE operations, mentioned earlier, allow changing from one machine representation to some other. Only 'time' and 'storage' characteristics need be known about the different MODES or machine representations since those are the only affects that can be detected by the user. Figure 3 represents the schema for a STDS.

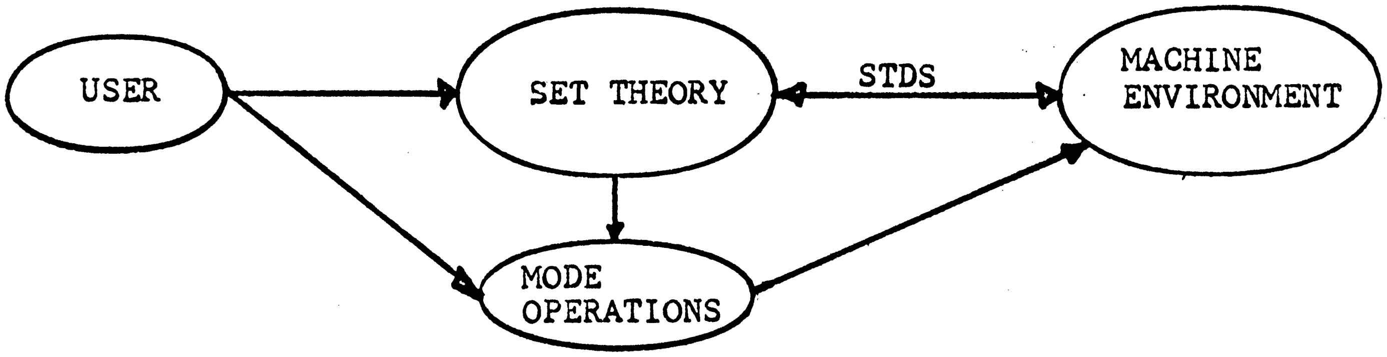


FIGURE 3

<u>Any</u> structure that allows referencing the data directly without dependence on artificial devices would give this kind of machine independence. The viability of such a structure depends on the expressive power allowed by the interaction of the operations. Here is the real strength of set theory. No argument can do it justice. Only first hand experience can demonstrate the inherent power of the set operations. For this reason, an interactive demonstration calling program for set operations was written for use in MTS. People who were introduced to it very quickly were doing retrieval queries they had previously thought to be impossible. (Examples appear in the appendices).

With set theory as a model the information environment acquires all the precision, generality, and formalism inherent in set theory. The general concepts proposed earlier may now be particularized starting with the separation and delineation of the information environment and the machine environment. The result is two separate and functionally independent spheres of activity, characterized in part by the following table:

INFORMATION ENVIRONMENT	MACHINE ENVIRONMENT
SETS	ADDRESSES
PARTITIONING	POINTERS
N-TUPLES	SORTS
NESTED SETS	RINGS
MEMBERSHIP	LISTS
RELATIONS	STORAGE MEDIA
SET OPERATIONS	RANDOM ACCESS
INDEX SETS	MULTILIST FILE
SUBSETS	PAGING
CARDINALITY	VIRTUAL MEMORY

Any collection of information can be represented by a set in the information environment, while the mode of that set is the particular machine-representation of that set in the machine environment. Changing the mode of a set clearly does not change the information content of the set but only how that set is represented in the machine environment. A legitimate mode can be modeled after any of the currently popular data organization techniques: trees, lists, inverted lists, ring structures, cellular multilist files, index sequential files, hash codes or content addressable organizations. In fact, any future machine-organization or hardware memory device can be utilized for a mode, or machine-representation of data. In all cases the information retrieval speed characteristics of the data organization will be preserved. However, due to the intrinsic advantages of set theory, the storage requirements will generally be far less when the data is organized with a STDS and the representation will <u>always</u> be machine relocatable. The latter property is possible by the consistent use of relative pointers and the complete avoidence of absolute pointers for representing information.

For an example of the above, let E be a set of employed persons and let F be a set of fathers. Then, to find the set A of all fathers who are also employed, A would equal the INTERSECTION of E and F. This operation takes place in the information environment. To actually accomplish this retrieval requires assigning modes to both E and F in a machine environment. Let E have, say, MODE(6) which may be assumed to have slow retrieval and small storage properties. Assign, say, MODE(3) to F and assume that it has fast retrieval characteristics but requires large storage. Let IN be a FORTRAN callable subroutine which performs the operation of INTERSECTION. Then "CALL IN(E, F, A)" gives A as the set of fathers who are also employed, and the mode of A will be the default mode. The point to be made here is that if the mode of E were changed from 6 to 3 or if the mode of F were changed from 3 to 6, or if both modes were changed, the resulting set A of IN(E, F, A) would be exactly the same, only the time to execute IN(E, F, A) and the resulting default mode for A might vary.

An important concept in set theory is that of "partitioning." This concept allows STDS to provide the facility to take maximum advantage of any storage medium when handling large data bases. Partitioning is the operation of subdividing a data base into several distinct and independent parts as a function of the information content of the parts.

For example, suppose that one must deal with a data base of, say, two million automobiles where each automobile has, say, 150 recorded data items. This data base might be contained on 20 magnetic tapes and require one and one-half hours per day to update. Suppose further, however, that it is the case that only about 30 of the 150 recorded items account for nearly all of the updating required.

If the data base were organized on tape as an STDS, the first pass over the set of twenty tapes could partition the data base into two parts:

4 tapes containing the partition subject to frequent change,

16 tapes of relatively static data.

With the data so partitioned, the daily update would now deal with only 1/5 of the tapes previously required. Furthermore, no loss of interrogation generality is incurred by this process. If "wild card" changes occur (i.e. changes that are not among the most frequent 30 items), these changes may be used to obtain dynamic partitioning, if desired. Or they may be maintained as a small set of "changed" individuals until it becomes economically or procedurally attractive to merge (in terms of Set Theory, union) them with the master data base.

Other facilities also follow from the basic nature of a STDS. Among the more important features are:

1) Any data base that is in machine readable form requires no redesign for use with STDS.

2) Most data bases in their raw input form (i.e. no structure associated) have enough redundancy and unused portions of data fields to permit a reduction in storage requirements by a factor of about four (based on current experience) when placed into the most compact STDS. Importantly, the data base in maximally compacted form in STDS would still permit total interrogation.

   In one specific case, a data base of 300,000 records of 120 characters/record was reduced to most compact form. Later when expanded for fastest retrieval, the expanded sets occupied only 1/4 of the space originally required by just the raw data (i.e. 9×10^6 characters for fastest retrieval and total interrogation versus $3 \times 10^5 * 120 = 36 \times 10^6$ characters in raw form).

3) Test runs of huge data bases may be expedited by using STDS to extract random samplings (i.e. subsets). These subsets may be used for experiments with partitioning and other schemes to obtain the fastest interrogation of the complete data base. Importantly, all STDS operations are completely compatible in both the subset and total data base cases.

4) Set Theory lends itself naturally to an English language superstructure, Many set operations already have English equivalents. For example:

   "and" → "intersection" "The set of people who are married and unemployed" is the same as "The intersection of the set of married people with the set of unemployed people."

   "or" → "union" "The set of blue or red automobiles" is the same as "The union of the set of blue automobiles with the set of red automobiles."

"but not" → "Relative Complement" "The set of people who are employed but not high school graduates" is the same as "The relative complement of the set of employed people with respect to the set of high school graduates."

"--- or --- but not both" → "Symmetric Difference" "The set of people who are employed or in school, but not both" is the same as "The Symmetric Difference of the set of employed people and the set of people in school."

5) The modularity of set theory may be carried into the design of a STDS. All of the set-theoretic operations may be implemented as subroutines. Each subroutine for either operations or modes (storage-representations) and independent of one another. Further, new operations (or modes) may be added at any time with no disturbance of previously implemented operations, modes or data already in existence.

Finally, the universality of set theory allows for the complete range of inquiry and complete management over any type of data. This fundamental fact together with the constructive demonstration of the existence of a canonical machine-representation for an arbitrary set provides the foundation for a Set-Theoretic Data Structure and its properties. However, the viability of any such implementation will depend on the transformation algorithms, the set operation algorithms, the storage media available, and most importantly the cleverness in the design of the different machine-representations.

D.1 EXPANDED DEFINITION OF STDS FACILITIES

```
#R SS81:STDS*
#EXECUTION BEGINS

 **  SET-THEORETIC DATA STRUCTURE:  INTERACTIVE DEMONSTRATION   **
      (12/6/69)


 FOR AN EXPLANATION ENTER "1": 1

 CONVENTIONS:
      C(A) = CARDINALITY OF A
      L(A) = LENGTH OF LONGEST N-TUPLE IN A
      U(A) = LARGEST DATUM NAME (INTEGER) IN A
      ALL ARGUMENTS MUST BE FROM 1 TO 4 CHARACTERS

 SET OPERATIONS AVAILABLE:
      UN(A,B,C)        IN(A,B,C)        SD(A,B,C)        RL(A,B,C)
      CV(A,C)          DM(I,A,C)        IM(I,A,B,C)      RS(I,A,B,C)
      NF(A,C)          RD(A,B,C)        CARD(A)          XREL(A,B,C)
      XPAN(I,A,B,C)    IGTJ(A,I,J,C)    IEQJ(A,I,J,C)

 NON-SET OPERATIONS AVAILABLE:
      SETH       LIST       FREE       MIN        INDX       SET
      GET        PUT        MTS        DATA       XSUB

 **  QUID(A....Z) GIVES EXPLANATIONS OF OPERATIONS A THROUGH Z.
 **  ATTENTION INTERRUPTS ARE FIELDED BY STDS*.

 ?QUID(UN,IN,SD,RL,CV)

 ####
 #
 #   UN(A,B,C)
 #
 #      C = A UNION B
 #      C IS THE SET OF ELEMENTS THAT ARE EITHER IN
 #           SET A OR SET B.
 #
 ####

 ####
 #
 #   IN(A,B,C)
 #
 #      C = A INTERSECTION B
 #      C IS THE SET OF ELEMENTS COMMON TO BOTH SET A AND SET B.
 #
 ####
```

```
####
#
#   SD(A,B,C)
#
#      C = SYMMETRIC DIFFERENCE OF A AND B
#      C IS THE SET OF ELEMENTS THAT ARE EITHER IN SET A OR IN S
#           BUT NOT IN BOTH.
#
####

####
#
#   RL(A,B,C)
#
#      C = RELATIVE COMPLEMENT OF A WITH B
#      C IS THE SET OF ELEMENTS IN SET A THAT ARE NOT ALSO IN
#            SET B.
#
####

####
#
#   CV(A,C)
#
#      C = CONVERSE OF A
#      C CONTAINS "REVERSED" N-TUPLES OF SET A.
#            IF <W,X,Y,Z> IS IN A, THEN <Z,Y,X,W> IS IN C.
#
####

?QUID(DM,IM,RS,NF)

####
#
#   DM(I,A,C)
#
#      C = I-TH DOMAIN OF A
#
#      C IS THE SET OF ELEMENTS THAT APPEAR IN THE I-TH
#            POSITION OF N-TUPLES IN SET A.
#
####

####
#
#   IM(I,A,B,C)
#
#      C = I-TH IMAGE OF B UNDER A
#
#      C IS THE SET OF ELEMENTS THAT APPEAR IN THE I+1
#      POSITION OF N-TUPLES IN SET A, ONLY IF THE CORRESPONDING
#      FIRST ELEMENT IN THE N-TUPLE IS CONTAINED IN SET B.
#
####
```

```
####
#
#   XREL(A,B,C)
#
#       C = RELATION EXTRACTED FROM A BY B
#
#       A IS A SET OF N-TUPLES, B IS AN INDX SET.
#       C IS THE SET OF N-TUPLES WITH L(C)=C(B) and WITH C(C)
#             EQUAL TO OR LESS THAN C(A), SUCH THAT IF <I,J> IS
#             AN ELEMENT OF B THEN THE I-TH ELEMENTS IN N-TUPLES
#             OF C WERE J-TH ELEMENTS IN N-TUPLES OF A.
#
####


?QUID(XPAN,IGTJ,IEQJ)

####
#
#   XPAN(I,A,B,C)       XPAN(I,A,B,C,V)
#
#       C =A EXPANDED BY B.
#
#       C IS THE SET OF N-TUPLES SUCH THAT:
#             <X(1),..,X(I),A(I+1),..,A(N),B(I+1),..,B(M)> IS IN C
#             IFF  N = L(A),  M = L(B),
#             <X(1),..,X(I),A(I+1),..,A(N)> IS IN A, AND
#             <X(1),..,X(I),B(I+1),..,B(M)> IS IN B.
#
####

####
#
#   IGTJ(A,I,J,C)
#
#       C IS THE SET OF N-TUPLES FROM A HAVING AN I-TH POSITION
#             LOGICALLY GREATER THAN THE J-TH POSITION.
#
####

####
#
#   IEQJ(A,I,J,C)
#
#       C IS THE SET OF N-TUPLES FROM A HAVING IDENTICAL I-TH
#             AND J-TH POSITIONS.
#
####
```

```
?QUID(SETH,LIST,FREE,MIN)

####
#
#   SETH(A)        SETH(A,U(A),L(A),C(A))      SETH(A,B,L(A),C(A))
#
#      A IS THE SET TO BE FORMED.
#      B IS AN INDEX SET SPECIFYING THE LARGEST DATUM NAME
#            (AN INTEGER) FOR EACH POSITION OF THE N-TUPLES OF
#      U(A),L(A),C(A) ARE RESPECTIVELY THE LARGEST DATUM NAME,
#            THE LENGTH OF THE LONGEST N-TUPLE, AND CARDINALITY
#            OF A.
#      WHEN A IS THE ONLY ARGUMENT, THE ELEMENTS ARE TO BE
#            ENTERED INDIVIDUALLY USING A (XIY) FORMAT, WHERE X
#            AND Y ARE INTEGERS.  IF DATA IS IN A FILE USE:
#            DATA(A).
#
####

####
#
#   LIST       LIST(A)       LIST(A,I,J)       LIST(A,I,J,V)
#
#      A IS THE SET TO BE LISTED.
#      I AND J ARE INTEGERS INDICATING THAT THE I-TH THROUGH TH
#            J-TH ELEMENTS OF A ARE TO BE LISTED.
#      WHEN NO ARGUMENTS ARE PRESENT, THE CLASS OF AVAILABLE SE
#            WILL BE LISTED.
#      V=1 ALLOWS FOR VARIABLE PRINT FORMATS.
#      V=-1 ALLOWS SETTING PERMANENT PRINT FORMATS.
#
####

####
#
#   FREE(A,...,Z)
#
#      A,...,Z ARE SETS WHICH ARE TO BE DESTROYED.
#
####

####
#
#   MIN(A,...,Z)       MIN
#
#      A,...,Z ARE SETS WHOSE STORAGE ALLOCATION IS TO BE
#            MINIMIZED.
#      WHEN NO ARGUMENTS ARE PRESENT, ALL SETS WILL BE MINIMIZ
#
####
```

```
?QUID(INDX,SET,GET,PUT,MTS,DATA)

####
#
#   INDX(A)      INDX(A,X(1),...X(C(A)))
#
#      A IS TO BE A SET OF ORDERED PAIRS, WHOSE DOMAIN ELEMENTS
#            ARE 1 THROUGH C(A), AND WHOSE RANGE ELEMENTS ARE TO
#            BE LISTED INDIVIDUALLY.
#      C(A) IS THE CARDINALITY OF SET A.
#      WHENEVER A ZERO IS ENTERED AS A RANGE ELEMENT, THE
#            REMAINDER OF THE SET WILL BE GENERATED RANDOMLY
#            BETWEEN 1 AND U(A).
#   X(1),..,X(C(A)) ARE RANGE ELEMENTS.
#
####

####
#
#   SET(A,B,...,Z)
#
#      A IS THE SET TO BE FORMED.
#      B,...,Z ARE INTEGER ELEMENTS OF THE SET A.
#
####

####
#
#   GET(A)
#
#      THE SPECIFIED FILE IS PUT INTO A.
#
####

####
#
#   PUT(A)
#
#      A IS PUT INTO THE SPECIFIED FILE.
#
####

####
#
#      MTS IS CALLED.
#      RETURN TO STDS* BY ENTERING "$RES".
#
####
```

```
?QUID(DATA,XSUB)

####
#
#   DATA(A,L(A),C(A))       DATA(A,L(A),C(A),V)
#
#      THIS COMMAND ALLOWS READING DATA FROM FILES.
#      V=1,..,10 PICKS A PRESET  INPUT FORMAT.
#      V=-1,..,-10 ALLOWS SETTING INPUT FORMATS.
#
####

####
#
#   XSUB(A,C)       XSUB(A,I,J,C)
#
#      C    IS A SUBSET OF CONSECUTIVE ELEMENTS OF A.
#      WHEN I AND J ARE GIVEN, C CONTAINS THE I-TH THROUGH
#            THE J-TH ELEMENTS OF A, OTHERWISE TWO ELEMENTS
#            ARE ENTERED AND C CONTAINS ALL ELEMENTS IN A
#            BETWEEN THESE ELEMENTS.
#
####
```

BINARY RELATIONS

A = {<c,a>,<c,d>,<a,e>,<a,f>,
<a,k>,<d,k>,<d,b>,<d,h>}

$$\begin{tikzcd}
& & c \arrow[dl] \arrow[dr] & & \\
& a \arrow[dl] \arrow[d] \arrow[dr] & & d \arrow[dl] \arrow[d] \arrow[dr] & \\
e & f & k & b & h
\end{tikzcd}$$

CARD(A) = 8

CONVERSE of A: CV(A,C)

C = {<a,c>,<d,c>,<e,a>,<f,a>,
<k,a>,<k,d>,<b,d>,<h,d>}

$$\begin{tikzcd}
e \arrow[dr] & f \arrow[d] & k \arrow[dl] \arrow[dr] & b \arrow[d] & h \arrow[dl] \\
& a \arrow[dr] & & d \arrow[dl] & \\
& & c & &
\end{tikzcd}$$

1. Find BEGIN and END points.

```
DM(A,B)  =  RG(C,B)      B = {a,c,d}
RG(A,E)  =  DM(C,E)      E = {a,b,d,e,f,h,k}
```

2. Find FIRST and LAST points.

```
RL(B,E,F)                F = {c}
RL(E,B,L)                L = {b,e,f,h,k}
```

3. Given points P, find successor points S.

```
IM(A,P,S)     If P = {c}  then  S  = {a,d}
              If P = {a}  then  S  = {e,f,k}
```

EXECUTION TIMES IN SECONDS ON IBM 360/67

		DOMAIN	CONVERSE	1 IMAGE	10%
#A=(2)	100	0.004	0.041	.0006	.004
(2)	500	0.019	0.302	.0006	.027
(2)	1000	0.035	0.685	.0007	.069
(2)	5000	0.148	5.142	.0013	.769

ITE RINGS

$$\begin{tikzcd}
 & & h \arrow[dr] & & g \arrow[ll] & & & \\
 & & & a \arrow[ur] \arrow[drr] & & & & j \arrow[dd] \\
 & f \arrow[urr] & & & & b \arrow[d] \arrow[r] & c \arrow[ur] & \\
p \arrow[d] & e \arrow[u] \arrow[l] & & & & i \arrow[dll] & & k \arrow[ul] \\
o \arrow[ur] & & & d \arrow[ull] & & & &
\end{tikzcd}$$

A = {<i,d>,<d,e>,<e,f>,
<f,a>,<a,b>,<b,i>,
<a,g>,<g,h>,<h,a>,
<p,o>,<o,e>,<e,p>,
<c,j>,<j,k>,<k,c>,<b,c>}

CARD(A) = 16

Given points B, find all reachable points C.

```
#LIST REACH
          SUBROUTINE REACH(A,B,C,T)
          INTEGER A(1),B(1),C(1),T(1)
          CALL UN(B,B,C)
      100 CALL IM(A,B,T)
          IF(CARD(T).EQ.0) RETURN
          CALL RL(T,C,B)
          IF(CARD(B).EQ.0) RETURN
          CALL UN(C,B,C)
          GO TO 100
          END
```

ACTUAL RUNS

```
CARD(A) = 1000     gave     CARD(C) = 12
CARD(B) = 1                 cpu-sec = 0.0207

CARD(A) = 5000     gave     CARD(C) = 992
CARD(B) = 1                 cpu-sec = 2.48
```

OPERATIONS EXTENDED TO N-TUPLES

$$A = \{ <p,q,r,s,t>, <k,z,m,n,o>, <a,b,c,d,e>, <u,v,w,x,y>, <f,g,h,i,j> \}$$

I-TH DOMAIN of A

DM(1,A,C)	$C = \{a,f,k,p,u\}$
DM(3,A,C)	$C = \{c,h,m,r,w\}$

I-TH RESTRICTION of A to B

Let $B = \{z,s\}$ then for
RS(2,A,B,C)

$C = \{<k,z,m,n,o>\}$

and

RS(4,A,B,C)

$C = \{<p,q,r,s,t>\}$

LET A = {<name,mother,father,spouse,sex>}

GIVEN x ,find all sisters of x.

let $X = \{x\}$ and $W = \{female\}$

RS(1,A,X,B)	B = {n-tuples with x in position 1}
DM(2,B,M)	M = {x's mother, m}
DM(3,B,F)	F = {x's father, f}
RS(2,A,M,C)	C = {n-tuples with m in position 2}
RS(3,A,F,D)	D = {n-tuples with f in position 3}
IN(C,D,G)	G = {intersection of D and C}
RS(5,G,W,H)	H = {n-tuples of G, female in 5 pos.}
RL(DM(1,H),X,S)	S = {sisters of x}

D.3 DESCRIPTION OF THE DEMONSTRATION FILES FOR STDS USED IN APPENDIX D.4

Universe: March 1967 Current Population Survey with Income and Work Experience Supplements Included (Bureau of Labor Statistics). Married, Spouse Present, Head and Wife of Family or Subfamily, Age of Wife 21 or over, Living in 96 of largest 104 Standard Metropolitan Statistical Areas (SMSA).

Item #	Unit	Characteristics	Coding
Family Files			
1.	Family	Family Code Number	1-11629
2.	Family	Primary Family	0
		Each Subfamily or Secondary Family[b]	1-6
3.	Family	Type of Family	
		Primary Family	1
		Subfamily	2
		Secondary Family	3
4.	Family	Weight[a]	0-999999
5.	Family	Presence of Own Children	
		None	0
		All 6-17	1
		None Under 3, Some 3-5, Some 6-17	2
		All 3-5	3
		Some Under 3	4
6.	Family	Relation of Wife to Head of Household	
		Wife	2
		Child	3
		Other Relative	4
		Non-Relative	5,6
7.	SMSA	CPS Unemployment Rate	
		0-9.7%	0-97
		Over 9.7%	98
8.	SMSA	BLS Employment Change, 1966-67	
		Under .1%	0
		0.1-9.7%	1-97
		Over 9.7 %	98

Item #	Unit	Characteristics	Coding
9.	SMSA	Relative Opportunities	
		.001-.997	1-997
		.998 or more	998
Wife Files			
1.	Family	Family Code Number	1-11629
2.	Family	Primary Family	0
		Each Subfamily or Secondary Family[b]	1-6
3.	Family	Race of Wife	
		White	1
		Negro	2
		Other	3
4.	Wife	Age	
		Age at last birthday	14-98
		Age 99 or over	99
5.	Wife	Labor Force Status, March	
		Not in Labor Force	1
		In Labor Force	2
6.	Wife	Employment Status	
		Employed:	
		At work full-time	01
		At work part-time	02
		With a job,not at work	03
		Unemployed:	
		Looking for work	04
		Temporary lay-off	05
		New Job	06
		New Job, School	07
		Out of Labor Force:	
		House	08
		School	09
		Unable	10
		Unpaid,less than 15 hrs.	11
		Other	12
7.	Wife	Recoded-Intermediate Hours	
		1-34 Hours	
		Usually full-time, Economic	1
		Usually full-time, Other	2

Item #	Unit	Characteristic	Coding
		Usually part-time, Economic	3
		Usually part-time, Other	4
		35-39 Hours	5
		40 Hours	6
		41-47 Hours	7
		48+ Hours	8
		Intermediate Duration of Unemployment not coded 1 or 2	9
8.	Wife	Intermediate Duration of Unemployment	
		Under 4 weeks	1
		4 weeks	2
		5-6 weeks	3
		7-10 weeks	4
		11-14 weeks	5
		15-26 weeks	6
		Over 26 weeks	7
		Not unemployed	9
9.	Wife	Years of School Completed	
		None	1
		1-4 elementary	2
		5-7 elementary	3
		8 elementary	4
		1-3 high school	5
		4 high school	6
		1-3 college	7
		4 college	8
		5 or more college	0
10.	Wife	FILOW	
		Negative or none	0
		Amount	1-24999
		$25,000 or over	25000

Husband Files

1.	Family	Family Code Number	1-11629
2.	Family	Primary Family	0
		Each Subfamily or Secondary Family	1-6
3.	Husband	Age	
		Age at last birthday	14-98
		Age 99 or over	99

Item #	Unit	Characteristic	Coding
4.	Husband	Labor Force Status, March	
		Not in Labor Force	1
		In Labor Force	2
		Armed Forces	9
5.	Husband	Employment Status	
		Employed:	
		At work full-time	01
		At work part-time	02
		With a job,not at work	03
		Unemployed:	
		Looking for work	04
		Temporary lay-off	05
		New Job	06
		New Job, School	07
		Out of Labor Force:	
		House	08
		School	09
		Unable	10
		Unpaid,less than 15 hrs.	11
		Other	12
		Armed Forces	99
6.	Husband	Recoded-Intermediate Hours	
		1-34 Hours:	
		Usually full-time, Economic	1
		Usually full-time, Other	2
		Usually part-time, Economic	3
		Usually part-time, Other	4
		35-39 Hours:	5
		40 Hours	6
		41-47 Hours	7
		48+ Hours	8
		Intermediate Duration of Unemployment not coded 1 or 2	9
7.	Husband	Intermediate Duration of Unemployment	
		Under 4 weeks	1
		4 weeks	2
		5-6 weeks	3
		7-10 weeks	4
		11-14 weeks	5
		15-26 weeks	6
		Over 26 weeks	7
		Not unemployed	9

Item #	Unit	Characteristic	Coding
8.	Husband	Years of School Completed	
		None	1
		1-4 elementary	2
		5-7 elementary	3
		8 elementary	4
		1-3 high school	5
		4 high school	6
		1-3 college	7
		4 college	8
		5 or more college	0
9.	Husband	FILOW	
		Negative or none	0
		Amount	1-24999
		$25,000 or over	25000

[a]Weight in file is 100 times the true weight.

[b]Each subfamily or secondary family within a primary family unit has a separate number. Subfamilies and secondary families are contiguous on the file to their respective primary family.

D.4 EXAMPLE USE OF STDS

```
 RUN SS81:STDS*
#EXECUTION BEGINS

 ** SET-THEORETIC DATA STRUCTURE:  INTERACTIVE DEMONSTRATION **
    (2/18/70)

 FOR AN EXPLANATION ENTER "1":
```

(See Appendix D.1)

```
?GET(H)
        FILE = CPSH1
  ENTER PRINT FORMAT: (9I8)
 DONE! L(*)= 9 C(*)= 5812   ( 0.2912 SEC)
```

H is the set of husbands (heads of households) (See Appendix D.3)

```
?GET(W)
        FILE = CPSW1
  ENTER PRINT FORMAT:  (10I7)
 DONE! L(*)= 10 C(*)= 5812   ( 0.3250 SEC)
```

W is the set of wives. (See Appendix D.3)

```
?SET(UNEM,4,6,5,7)
   * DONE!  L(*)=   1  C(*)=   4  ( 0.0036 SEC)
```

UNEM is the set of codes for "UNEMPLOYED"

```
?RS(5,H,UNEM,UH)
 DONE!  L(*)=   9  C(*)=  95   ( 0.4715 SEC)
```

UH is set of unemployed husbands

```
 ?XPAN(2,W,UH,WUH,1)
 DONE!  L(*)=   17  C(*)=  95 (0.5356 SEC)
```

WUH combines (matches) the wives of unemployed husbands in a set of 17-tuples (wife-husband relationships)

```
 ?IGTJ(WUH,4,11,OLDR)
 DONE! L(*)=  17  C(*)=    13( 0.0023 SEC)
```

OLDR is the set of wives of unemployed husbands who are older than their husbands.

```
?RS(6,OLDR,UNEM,UNW)
 DONE!  L(*)=   17   C(*)=   1 ( 0.0023 SEC)
```

UNW is the set of unemployed wives of unemployed husbands who are older than their husbands.

```
?LIST(UNW,1,1,-1)
    ENTER OUTPUT FORMAT:    (5I10)
      2006         0         1        57         2
         5         9         6         3      6520
        53         2         5         9         6
         6      6321
```

By examining this with the codes in Appendix D.3, it may be *seen that we have located a 57 year old wife of a 53 old husband such that the FILOW [family income less own wages] for each is in excess of 6000 dollars. This may be due to pensions, interest, dividends, capital gains or other sources, explaining the basis of this data sample instance. Observe the CPU times used to find this result from an initial set of 5812.*

Total time to do STDS operations = 1.644 sec.

```
?MTS
 #SIGNOFF
 #OFF AT 18:11.26
 #ELAPSED TIME       1400.7       SEC.← due largely to typed comments!
 #CPU TIME USED        10.471     SEC.← due largely to program "loadin
 #STORAGE USED        929.573     PAGE-SEC.        and "relocation"!
 #DRUM READS          107
 #APPROX. COST OF THIS RUN        $2.42
 #FILE STORAGE        841 PG-HR        $.12
```

The above is an actual STDS session retyped for greater readability using italics to distinguish the comments.

UNCLASSIFIED
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) The University of Michigan CONCOMP Project	2a. REPORT SECURITY CLASSIFICATION Unclassified
	2b. GROUP

3. REPORT TITLE
CONCOMP: Research in Conversational Use of Computers

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

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

6. REPORT DATE December 1970	7a. TOTAL NO. OF PAGES 19 + Appendices	7b. NO. OF REFS --
8a. CONTRACT OR GRANT NO. DA-49-083-OSA-3050	9a. ORIGINATOR'S REPORT NUMBER(S) 07449-3-F	
b. PROJECT NO.		
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) None	
d.		

10. DISTRIBUTION STATEMENT
Qualified requesters may obtain copies of this report from DDC.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency

13. ABSTRACT

This report describes the final research results of the CONCOMP Project: Research in the Conversational Use of Computers, which was funded from 1965-1970. This research involved the design, development, and testing of computer programs for graphical input of problem statements and graphical output of results from a computer; the application of the techniques so developed to speech synthesis, systems design research, and research in the logic of computers; the study, design, implementation, and testing of systems for describing graphical operations within the format of procedure-oriented computer programming languages. All of this work was predicated on the availability of IBM 360/67 hardware and software. When TSS was unavailable, CONCOMP undertook two additional tasks: (1) development of the conversational aspects of an operating system for the central computing facilities to support effective man-machine interaction; (2) development of an effective hardware interface for the support of the remote terminal devices.

UNCLASSIFIED
Security Classification

UNCLASSIFIED
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	
interactive computing						
Data Concentrator						
remote terminal devices						
computer graphics						
computer-aided design						
audio-response unit						
cellular automata						
data structures						
graphical languages						
MAD/I language						
extensible languages						
biological simulation						

UNCLASSIFIED
Security Classification