Title
GRAPHICS MODELING TECHNIQUES IN COMPUTER AIDED DESIGN

Author
Holmes, Harvard H.

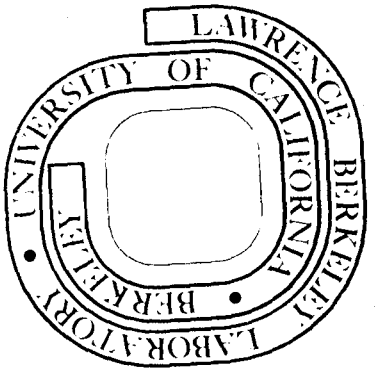
Publication Date
1975-11-01

GRAPHICS MODELING TECHNIQUES IN COMPUTER AIDED DESIGN

Harvard H. Holmes
(Ph. D. thesis)

November 1975

Prepared for the U. S. Energy Research and Development Administration under Contract W-7405-ENG-48



LBL-4240
c.1

DISCLAIMER

TABLE OF CONTENTS

ACKNOWLEDGEMENT

I am indebted to my thesis committee, Professor H. B. Baskin, Professor D. Ferrari and Dr. L. P. Meissner, for their encouragement and guidance, to D. M. Austin for many helpful suggestions and collaboration in the programming of the prototype, and to the users at Lawrence Berkeley Laboratory for their suggestions and motivation, specifically Ivan Wood and Horace Warnock of Electrical Engineering Drafting, and Michiyuki Nakamura, Richard LaPierre, John Mendes, Don Evans and Frank Neu of Electrical Engineering Research and Development. I am also indebted to James Baker and Carl Quong, Math and Computing, LBL, for support and encouragement; and to Virginia Franks for preparing the manuscript.

Finally, this project would never have been completed without the encouragement and support of my wife Susan.

This work was supported by the Energy Research and Development Administration under contract W-7405-eng-48.

GRAPHICS MODELING TECHNIQUES IN COMPUTER AIDED DESIGN

Harvard H. Holmes

Lawrence Berkeley Laboratory
Berkeley, California

ABSTRACT

Schematic diagrams form a natural medium of communication in a wide range of problem areas. In this thesis, we will describe a comprehensive approach to problem solving using schematic diagrams as the interface between man and computer.

Past efforts at computer aided design have been hampered by an approach which combined the man-machine problem-description interface with the problem analysis portion of the system. In this thesis, we set forth a methodology which separates these two aspects of computer aided design. By recognizing those topological properties of schematic diagrams that are common to a wide variety of disciplines, GMS is able to provide a single man-machine problem description interface for use in a wide variety of problem solving disciplines. In addition, GMS includes intermediate data structures and preprocessing facilities that form a natural interface and starting point for the creation

of additional analysis capabilities are described.

The problem-definition interface supports two main activities: the creation of elements, and the interconnection of these elements to form diagrammatic models. These elements are the basic building blocks for creating models. In the past, the elements have been embedded rather deeply in the software. Thus, the creation and description of elements was done by the system designer or, at best, was relegated to a separate phase which required substantial familiarity with the software. The main difficulty in creating a new element was to communicate to the analysis portion of a system the exact meaning of the element. Our approach, on the other hand, makes it easy for a user to describe the meaning (semantics) of a new element in a natural way. If the new element is a primitive (containing no other elements as components), the semantic description is given in analytic form (e.g. a formula), or in empirical form (such as a table of numbers). If the new element is composite (a combination of previously created elements), its semantics are defined implicitly by the semantics of the component elements along with the topology of the interconnections. A convenient representation of the topology of the interconnections is given by a routine which traces lines in a drawing and recognizes nets of joined lines. A translator is described which produces explicit semantics of composite elements from the semantics of the component elements and the topology. A complete prototype

implementation for this part of the system is described.

The problem-solving portion of this methodology allows a wide variety of analysis techniques which can be flexibly combined to solve a particular problem at hand. This approach recognizes that no workable scheme for automatically constructing computer programs has been developed. Nevertheless, several improvements in present techniques for constructing programs can be made to prepare for such schemes. These improvements include flexible data structuring facilities for programming languages and generalized user interfaces. An initial set of data structures and analysis functions for constructing programs is described.

Another aspect of our work is to demonstrate how existing problem solving systems can be extended using the graphics problem description interface. These extensions serve to tailor analysis packages to a spectrum of disciplines where such analysis techniques are appropriate. This gives the user the impression that he is using an analysis routine which has been specifically constructed for his problem.

1. AN INTRODUCTION TO DIAGRAMMATIC MODELING

In many areas of design, symbolic or schematic diagrams are the most widely used representations for the statement of a problem or the representation of an idea. Diagrams have been utilized for a wide variety of disciplines, in which instruction in the discipline is based on diagrammatic representations. Symbolic diagrams have thus become a rather universal means of communication within disciplines. In addition, the diagrams themselves have developed into powerful tools for guiding problem formulation and solution.

Once computer aided design (CAD) techniques and computer graphics techniques had succeeded, it was only natural to combine these two techniques and provide graphical input to computer aided design programs. There are many such programs in existence, but their growth has not been as rapid as the computing world expected. The reasons for this slow growth are many, including primarily an inability to directly use previous work [BASK68], and the high cost of graphics hardware. The inability to use previous work results in a high cost for developing new applications, and this difficulty is addressed in this work. The high cost of graphics hardware has largely disappeared; mini-computer displays are now relatively inexpensive and operating systems are now much better suited to support interactive computing.

The minimum level of software design which allows the

incorporation of previous work is one which uses common subroutines in each successive graphics input CAD program. This level of design has not been very effective, with most subroutines being at the level of common graphics functions. A significant step forward in the organization of graphics software design was provided by Baskin [BASK68] in developing a conceptual model for such software design. Baskin suggested that a graphics CAD facility be organized as five separate subsystems or modules for: (1) creating elements, (2) diagrammatic modeling, (3) analyzing problems represented as diagrammatic models, (4) revising analysis procedures, and (5) providing output. Two other key ideas in this paper were: that semantic description of an element could be done in terms of three definition mechanisms; and that a large class of diagrams could be included in a single generalized topological framework.

Many of these ideas were reflected in contemporary and subsequent work. Software modularization is evident in an experimental CSMP [BREN66], in GINA [MAGN67], in DESIGNPAD [BELA71] and in a graphic version of ECAP [HOGS67]. Efforts to handle diagrammatic models in a general way are evident in the experimental CSMP and in DESIGNPAD. One semantic description mechanism appears in the Simulation and Modeling System [GEAR70], but as an extension to the modeling capabilities, rather than as a part of the element description facility. This thesis undertakes to extend and revise these

ideas and describes an implementation of the graphic input which includes such ideas.

1.1 DIAGRAMMATIC MODELING DEFINED AND ILLUSTRATED

1.1.1 Use of Diagrams for Modeling

A model, in the most general sense, is an abstract representation of reality; a diagrammatic model is one which is presented as a stylized drawing or diagram. Diagrams take many forms according to the customs of the many diverse disciplines which use them. Nevertheless, the conventional tools of their construction, namely pencil and paper, have forced the great majority of these diagrams into a common format. Observing Figure 1, we see that a diagrammatic model is composed of elements, interconnected by lines, together with alphanumeric annotation. To the designer, the elements are the building blocks of his model. The lines show the relationships between elements of the model. Usually, only the topology of the relationship is important, since other representations with the same topology would be considered equivalent. Almost always, a single element is used for a large class of similar objects; when the designer uses an element, he identifies the appropriate member of the class using some annotation. Annotation is also used for commentary and for identification.

The choice of elements is often dictated by the

F I G 1

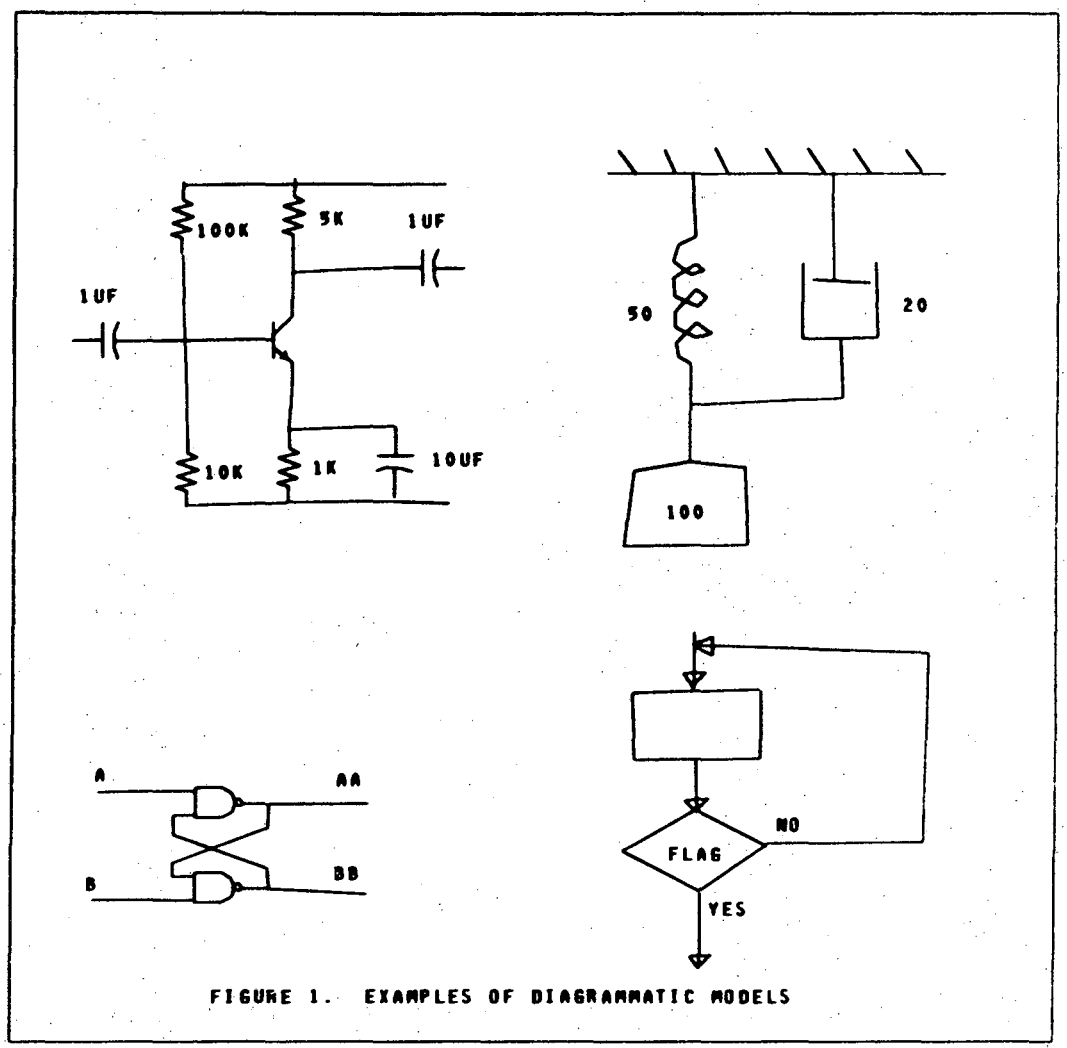


FIGURE 1. EXAMPLES OF DIAGRAMMATIC MODELS

conventions of the particular discipline involved. These conventions have been established by long experience and they are usually very effective. The convention usually specifies both the visual symbol and the class of objects which it represents (i.e., the meaning or semantics associated with the visual symbol). In electrical engineering, these choices have become so standardized that templates of the various shapes are available as drawing aides, and documentation specifications may even establish the exact size to be used for the symbols. Semantic conventions are equally complex and well developed, although they may change with the application. Thus, to take a very specialized example, the symbol in Figure 2A may at times represent a physical resistor with its associated inductance and capacitance. At other times, it may represent a pure resistance so that a physical resistor must have its inductance and capacitance shown explicitly, as in the composite element shown in Figure 2B.

Note that the components of the composite element have separate meanings (semantics) of their own, which have a role in establishing the semantics of the larger element. This is only one case of the more general existence of hierarchies of diagrammatic models, in which the detail of the model is adjusted to fit the necessities of the application.

The primary virtue of having hierarchies of models is to enable an easy comprehension of the model by using several layers of abstractions with each layer based on the next lower

F I G 2

FIGURE 2A. SIMPLE SYMBOL FOR A RESISTOR AS A PRIMITIVE ELEMENT

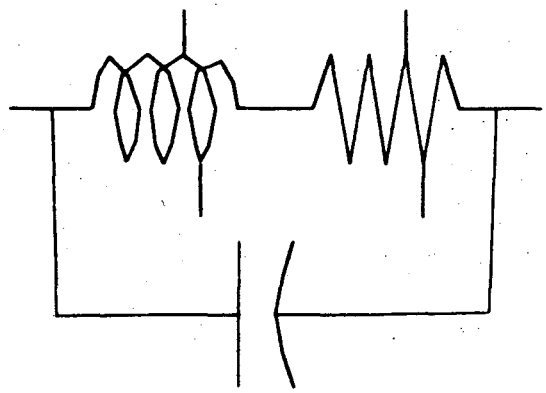


FIGURE 2B. MORE COMPLICATED MODEL FOR A RESISTOR AS A COMPOSITE ELEMENT

level. This facilitly uses man's language and abstract reasoning skills to break a complex problem into understandable subunits and understandable compositions of these subunits. We will return to this line of observation later (Section 1.2.1, 1.2.2). For the present, we simply wish to note that these layers can be observed in any particular discipline as a fundamental tool for problem solving.

The use of diagrams to show relationships among elements of a model brings with it several advantages and disadvantages. The primary advantage is that the irrelevant relationships may be removed from the diagram leaving the designer free to concentrate on those deemed relevant. To use an electronic circuit as our example again, the designer can concentrate on signal flow and not worry about the actual three dimensional packaging of components in the model. A secondary advantage is that a three dimensional situation can often be easily reduced to two dimensions. Of course, the abstraction inherent in modeling and the use of pencil and paper have provided mutual reinforcement of this capability. Finally, we can observe that even a physical situation that is basically two dimensional in its spatial configuration can often benefit from using abstract diagrams rather than spatial configuration to convey relationships. The field of optics provides examples where the symmetry of lenses allows a convenient two dimensional representation, yet a more schematic approach has often been taken.

The primary disadvantage of using diagrams to show relationships is the converse of the advantage: they may omit some relationships which are significant. This may result in the design of a model which cannot be physically realized. While these tradeoffs are inevitable, the situation is not hopeless - the usual remedy is to incorporate more relationships explicitly in the model. In an electronic circuit diagram, for example, some of the circuit delays may be incorporated explicitly by replacing a line by a modeling element which represents time delay. The amount of delay would be set by the designer, who can estimate it from packaging considerations.

1.1.2 Earlier Modeling Programs

In early diagrammatic modeling systems, the semantics of primitive elements were quite rigidly built into the software and the propogation of semantics from elements to combinations of elements was, in most cases, straightforward.

Historically, the first graphical modeling program in the sense of this thesis is SKETCHPAD [SUTH63]. It combines an interactive drawing routine with a scheme for evaluating and satisfying constraints on the drawing. It manipulates lines, constraints, and subpictures (which may include other subpictures). A versatile copy function allows combinations of constraints to be copied from one instance of a subpicture to another. These constraints, the semantics of SKETCHPAD,

are created and incorporated into the data structure via the same graphic techniques which are used to create the original drawing. We may characterize SKETCHPAD by noting that a single data structure and a single analysis routine are used, with the semantics given implicitly by the analysis routine.

Following SKETCHPAD, there appear programs designed to support more interesting analysis routines. Examples of such programs are CIRCAL [DERT67], CADIC [PRES67] and CADD [DERT65]. These programs may also be characterized as having a single data structure used by both the drawing phase and the analysis phase.

Baskin and Morse [BASK68] created an experimental CSMP in which the graphic input functions and the analysis functions were separated and implemented as separate modules. It is noteworthy that an existing interactive drawing program (DIM [RIEK67]) was used. This demonstrated rather clearly that graphic input facilities did not have to be tailor-made for each particular application. DIM allowed the user to create subpictures from lines and other subpictures in a very general fashion. DIM also provided rotation and scaling of subpictures. For use in the modeling system, DIM was given a basic set of entities (predefined primitive elements) which corresponded to functions available in the analysis package. The user created models using lines and these entities. A second overlay generated a topological description of the model in the form required by the analysis package. A third

overlay, again based on an existing program, did the analysis and produced the output.

An extension of this work, DESIGNPAD [BELA71], concentrated more thoroughly on the interactive drawing package and the topological analysis routines, elaborating upon the intermediate data structure for use by analysis programs.

2250 ECAP [HOGS67] and GINA [MAGN67] are examples of graphic drawing packages which are used interactively to produce input for "standard" batch mode circuit analysis programs. Each of these has a predefined menu of circuit elements, from which a circuit diagram is constructed. GINA is able to change its output format to suit any of several circuit analysis packages. J. L. Franklin and E. B. Dean [FRAN73] describe a system similar to GINA, but with the added ability to nest diagrams to form a hierarchy of models.

BIOMOD [GRON71] is a system designed for biological problems; it uses CSMP for its analysis phase. In BIOMOD, all symbols are shown visually as rectangles (a disadvantage), but a more complete set of semantic description facilities is provided, including CSMP primitive elements, algebraic, differential and chemical equations, and Fortran statements. Symbols may also be combined to form hierarchies (implemented by means of macros). BIOMOD offers good facilities for semantic description (although limited to the analysis

facilities of CSMP), but no facilities for user-drawn primitive symbols.

The principal shortcoming of all of these earlier programs is their dependence on a single analysis program, or a single class of analysis programs. Thus, the semantics of their primitive elements are built into the system and the semantics of composite elements can be generated in a uniform manner using the topological description. Although this allows them to sidestep the difficult question of semantic description, it precludes the generalization of these systems to allow multiple forms of semantic description that might be required for various fields of application. Most of the earlier systems also exploit the specialized nature of their application to simplify the graphics programming. The ultimate cost of this specialization is a lack of transferrability of the software.

1.2 A CONCEPTUAL FRAMEWORK FOR INTERACTIVE CAD

This section describes the Graphics Modeling System (GMS) in terms of the facilities which are necessary for very general problem formulation and solution. Many of the parts of GMS can be viewed as extensions of similar facilities in these earlier programs, generalized and made more systematic to serve a wider variety of disciplines. The goal is to make GMS so general that it can be modified online to suit whatever analysis the user wants to make. In addition, the

organization of GMS should be straightforward so that even an inexperienced user can adapt GMS to his problem.

1.2.1 The Art of Design

The art of design is the successful combination of synthesis and evaluation. Design proceeds by a process of synthesis or proposal followed by an evaluation and refinement of the proposal. Refinement is merely a variation of synthesis, and so the designer repeats these steps over and over. First, a synthesis and then a evaluation; then a return to the synthesis phase to improve the solution using the results of the evaluation to guide the synthesis. In the early stages of design, these two activities are carried on entirely within the imagination of the designer. They become a series of "thought experiments." The designer mentally proposes some situation; then he explores it to see if it will provide the desired result.

At some point, the designer is ready for a more concrete evaluation; he is ready to test his ideas using a CAD system. For this purpose, the CAD system must be fast and flexible (at the cost of some accuracy, perhaps). It must respond quickly to allow many ideas to be tested and it must respond to each idea fast enough to promote a steady flow of invention. If he must wait too long, the designer becomes bored or impatient, his attention wanders, and the atmosphere of creativity is lost. At this early stage, the designer is likely to shift

his attention from one aspect of the design to another; thus the CAD system must be flexible in order to accommodate these several aspects of the problem and to allow easy modification of the design.

An essential part of the art of design is the ability to partition the problem. The designer breaks one large problem into a number of smaller problems. This subdivision is evident in practically any problem solving situation. It is a key part of design; a proposal is just a synthesis of subparts. One central activity in design is the sub-division of the problem into satisfactory (more tractable) subparts. A most important counterpart of this ability to subdivide the problem is the designer's repertory of known solutions. The designer with a wide knowledge and understanding of existing solutions is able to use this knowledge to greatly extend his effectiveness. The art of design then becomes a search through a tree structure. At each stage, a branch is subdivided and each of the subbranches are explored. In most cases the designer can immediately evaluate the subparts of the design from his prior knowledge.

In one common design methodology, this subdivision and evaluation takes place in "depth first" manner; that is, as a task is subdivided, one of the subtasks is selected and pursued to its conclusion before effort returns to any of the other highest-level subtasks. This is a very effective way of pursuing the bottleneck in a design, that difficult aspect

which separates it from known solutions. At each subdivision, most of the subtasks are recognized as having known solutions; they can thus be quickly discarded from the part of the problem that needs further consideration. In this way irrelevant detail is avoided while pursuing the heart of the problem to any necessary level of detail. This approach makes good use of man's ability to organize and recognize patterns, while, at the same time, it minimizes the amount of detail which must be managed since man is limited in this regard.

As the design becomes established, there is a need for a more exact evaluation and refinement of it. At this point, the designer turns to mechanical and technical aids. First the design is committed to paper; then the designer begins to refine and evaluate his proposal with the aid of rules of thumb, technical formulas, and the use of a computer. In the past, the use of a computer, allhough desirable or necessary, was delayed until the last possible moment. This was an unhappy result of the great effort required to obtain a computer evaluation of the design. The designer did not want to make this effort until he was reasonably sure that the design was nearly correct. The use of the computer, although necessary, was so costly that alternate solutions could not be pursued unless the original was unsatisfactory. This led to barely adequate designs and stifled the search for innovative, superior designs.

1.2.2 Goals of CAD Systems

The goal of CAD systems should be to make the computer so easy to use that it becomes an aid to evaluation and refinement from the very beginning of a design. This requires a CAD system which is easy to communicate with and which suppports subdivision and evaluation strategies that are natural and convenient for the designer. The technique of diagrammatic modeling can be an important part of CAD systems because it contains these necessary facilities. Ease of communication is inherent in the use of schematic diagrams, since these are the first choice for the communication of ideas in many disciplines. The other features of a diagrammatic modeling technique must allow for experiment, design, evaluation and refinement at a variety of levels of detail. The incorporation of a hierarchical structure into the elements of a diagrammatic model is the best technique for supporting the experimental partitioning and synthesis of the subparts of a design. The designer must be free to construct his model as his attention directs and to return to and extend sections of the model with progressively more and more detail. A hierarchical structure, together with the ability to redefine elements or to have alternate definitions for elements, allows the designer to establish a schematic diagram with only very rough notations of what each element should do. Thus, he can begin with a very simple definition for some elements, and then at a later time he can return and redefine

them in terms of simpler subelements. This allows the designer to concentrate on crucial problem areas and pursue them to the necessary level of detail, while temporarily ignoring the rest of the problem. Later, other parts of the problem can be elaborated upon without disturbing earlier parts of the design.

In addition, a hierarchical organization allows one aspect of a problem to be removed from the rest of the problem and tested out of context. Thus, the designer can perform his "thought experiments" until a particular aspect reaches a level of detail which he needs help in evaluating. He can describe this one part of the system and test it, using his mental design to guide him in describing the environment of this part. For example, in the design of an FM radio, for which a new detector is proposed, the designer divides the radio into an RF section, an IF section, a detector and an amplifier. He immediately dismisses all but the detector as being known. A detector is then proposed and described to the CAD system; from experience the designer can describe the input to the detector and recognize good or bad output. Thus he does not even need to include the other parts of the radio in the description he gives to the CAD system. Alternatively he can include them without describing their internal structure in detail.

If the designer has been using a CAD system for some time, then some of his prior experience can be incorporated

into a library of previously designed parts. A diagrammatic modeling facilitly can lend itself to easy management of this library. As earlier designs are completed, they are stored and their individual parts are reused in later designs.

1.2.3 Need for Separation of Tasks

Consideration of the proposal and evaluation aspects of design suggest that for a CAD system, the proposal aspect can be best supported by a good communications medium, while evaluation requires the services of more conventional mathematical programs. Since use of the system in an interactive descriptive mode alternates with mathematical evaluations, it appears feasible to provide separate software in the CAD system for these two functions. The critical requirement is that the two functions not be required simultaneously. In almost all cases, there is no need for mathematical evaluations during the interactive description phase and vice versa. Thus, these two functions can be organized as two separate software systems with communication through a common data base.

The principal advantage of this separation of tasks is that a wide variety of analysis systems may share the same graphics facilities for problem definition. This will lead to a better graphics product, since the graphics facilities need to be designed and implemented only once. Thus, we can justify doing a more complete job than we might otherwise have

done and we can include features which we might omit if they were to be used with only a single analysis procedure. Equally important is greater convenience for the user, due to the fact that he can use the graphics facility as a common interface. He is no longer required to learn a variety of languages in which to state his problem, and yet he can use a wide variety of analysis procedures. Since the same graphics facility is used with a variety of analysis procedures, it is also a simple matter to use the same problem description with more than one analysis procedure. That is, a particular problem needs to be described only once to be available as input to several analysis procedures.

In addition, the use of separate subsystems makes the use of intelligent terminals for the graphics subsystem a very attractive possibility. This reduces the real time load on the central facility and simultaneously provides even better response for the most common interactions. The use of an intelligent terminal may also allow the use of a low-bandwidth connection between the central facility and a remote user. Without the use of an intelligent terminal, a high bandwidth connection to the central facility would be required, which might be uneconomical or impossible.

But the greatest benefit by far accrues to the analysis programs which can use this common graphics facility. An analysis program that is not used with a separate graphics facility must either include a graphics facility within the

analysis program or use some external form of problem description, such as card images. Incorporation of a graphics facility into analysis systems introduces problems of real time interaction and consequent hardware and operating system dependencies.

Use of the data structures of a common data base, if they are well thought out and carefully designed, is preferable to many of the schemes that have been widely employed to encode schematic diagrams in a batch processing environment. Those schemes use data structures based on card images in which code numbers or similar techniques convey the topological information. They are tedious to use and very prone to error, as well as being very difficult to update to conform to changes in the original schematic diagram. By using a translator, as described subsequently, one can adapt the "standard" interface data structure to a wide variety of card image-based schemes (if necessary), allowing batch programs to be used with the graphics facility.

1.2.4 The Graphics Facility

The function of the graphics facility is to allow CAD problems to be conveniently described as schematic diagrams. The methodology of this description was suggested by Baskin [BASK68], who proposed three types of element descriptions. Two types of semantic description mechanisms are provided for primitive elements: an analytic or relational expression (e.g.

a formula) and an empirical or explicit relation such as a table of numbers. The semantics of composite elements are given by the interconnection (topology) of component elements (and the semantics of the component elements).

To support this methodology, the following software components are required: (1) an interactive graphics editor, (2) an interactive text editor, (3) a topological analysis module, (4) data management for primitive elements with empirical descriptions, (5) a translator to propagate the semantics of primitive elements up through the hierarchy and to provide customized data structures for output, and (6) filing and retrieval functions.

1.2.5 The Analysis Facility

The analysis subsystem uses the problem description and produces answers for the user. Although ideally the analysis subsystem should include a facility for constructing new analysis procedures as simply as new modeling elements can be constructed, the state of the art in automatic programming has not yet reached this ability. Thus, the analysis facility will include a set of preassembled packages, together with a large set of more elementary routines to use as building blocks for new analysis packages. These building blocks will include mathematical routines, data output and display routines, and user interface routines. The system should provide an easy-to-use, interactive facility to guide the user

in assembling these routines to do what he wants. One approach to this problem is evident in the Dynamic Modeling System [PROJ72] in which subroutines from several different languages can be combined in a single system. Perhaps the user can use the graphics interface to describe programs. The facilities needed here for program assembly must be on a higher level of abstraction than that provided by GRAIL [ELLI69], with most of the tedious details of programming and data structure being taken care of automatically for the user (who is not a programmer).

Two examples may illustrate some of the variety of analysis techniques which can be applied. Many problems fall naturally into one of two categories with respect to the computational procedure required to evaluate them: sequential problems and "gestalt" problems. Sequental problems are characterized by an inherent modularity: that is, the model is composed of elements with a definite input-output relationship; computation performed on the inputs for each element yields outputs which serve as inputs to another element. Such a configuration can be recognized in analog computers, digital logic, compartmental flow models, queueing situations, pert charts and flowcharts. Analysis routines for these problems must organize the sequence of modules for evaluation.

It often happens that the analysis routine for sequential problems must deal with the problem of closed loops caused by

feedback. Numerical integration problems, for example, are often solved by reduction to integral equation form, the graphical equivalent of which is to break the loop by assuming that the outputs of the numerical integration are known in advance. Then using these known values, new inputs to the integration routines are computed. During execution, the analysis routine must ensure that the old values are adequate approximations to the new values. For some other problems, for example, digital logic, closed loops may be broken by the inclusion of time delay in the loop.

Once the proper sequence for the modules has been found, they may be evaluated interpretatively (CSMP), by compilation and execution MIMIC [CONT68], or by translation to another language, DSL/360 [SYN68], where Fortran is the target language. The evaluation routines must include the necessary control routines, and usually also provide utility routines, such as those for integration, time-delay and the trigonometric functions.

Gestalt problems, on the other hand, lack the modularity exhibited by sequential problems. Some electronic circuits and bridge and building structures are examples of such problems. Efficient analysis of the behavior of such a structure as a whole cannot be accomplished by the sequential evaluation of computation procedures associated with the component elements. The analysis of such a problem requires that the description of the elements be reorganized into a set

of relationships (e.g. equations), which describe the behavior of the system as a whole. These equations can then be evaluated, often by the same techniques (e.g. numerical integration) used for sequential models. We will return to this examination of analysis routines and packages in Chapter 4.

1.3 AN IDEALIZED MODELING SYSTEM

1.3.1 Defining a Primitive Element

We shall begin our explanation of GMS with the creation of a new primitive element, one which is not composed of other elements. Although a library of elements is maintained, it often happens that a new element is required for a particular problem. A new element is created by giving it a name. A semantic description of the element must then be given which can be understood by the intended analysis routine. For primitive elements, two mechanisms for semantic descriptions are provided: (1) the analytic or relational description of an element by means of text, and (2) the empirical description of an element by means of a numeric or empirical relationship between variables described by some data set.

The analytic description is created using the text editor. It is often used to specify an analytic relationship between variables using a mathematical notation suitable for the desired analysis routine. For example, a resistor may be defined by I = (V1 - V2)/R; or a NAND gate: D = NOT (A + B +

C). For some applications, a suitable description can be obtained from past experience, either with the modeling system or with similar analytic procedures. For other applications, experimentation with alternate descriptions may be a significant modeling activity. An analytic description may also give a functional relationship implicitly rather than explicitly, by referring to a procedure supplied by the analysis routine:

$$x = f(y,z)$$

or

R101 3-7 1100

where the procedure itself is specified by the first character "R".

The empirical description form is used to associate a data set with a primitive element. For example, a transistor can often be best described by some of its characteristic curves. Simple data sets can be generated or edited using the graphic facilities of GMS.

A symbol (the pictorial or visual representation of the element) may also be associated with the name. When an element is used as a component of a larger element (model) a symbol is required for use in the specification of interconnections among the components of the model. The user creates the symbol with the line drawing commands of the graphics editor (Figure 3A illustrates resistor, capacitor and

F I G 3

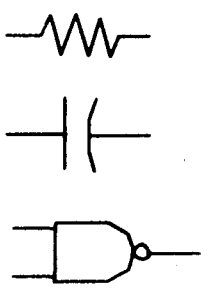


FIGURE 3A. PARTIALLY COMPLETED SYMBOLS

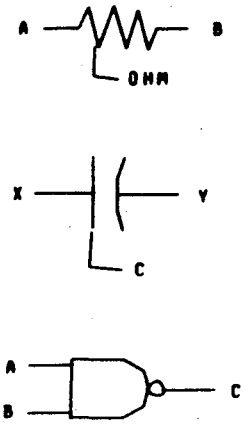


FIGURE 3B. SYMBOLS WITH ATTACHER POINTS ADDED

NAND gate symbols), and he specifies points on the symbol which are available for connection to other symbols in a model. These points are called attacher points, since they specify where lines in a model can be placed to attach the symbol to other symbols. Each attacher point consists of this position together with a character string which identifies the variable (in the analytic or empirical description) which is to be associated with this position. Each attacher point of a symbol has this unique character string and a unique coordinate location relative to the symbol. Figure 3B illustrates resistor, capacitor and NAND gate symbols with attacher points added.

1.3.2 Creating a Composite Element

A composite element is defined in terms of other elements, allowing users to create hierarchies, to assemble collections of elements and use these collections as single new elements. They can build layer upon layer of composite elements, with each layer an abstraction of the layers below. On the other hand, a large problem can be broken into smaller and smaller subunits, with the arrangement of subunits being easily comprehensible at every stage. Thus, within this framework, the user can work from the top down or from the bottom up. In working from the bottom up, new elements are created from combinations of existing ones, creating larger and larger buildlng blocks for larger and larger problems.

In working from the top down, elements already defined may be refined by replacing tentative or experimental definitions by more exact definitions composed of other elements. By analogy to programming languages, the creation methodology for composite elements is called macro definition.

We use the term model as a convenient shorthand for a composite element. Thus "model" refers to the combination and "element" to the components. Strictly speaking, of course, a model is just another element and can be used as a component of a larger model.

After giving a name to the composite element, the next step is to select the elements which are to be used in it. These elements are represented by their symbols. These symbols must then be connected by lines joining the appropriate attacher points. A set of connected lines is called a net. A single object or two objects juxtaposed (without lines) may also be considered nets. A net will usually join attacher points from different symbols, but two or more attacher points of the same symbol can be joined together if desired. Finally, labels are added to the model to identify particular nets. (These labeled nets are associated with corresponding attacher points in the symbol (discussed next) of the composite element. If no symbol has been given or a label does not match an attacher point on the symbol, then the label is assumed to denote a constant or specific name for the net, and the character string is made

available to subsequent analysis routines.)

A composite element may also be given a symbol to allow it to be used as a component in larger models. This symbol has its own attacher points that will be used to connect it into larger models. Each attacher point is identified by its associated character string. The net (in the model description) which corresponds to a particular attacher point (in the model symbol) is identified by a label with a character string which matches the character string of the attacher point. Values entering a symbol as formal parameters through an attacher point propagate down to the matching net and become actual parameters to the sub-symbols.

This character string matching is required because the symbol (picture) of the model (composite element) is not inherently identified with the collection of pictures of its component elements and their interconnections, that is, with the graphical object generated in the description of the model. The attacher points have identifying character strings in the symbol of the model. These are identified with nets in the composite model description by giving matching labels to the nets.

In many respects, the propagation of semantics in GMS is parallel to the propagation of semantics in conventional programming syntax. By analogy with the use of procedures, the semantic definition of an element forms the body of the

procedure and the symbol forms the procedure declaration. The symbol establishes both a pictorial name for the procedure and the formal parameters. As with conventional procedure declarations, the symbol provides the correspondence between external and internal references to the parameters. External references are made according to the order of parameters, or in the case of symbols, by coordinate location. Internal references are made to the corresponding character strings. Procedure declarations give an implicit correspondence between the external order of the parameters and their internal character string referents. For symbols, the attacher points provide a correspondence between coordinate locations with respect to the symbols (for external reference) and character string identifiers (for internal reference). Thus, in a conventional syntax for calling procedures the actual and formal parameters must agree in order (and number), while in GMS they must agree in coordinate location.

1.3.3 Graphics Features

Thefundamental graphic operations are the creation of lines and alphanumeric annotation, the selection and positioning of symbols, and the erasure of these elements. Lines are used for the outline of symbols and for creating nets in composite elements. Lines can be created in a point-to-point fashion (most suitable for nets) or as multiple segments approximating a free hand curve. Alphanumeric

annotation is used for attacher points and for labels. It is constructed and positioned with keyboard and lightpen. A number of character sizes are provided. Symbols are used in composite elements, and each instance of a symbol has a size and an orientation.

A number of graphics features are provided as aids to drawing. Among the most important features are zooming, clutter suppression, grids, and a subpicture facility. The zoom facility allows the user to enlarge any portion of the screen to any desired size. Typically, symbols are enlarged by a factor of 16 to 64 for greater ease in drawing (where a magnification factor of one displays the entire drawing area on the screen). Models are then created at a somewhat reduced size to allow more symbols to be visible at any one time. Clutter suppression operates in conjunction with the zoom. Its effect is to suppress the display of character strings from the screen when they would be too small to be legible or they would overlay one another in a cluttered manner if made large enough for legibility. When a symbol appears in a model (at a smaller size than it was created at) the character strings associated with its attacher points are usually suppressed. A grid can be overlayed on the drawing to help the user draw his lines in exactly the right orientation, or to help him make one line twice as long as another, et cetera. The grid can also be calibrated to provide dimensional accuracy on the finished drawing. (An implementation is

currently in use as a drafting aid to produce printed circuit boards.) The subpicture facility allows a part of a symbol to be drawn separately and then used in several places. For example, computer logic diagrams often use small circles to denote inversion at the input or output of elements. These circles can easily be drawn as subpictures and then called as needed. Subpictures can also be rotated or scaled as desired.

In addition to the graphics and text editors, it is also necessary to have file management operations which allow the storage, retrieval and manipulation of sets of elements (libraries). These operations also allow the deletion and replacement of individual elements within libraries.

1.3.4 Topological Analysis of Composite Elements

A topological data structure is created by scanning the graphics representations of macro definitions of composite elements. Working from coordinate information, the topological analysis module decides which lines are connected, then traces out the nets, and provides a structure in which the relevant topological connections are displayed. From this structure, other modules (Section 1.3.5) can determine either the connectivity of each attacher point associated with an element, or the set of attacher points and labels associated with each net. That is, either the elements or the nets can be the starting point for information retrieval by other

modules.

1.3.5 The Role of the Translator

The primary purpose of the translator is to provide for semantic propagation as described earlier (Section 1.3.2) and to reformat the results into the various forms required by analysis routines. A common form is text on card images similar to conventional programming syntax. For this form, the translator allows the user to specify a subroutine or macro syntax (for example) for the output. Starting with a model to be analyzed the translator creates the appropriate initialization and then compiles each instance of a component element into the specified call syntax with the actual parameters taken from the labels (names) of the nets in the model. If no name was given to a net, then one is created. For each distinct element used in the model, a subroutine for that element is constructed. A subroutine declaration is created using the attacher points to identify the formal parameters. If the element has an analytic definition, it is simply copied after the subroutine declaration. If an empirical definition is given, a user-specified statement is constructed. If a macro definition is used, calls are compiled for the component elements as in the highest level model.

When a model is to be analyzed, the model and the analysis module are identified to GMS, which provides the

proper translation and then passes control to the designated analysis module. A simple extension to the selection command allows several models to be selected as a group, if the user finds that more convenient than incorporating them in a single higher level model.

Since GMS treats all models similarly until one is selected for analysis, there is no difficulty in selecting parts of a larger problem for analysis or in maintaining several variant approaches to a particular problem. The only difference to GMS is that a model which is selected for analysis need not have a symbol, since a symbol is needed only for use at a higher level. GMS can accept any model as the top level model for analysis, and conversely, such a model can have a symbol drawn for it and then it can be used in the same way as any other element. This uniform treatment preserves the open-endedness which is necessary for flexibility and continued expansion of the system.

The secret of success of GMS lies in the fact that the propagation of semantics from components to composite model is independent of the nature of element semantics. Thus although the element semantics must be tailored for an assumed analysis program, nevertheless a common translator can propagate semantics for any analysis program.

The necessary information for element semantics consists of formulas or element descriptions in the form of text or of

numeric data also input as text or as a curve (graphically). The information needed to propagate the semantics consists of the interconnections. At a given hierarchical level, these are just lines (nets) that join symbols (at attacher points). Between levels there are labels that relate internal nets to external attacher points. GMS translates the graphics (line-and-label) topology description into a macro and subroutine parameter form that can be adapted to all analysis routines.

1.4 EXAMPLES OF MODELING

These examples illustrate just a few of the many problems which can be conveniently represented by a graphics diagram. They were created using PICASSO [HOLM72], which is our prototype realization of a GMS.

1.4.1 PERT Diagram Example

A PERT (Program Evaluation and Review Technique) diagram is a representation of job scheduling designed to identify the minimum time to complete a job and to isolate the "critical path", the set of job steps which, if delayed, would delay the whole project. In addition, the PERT diagram may identify, for each step not on the "critical path", the amount of scheduling leeway that can be permitted without delaying the entire job. Figure 4 illustrates a simple PERT diagram. In this figure, each circle represents a job step and the length

FIG 4

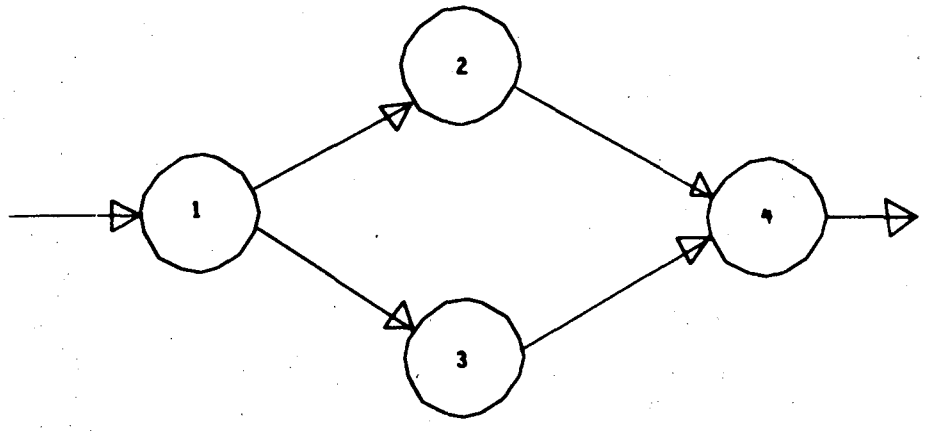


FIGURE 4. A SIMPLE PERT DIAGRAM

of the job step is indicated by the number within the circle. The job step at the tail of an arrow must be completed before the one at the head is begun. With these conventions in mind, we would like to compute the time required to complete the job. We will use MIMIC for the analysis. Each line connecting job steps is associated with the time it takes to complete the job steps ahead of it. In this example we assume that each job step has at most 3 immediate predecessors. (This is no restriction.) Thus, for a single job step with three job steps directly ahead of it, the time of completion T is given by

$$T = \max(A, B, C) + D$$

where A, B, C, are the completion times of the jobs directly ahead of this one and D is the time for this job step. This formula is the semantic description of JOBSTEP and can be understood by the analysis routine. Now note that we can use this same formula for job steps preceeded by less than three jobs if the missing completion times are set to zero. Since PICASSO's translator can assign default values (Section 2.6.4) to unused attacher points we can use one element for all job steps, with unused inputs having a default value of zero.

We can now apply these ideas at the console, beginning by creating the "JOBSTEP" element. First we select the command for a new analytic description. At this point we may either choose an existing element (for modification) or create a new

element. We do the latter and type in the new name, JOBSTEP. The semantic description is then entered by typing the formula

$$T = MAX(A, B, C) + D$$

in the appropriate columns of a card image.

When the semantic description is complete, we select the command to draw a symbol for an element. We designate the name of our element, JOBSTEP. After we set the zoom factor (Section 1.3.3) and ask for a grid, we draw the circle and the lines shown in Figure 5B. The attacher points (A,B,C,D and T) are then created, by first typing in each character string and then positioning it with the lightpen. Note that some attacher points include default values, and that attacher points are positioned at graphics features (e.g. endpoints of lines) which can be used to locate them if the accompanying character strings are suppressed to prevent clutter (Section 1.3.3).

Now we have the primitive element we need to create a model. We select the command for a new macro description and supply a name for our model. We ask for the JOBSTEP symbol by typing its name and, working from a rough sketch, we position it with the light pen. Eight JOBSTEP symbols are required in this case. The length of each job step is specified by a label at the attacher point D (inside the circle). Labels are created by typing the associated character strings and positioning them with the light pen. The labels inside the

F I G 5

1 T MAX(A,B,C)+D
2

FIGURE 5A. ANALYTIC DEFINITION FOR JOBSTEP ELEMENT

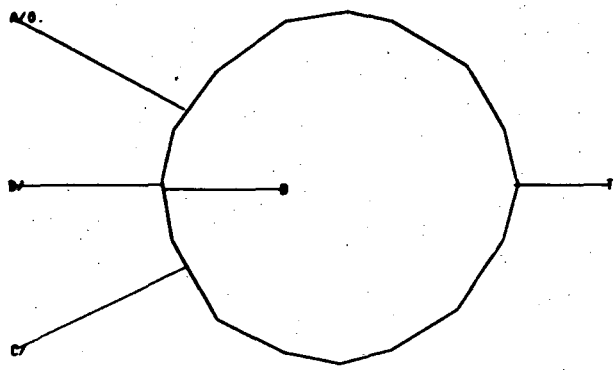


FIGURE 5B. SYMBOL FOR JOBSTEP ELEMENT

circles will be interpreted by the analysis routine (MIMIC) as constants. The job steps are then connected together by lines.

We now realize that the output (the cumulative completion time) of the final jobstep element must be printed in order to have a useful analysis. We suspend work on our model and create another primitive element referring to one of MIMIC's built in functions for printing results. Armed with this output element, we return to our model, add the output symbol to the model and connect it up (Figure 6).

Another command selects the model for analysis, and provides the translator with specifications needed for the propagation of semantics. These specifications also include job control language (JCL) which executes the analysis program and returns control to GMS. The text editor subsystem of GMS is used to examine the output (Figure 7).

1.4.2 Circuit Diagram Example

For this example, we shall assume a more knowledgeable user than previously. In particular, we shall assume that he is primarily interested in using GMS to translate a drawing into card images for input to SPICE [NAGE73], a circuit analysis program. His problem is simplified because SPICE determines his text definitions and electrical engineering conventions determining his symbols. He creates names,

PERT

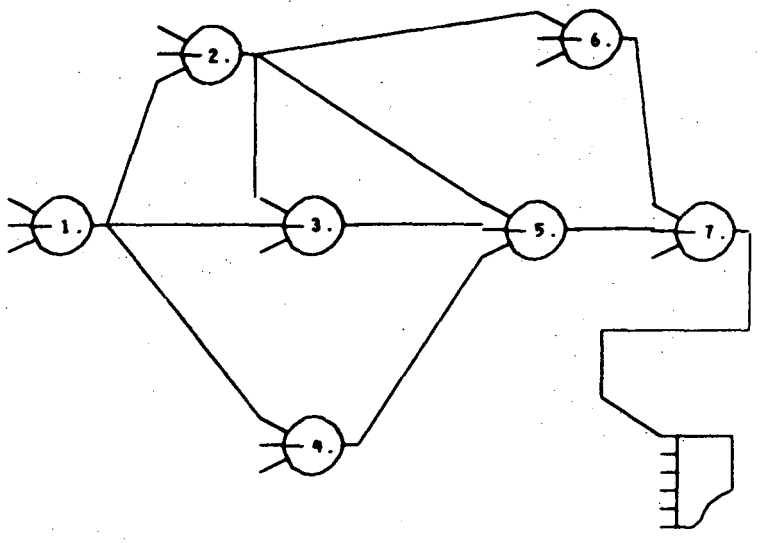


FIGURE 6. PERT DIAGRAM READY FOR ANALYSIS

F I G 7

```
 1 1          CU MIMIC OVERLAY VERSION OCT 27, 1969
 2                              **MIMIC SOURCE-LANGUAGE PROGRAM***,
 3
 4
 5            $IAM       MIMIC RUN OF PERT PRODUCED BY MIMVERT
 6                       G001       MAX(0.)+ 1.
 7                       G002       MAX(0.,G001)+ 2.
 8                       G003       MAX(G002,G001)+ 3.
 9                       G004       MAX(G001)+ 4.
10                       G005       MAX(G002,G003,G004)+ 5.
11                       G007       MAX(G002)+ 6.
12                       G006       MAX(G007,G005)+ 7.
13                                  OUT(G006)
14                                  END
15 0*** NO FIN STATEMENT FOUND. MIMIC ASSUMES - FIN(T,TZERO) - ***
16
17 1
18  G006   = 1.800000E+01
19
20
21
22
23
24                  FIGURE 7.  PERT DIAGRAM ANALYSIS
25
```

semantic descriptions and symbols for three primitive elements -- a resistor, a capacitor, and a transistor -- using common symbolism (see Figure 8). Extra lines have been added to each of these symbols in order to clarify where the element value and the element identification (XXX) are to be placed. The textual semantic description for each of these elements is contrived to produce valid input for SPICE. For the resistor, the following definition is satisfactory

R->XXX IN1 IN2 VAL

the R -> ensures that whatever identification is chosen will be presented to SPICE with an R (prefix) in front of it, thus allowing SPICE to recognize that it is a resistor. Similar descriptions are used for the capacitor and transistor elements:

C -> XXX IN1 IN2 VAL

Q -> XXX NC NB NB MNAME

Then the user creates a composite elment (model) named ONETRAN. After first positioning the symbols for the resistors, the capacitors and the transistor, he connects the symbols together with lines. Parameters for the individual elements are provided by labels positioned at the appropriate attacher points (Figure 9). Each set of connected lines (net) joins together a group of attacher points. These connections comprise the topological information in the diagram.

F I G 8

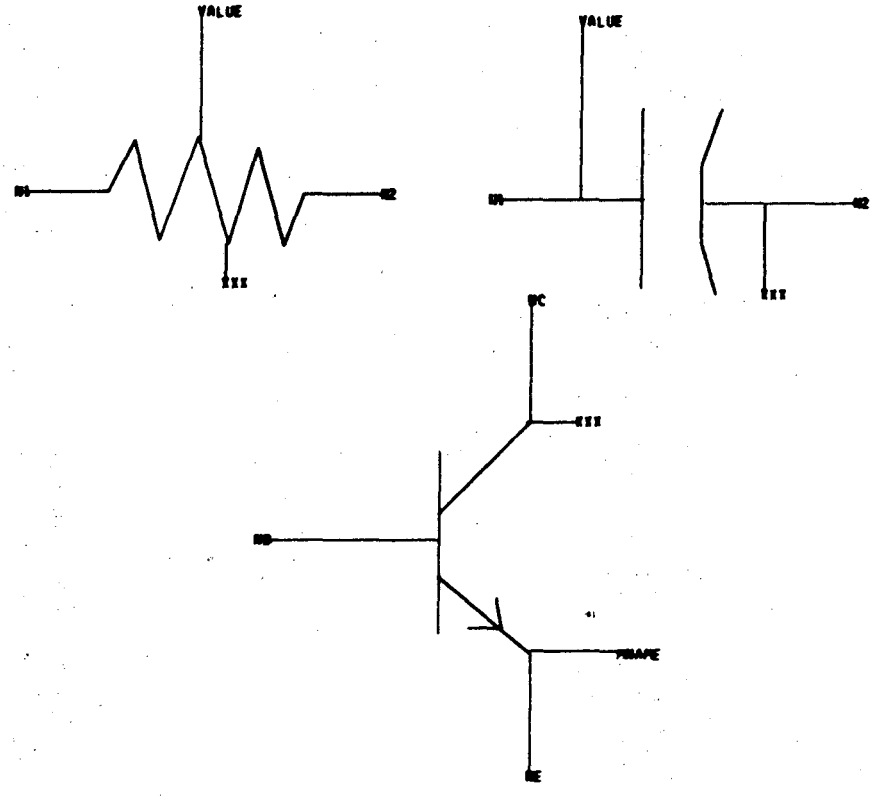


FIGURE 8. SYMBOLS FOR CIRCUIT ANALYSIS

ONETRAN

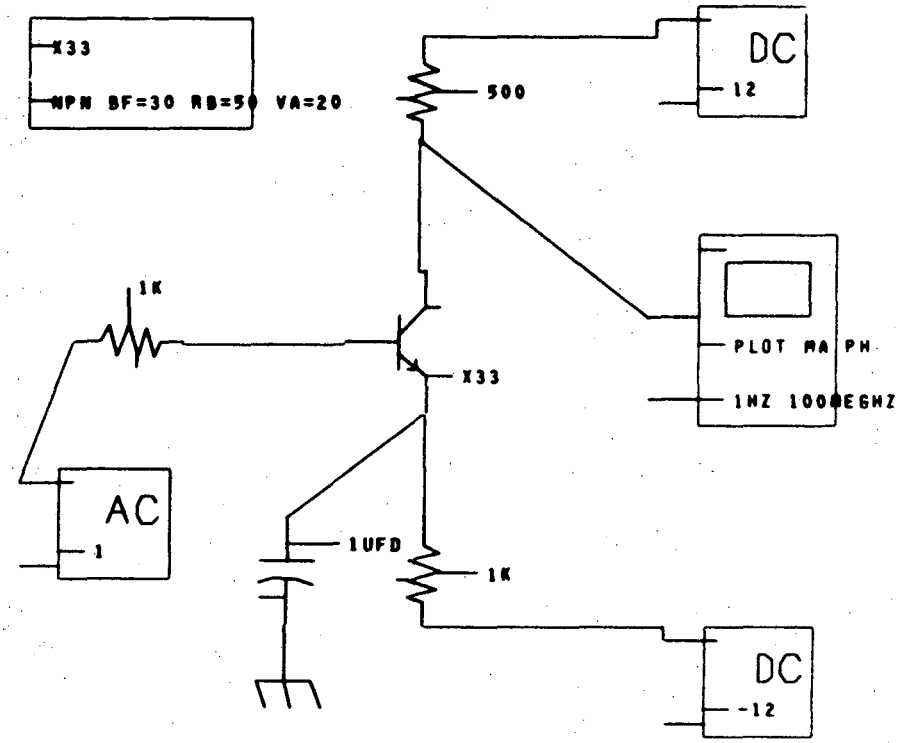


FIGURE 9. ONE TRANSISTOR AMPLIFIER

The translator performs the semantic propagation by expanding the model (macro definition) in terms of the semantics of its component elements. In this case, the propagation is quite simple, since only one level of semantic propagation is involved. The translator assigns a name to each net (as identified by the topological analysis routine). Labeled nets are assigned the label as their name. Unlabeled nets are assigned a name which is generated internally by the translator. Each net name then becomes the actual parameter which is used (substituted) in place of the formal parameters identified by the attacher points. Figure 10 illustrates how formal parameters in the element descriptions have been replaced by actual parameters from the macro description. The results of this analysis are shown in Figure 11.

1.4.3 Digital Logic Example

This example demonstrates the power and flexibility of GMS by using it to "redefine" (adapt) an existing language (MIMIC) to handle a digital logic problem, for which it was not designed. Without GMS this would not be practical, but with GMS the user can create a set of elements and then work with the symbols for those elements, unencumbered by constant attention to the intricate details needed to "redefine" the existing language. Since elements are parameterized, only a small number of primitive elements may be needed. We begin with the definition of a NAND gate, a gate whose output is

ONETRAN

```
 1 SPICE RUN OF ONETRAN
 2 V08    01 0   AC  1
 3 .MODEL X33    NPN BF=30 RB=50 VA=20
 4 R09    01 02  1K
 5 V03   03 0 DC 0
 6 C10    04 03  1UFD
 7 Q11    05 02 04  X33
 8 R12    06 05  500
 9 R13    04 07  1K
10 .OUTPUT V14    05 0    PLOT MA PH
11 .AC DEC 10   1HZ 100MEGHZ
12 V15    06 0   DC  12
13 V16    07 0   DC  -12
14 .END
15
16
17
18
19
20
21
22
23            FIGURE 10.  TRANSLATOR OUTPUT FOR AMPLIFIER
24
```

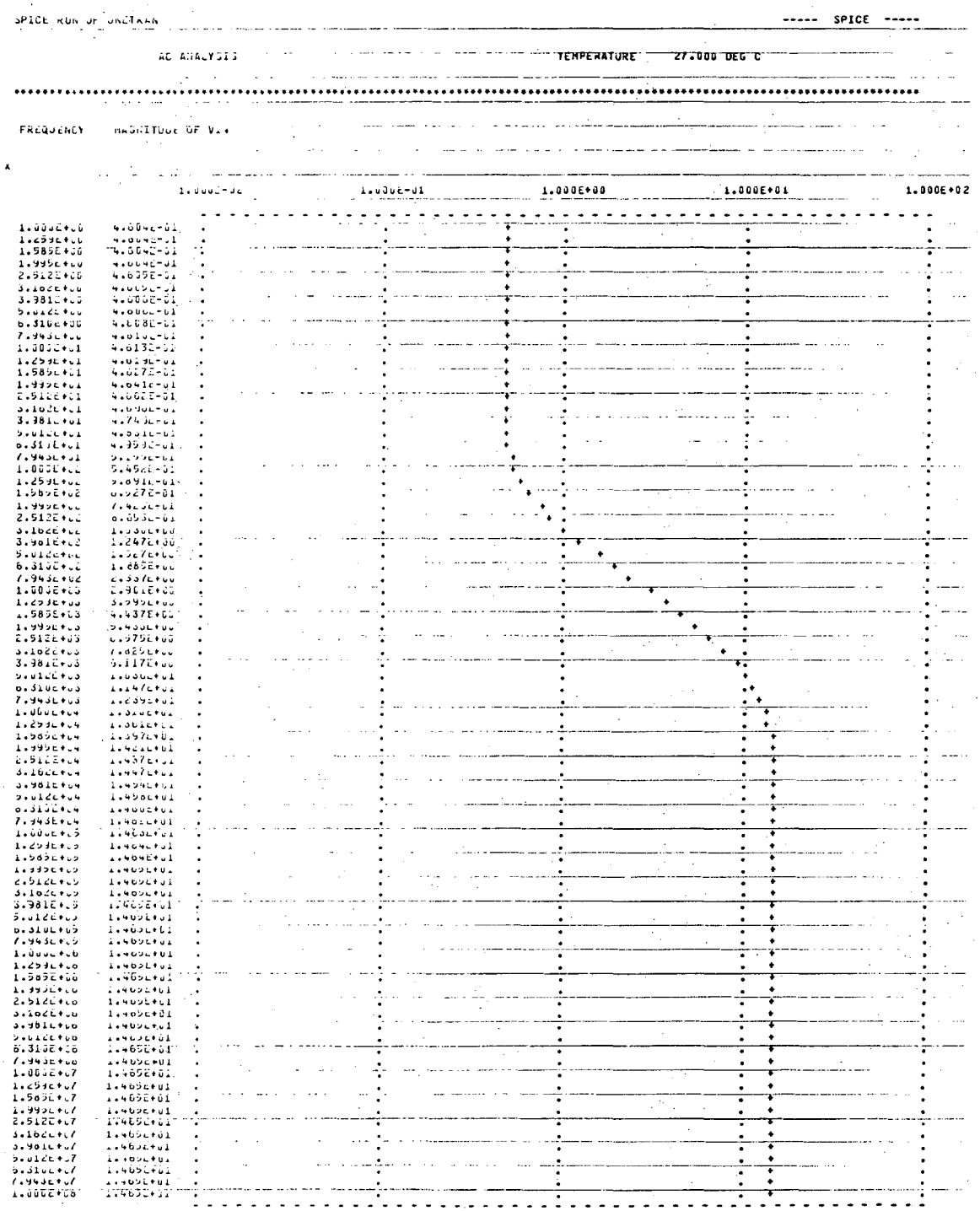


FIGURE 11A. RESULTS OF SPICE ANALYSIS - MAGNITUDE

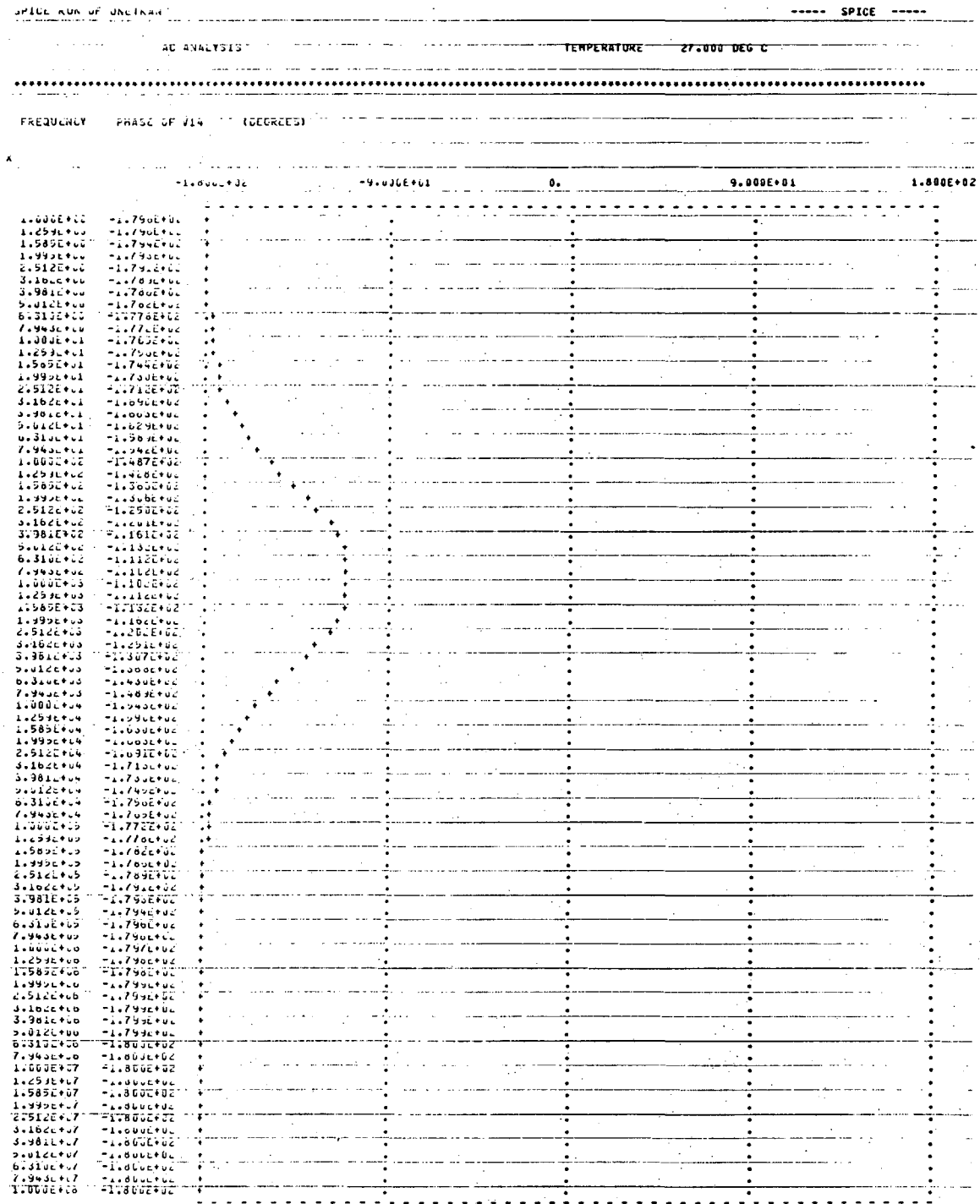


FIGURE 11B. RESULTS OF SPICE ANALYSIS - PHASE

false if and only if its two inputs are both true. Our first convention is to represent true values as analog signals with a value of 1.0 and false values as signals with a value of 0. Then the arithmetic expression

$$Z = 1.0 - X * Y$$

is contrived to compute the output Z of a NAND gate with two inputs, X and Y. Using this NAND gate as a primitive element we can construct a model to function as a half adder (see Figure 12A). In order to test it, we need a means for assigning inputs and observing outputs. We do this with some MIMIC functions (see Figure 13). The results are shown in Figure 15.

However, when this NAND gate is used in situations with feedback, MIMIC produces a diagnostic which usually indicates a non-physical situation. In this case, our model neglected the fact that physical gates have an inherent time delay and we revise the text of our semantic description to include this. (MIMIC's TDL (time delay) function requires a third argument (10.0) for storage allocation purposes.)

$$Z = TDL\ (1. - X * Y,\ TD,\ 10.0)$$

Here we have introduced a parameter, TD, which is global since no attacher point occurs to make it a formal parameter, and we must be careful to give it a value in our model. This allows us to simulate "slow" or "fast" logic by manipulation of this

H A L F

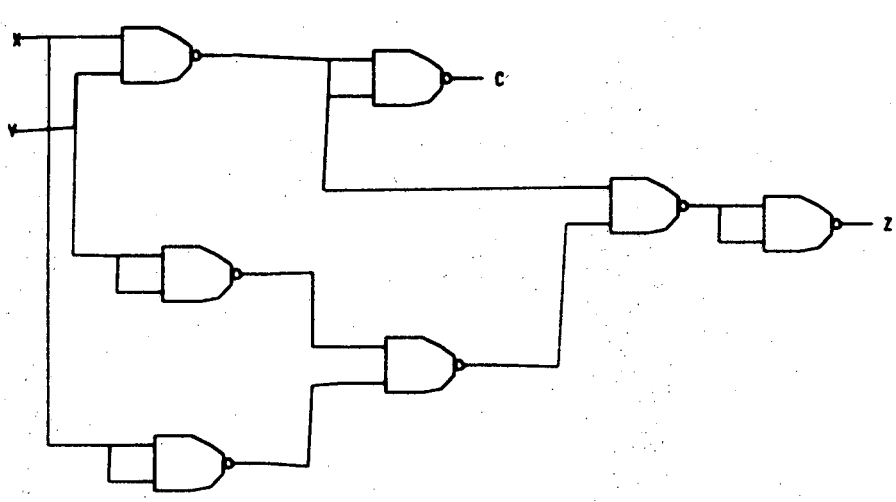


FIGURE 12A. HALF ADDER FROM NAND GATES

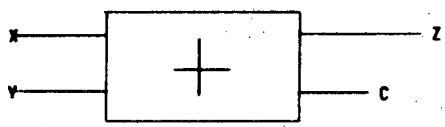


FIGURE 12B. HALF ADDER SYMBOL

HALFTST

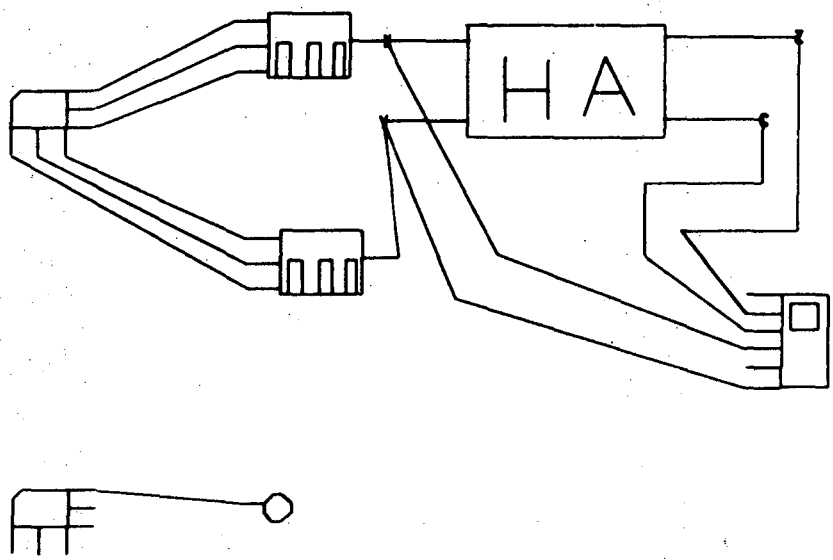


FIGURE 13. HALF ADDER TEST CIRCUIT

HALFTST

```
 1 $IAM      MIMIC RUN OF HALFTST PRODUCED BY MIMVERT
 2           HALF     BML(C      ,X      ,Y      ,Z      )
 3           G001    =FIX(TDL(1.-X*Y,TD,20.)+.5)
 4           G002    =FIX(TDL(1.-X*X,TD,20.)+.5)
 5           G003    =FIX(TDL(1.-Y*Y,TD,20.)+.5)
 6           C       =FIX(TDL(1.-G001*G001,TD,20.)+.5)
 7           G004    =FIX(TDL(1.-G003*G002,TD,20.)+.5)
 8           G005    =FIX(TDL(1.-G001*G004,TD,20.)+.5)
 9           Z       =FIX(TDL(1.-G005*G005,TD,20.)+.5)
10                    EMA
11                    PAR(X1,X2,X3,Y1,Y2,Y3)
12                    PAR(TFIN,PI,TD)
13                    FIN(T,TFIN)
14           X       =FSW(SIN(PI*(T-X1)/X3)*SIN(PI*(T-X2)/X3),1.,0.,0.)
15           HALF     CML(C      ,X      ,Y      ,Z      )
16           Y       =FSW(SIN(PI*(T-Y1)/Y3)*SIN(PI*(T-Y2)/Y3),1.,0.,0.)
17                    PLO(T,Z,C,X,Y)
18                    END
19
20
21
22
23
24
25          FIGURE 14.  TRANSLATOR OUTPUT FOR HALF ADDER
26
```

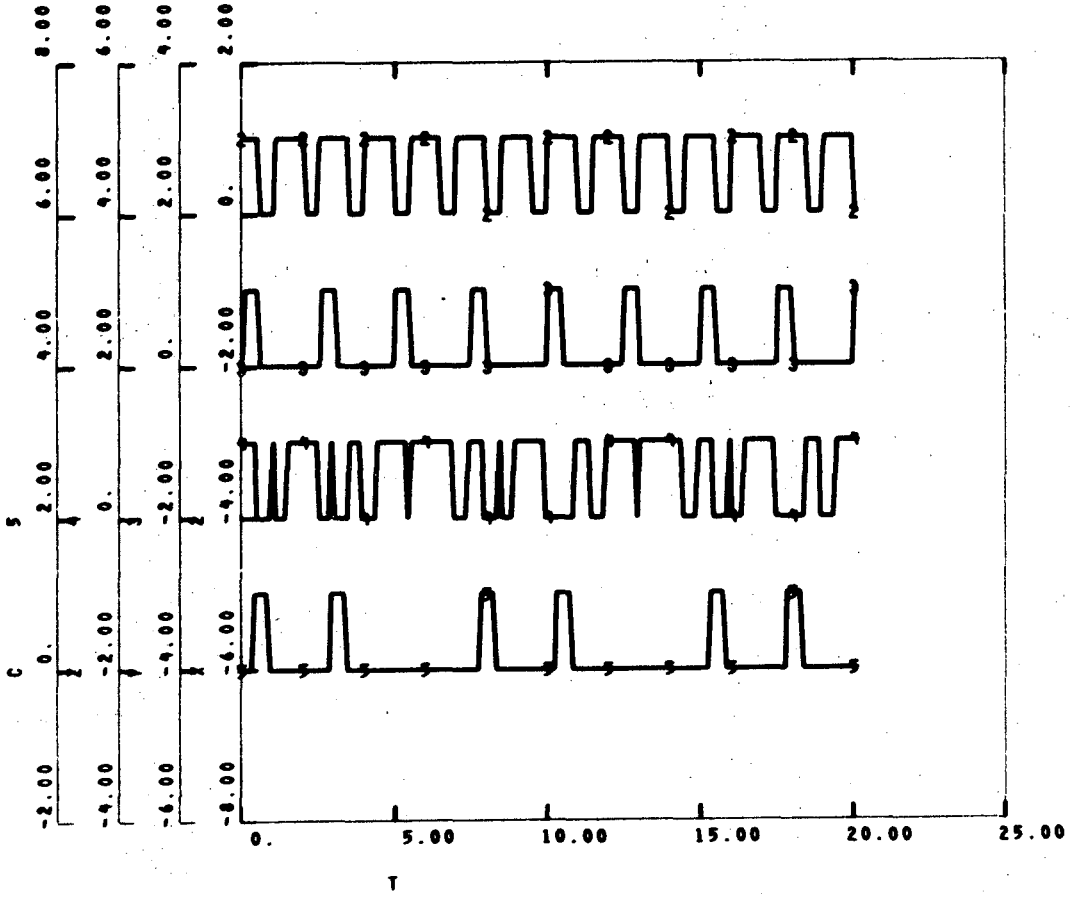


FIGURE 15. RESULTS OF HALF ADDER TEST

11/02/75. 14.45.54.

parameter.

The point of our description is to emphasize that refinements can be made to the semantic descriptions of elements without disturbing models which use them as components. This is essential to allow progressive refinement of problem descriptions. A further refinement (to correct roundoff errors) is shown below:

```
Z = FIX (TDL(1. - X * Y, TD, 10.) + .5).
```

This particular element, NAND with time delay, is used in a more complicated logic problem, designed by Richard La Pierre (Figure 16). More complete documentation for this problem is available from the PICASSO user's manual [AUST72].

1.4.4 Compartmental Modeling

Compartmental modeling is a technique in which fluids are modeled as if contained in a discrete set of compartments with channels between them. It can be applied in diverse situations from many fields, including biology, chemistry and engineering. We shall take the following example from an engineering application, an analysis of water and pollution flows in the San Francisco Bay area. Our model of the bay will be a series of compartments, connected by channels. A roogh sketch of the configuration reveals that no compartment is connected to more than three channels, so we designate three possible channels for each compartment, labelled, A, B,

P I E R R E 2

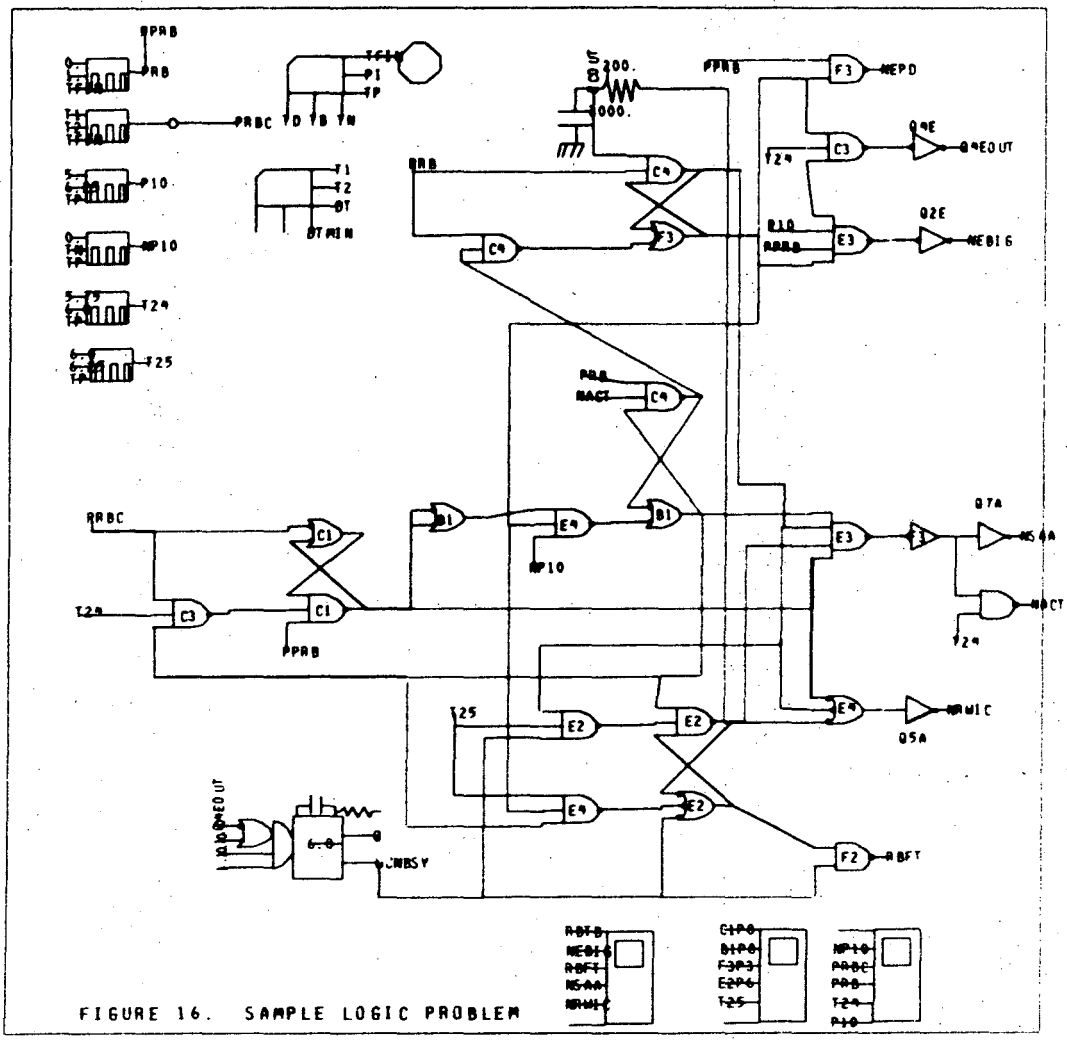


FIGURE 16. SAMPLE LOGIC PROBLEM

and C, and we use only as many as are needed in each case. The amount of water in each compartment, V, is the time integral of the net flow into it from the three channels:

$$V(t) = V(0) + \int (L_A(t) + L_B(t) + L_C(t))\ dt$$

Each channel connects two compartments, which we designate as A, the upstream compartment, and B, the downstream compartment. The amount of water flowing in a channel is proportional to the difference between the heights of the compartments, H_A, and H_B, and to the cross-sectional area CA of the channel:

$$L(t) = (H_A(t) - H_B(t)) \cdot CA$$

But the height of the water in a compartment is the ratio of the volume V to the base area AREA:

$$H(t) = V(t) / AREA$$

In the same manner, we analyze the amount of pollutant P in a compartment:

$$P(t) = P(0) + \int (F_A(t) + F_B(t) + F_C(t))\ dt$$

where, for each of the three channels A, B, and C,

$$F(t) = \begin{cases} M_A(t) \cdot L(t) & \text{if } L(t) > 0 \\ M_B(t) \cdot L(t) & \text{if } L(t) \leq 0 \end{cases}$$

Here M_A and M_B are the concentrations of a pollutant in the

two compartments. The assumption is that each compartment is well mixed; that is, the concentration of pollution is uniform throughout the compartment. This concentration is therefore simply the amount of the pollutant divided by the volume:

$$M(t) = P(t) / V(t)$$

These two elements, the compartment and the channel, are the primary elements needed for the analysis. However, a number of other elements are needed : variations of these primary elements to suit special needs, and utility elements for input, output and termination. The two special elements required are a compartment whose height is a sinusoid and whose pollution concentration is zero (to serve as the ocean), and a channel whose flow is independent of height (which can be used as a river). The utility elements required are constructed using analytic definitions which refer to standard predefined MIMIC functions for input (Figure 17), output (Figure 18) and termination.

We can now enumerate the parameters which must be provided for a compartment: three outputs (the height of the water H, the amount of pollution P and the concentration of pollution M), and nine inputs, that is, three initial conditions (volume V(0), base area AREA, and initial pollution P(0)) and a pair of inputs for each of three channels (water flow, L_A, L_B, L_C and pollution flow F_A, F_B, F_C). Each channel has four outputs (two water flows L_A and L_B and two pollution

FIG 17

```
1          PAR(P1,P2,P3,P4,P5,P6)
2
```

FIGURE 17A. ANALYTIC DEFINITION FOR PARAMETER INPUT

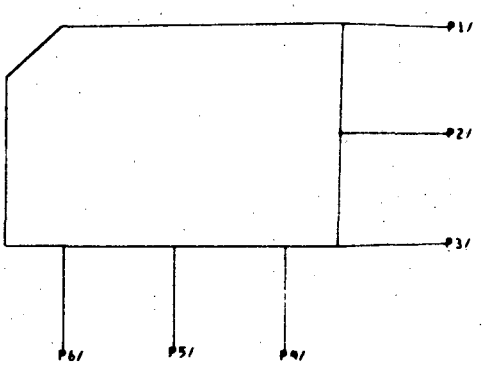


FIGURE 17B. SYMBOL FOR PARAMETER INPUT

F I G 1 8

```
1                    PLO(IN1,IN2,IN3,IN4,IN5,IN6)
2
```

FIGURE 18A. ANALYTIC DEFINITION FOR PLOTTED OUTPUT

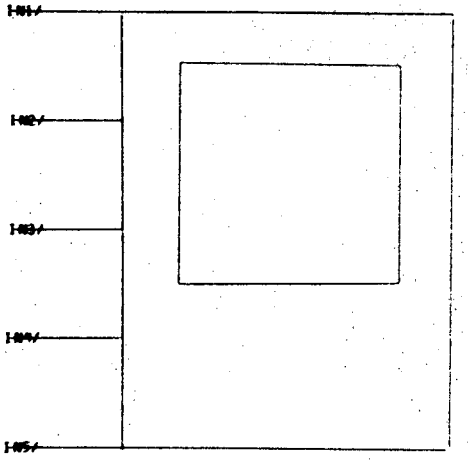


FIGURE 18B. SYMBOL FOR PLOTTED OUTPUT

flows F_A and F_B), and five inputs (two heights H_A and H_B, two pollution concentrations M_A and M_B, and the value for the channel cross sectional area). Note that the net flow in the channel is zero; we have assumed that the channel has no storage capacity.

The user has now reached the point where GMS can help him. He has formulated the problem and derived the mathematics necessary for its solution. Most importantly, he has established a conceptual framework and a set of conventions to guide him in further development of the model.

The actual example shown here uses a special string substitution feature to reduce the number of nets (Section 2.6.7) needed to join compartments and channels. Since the translator will substitute the name of a net for the formal parameters named on the attacher points, we can use a concatenation feature in the translator to prefix each of several identifying characters to the string which represents the name of the net. The effect of this (in this case) is to provide four variables from a single net name. If our character string manipulations are consistent among the various elements, we can use these four variables as if they were connected by four parallel nets. For the compartment the formal parameters for connection to the three channels are A, B and C. The variables associated with parameter A would be LA (for the flow), HA (for the height) FA (for the pollution flow) and MA (for the concentration). Other variables are

renamed as follows: VOL for the initial volume V(0) and POL for the initial pollution P(0). Similar manipulations are made in the channel parameter names.

The user must now create the primitive elements, including names, semantic descriptions, and symbols for the compartment (Figure 19), the channel (Figure 20), and the special variations of these elements. The utility elements must also be constructed, unless they can be borrowed from one of the existing libraries of elements.

Then the model (Figure 21) must be created, including its name, placment of the various symbols, and their connections. Attention must be paid to the assignment of parameters to the elements, to be sure that all the channels have base areas, initial volumes and so forth. The values used in this model were obtained in part from a report on the San Francisco Bay by the Kaiser Corps of Engineers [KAIS69].

The translator function in GMS will provide a card image interpretation of the semantics of the model and its component elements. Figure 22 gives a partial listing. Control then passes to MIMIC (via JCL) for analysis and display (Figures 23 and 24 show typical outputs). When MIMIC is terminated, control returns to GMS for more work on the model or termination of the session.

N O D E

1	V	INT(L→A+L→B+L→C,VOL)
2	HT	(V-VOL)/AREA
3	H→A	HT
4	H→B	HT
5	H→C	HT
6	P	INT(F→A+F→B+F→C,POL)
7	MT	P/V
8	M→A	MT
9	M→B	MT
10	M→C	MT
11		
12		
13		
14		
15		
16		
17	FIGURE 19A. ANALYTIC DEFINITION FOR A COMPARTMENT	
18		

F I G 1 9 B

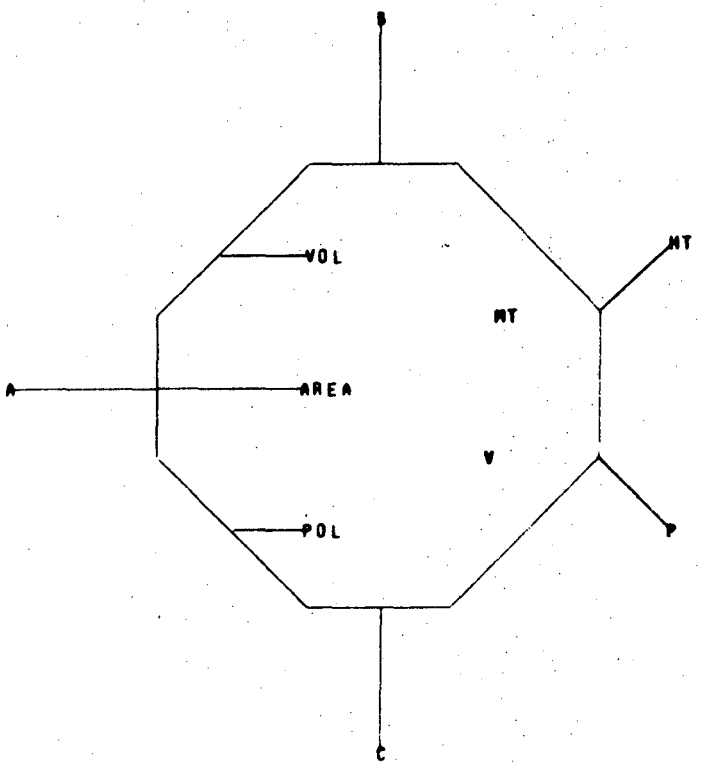


FIGURE 19B. SYMBOL FOR A COMPARTMENT

CHAN

```
 1          DH        FTR(M→A-M→B,TAU)
 2          L→B       DH*CA*KF
 3          L→A       -L→B
 4          ZM        MAX(DH+KD,0.)*M→A
 5          F→B       (ZM-MAX(-DH+KD,0.)*M→B)*CA*KF
 6          F→A       -F→B
 7
 8
 9
10
11
12
13          FIGURE 20A.  ANALYTIC DEFINITION FOR A CHANNEL
14
```

FIG20B

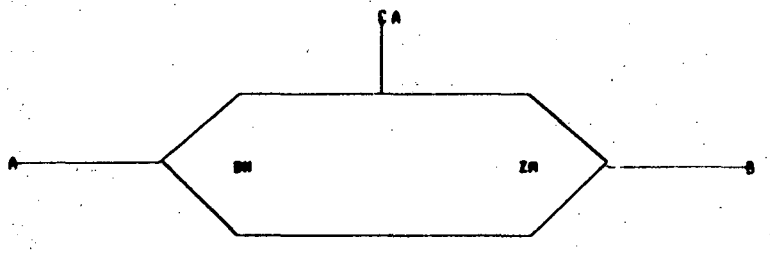


FIGURE 20B. SYMBOL FOR A CHANNEL

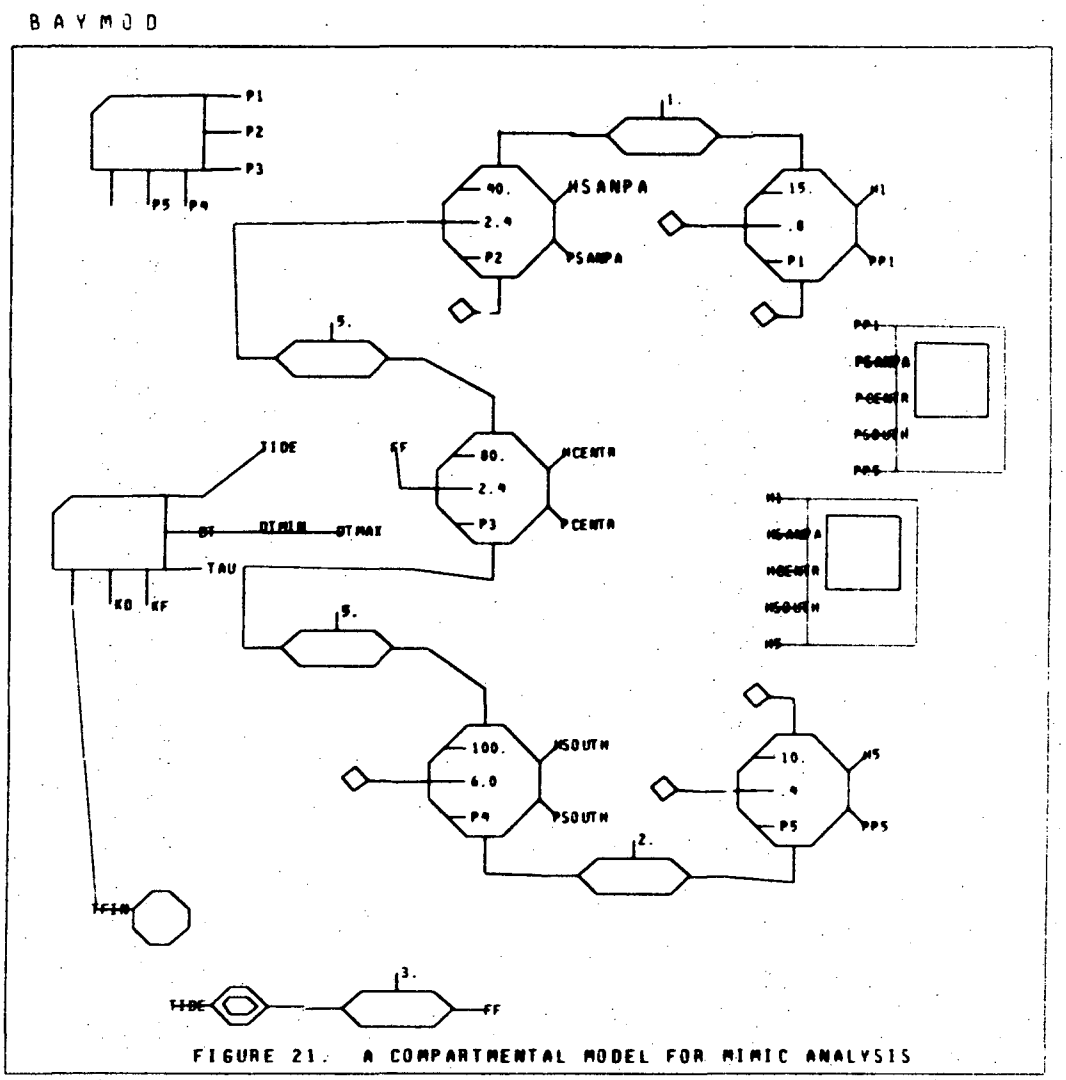


FIGURE 21. A COMPARTMENTAL MODEL FOR MIMIC ANALYSIS

BAYMOD

```
 1 $IAM       MIMIC RUN OF BAYMOD PRODUCED BY MIMVERT
 2            DTMIN     DTMAX
 3            DT        DTMAX
 4                      FIN(T,TFIN)
 5            HG004     TIDE*SIN(T*.5)
 6            MG004     0.
 7                      PAR(TIDE,DTMAX, TAU, KF, KD,TFIN)
 8            G003      FTR(HG001-HG008,TAU)
 9            LG008     G003* 5.*KF
10            LG001     -LG008
11            G006      MAX(G003+KD,0.)*MG001
12            FG008     (G006-MAX(-G003+KD,0.)*MG008)* 5.*KF
13            FG001     -FG008
14                      PAR( P1, P2, P3, P4, P5)
15            G007      FTR(HG002-HG009,TAU)
16            LG009     G007* 5.*KF
17            LG002     -LG009
18            G010      MAX(G007+KD,0.)*MG002
19            FG009     (G010-MAX(-G007+KD,0.)*MG009)* 5.*KF
20            FG002     -FG009
21            G013      FTR(HG004-HFF,TAU)
22            LFF       G013* 3.*KF
23            LG004     -LFF
24            G014      MAX(G013+KD,0.)*MG004
25            FFF       (G014-MAX(-G013+KD,0.)*MFF)* 3.*KF
26            FG004     -FFF
27            G017      INT(LG015+LG009+LG011, 100.)
28            HSOUTH    (G017- 100.)/ 6.0
29            HG015     HSOUTH
30            HG009     HSOUTH
31            HG011     HSOUTH
32            PSOUTH    INT(FG015+FG009+FG011, P4)
33            G016      PSOUTH/G017
34
35          FIGURE 22.  PARTIAL LISTING OF THE COMPARTMENTAL MODEL
```

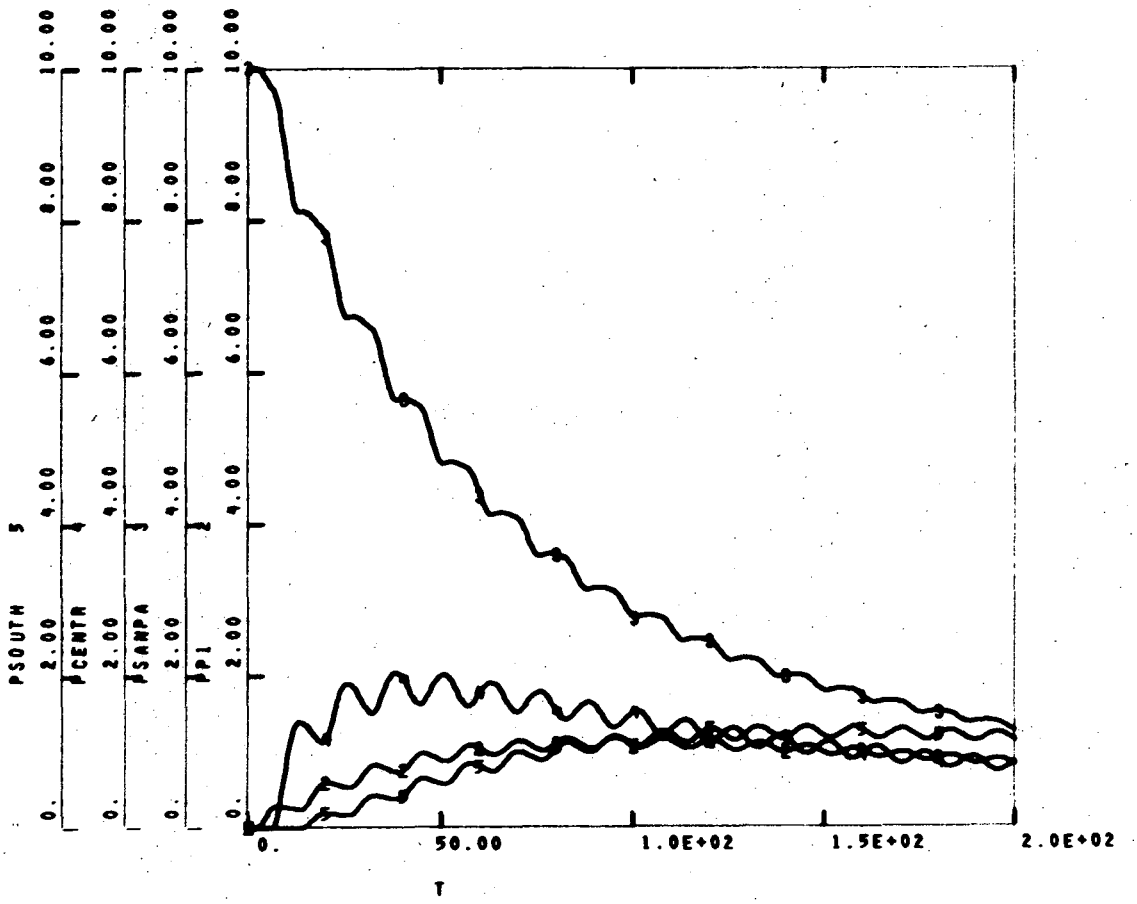


FIGURE 23. 200 HOURS OF POLLUTION HISTORY IN 4 COMPARTMENTS

11/02/75. 16.25.03.

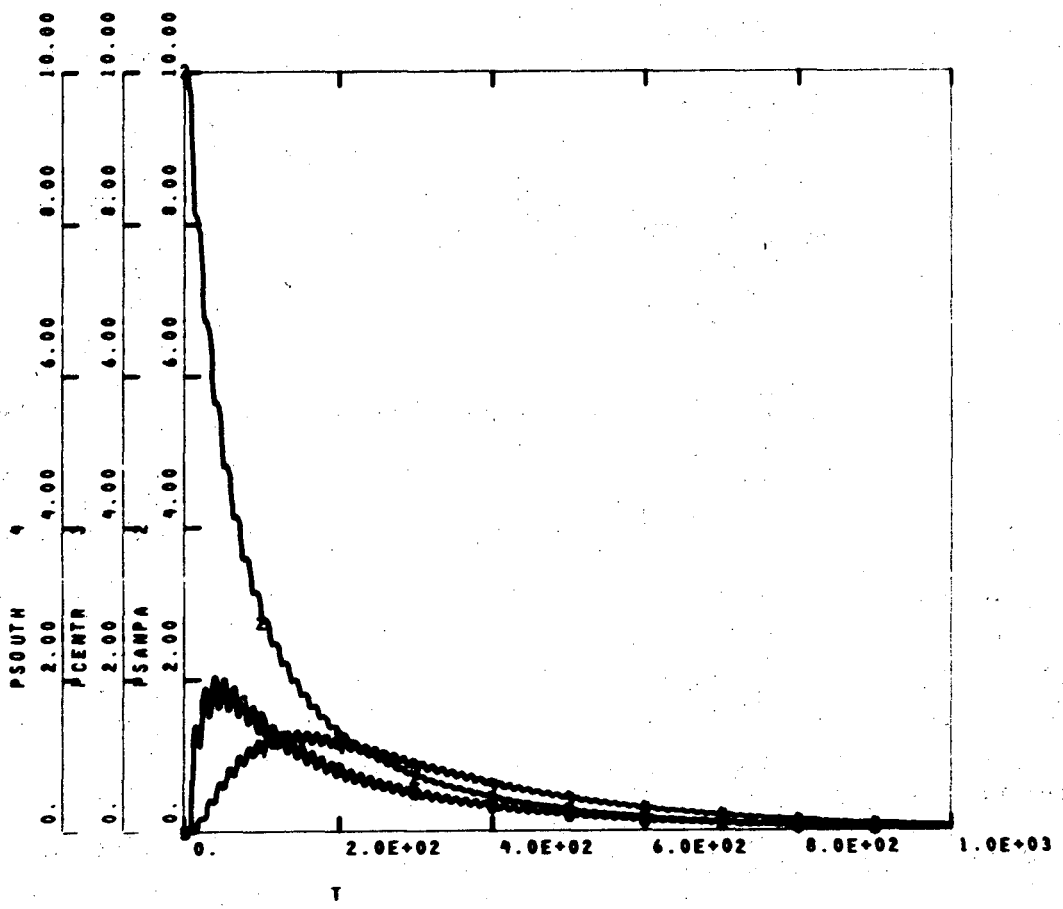


FIGURE 24. NEW PARAMETERS SHOW EFFECTS EXTENDED IN TIME

11/02/75. 16.30.21.

2. A GRAPHICS MODELING SYSTEM

2.1 SOFTWARE ORGANIZATION (AND OPERATION)

We have argued that a CAD system can be divided into a problem definition module and a problem-solving analysis module. Figure 25 illustrates the information flow for such a system. We have shown a CRT and a keyboard as the man-machine interface. For maximum effectiveness such an interface could also include a lightpen, tablet, or other auxiliary devices. The illustration shows GMS on the left being used to construct primitive elements and models, and storing them in a library. The analysis section is shown to the right, with its input obtained either directly from the element library or indirectly via the translator and an intermediate file (shown dotted). The analysis section produces an output file which is then interpreted by the display routines. In many cases, the display routines will be combined with the analysis routines. (Ultimately, however, I think that a set of general purpose display routines will prove to be more powerful than an individual set for each analysis package. The reasoning behind this conclusion is analogous to that leading to a separate GMS rather than one for each application.)

FIG25

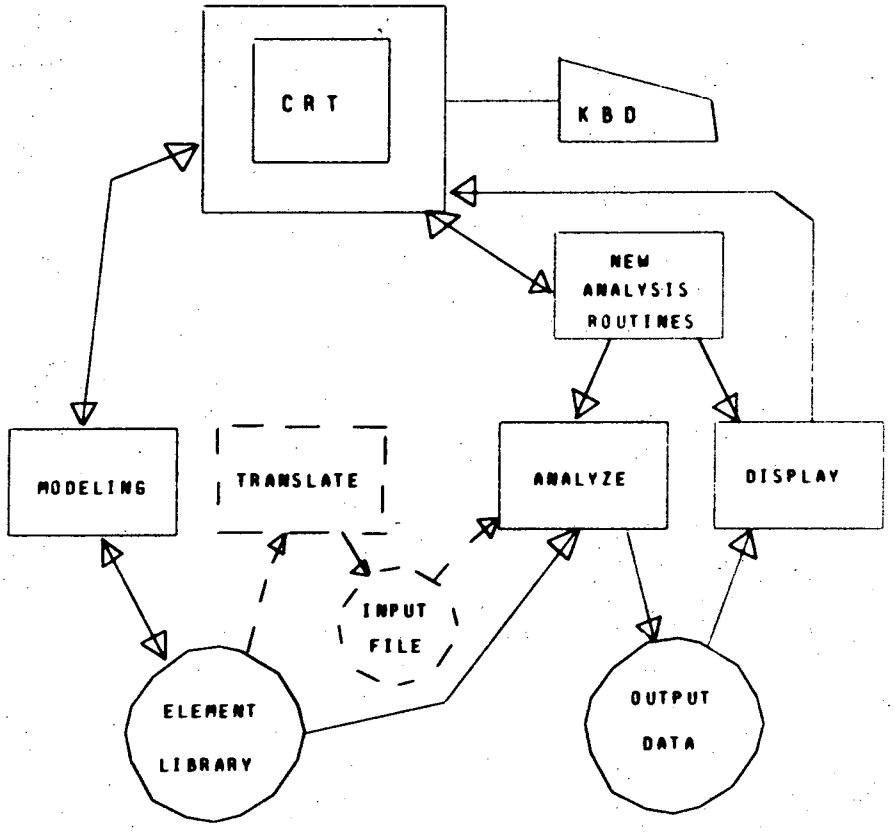


FIGURE 25. GMS-CAD INFORMATION FLOW

2.1.1 GMS Information Flow

Figure 26 shows the information flow within GMS itself. The graphics editor accepts information from the user, modifies the edit buffer, and updates the display. The text editor performs similarly on text. The edit buffer is simply a reserved part of memory in which addressing of individual items is very simple. (Details of these data structures are provided in Sections 2.2 and 2.3.) The filing module is responsible for storage and retrieval from disk and for loading the edit buffer from the element library area in memory. The filing module is the highest level routine, in the sense that it loads the edit buffer and activates the other routines. It also maintains the directory structure which ties together the symbols and semantic descriptions of elements.

GMS, in the conceptual sense, is a methodology for the problem-description phase of CAD activities. GMS, in the programming sense, is a data structure and a set of modules which support the design techniques. In the conceptual sense, we have found it convenient to describe activities such as creating primitive elements, creating composite elements and translating models for analysis. In the programming sense, the creation of primitive elements is not concentrated in a single module, but rather spread over the filing, text editor and graphics editor modules. In the same manner, the graphics editor is not limited in responsibility to just one task, but

F I G 2 6

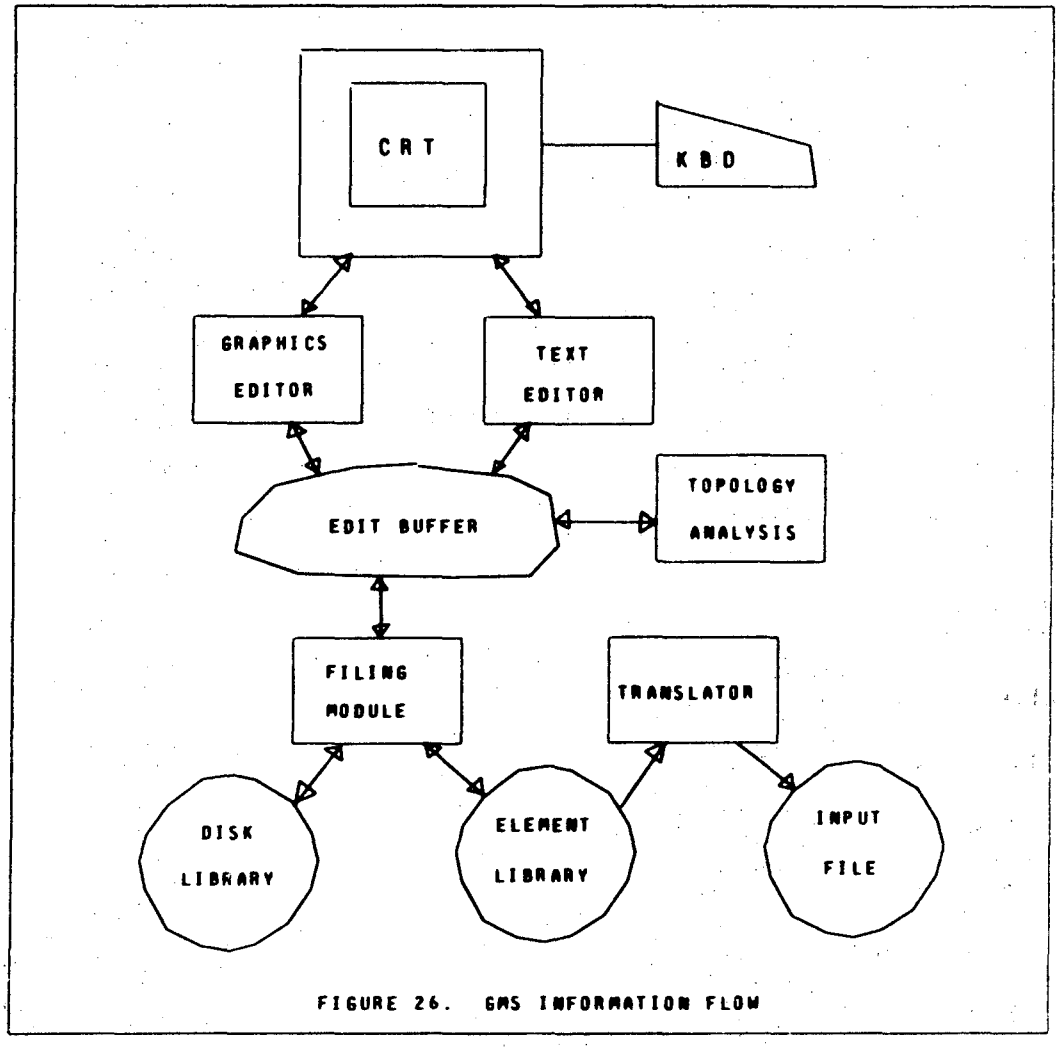


FIGURE 26. GMS INFORMATION FLOW

must edit symbols, macro definitions and empirical descriptions; a single common editor is preferred to a separate editor for each task. This requires that a common set of graphics primitives be found for these three tasks.

2.1.2 Graphic Primitives

The design of GMS requires the specification of the graphics primitives to be used and their correspondence to our description of element symbols, model descriptions and empirical descriptions. In the case of model descriptions, the correspondence is obvious: subpictures for symbols, lines for nets, and alphanumeric annotation for labels of nets. For subpictures, we have added the useful features of scaling and rotation. In general, alphanumeric annotation could include both a coordinate location for the attachment (to some feature) of the string and a separate location for the display of the character string. This latter location would be strictly a graphics feature, but could increase the legibility of dense displays. In the prototype, only one location is allowed; the coordinate location for attachment is also used as the origin of the first character of the string for display. Also in the prototype, the entire annotation is the unit of editing, precluding either the replacement of individual characters within the string, or moving the string once its position has been confirmed. This has not been inconvenient, since most annotation is short.

Our choices of graphics primitives are motivated by a desire to choose high-level primitives which are still compatible with easy editing (of our type of drawings) and easy interpretation (for topological analysis). Thus, if lnes were created explcitly from points, the flexibility gained would not compensate for the greater complexity in using and analyzing the drawing. At the other extreme, if sets of joined line segments were the units of editing, the user would find it more difficult to change one line segment in a group, although the topological analysis might be easier or faster.

Another motivation for our choice of graphics primitives is to allow the use of the same graphics primitives and editor for drawing element symbols. For use in element symbols, lines correspond to lines and alphanumeric annotation to attacher points. The graphics elements are the same, but the meaning is different. Lines have no meaning other than as a visual element, while annotation is now used for attacher points instead of labels.

The correspondence between graphics primitives and data sets for empirical definitions of primitive elements is the least well developed of the prototype's capabilities. While lines and alphanumerics enable nearly all charts and graphs to be reproduced in a visual sense, it is not always clear how these charts and graphs should be converted to data sets (e.g. tables of numbers) and vice versa. The prototype system requires that empirical descriptions have the form of a single

curve, single valued on the x-axis. Scales may be given for both axis and either axis may be specified as linear or logarithmic. A generalized I/O facility in the prototype can convert any drawing to a card image description of its graphics primitives and vice versa. This feature can be used to incorporate simple tables into empirical descriptions.

For the creation of text, the prototype has followed the example of most common simple text editors. The units of editing are the line and the character. No problems were encountered (or expected) in this approach.

2.2 DATA STRUCTURES AND STORAGE

The purpose of a data structure is to allocate storage to the various pieces of information which must be stored (items) and to provide access paths for use explicitly by the program and implicitly by relations within the data (links). For GMS, there is a natural grouping of the items into blocks. This grouping is natural in the sense that links in GMS always refer to a complete block and not to items within a block.

2.2.1 Types of Blocks

The four types of blocks are graphics blocks (items are lines, alphanumeric annotation and symbol references), text blocks (items are lines of text), topology blocks (items are nets, labels, symbol references and attacher points) and

empirical blocks (items are pairs of numeric values).

A primitive element with an analytic description has a graphics block for its symbol and a text block for its semantic descriptor. A primitive element with an empirical description has a graphics block for its symbol and a graphics block for its semantic description. In principle it also has an empirical block containing data points that comprise a tabular representation of the semantic description graphics block; this block is not formed until needed since it is a lot of work to keep it up to date when the graphics changes.

Each composite element has a graphics block for its symbol (if any) and a graphics block for its semantic description. In principle, it also has a topology block containing a rearrangement of the graphics block lines into nets; this block is not formed until needed.

The translator also uses templates (Section 2.6.3), the data for which is stored in the text block. These are created in the same manner as the analytic description text blocks used for primitive elements.

The normal display operation is a sequential scan of a graphics block or of a text block in the edit buffer, interpreting each item and generating the specified picture or text string display. Topology and empirical blocks are processed only by the translator; they contain auxiliary information (topology or data points) needed for translation

but not for display.

The strategy used in the prototype GMS is to allocate memory at the block level and to keep track of these blocks with a directory (Figures 27, 28). For the sake of simplicity, the prototype stores each block in the element library, a contiguous area of main memory. When a block is to be edited, it is moved to the edit buffer and the following blocks are moved up to reclaim the space. When editing is finished, the block is moved from the edit buffer to the end of the element library area. (The amount of memory available to the program and thus the size of the element library area can be changed by a request to the operating system.) This technique eliminates the need for a separate garbage collection phase and requires a minimum of memory. In use, editing seems to cluster within a few blocks, and these blocks move to the end of the storage area and reduce the amount of storage shuffling required subsequently.

2.2.2 Data Structures for Graphics and Derived Blocks

A design decision in the prototype was to include no auxiliary or secondary information in the graphics data structure. That is, while a graphics block is being edited, no secondary information (e.g. nets) is derived from the pictorial information. A separate block is used for this information and a conversion module is executed when the

FIG27

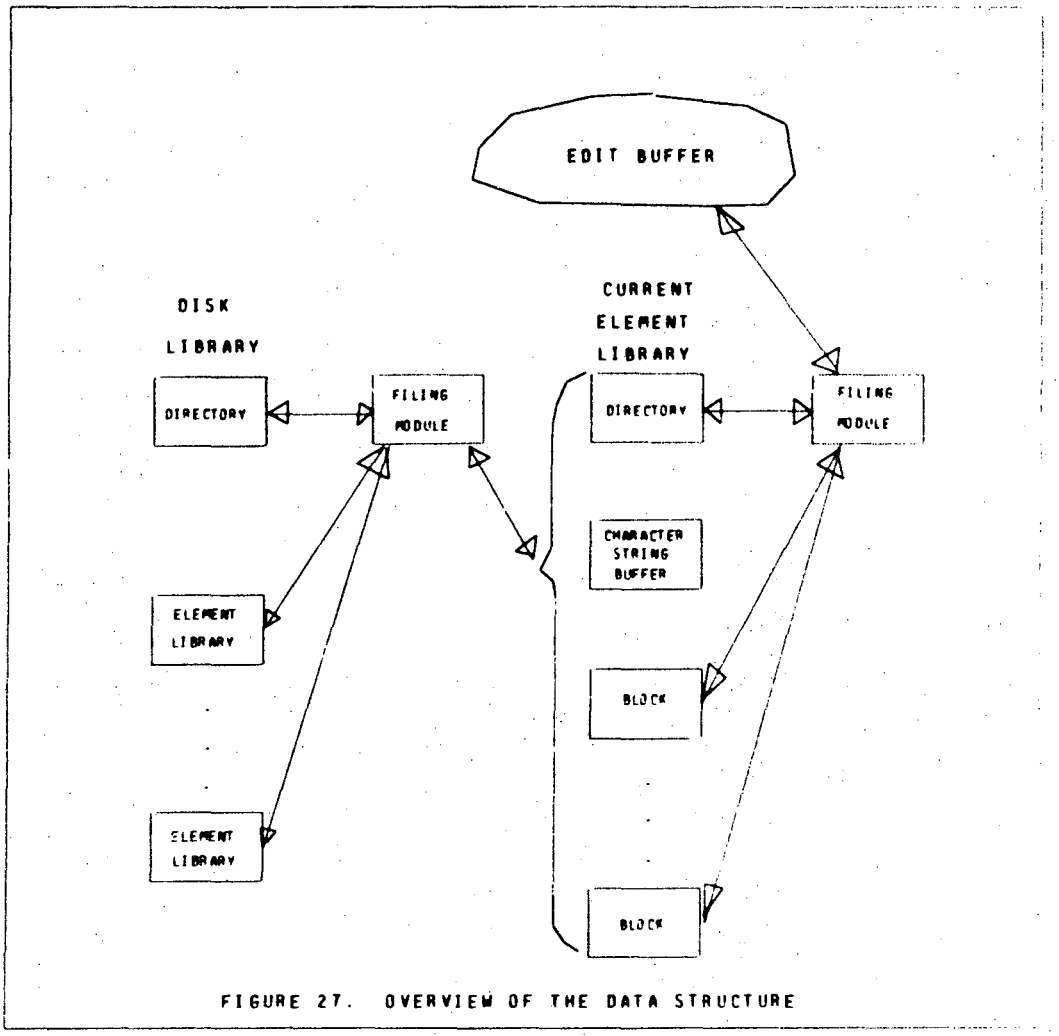


FIGURE 27. OVERVIEW OF THE DATA STRUCTURE

FIG28

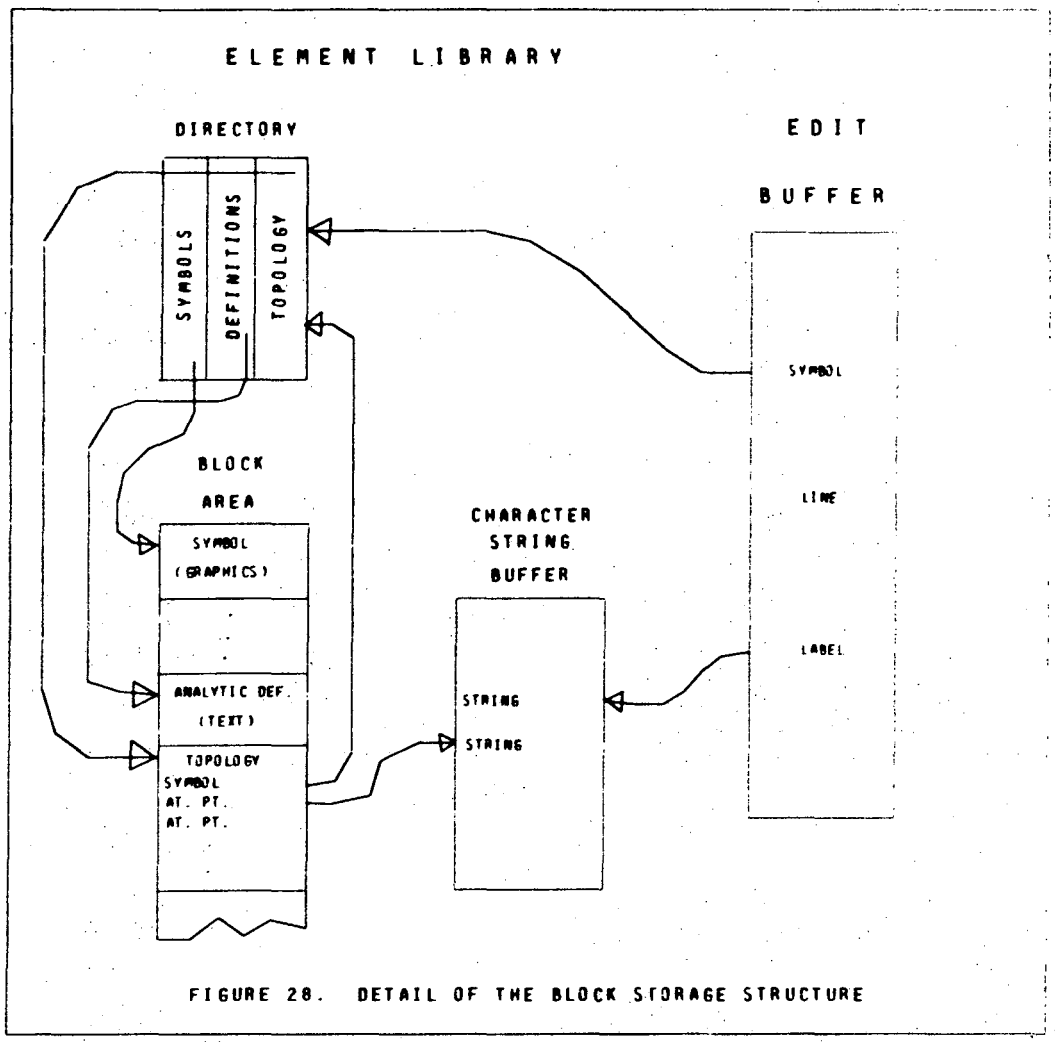


FIGURE 28. DETAIL OF THE BLOCK STORAGE STRUCTURE

secondary information is needed (if changes have been made since the last conversion). In the prototype, this secondary information is the topology and the empirical analysis of the drawing, produced by the topology and empirical routines. The advantage of this separation of data structures is that both structures are easier to interpret, less computation is required during graphics editing, and programming is reduced and simplified. The disadvantage of this approach is that no topological information can be given to the user during editing. For example, he cannot be warned that a particular connection might be invalid. He can be warned later, of course, but effort may have been wasted in the meantime on the editing of erroneous data.

The separation of data structures also provides more insight into how an existing graphics editor might be incorporated into a GMS.

Within a block, the individual items have been arranged to meet the needs of each particular type of block. For graphics blocks (Figure 29), each item consists of five fields. The first field gives the type of the item (line, alphanumerics or symbol reference); the second and third fields give the x and y coordinates of the origin of the item; and the fourth and fifth fields give the endpoint coordinates for lines, or a size/orientation field and a pointer field for alphanumerics and symbols. For alphanumerics, the pointer refers to a character string buffer, a reserved area of fixed

FIG29

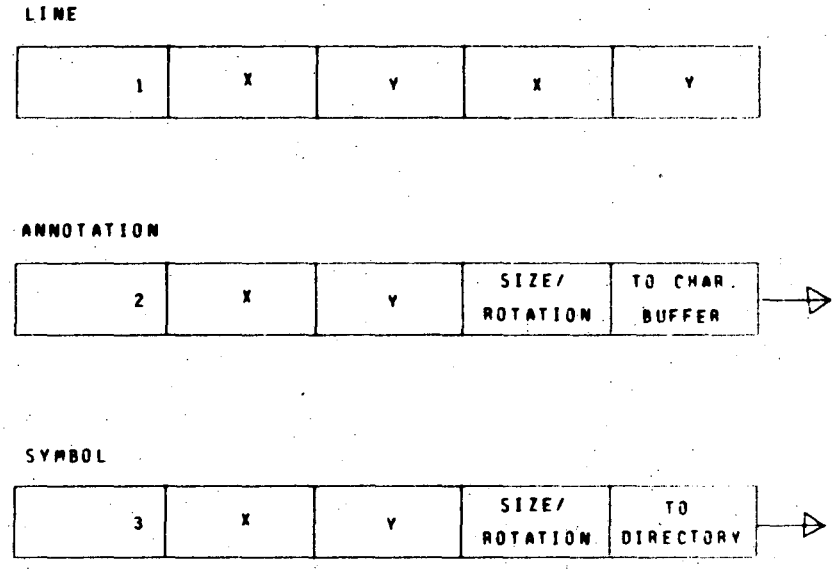


FIGURE 29. GRAPHICS ITEMS DATA STRUCTURE

size where the character string for annotation is stored. (See Section 3.3.2 for improvements.) For symbol references, the pointer field contains a link to the block whose items are the lines and alphanumerics of the symbol. More details can be found in Sections 2.3, 2.5 and 2.6.6.

For blocks containing a topological analysis, the items are similar to graphics items. Lines have been removed by the topological analysis module and their information is now conveyed by net numbers. Attacher points are now included, distinguished from labels by a different type number (and a different function). Label and symbol items are similar to the graphics format, with the fourth (size/orientation) field replaced by the net number.

Blocks containing a data set for an empirical definition are composed of a sequence of x,y coordinates, stored in the hardware floating point format.

2.2.3 Data Structure for Text Blocks

A text block is composed of lines of text with eacy line terminated by an end-of-liine character. Ten characters are stored in each computer word. Trailing blanks in each line are removed and the last (partially filled) computer word is filled with blanks and the end-of-line character. Thus, each line of text occupies an integral number of computer words.

2.2.4 Filing Module

While the element library contains all the information associated with a set of elements, it is necessary to provide permanent storage for this data. Although a simple disk writing routine might suffice, the prototype has been provided with the capability to store and access several element libraries, collectively known as the disk library (Figure 27). The user may copy the current element library area to the disk library as a named element library. Named element libraries may be loaded, erased, or appended to the current element library.

Ordinarily, the user's first action after the prototype begins execution is to select an element library to be read in (loaded) from the disk. A fixed-length directory is stored on the disk with a name for each element library and a disk address for it.

A copy of this directory resides in main memeory, although changes to the directory are immediately made to the disk copy also. This ensures that when the program or hardware crashes, the disk can be read to recover the most recent version of the element libraries.

At the disk address referenced by an element library name, there is a short record giving the lengths of the element library directory, the element library character string buffer and the element library blocks. These objects

then follow, using as much disk space as required. When a named element library is to be read into the current element library area, the name is designated, the program looks up the disk address, and reads the length information. At this point, more memory is requested from the operating system if it is required, and the data is then read into memory. If an elment library is being appended to an existing element library, then the disk is read into memory following the existing element library area. Within the appended sets of elements, the pointers to the directory and to the character string buffer are incorrect by a fixed offset. A subroutine scans the entire data structure and revises these pointers to correspond to the new locations in the directory and the character string buffer.

When a named element library is to be erased from the disk library, the directory entry for that named element library is simply deleted. When the element library is to be stored on the disk, more disk space is allocated and the element library is copied out. The directory is updated both on the disk and in memory. Garbage collection of the old information is performed at the end of each run but the user has the option of skipping it.

2.3 THE PROTOTYPE GRAPHICS EDITOR

The graphics editor operates on the edit buffer. The filing module copies a block from the element library into the edit buffer. The editor displays the items in the buffer, accepts requests to add or delete items, and displays the revised buffer.

2.3.1 Data Structures Used by the Graphics Editor

This section describes the requirements which must be satisfied, and the resulting data structure. First, the data structure is used for display of the picture; second, the data structure must identify items from graphics input (that is, act as a look-up table) and third, the data structure must be analyzed to form nets. These three requirements, together with the need for easy modification of the data, guide the evaluation of proposed data structures.

The dominant influence on the design is whether the data structure is to be interpreted by hardware or software for display of the picture. If the data structure is to serve as a conventional hardware-interpreted display list for a refresh display, then the alternatives are practically elminated and the other aspects of the data structure are fitted in as well as possible. In the prototype, the data structure was to be interpreted by software and these other aspects strongly influenced the design. Software interpretation is not

altogether bad, however, since it simplifies zooming and providing a large work area. The alternative is to provide several graphics "pages" for a drawing, but the user shouldn't be forced to divide his drawing if he doesn't want to. Ideally, both schemes should be provided.

The second most frequent use of the data structure is as a look-up table from graphics input to data item locations. As before, the hardware available can make a big difference. In particular, if the hardware provides a pointer to the display item detected (by light pen or special tablets equipped with comparators), then the look-up is much easier. In our case, only the x,y coordinates of the item detected are returned. In order to provide the look-up without an exhaustive search, the prototype stores its graphics items in the edit buffer according to the x coordinate of the item. That is, when an item is to be stored, its x coordinate (suitably scaled) is used as the index in the array where the item is stored. When an item is selected by the user, the coordinates are used as the index to retrieve the item. Collisions are dealt with in a manner used by many hashing schemes: an item colliding with another is stored in the next sequential unused (open) location. The storage scheme must also recognize this convention in its searches: beginning at the index for the coordinates given, it scans sequentially until the desired item or an open word is found. If an item is erased, it is replaced by an "empty" but non-open word.

The third requirement for the GMS data structure, to allow a net search, is quite similar to the second, since a coordinate-oriented look-up is involved. If the look-up for graphics input has been performed by the hardware however, as in DIM for example, some auxiliary table is usually required for this search.

In its ease of modification, the structure used by the prototype is superb, since there are no linkages or directories to be updated and neither deletion nor insertion requires existing entries to be moved.

Summarizing our implementation of the edit buffer structure, its advantages are ease of look-up and net searches, and ease of modification. Its disadvantages are its fixed size and its bias toward software interpretation for display.

2.3.2 Implementation of Graphics Editor Commands

The graphics display is divided into three areas: an area for the display of the edit buffer, a menu area (the right hand side of the screen), and an area for the display of status and instructions (top of screen). Commands are initiated by pointing to the appropriate menu entry. For each command, instructions and (possibily) a new menu are provided. The major commands are given below.

1. The first command required to edit a drawing is given

to the filing module, to retrieve a particular graphics block from the element library for editing. The filing module retrieves each item from the block and stores it in the edit buffer according to its x-coordinate. The block is then deleted from the element library. The topological analysis or empirical analysis is also deleted if any exist.

2. To draw a line, one specifies a sequence of points (e.g. by light pen). The first point begins the line and subsequent points are joined to the previous ones to create a joined line segment. If coordinate input is received when no command has been selected, the line command is assumed. This is the only default command. There is a hardware mode which provides a continuous stream of points (from the tracking cross). The software organizes these points into line segments.

3. To enter alphanumeric annotation, one types the character string on a keyboard, and then enters a position via the lightpen. At this time the annotation appears, but it can still be moved about, rotated and changed in size, using an auxiliary menu which replaces the primary menu for the duration of this command. (During this manipulation, only the changing annotation is rewritten in the display hardware, using an addressing capability in the display hardware.) When the user is satisfied, a "confirm" signal is given and the annotation is frozen. It can no longer be manipulated, except by erasing it and creating it again. At this time, the

primary menu resumes.

4. When the user wishes to add a previously defined element symbol to the graphic description of a composite block, the operation is similar to that for annotation. The user is instructed to type the name of the element whose symbol is desired. (Only enough characters for unique identification need be given.) the user is then directed to enter an initial position. The symbol appears here but it can still be moved about, rotated, and changed in size, using an auxiliary menu. When the user is satisfied, the symbol is frozen.

5. To erase an item, the user points to the item and the x coordinate is used to find the item in the edit buffer. The item is blinked and then erased upon confirmation. If more than one item is located at the given coordinates, each one is blinked in turn for the user to select the proper one. A variety of options aid in this selection, including a choice of what kind of item is to be erased, and whether confirmation is required or not.

6. When the zoom parameters are to be changed, the display is redrawn at a magnification factor of one and a square is drawn on the screen outlining the area displayed previously. The square can be moved and changed in size with the light pen until the desired area is within the square. The display is then redrawn with the desired area filling the

screen. An alternate set of commands allows the square to be moved in any directon, in steps of the square width.

7. A grid is provided as an array of dots. The user specifies (by typing a number) the spacing between dots (with a spacing of zero denoting no dots).

8. When the user is finished editing, the items in the edit buffer are copied to the end of the element library and the new graphics block is entered in the directory under the name selected earlier.

2.3.3 The Software Graphics Interpreter

The software graphics interpreter provides the link between the structure used by the editor and the capabilities of the actual hardware (see Figure 30). The prototype uses some simple system routines to generate the actual display commands.

As illustrated in Figure 30, the interpreter scans the edit buffer to get the next item. Lines and annotations are transformed according to the current size, rotation and zoom parameters, clipped and drawn. Symbol references cause the current size and rotation values to be stored and new ones constructed from the size and rotation values in the symbol reference. Then the interpreter scan is directed to the symbol block until the end of the block is reached. At the end of the symbol, the previous size and rotation values are

FIG30

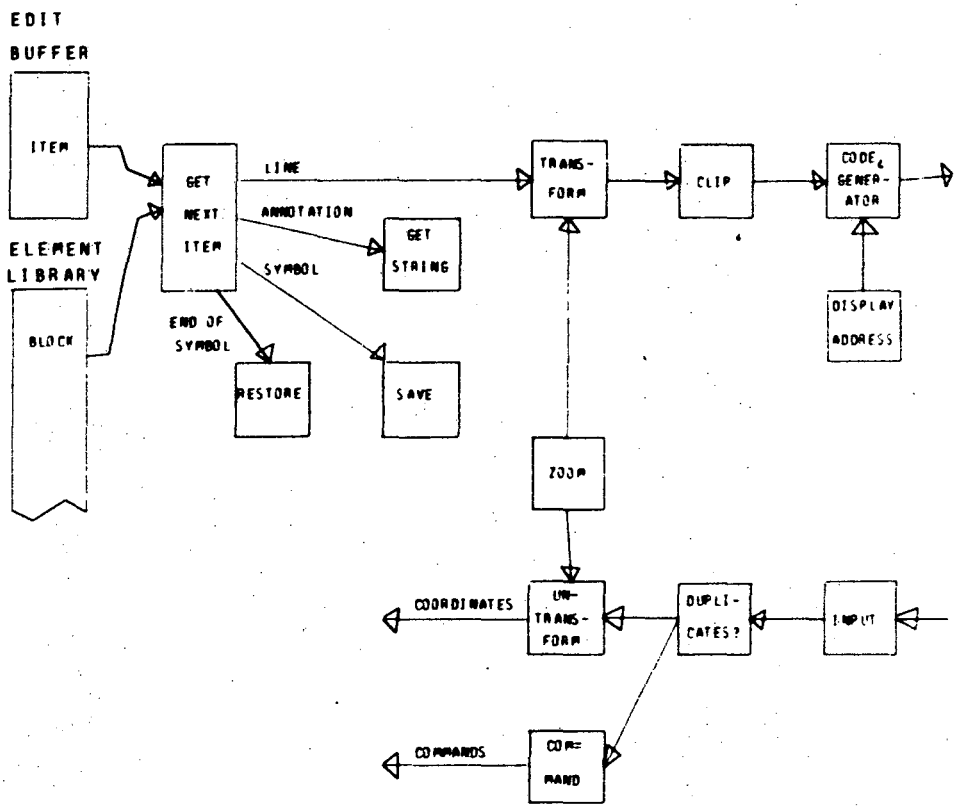


FIGURE 30. GRAPHICS DISPLAY INTERPRETER

restored and the interpreter resumes the scan of the edit buffer.

The interpreter can also be instructed to display a single item. This feature is used when new items are being added to an existing display.

2.4 TEXT EDITOR

The text editor was designed to be the essence of simplicity. To that end, it interacts only with the keyboard, and each line of text is identified by a number displayed with it. When a text block is to be edited, it is moved to the edit buffer and each line is filled out with blanks to be 80 characters long.

The general form of a command is

<string>;<string>;<linenumber><commandchar><commandchar>

where each of these elments is optional. We will use n as a shorthand for <line number>. The default line number is the current one, and the default command character is I (Insert); so the command

<string>

will insert <string> at the current editing position. Other commands are

<string 1> ; <string 2> ; nA

to replace (alter) <string 1> by <string 2> in line n.

; nD

to delete line n. (;D will then delete the next line.)

<string> ; n I

will insert <string> before line n.

$k_1, k_2, k_3, \ldots, k_n$; T <tab char>

will define tab stops at $k_1, \ldots, k_n$ activated by <tab char> in the input. For example, 7;T* followed by *X = Y will place X in column seven.

;nP

will start the display at line n. If more than 35 lines are in the edit buffer

;P

will start the display 35 lines (one screen full) beyond the present starting position recycling from the end to the beginning.

<string> ; Q

will apppend the file <string> to the current text, beginning at the current position of the file.

<string> ; X

will rewind the file <string>.

<string> ; W

will write the current text block on file <string>. X, Q, and W may be combined. For example,

ZAP;XWXQ

will copy the text buffer onto the end of itself using ZAP as a temporary file.

;R

will exit (Return) from the editor. The trailing blanks in each line of text are replaced by an end-of-line character as the lines of text are moved to the element library. The entire collection of lines is stored as one text block in the element library.

2.5 ANALYZING THE TOPOLOGY

2.5.1 Data Structure

The topological data for a composite element is extracted from the graphics block for the macro definition of that element. The topological data structure adds explicit connections between items, which, in the pictorial representation, are only implicit (identical in coordinate values).

The topology of a drawing is represented using the items of a topology block (see Figure 31), whose data structure is analogous to that of the items of a graphics block. Although the graphics items are designed for display, a similar format works well for the topological items.

To represent the topology of a drawing, each net in the drawing is assigned a sequence number, beginning with 2 (1 is reserved for a special case). The topology module then generates a block (Figure 32) in the element library. The first two words contain the number of nets and the number of labels found. Next comes a list of items corresponding to all of the labels in the drawing; these are in alphabetical order. The character string for the label is not duplicated, since the pointer field in the item is copied from that of the corresponding graphics items; thus it references the same string used in the drawing. The rest of the list is composed of groups of items, with each group representing an instance of a symbol within the drawing. A symbol item heads each group to identify which symbol is being used. Attacher point items follow, one for each attacher point on the symbol. Each attacher point refers both to the formal parameter (the character string) and (indirectly) to the actual parameter (net number) to be associated with this formal parameter. The special net number 1 is reserved for those nets which have only one node. These are typlically attacher points which are unused in a particular instance or labels which are used as

FIG31

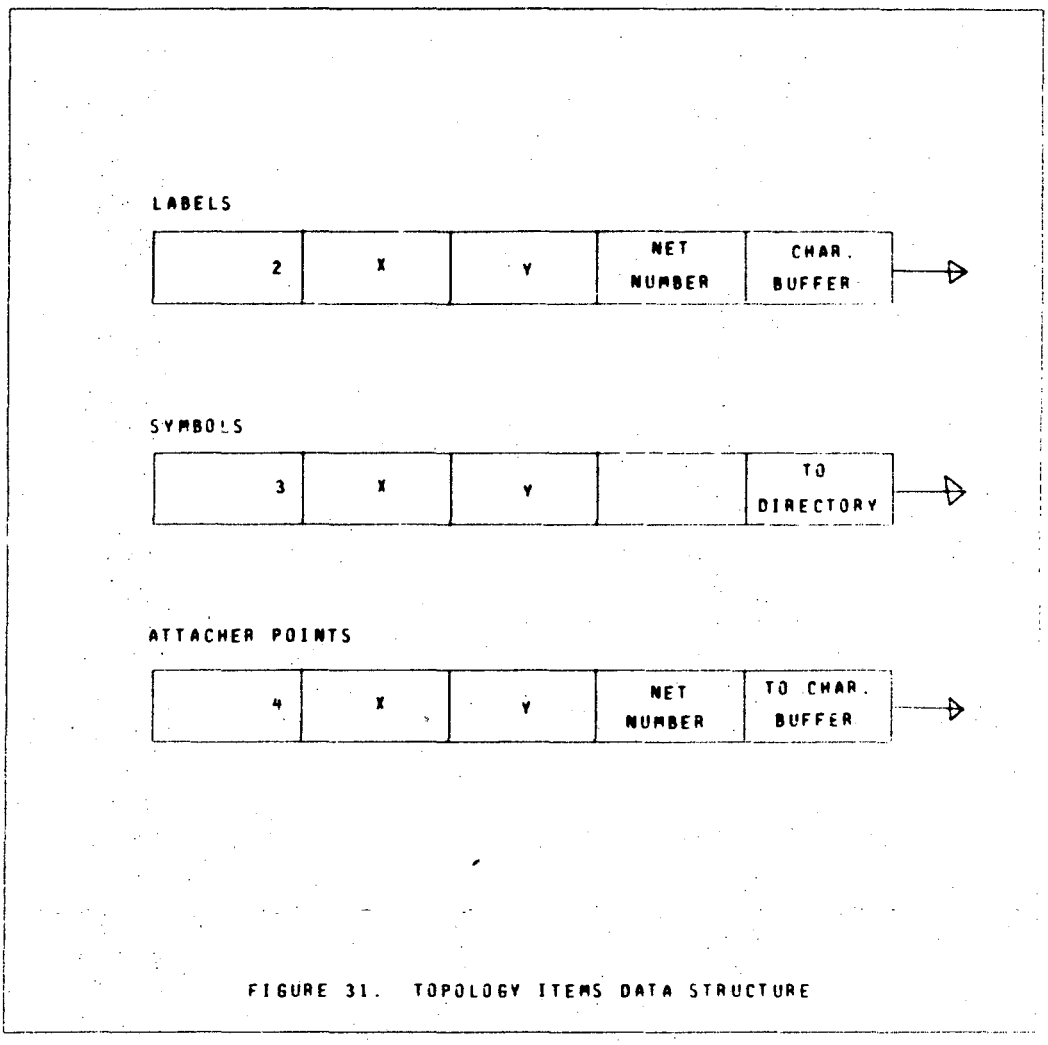


FIGURE 31. TOPOLOGY ITEMS DATA STRUCTURE

FIG32

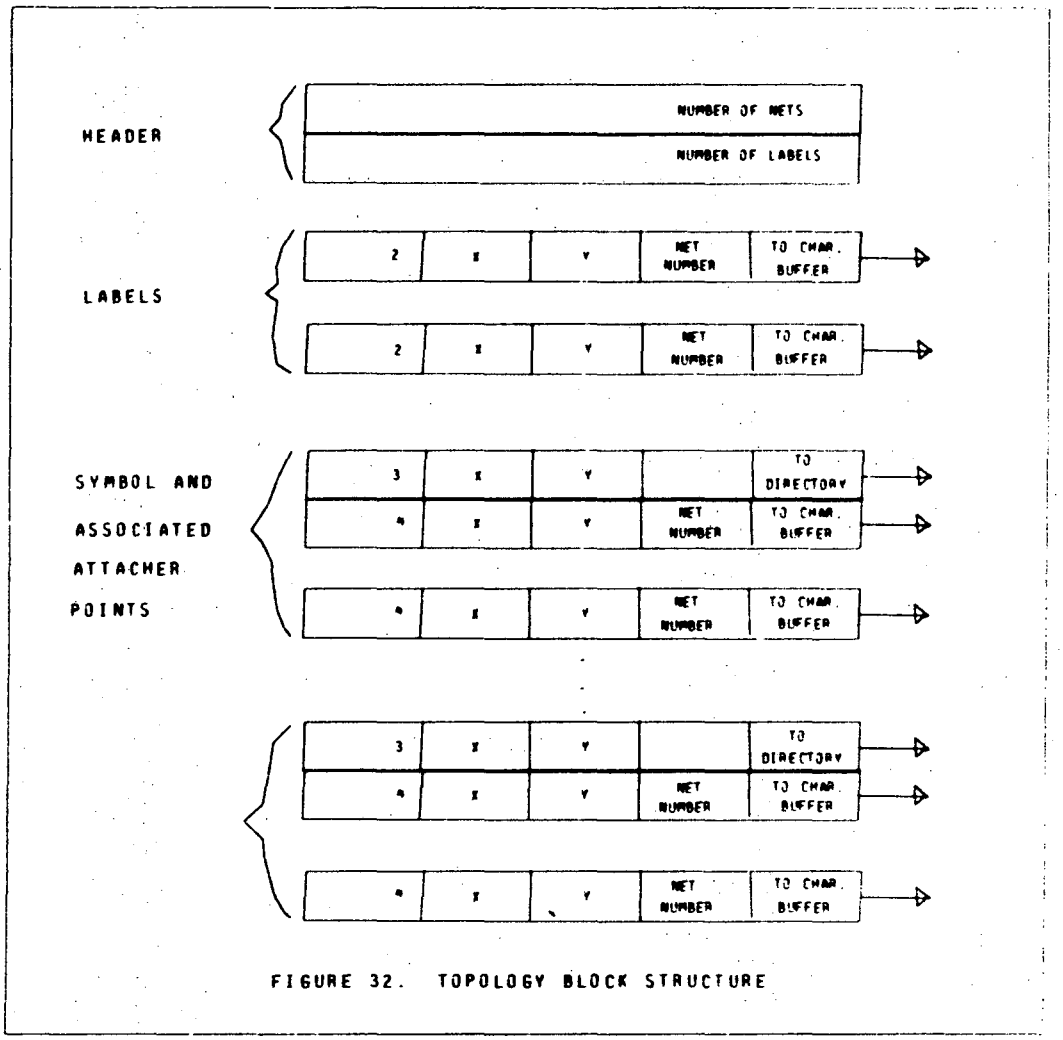


FIGURE 32. TOPOLOGY BLOCK STRUCTURE

comments in the drawing. These nets receive special treatment from the translator.

2.5.2 The Topological Analysis Process

When analysis of a model is desired, the filing module ensures that all composite elements have topology blocks. It scans all the entries in the directory and calls the topology module (TOPO) for any which need a topological analysis. We may note that the filing module erases the out-of-date topological analysis only when a macro definition (composite element) is edited. Thus, most of the topological analyses may have already been done.

To prepare for the topological analysis of a drawing, the filing module copies the graphics block into the edit buffer, indexing each item (according to its x coordinate) into its proper location. TOPO now scans the edit buffer and preprocesses each line and symbol item. For lines, a "reversed" (endpoints interchanged) line is created and stored as a special element within the edit buffer. These "reversed" lines are indexed into the working area at the location corresponding to the x coordinate of the terminal point of the original line. TOPO can now retrieve both the initial and terminal points of a line by indexing.

During this pass, the symbol items are also processed to create the attacher point items. TOPO finds the symbol using

the directory and scans the symbol to find each of its attacher points. Then, using the location, size, and orientation of the symbol just as if it were drawing it, the routine computes where each attacher point appeared on the drawing. TOPO then creates an attacher point item and stores it in the edit buffer at the location appropriate to its coordinates. A list structure which links each attacher point to the proper symbol item is also created. At the conclusion of the first pass, all of the relevant items have been placed in the working area, each indexed by its x coordinate location.

The second pass creates the nets. Starting with the first line, label, or attacher point (from left to right), TOPO assigns a net number to the object and saves its coordinates. If the object is a line, the coordinates of the opposite endpoint of the line are placed on the coordinate stack and the "reversed" line is deleted from the edit buffer. TOPO then searches the edit buffer for other objects which are close (within a certain tolerance) to the current coordinates. If any are found, they are also labeled with the same net number. If the new item found is a line, it is treated as before; that is, the endpoints are again stacked and the "reversed" line deleted. When the search is finished, TOPO examines the coordinate stack for new search coordinates. Ultimately, all the line segments are thus traversed. When a net has been completely traced out, the routine checks the

number of nodes (attacher points or labels) on the net. If only one node was found, that node is marked with the special net number 1. Finally, the net number is advanced and the routine scans for another new item, avoiding, of course, items already having a net number.

A third pass over the edit buffer extracts all the labels and moves them to the topology block, sorting them into alphabetical order.

The fourth pass extracts symbols and attacher point items. For each symbol, all of its attacher points are found and moved to the topology block, sorting them into alphabetical order. The use of an alphabetical sort for the attacher points provides a canonical ordering for them. This permits the translator to produce output in the order in which it encounters the attacher points, taking advantage of this consistency in the order of the parameters.

2.5.3 Other Data Structures

Other model builing programs have typically used much more complicated data structures for the representation of topology. A ring structure is used by many, including SKETCHPAD and CSMP. Several matrix structures were considered for this program, but were rejected as being less effective. The structure chosen was designed so that the translator could make one pass over the topology block and produce the required

output.

Conversion to one of the more complicated data structures, from the data structure as it already exists, is an easy step, because this structure is adaptable as an intermediate form for many other topological data structures. We conclude this section by describing how the present structure can easily be converted into a ring-type data structure. First the array of labels must be enlarged to provide a header item for each ring of labels and attacher points. Each label should be inserted in the array using its net number as the array index. Each header item is given a point to itself. These header items each form a degenerate ring. The array of symbols and attacher points can then be scanned and each inserted in the appropriage ring. For each attacher point encountered, the net number is replaced by the pointer in the corresponding header and the header is updated to point to this item. At the conclusion of the scan, the header will have a pointer to the last item of that net and the pointers will link upwards, eventually returning to the header to complete the ring.

2.6 THE TRANSLATOR

The purpose of an idealized translator is to "propagate semantics", i.e. to incorporate the semantics of component elements into a complete semantic description of a composite element. The translator should provide a variety of schemes

to represent the semantics. In the prototype translator, we have selected a subroutine or macro notation as one of these schemes, yet we have provided considerable flexibility within this framework. This notation provides an interface to a larger number of existing analysis routines which expect unit record (card image) input.

This input must be directly usable without requiring modifications of the analysis routines themselves. This requires a high degree of flexibility in the translator, but we feel that this is justified, for there are a large number of analysis programs which expect card image input and which deal with problems having a diagrammatic representation. These programs are the backbone of present computer usage in various fields, and it is impractical to rewrite them.

2.6.1 Notation

Central to the operation of the translator is the selection of a suitable notation for the description of the hierarchical structure of the model to be analyzed. This notation must also provide for describing the network structure encountered within the model and within subsidiary elements.

For the prototype, the use of a subroutine (or macro) notation is well matched both to the graphics methodology and to the usual input conventions of analysis programs. For

example, it is fortunate that SPICE (see Section 1.4.2) requires each component to be given together with the nets it is connected to. This is entirely parallel to the graphics methodology. An alternative would be to require a list of components connected to each net. That is, the information associated with an element would no longer be collected in a single place. This would be more difficult to provide, since it conflicts with the use of elements as the organizing concept, a scheme natural to a subroutine notation. An advantage of subroutine notation is that it is familiar to almost all users and its use to represent hierarchical relationships is well understood.

We will now describe the detailed operation of the translator in sections which explain and illustrate each of its major features.

2.6.2 Overall Operation of the Translator

The translator treats the model and the elements within it as a complete entity, and its output consists of all the information necessary for an analysis routine. A table of templates specifies formats required. The output is in the form of card images and comprises three files: a control card (JCL) file, a program file, and a data file.

In a first pass, the translator scans the model and compiles a list (a "load" list) of all the elements used

within it. Composite elements are scanned so that all elements which are used (directly or indirectly) in the model are included in the list. The list is then sorted according to the depth of each element within the hierarchy, ensuring that the output of each element will be in the proper order for those compilers and assemblers which require that macros or procedures be defined before they are used.

The second pass of the translator creates the actual output and writes it on the designated files. Each instance of an element in the model (i.e. the top level element) is transformed to a subroutine or macro call notation. The translator creates the call using the name of the element and the proper actual parameters. The actual parameters are created from labels which appear on nets connected to the attacher points. That is, if element X with attacher point A appears as a component in a model, and the net connected to A is labeled B, then the object X is invoked with actual parameter B (e.g. CALL X(B) is created). After the top level model has been converted into card images, the translator must include all of the elements which have been used as components. Each component element begins with a translator generated header card giving the formal parameters, followed by the body of the semantic description: a series of calls if the element is composite, a copy of the text if the element has an analytic definition, or a statement constructed from the table of templates if the element has an empirical

definition.

The translator has created subroutine calls or macro calls as directed by the format table. Now, if they are macro calls, the choice is whether to pass them to the analysis routine for expansion or to have the translator expand them. In the prototype, we choose to have the translator expand them to one level only. This was done for several reasons. First, a single level expansion can be manually forced into a complete expansion, which requires several passes but at least gives a semiautomatic process to enable use of analysis programs with no subprogram or macro capability. Second, this expansion is adequate for simple models having only one level, or for testing the lowest level of a hierarchical model. Third, a single level expansion has some has some of the aesthetic properties of the full expansion: in particular, the output is cleaned up by eliminating the large number of one-line macros which otherwise occur. Fourth, an expansion moves part of the burden from the analysis routine to the translator. If the same model is to be analyzed several times, this can reduce processing time. Note that since the subroutine call or macro call format is only a notation, the same expansion occurs in a model which is to be translated into subroutine notation. Subroutine calls on primitive elements are replaced by the text of the primitive element, with actual parameters substituted for formal parameters.

2.6.3 A Table Driven Translator

To be as complete as possible, the translator attempts to supply many of the details required to make the text output into a fully complete input for analysis routines. These details are supplied by a table of templates. This table is constructed with the text editor. In the prototype, a dummy element is created with an analytic definition to hold the table. When an element is to be translated, the name of the element is given; if a second name is also given, then the second element is assumed to contain a table for the translator as its analytic description. If no second element is given, a default table (the table of MIMIC templates) is used.

In the prototype translator, the controls provided by the table are very simple: an asterisk is replaced by the name of an element, and a left parenthesis triggers the construction of a list of parameters complete with right parenthesis also. The first two templates are a header and a trailer card for the highest level model. For CDC Fortran, typical templates would be

```
PROGRAM * (INPUT,OUTPUT)
END
```

often the header card is used only as a title (see the SPICE example, Section 1.4.2). If it gets in the way, it can often be made to appear as a comment card. Putting the name on this

card is useful for documentation.

A similar pair of templates is used for subroutine or macro definitions. In this case, however, the translator must supply the formal parameters as well as inserting the name of the element. A typical template for a MIMIC macro would be:

```
* BMA(
```

where BMA is the MIMIC mnemonic for begin macro. The trailer template would be

```
EMA
```

For a Fortran subroutine the corresponding pair would be:

```
SUBROUTINE * (
RETURN ; END
```

Note that the prototype translator allows only one line for the trailer (terminator) of a subroutine or macro definition, so we have used non-standard Fortran. A subroutine call template which is analogous to the subroutine header template must be provided, for example, MIMIC's call macro

```
* CMA(
```

or the Fortran CALL statement

```
CALL * (
```

Other information found in the translator directs the

treatment of empirical data. For each set of empirical data, the translator completes a template for reading this data. Each time a particular empirical table is referred to, the translator completes a template supplying the name of the table and the parameters for table look-up. The translator ensures that the data input statements will occur in the same order as the data, and it can specify the length of each empirical table to the data input routines. The input template for MIMIC is "constant function"

```
* CFN(%.0)
```

The percent sign is another special character. In this case it is replaced by the number of data items. Empirical data has not been used with analysis programs other than MIMIC.

The table contains several flags and directives as well as the templates. These are specified on a single card image in fixed fields of 10 characters each. A continuation character and column may be specified for use when the output would otherwise extend beyond column 72. The disk files which are to be used for the text and the empirical data are specified, as well as positioning for these files. The files may be rewound either before or after the translator output is written on them, and an end of file indicator may be optionally written. For example, to use files PROG and DATA, to rewind PROG before writing, endfile it, and rewind DATA after writing, the following specification is used:

PROG/R EOF DATA/BR

The notation used reflects file position options used in the local operating system. The table of templates also specifies an initial character for translator-created names. The translator uses this initial character and appends a sequence number. For most analysis programs, the choice of initial character is of little consequence. Where numeric names are needed, either a number or a blank can be used.

A flag determines the order in which element definitions are written on the translator's output file. A "down" flag will output the hierarchy from the top down, beginning with the model and ending with the primitive elements. This order is customary in Fortran programs. The directive "up" will cause the hierarchy to be scanned from the bottom up, producing output for the model last. For the benefit of most macro processors (notably MIMIC) this mode of operation of the translator ensures that on the translator output file the definition for each macro will preceed its use.

The final portion of the template specification is a set of control cards (JCL) which are written to a system file for execution when the translation is completed. This JCL will typically call an analysis program into execution, handle errors in execution, and return control to the prototype at the termination of the analysis program.

2.6.4 Nets and Labels

The first operation in the second pass of the translator is to assign names to all the nets. To the translator, a net represents a variable which is an actual parameter to the various component elements. If the net is labeled, then the label is taken as the name of the net. If no label is present, then the translator creates a unique name using the initial character specified in the table of options to the translator.

One-node nets are flagged by the topological analysis routine and receive special handling by the translator. They can be either isolated labels or unused attacher points on symbols. Isolated labels are simply ignored by the translator; their function is to help explain the picture and they are not required by analysis routines. When the translator recognizes an unusued attacher point, it searches the character string associated with the attacher point to determine whether a default string has been specified (see example, Section 1.4.1). If no default string is specified, the translator generates a name and supplies it as the actual parameter. This treatment of unused attacher points encourages the creation of more general, more flexible elements.

2.6.5 Details of Element Processing

After names are assigned to all of the nets, the translator scans the list of elements created by the topological analysis. Each element is processed in turn. If macro expansion is not being performed, then the template for subroutine or macro calls is copied to the output. If an asterisk is encountered, the name of the element is substituted. If a left parenthesis is encountered, the list of actual parameters is created. From each attacher point, the net number is extracted and used to find the net name. These names are written to the output, separated by commas and enclosed in parentheses.

If a macro is being expanded, then the element being called is a primitive element (since only bottom level elements are expanded). If it has an analytic definition, then the text of that definition is broken into tokens, either names (strings of consecutive alphanumeric characters) or operators (non-alphanumeric characters). Each name is matched with the list of attacher point names; if it matches, the net number (from the attacher point) is used to find the net name. The net name is then substituted for the name token. Operator tokens are passed unchanged to the output with two exceptions: (1) the concatenation operator is not passed to the output and (2) where a string of blanks appears, blanks may be added or deleted to try to preserve the column spacing of the original analytic definition. This treatment of blanks

facilitates the use of fixed-field languages.

2.6.6 Treatment of Empirical Data in the Prototype

Empirical data is stored in the data structure as a list of coordinates extracted from a "hand drawn curve". There are three phases in the treatment of empirical data: first, the actual data must be placed on a file where it can be easily referenced; second, statements must be placed in the translator output to read the data; and third, routines must be called to locate data values when required.

The prototype provides a complete treatment of empirical data only for the MIMIC analysis routine.

For MIMIC, the translator assembles a list of needed data sets during the first pass of the hierarchy scan. During the second pass, input statements are generated which tell MIMIC to read the data sets. These statements preceed any other references to the data set, and only one statement per data set is provided (no matter how many references are subsequently made to the data set). For each reference to the data set, a table look-up statement is created referring to a data set which has been previously read. After the second pass, the needed data sets are written to the appropriate file using a format specific to MIMIC. The order of the data sets agrees with the order of input statements.

2.6.7 Concatenation and Coordinates

The translator has two special features which have been useful in some instances. At present, these features are implemented only for analytic definitions, but the extension to labels within macro definitions is straightforward.

The ability to concatenate two or more strings has been provided by the use of a special concatenation character, the right arrow (Section 2.6.5). This character serves as a break character, but is not copied to the output. Thus, two strings may be concatenated. This features is useful primarily when one of the character strings is a formal parameter. The translator will first substitute an actual parameter for the formal parameter and then perform the concatenation, forming a new name from the net name supplied. This technique has been used to form a whole family of names from a single net name (see Section 1.4.4). Used with care, this allows a number of variables to share a single line in the model; that is, a single line can be made to represent the flow of several pieces of information when the analysis routine does not allow arrays to be used.

Another use for the concatenation facility is to prefix (or suffix) a character to strings, where a particular character has meaning to the analysis routine. (see the SPICE example, Section 1.4.2)

A second useful feature is the option of using, within a

composite element definition, the actual coordinates of a component symbol attacher point. If the attacher point name in an analytic definition has the characters #X or #Y appended to it, the translator will retrieve the actual X or Y coordinates of the attacher point for the particular instance being expanded. In this way, parameter values are derived from the position on the diagram rather than from labels. A mundane, but very practical example of this technique is used in circuit board layout. In this case, definitions are contrived to print out the coordinates of pads used for connection to circuit elements. A paper tape to drive an automatic drill is generated directly from these coordinates.

At this point, it is well to note that this technique is only a glimpse of the problem of preserving graphical information in the topological data structure. This technique represents only an ad hoc answer to a specific need, and does not derive from a general approach to the eventual solution of this problem. The real goal, in fact, is to allow reversal of the analysis, including error messages or even new graphics configurations, to be presented in the context of the original input.

3. AN EVALUATION OF THE PROTOTYPE GMS

Our evaluation of the prototype GMS can perhaps best begin by comparing it with the idealized GMS. The idealized system describes a single man-machine problem-definition interface for use in a wide variety of problem solving disciplines. The prototype has demonstrated such an interface for a variety of analysis routines. The idealized methodology for creating elements from two types of primitive elements and one compound element has been verified by the prototype. The idealized system has proposed a topological interpretation and a hierarchical structure. The prototype has demonstrated one feasible solution to the analysis of topology and the propagation of topological structure through a hierarchy of models. It has shown how simple manipulations on character strings provide consistant resuls without concern for whether the character strings are formal parameters, variables or constants.

In addition, the prototype has demonstrated a feasible notation and translation method for this notation which enables the conversion of internal structure to card images in a flexible way.

The prototype GMS has demonstrated that these elments of a modeling system can be provided at a reasonable cost. The programming time required for the prototype GMS was approximately 12 man-months. We estimate that adding similar

facilities to a single applications program would have required six to eight man months and would not be any cheaper to use. Thus, if two or more applications can use such graphics facilities, then it is economically sound to program a graphics facility.

Based on our experience with the prototype GMS, the next sections describe user reaction to the prototype, a survey of analysis routines available to the prototype, and improvements to the prototype GMS that are possible within its present structure.

3.1 USER EVALUATION OF THE PROTOTYPE GMS

In attemption to evaluate the prototype GMS, we undertook a survey of opinions and experience of as many of the users as could be located. A questionnaire was prepared and circulated, the results of which are in Appendix A.

Most of the users of the prototype were employees of the Lawrence Berkeley Laboratory, and a few were students at the nearby Berkeley campus of the University of California. Most of them had programmed or used a computer. In addition to an informal open invitation to the Laboratory staff, a few staff members were approached and asked to provide test problems. These members were aided by the author and others in setting up and solving their problems. Most of these users were in the Electrical Engineering Department of the

Laboratory. The majority of these problems were in digital logic design. Since a simulation or analysis program for digital logic is not currently available at the Laboratory, these problems were solved by simulating the digital logic with an analog simulator. The substitution was successful, however, and several limited size problems were solved.

Users with problems in analog simulation were quite pleased with PICASSO. Mark Horovitz [HOR072] provided an unsolicitied evaluation in one of his publications, in which he described a biological model and the modeling facilities of LBL. He made the following remarks:
[Although the author is listed as a co-author of this paper, the opinions expressed are entirely those of Mark Horovitz.]

Evaluation of the System

How easy it is to construct models by using the PICASSO program? Skill is required to choose and define the primitive elements so that they yield neat, natural buildling blocks for a class of models. Once the primitives have been defined, model structures can easily be built. The library storage facilities for graphics models are very convenient. To illustrate some of the features of the system, let us suppose that we want to give a user an introduction to compartmental models. We can take a one-compartment model out of the library, analyze it - examine the equations produced in the analysis phase, execute a simulation run - examine the results, change the parameters, and run it again. Then we can pick a two-compartment model and go through the same cycle. Next we could build a model of real interest to the user or look at more complete stored models - all in one session at the console. Starting out in this way, the new user does not have to spend a great deal of time learning about the system before being able to tackle problems of interest to him.

Extensions

It is easy to accommodate analysis of other languages, and we expect this aspect to proliferate. It seems to me that this should be encouraged, provided a processor for the language is available on our machine. If one is aiming for ease of use, then a new PICASSO user with some experience of modeling, using for example, GPSS or DYNAMO, should be encouraged to continue by generating PICASSO models which are executed via GPSS or DYNAMO.

In the present system, if I wish to construct a model with the same functions and analyze it via either MIMIC or FORTRAN, I have to generate two sets of definitions and names. In other words, I may have one visual representation for the model, but I need two sets of names. Example: for an adder I could have ADDERF with a FORTRAN definition, and ADDERM with a MIMIC definition. A later version of PICASSO will permit multiple definition of symbols. It is also hoped to add features so that animiation of diagrams will be made easy.

Several students did classwork and other projects using the GMS prototype, including class assignments in Engineering 111, that most students do using CSMP, a less powerful system on the Berkeley campus.

It is clear that GMS will be successful only when applied to a suitable problem; that is a problem usually described symbolically and for which an analysis procedure exists. It is helpful if the analysis procedure is currently in use. In this case, there is little difficulty in showing users how to use GMS. On the other hand, our conversations with users revealed that designers did not always design as we expected. For example, a digital logic designer related that he designed with boolean equations and let the draftsman develop the logic diagrams. He still relied on the logic diagram to some degree however, especially for problem areas. The designer of

control systems may prefer to work directly with the poles and zeros of the desired transfer function rather than with a block diagram. This does not mean that GMS is not applicable; it only means that we might do well to direct our attention to the draftsmen as well as the engineers. Indeed, PICASSO has been used successfully as a drafting aid for logic diagrams and for analysis of these diagrams.

In general users felt that the graphics interface was adequate, although clearly not optimum. The author concurs and work is under way to improve it (Section 3.3). A significant part of the frustration is due to the hardware and operating system. When the host, a large batch processing system, is lightly loaded, response is excellent. When the system is heavily loaded, swapping delays are a significant source of frustration. An intelligent terminal is an obvious cure for these problems.

The cost of using PICASSO varied from $10 to $20 per hour of terminal time. Connect charges are approximately $6 per hour; the remaining charge is primarily for I/O and reflects the variations in working speed from novice to very experienced users. At $10 per hour, a GMS is cost effective at practically any task which it can perform. Only two users (out of ten) found alternate systems easier or cheaper. In the first case, inadequate documentation was cited as a difficulty in using PICASSO; inquiry revealed that the user did not have a complete set of the available documentation.

The second involved the use of PICASSO as a graphics editor only; for this application an editor was written which was cheaper to use. More than six months of daily use were required to amortize the cost of the new editor, however!

In summary, users were excited by the prospects inherent in PICASSO and impressed by its capabilities. The main complaint related to reliability problems in the hardware and the operating system, subjects which are beyond the control of the GMS designer. Users were also of the opinion that the interface lacked polish, although after several hours experience, they felt comfortable with it. Finally, my own strongest feeling as a user was that the operating system was poorly matched to the very high I/O rate required for interactive drawing and manipulation. An intelligent terminal or an intelligent concentrator should be provided to buffer these interactions into larger chunks.

3.2 A SURVEY OF ANALYSIS ROUTINES

In this section on supporting software, we will describe the analysis routines which have been interfaced to the prototype GMS or for which an interface would be useful and easily constructed. We will describe the analysis systems, what problems they are suited to, and the effectiveness of the GMS-analysis system combination compared to manual preparation of input. We also give the translator template used with these systems, and any problems encountered. Finally, we will

describe the operating system support which the prototype GMS uses.

The analysis systems used or proposed for use with the prototype GMS include MIMIC, a continuous system simulator, SPICE, an electronic circuit analysis program, a wirewrap tape generator, GPSS, a discrete event simulator (not described), and language compilers, e.g. Fortran, ALGOL, and QUEL, a lower level interactive language.

3.2.1 A Continous Systems Simulator: MIMIC

MIMIC, the first analysis system to be interfaced with the prototype GMS, is a simulator of continuous systems which accepts a set of equations and evaluates them iteratively as a function of an independent variable (time). MIMIC is used primarily in the solution of differential equations. MIMIC is most valuable in cases where the behavior of each element in a problem is known and the behavior of a collection of elements is to be simulated. The equations describing computation for each element are entered by the user, as formulas that can include functions available in MIMIC. MIMIC sorts the equations into an efficient computational order by placing computations which yield an intermediate result before computations which use that result. This sorting and the special treatment of integration, time delay and other special functions gives the impression that MIMIC evaluates all the equations in parallel.

Almost any MIMIC problem which is visualized as a diagram is well suited to using GMS for input preparation. The first use of GMS for a particular problem is often more work than coding the problem directly in MIMIC's input language. This is due to the extra work required to create primitive elements suitable for the problem. If primitive elements are already available, then using GMS is significantly faster than coding the problem directly in MIMIC's input language. In addition, the GMS version is much more likely to be free from syntax and typing or coding errors. Variable names are particularly prone to spelling errors and GMS is a great help in this area. In GMS, a variable name is given only once by the user and is propagated, always correctly, by the translator. When a model is to be modified, making the changes graphically is much faster than making the corresponding changes via MIMIC's input language. GMS makes it easy to group elements and to work with these groups as single units. Also important is having GMS propagate changes in the semantic description to all places where it is used.

The first step in using the translator for MIMIC diagrams is to become familiar with the input format for MIMIC statements. MIMIC uses fields beginning at columns 1,2,10, and 19 of the input card images. The editor has tab stops which aid in putting text in the correct fields but the translator does not check this. The template for the translator is then created, using the documentation for

guidance.

There were two problems in using MIMIC with the generalized translator. The first problem is that MIMIC does not allow statements to extend beyond a single line. In a few cases, the substitution of long strings for short ones caused a line of text to get too long. In these cases, long formulas were rewritten as two short ones. The second problem is that MIMIC allows only 6 formal parameters in its macro declaration. Extra parameters must be declared on succeeding lines with a different keyword. We did not think that this syntax was worth implementing as a generalized facility, so the translator was modified internally to handle this case.

```
$IAM        MIMIC RUN OF * PRODUCED BY MIMVERT
                    END
                *   BMA(
                    EMA
                *   CMA(
                *   CFN(%.0)
                  DATA/R     NOEOF      DATA      G      UP

TIM.
REWIND,DATA,MIM.
SFL,70000.
SCP,A=100.
MIMGO,DATA,MIM.
EXIT.
CXIT.
FIN.
SCP,A=0.
COPY,MIM/RBR,OUTPUT.
REWIND,DATA.
SFL,55000.
DRAW.
```

3.2.2 Electronic Circuit Analysis: SPICE

SPICE was selected as a typical electronic circuit analysis package for a number of reasons. First, the input format is relatively simple and consistent, yet it is quite typical of those used by the majority of circuit analysis packages. Second, it produces both frequency domain and time domain response, thus serving a larger group of potential users. Finally, it has some graphics output routines.

The input to SPICE is a list of the circuit elements, their connections and values. SPICE then analyzes the connections to construct differential equations (Section 1.2.5) which give the behavior of the circuit. These are then solved numerically to provide the desired response. SPICE is well suited to small-signal analysis of circuits with 40 to 50 elements or less. This will encompass two or three stages of a typical amplifier, but not an entire piece of electronic equipment.

In the case of SPICE, the primitive elments are trivial to construct. The typical user knows exactly what the symbol for a resistor, capacitor, etc. should be, and the analytic definition for these elements is obvious from the SPICE documentation. There is a time saving even for the first application, since the element library is so easily constructed. When a circuit is to be changed, the user is far ahead with GMS. Although some changes are easy to manage in

SPICE's input language, such as adding or removing a component, other changes are rather difficult. For example, if two nodes are combined, all of the components for one of the nodes have to be changed to have the other node number. This tedious and error prone operation is automatically handled by GMS when the corresponding change is made graphically to the model.

Setting up the translator for use with SPICE is quite simple. SPICE does not support macros or subroutines, so the user does not have this detail to worry about (but hierarchically deep circuits are not very practical). SPICE identifies the type of each component by its first letter, so the concatenation facility is used to append the correct first letter to the component names. The translator provides for setting the first character of translator-provided names (Section 2.6). This character is set to a blank so that these names will be numeric strings as required by SPICE. The control cards (JCL) required are taken from the SPICE documentation and modified slightly to return control to GMS at the termination of SPICE.

```
SPICE RUN OF *
.END
```

```
        DATA/R      NOEOF      DATA                UP

REWIND,DATA.
SPICE,DATA.
PTSS,D.
```

```
CXIT.
EXIT.
FIN.
DRAW.
```

3.2.3 Wirewrap

A wirewrap program accepts a wiring list for electronic circuitry and generates a tape which drives an automatic wiring machine. The primitive elements for the wirewrap program are modules with various numbers of pins, e.g. gates, flip-flops and other components. The symbols used for these elements are similar in shape to the actual physical modules. Each module has an analytic definition which, when translated, lists the module type, name, location, pin numbers and signal names. Macro's or subroutines are not required, so the translator is quite easy to set up.

The only problem encountered was that the input for the wirewrap program must be sorted by signal name. Since the translator does not have this capability, a conversion routine was written locally. The conversion routine is straightforward and is written in SNOBOL, although a standard sort routine would be more efficient.

The effectiveness of this experimental project was questionable, since the user created a diagram of the actual pin arrangement and wired that from the circuit diagram, rather than reproducing the circuit diagram in GMS, from which the same information could have been derived. There were no

problems in this application, since the conversion module was designed with the translator output in mind.

3.2.4 Language Compilers

Another class of analysis systems is represented by the language compilers, such as Fortran, ALGOL, and PL/1. We have experimented with the prototype GMS as a flowchart analyzer. It is not adequate to handle all of the details of a language such as Fortran in a convenient manner. While it is easy to set up the translator to produce statements in a desired format from a drawing, there are several problems in the overall approach. First, there is no mechanism to generate DIMENSION and other declarations from an analysis of the flowchart. This can be circumvented by requiring the user to include declarations in the flowchart. Second, the graphics handling of DO loops and related control structures is difficult since the contents of the DO loops and control structures can vary so much in size. This problem will decrease as experience is gained. Variable sizes for symbols could be used to good advantage here. Finally, there is no mechanism for ordering statements. Our experiments have used GO TO statements at the end of every statement to control the flow of execution. This is inefficient and results in a program which is impossible to read. Nevertheless, we have produced some experimental programs which have compiled without error. A diagraph analysis could easily be used here

to generate an efficient statement order.

```
      PROGRAM *(INPUT,OUTPUT,FILM)
      END
      SUBROUTINE *(
      END
      CALL *(
      DATA */
     *     DATA     NOEOF     DATA     9     DOWN
```

3.2.5 Lower Level Interactive Languages: CUPID

Another area of language processors with considerably more promise is the use of GMS to produce input for lower level interactive languages. In particular, CUPID [MCD075] is a system using GMS to produce input for QUEL [HELD75], an interactive data base inquiry language. Since the relationships between items in a data base are often described by use of a diagram, GMS is well suited to conversion of this diagram directly into input for a lower level inquiry language. This experiment has been quite successful.

3.2.6 Operating System Support

The operating system support which is used by the prototype GMS is commonly found on most medium to large systems. In particular, some features the prototype uses are (1) dynamic memory allocation to vary the size of its operating partition, (2) random access to disk storage, (3) operating system graphics modules, and (4) the ability to change the job's control record to specify execution of

analysis routines after GMS terminates. Abilities (1) and (2) are provided on almost all systems with perhaps (1) being provided by virtual memory. GMS uses the system graphics modules at a rather low level: the graphics modules used allow one to draw lines and characters, overwrite part of a display if it is a refreshed display and to read coordinate input from various hardware devices. The job control facilities provided by the operating system allow a job to change its own control record. This allows another program to be executed with control returning to GMS as directed. This is similar to spawning another process, except that the new process runs sequentially with GMS rather than in parallel with it. This feature is necessary to provide a satisfactory measure of control and simplicity in operating the analysis programs from within GMS. It allows the job control sequence for each analysis program to be stored within GMS.

3.3 IMPROVEMENTS TO THE PROTOTYPE GMS

There are many improvements which can be made to the prototype GMS within its present structure, that is, as incremental changes, without requiring a redesign of the entire system. These suggestions are not so much criticisms of the system as it stands, but, rather the insights of hindsight and plans for continued development.

3.3.1 Modularity

In several areas of the prototype, the code is not ideally structured. Better structure could be achieved by greater attention to control structures and the assignment of functions to modules. This would facilitate comprehension, maintenance and modification. A specific benefit would be the easier conversion of parts of the system to use new hardware, e.g., an intelligent terminal.

The most difficult design problems involving modularity of software are in the interface with the data structure. Since the components of the data structure are related in ways which must be recognized by individual modules, it is not possible to isolate all the modules from the data structure. The proper approach is to provide sub-modules for each of those functions which require an interface with the data structure. For the graphics editor, for example, such sub-modules would include (1) displaying an item or several items from the data structure on the screen, (2) identifying an item from light pen or other input, (3) adding or removing an item from the data structure, and (4) composing items from their constituent fields or decomposing items into separate fields. Another set of sub-modules could be used in the editor to display commands and to interpret command input. These sub-modules can easily be combined to get the graphics editor. All that is left is to create the menus of commands, use a sub-module to display and interpret them and a giant

"case" statement to accomplish each command. Each command requires only the collection of the necessary parameters (from an I/O sub-module), which are then passed to the data structure sub-modules. In this way, the details of command display and interpretation are separate from details of I/O and also from details of the data structure.

3.3.2 Improvements to the Graphics Editor

In the graphics editor, we would revise the treatment of character strings to store them with the rest of the display items rather than in a separate buffer. This would both eliminate the danger of overflow for large element libraries and reduce the space required for small ones. The storage of symbols would be changed so that a line segment which began at the end of another segment would be displayed consecutively if possible. This would reduce both transmission time and flicker.

A menu of symbols would be provided, so that the user could choose them with the light pen rather than typing in the name of the element. Presently, the user must remember the available elements.

We would allow global scale changes on drawings; an entire drawing could be enlarged or reduced by some factor. This is especially important for the novice, who often finds that his initial set of symbols was too large or too small.

The entire drawing could also be translated to allow more room at some side of the drawing.

More control over the character string display can be provided. At a minimum, it should be possible to delete from the display the attacher point strings associated with symbols in a composite element.

A more drastic change is the use of an intelligent terminal to implement the graphics editor. The use of intelligent terminals is motivated by a desire to improve response time, save money, and bring the graphics facility closer to the user. Response time is improved by providing immediate response without waiting for transmission time to the host computer or for scheduling delays within the host computer. Money is saved by reducing computation at the central site or timesharing facility. In remote areas (i.e., user areas) a high bandwidth connection is not feasible; a non-intelligent terminal could not provide adequate response. To achieve significant bandwidth reductions, a substantial part of the data structure must be moved to the intelligent terminal.

3.3.2 Topological Analysis

For the topological analysis, we expect that schemes for the incremental compilation of topology will be investigated. These schemes will use more elaborate topological data

structures, in exchange for a reduced computation time. The more elaborate structures are needed to preserve temporary information which is now discarded when analysis is complete and recreated for each new analysis. We also expect that the data structure describing the topological analysis will be improved. The topological analysis will also incorporate connectivity checking, either from auxiliary user supplied data or from a data set supplied as part of the analysis package. It will warn the user of non-connected or ilegally connected lines.

The topological analysis would be revised to include a graphical "symbol (name) table." This would have many advantages (see 2.6.7); for example, it would enable error messages generated by analysis programs to be related to the graphics structures which caused them.

3.3.4 Translator Improvements

Development of the translator will be guided by the recognition of parallels with assembly language macro processors. These parallels will provide guidelines for a more consistent control specification, for incorporating new facilities and also for constructing the software itself. The translator needs to have a better syntax and semantics for the control of the translation process. It also needs to be made recursive and given better facilties for incorporating data sets into models. We would experiment with an output notation

more suitable for computer interpretation, perhaps a sequence of pointers comprising one of the more common list structures. Beyond this, the translator will become a data base inquiry and conversion routine which provides a conversion from the topological data structure into the exact structure required by any particular analysis routine. Eventually, such a data structure conversion module will be separated from the GMS and will be viewed as a general tool for the development of analysis routines rather than as a special purpose part of the graphics interface. It will be used wherever needed between analysis modules, not just between the graphics system and the input section of an analysis module.

3.3.5 Operational Improvements

The prototype GMS also needs some operational improvements to make it easier to use. The most important of these is the automatic preservation of the element library when the program terminates to execute an analysis program. The elment library would also be automatially restored as part of the program initialization. These operations must be done manually at present. The startup setting for the zoom parameter should also be changed to a more useful value.

Some extra commands should be added so that users could avoid following the full hierarchical command path. In particular, a single command should switch back and forth between the symbol and the semantic description of an element.

A command should also be added to the graphics editor which would store the current element, translate it, and start the analysis routine all in one operatin. When the analysis is finished and control returns to GMS, the initialization procedure could return to the graphics editor with the element just analyzed, ready for any changes.

Finally, some on-line help and tutorial commands would be useful.

4. FUTURE CAD SYSTEMS

4.1 WHERE WE ARE NOW

This chapter explores some of the problems inherent in the continuing development of CAD systems. It is clear that the need for CAD software is growing faster than the supply of programmers and funding for software developmment. CAD software development must therefore rely increasingly on techniques for increasing programmer productivity. We feel that the term "structured programming" [DIJK72], is applicable to almost all of these techniques, although various authors have used the term in more restricted contexts.

4.1.1 Types of Existing Systems

We can divide CAD software into three general areas: data structure and data management techniques, computational techniques, and user interface techniques. Examples, are described which are effective in each of these areas.

Architecture is a design area in which data structure and data management techniques predominate. These systems typically have a rather large data base, but only modest requirements for complex computations. There are single architecutral systems which have subsystems for the design and checking of space utilization, structural details, architectural aesthetics, bills of materials, and building

codes. A variety of subsystems are required to support the multitude of overlapping considerations which influence such a design. The architectural designer may change rapidly from one design aspect to another in this way. For example, he may change a room, then check the new space utilization and view a perspective drawing; change some structural details, then see how costs are affected and check for compliance with the building codes. To support this switching from one subsystem to another, the overall architectural CAD system must be modular and it must have a very general data structure and data management facility.

The predominant type of CAD systems are those which are used for their analytic capacity. Examples of such systems are NASTRAN, for structural analysis, TRANSPORT, for accelerator magnet design, and SPICE, for electronic circuit design. These systems operate on modest amounts of data (from a data management point of view), so they have tended to use data structures formulated to facilitate the required computation. Another characteristic of these systems is that they are very specific; they concentrate very thoroughly on a very small problem area. The algorithms used by these computational systems show that they have a good theoretical framework. Nevertheless, they recognize a large number of special cases, often at substantial cost in software. NASTRAN, for example, recognizes beams, plates, cylinders, and many other shapes.

Interactive CAD systems are rapidly moving from an academic to an industrial environment. While the data capacity and the computational capacity of these systems has been modest, industry is finding that in many cases an interactive facility is cost effective. The GMS described in this work, together with one of the analysis routines forms a cost effective combination for many problems. While the ease of use of such a system will often encourage more analysis and hence more computing, the time saved by the user will usually more than compensate for the added computation cost.

4.1.2 Trends in CAD Systems

As CAD systems grow in scope, they must incorporate methods and techniques which are increasingly general if they are to increase in scope without a corresponding increase in complexity.

Data structures which are formulated to meet the needs of some particular computation are usually optimum in terms of computer utilization. They are usually not optimum in terms of programmer utilization. As software development takes an increasing portion of CAD project cost, the best course will swing away from tailor made structures toward general techniques. In data structures, these general techniques are evident in the well established computer utility systems for data base management. Generalized approaches to data structures are also being provided by improved programming

languages, in which the compiler assumes the detailed management of an increasing variety of data structures.

In the computational area, generalization will occur as special cases are combined into a common theoretical framework. This combination will be a result of work in the individual disciplines and mathematics in general. As these generalizations are incorporated into systems, new special cases will constantly arise; we can rest assured that special cases will constantly be with us as long as CAD systems grow in applicability.

Systems in which data base inquiry and report generation predominate also have a need for more complex analysis facilities. These systems will adopt computational techniques from work in analytic areas.

Interactive facilities are among the least well developed aspects of CAD systems. While a great deal of research has taken place on man-machine communication, little of this research has filtered down to the level of the average applications programmers where it can be applied as simple, convenient subroutine packages. Subroutine packages to simplify the control of the man-machine dialogue are becoming available. These include primarily syntax checking, and error handling. To go much beyond this, the content of the data structure must be available to these interfacing routines. This has not happened in a general way yet.

The software tools for continuing development of CAD systems includes both subroutine packages and compiler development. The two most critical areas seem to be in data structures where compiler development would be most useful, and interactive techniques, where software packages seem to be the best first step.

4.2 GOALS FOR FUTURE CAD SYSTEMS

The goal of a CAD system is to enable a user (at a console, perhaps) to perform some manipulations upon his data without understanding the layers of computer software between what he sees and what is actually going on. To support this capability, the first requirement is for data structures appropriate to the user's view of the world. A second requirement is for manipulative routines to perform the desired transformations, analysis and synthesis. A third requirement is for user interfaces to provide controls for the manipulation routines and display of the data.

Viewed in terms of computer software, we can add another requirement for implementation tools, (e.g., compilers, subroutine packages) which will allow such systems to be conveniently built. In this section we shall discuss each of these requirements in turn to develop some idea of the extent and depth of each topic. In Section 4.3, we shall offer some ideas about the necessary elements of the system design.

4.2.1 Data Structure Goals

A data structure should reflect the structure of a problem, allow efficient manipulation of the data, and provide rapid access to the data. When we say that a data structure should reflect the structure of a problem, we mean that it should be easy to interpret the data structure and provide a display which is readily understood by the user. This display should allow him to easily visualize the results of various manipulations which he may perform on the data. For example, consider a program to compute income tax. A good choice of data structure might be one which is parallel to the IRS form with which we are all familiar. Within a category called Identification, we have Name, Address, Occupation, and so forth. A category called Deductions will contain sub-categories Medical, Taxes, etc. The Medical category can be further broken down into Drugs, Doctors, Hospitals, and others. The user is now able to make statements such as: subtract Total Deductions from Income. The information in the data structure should be displayed in a way which makes clear the results of such operations.

Data structures should be flexible to support efficient manipulation. Even within a single discipline, a variety of structures may be required. For example, a cartography application might require the storage of a map. An array of boundary points of each area would be suitable for cloropleth mapping (shading) applications but unsuitable for determining

which areas are adjacent, as might be required in neighborhood analysis. A structure suitable for finding adjacent areas may, in turn, be unsuitable for storage of individual boundaries.

Older programming languages, (e.g., Fortran, Algol60) have been limited in the variety of data structures which they support. They have provided only the simplest language constructs and data structures. The programmer has been forced to build his own superstructure using the compiler provided structures as primitives. Newer languages (e.g., Algol68, Pascal, EL1) have provided not only a greater diversity of structures, but also tools for more conveniently creating new structures. Languages with extension facilities can also provide a compact notation for describing operations on these new structures. Currently, these newer languages are not as efficient as the equivalent programmer defined structures using the older languages. As compilers improve, this inefficiency will be accepted as a compromise which improves overall programmer productivity. Many operations could be written more compactly using such a facility. The programmer could concentrate on data manipulation without wasting programming effort on the manipulation of high level structures using primitive operations meant for arrays.

Currently, a program is composed partly of manipulation of a data structure mechanism and partly of implementation of such data structures, the two parts being so mixed together

that it is sometimes difficult to separate them. It has been said that a revised algorithm can produce order-of-magnitude increases in program speed, whereas re-coding a problem rarely improves the speed by more than a factor of two. Almost always, such a revised algorithm depends on a revised data structure which allows more efficient computation. In many cases, the newer languages should allow the replacement of data structure and operator definitions within an algorithm without recoding the entire algorithm.

What we have described above is an access structure, which is what the user/programmer sees. At a lower level, there is a storage structure onto which the access structure is mapped. The choice of a storage structure depends on its compatibility with the access structures, the speed of access and the relative compactness of the storage scheme. It is quite possible that two schemes with the same access structure could require different storage structures due to varying amounts of computing, different frequencies of mass storage access, or different sizes of the data structures. A single access structure or notation should be allowed to manipulate several physical structures, for example, the same notation for sequential access could apply to either vectors or lists.

It would also be useful to allow more than one logical structure or notation to be used with a particular physical structure. A representation of a street map, for example, could be manipulated either with a logical structure of

streets and intersections (arcs and nodes), or with a logical structure of street names (vectors). An important start in this direction has been made by schemes for multiple keys access to items with several attributes.

4.2.2 Analysis Techniques

Analysis routines supply the brains of a system. These are the algorithms which the programmer supplies for the manipulation of the data; they are controlled by the user interface routines. There are many features which make an analysis routine convenient to use: accuracy, speed, documentation, and error control. Most of these algorithms perform an operation which can be defined mathematically. Accuracy is a measure of how little the results deviate from the mathematical definition. Inaccuracies are caused by the finite precision of the hardware and the approximations necessary for efficient computation.

Speed is a requirement which usually conflicts with accuracy. Speed is obtained primarily by using the fastest algorithms and secondarily by carefully coding the algorithm.

Documentaion must explain all the aspects of a routine covered here: accuracy, speed, error control, and other modules required, and it should give the algorithm used.

To achieve error control, a routine must validate its input and monitor the algorithms used to ensure their correct

operation; errors must be reported back to the superior routine.

4.2.3 User Interface

The user interface is the most important part of a CAD facility, both in the amount of effort required and in its impact on the user. The user interface must instruct the user in the manipulations available within a particular program and it must carry on a dialogue with the user during operation of the program. Documentation is part of this user interface; it is unreasonable to expect all aspects of a program to be explained as part of its operation. This documentation should be the primary reference material for a program. The user should never have to consult a "listing" of the program. The documentation should explain the domain of applicability, the algorithms used, any restrictions, the commands and their parameters, and describe the possible outputs. The program itself should provide instruction in its use. For the novice, a list of commands should be provided with the most likely commands noted; when parameters are required, the range of acceptable values should be given and a default value provided if possible. For the expert, it should be possible to abbreviate these messages. Each input should be checked and in case of error the user should be allowed to correct only the value in error.

When output is presented, the user should always be able

to suspend or terminate the output, in case he has asked for too much output or for the wrong item.

Every interactive program should have some means of preserving its state, so that it may be interrupted and then continued at or near the point of interruption. This allows the user to suspend the program, use some other facility or stop work for the time being, and return to the task at some later time.

4.2.4 Implementation Aids

Implementation aids are the editors, compilers, and other mechanized aids to program development (including documentation). After the choice of hardware, the most far-reaching decision among implementation aids is the choice of a language. No language is ideal, but the existing body of software provides a powerful incentive to choose an existing language as a starting point. Such an approach also avoids the work required to design the language and perhaps also the work required to implement it. Ideally, the language could gradually be provided with extensions, first implemented by a preprocessor, for the source text, then subsequently by revising the compiler. In addition, a programmer would expect to use extensions included in the text of a routine, if the language permitted.

Debugging aids are often supplied for working with

assembly languages but they are rarely supplied for a high level language because they must incorporate practically the entire compiler. A feasible alternative is an interpreter and/or incremental compiler.

4.3 PROPOSALS FOR FUTURE CAD SYSTEMS

In this section, we will provide enough detail so that individual components can be identified. While we will not attempt to design these components, we will identify some which can be adapted from currently available software and others which must be developed from scratch. A CAD facility cannot remain static if it is to serve the changing needs of its users. Thus, we cannot say what should or should not be included in a system. A system is complete or incomplete only with respect to its community of users.

From our point of view, a system is a collection of modules for data access, for data manipulation, and for user interfacing. Within this system, there are also complete programs which include one or more modules from the groups given. Our job is to maintain these modules and programs and to provide implementation tools and guidelines so that new modules and programs can be created. This view suggests that system management is an administrative task rather than a programming task, although it is programming which we want to manage. A concept relevant to this management is "structured programming". One basis for structured programming is "the

realization by management that it should incur short-range costs in order to achieve long range benefits" [MEIS74]. Our proposals fall within the domain of structured programming; they will incur short-range costs in the hope of long range benefits. Our proposals are also administrative: they require disciplined programming, a discipline which will free the programmer to be more creative in the long run.

4.3.1 Data Structure Proposals

Within a data structure, there are at least three levels at which a piece of information can be considered. These levels are (1) a physical or storage structure, which provides access at the hardware and operating system level; (2) a logical or access structure related to the physical structure by compiler or programmer provided access mechanisms; and (3) a display or presentation structure which is presented using the nomenclature of the user's discipline. The physical data structure is provided by the hardware and the operating system; it usually takes the form of random access to small amounts of information (main memory), or indexed sequential access to larger amounts of information (secondary memory). The logical data structure is the most important level of data access, since most of the programming in the system will refer to the structure at this level. The function of this level is to provide a convenient set of concepts and the corresponding notation to simplify manipulation of the data structure. A

simple example of a structure at this level is a matrix, or two dimensional array. The programmer uses this notation because it is convenient to his problem, while the compiler provides the transformation (or mapping) to the physical data structure which is actually available, usually sequentially organized memory. The highest level of data access is the presentation to the user. This may parallel the structure of level 2 but it must be provided with labels and identification to make it meaningful to the user. To extend the example of the matrix, the user must see not only the numbers, but also labels for each row and column, e.g., the rows could be labeled with county names within a state, and the columns could be labeled Total Population, White Population, Black Population, etc. The point of the third level of data access is that the concepts for the second level of data access are common to a wide variety of applications while the third level must change from user to user.

The programmer must discuss the user's needs, and then select those logical (compiler-provided) structures which are appropriate for problem definition and manipulation. The programmer then constructs the user interface which augments the logical structures by adding the appropriate terminology and symbology.

There are two aspects of data structure as currently implemented in high level languages which need to be improved. A greater variety of data structures must be provided and more

isolation must be provided between the logical structure as seen by the programmer (user) and the physical structure as implemented by the compiler or data access module. A greater variety of data structures serves the needs of the programmer in an obvious way; he is better able to choose a data structure which reflects the structure of the problem, allows efficient manipulation, and provides rapid access to the data. Improved isolation between logical and physical structures allows a variety of logical structures to address a particular physical structure. This allows the programmer to use an appropriate notation for each subproblem to decrease the amount of coding required and increase the maintainability of the code by making the operations more obvious. It will also reduce the amount of reformatting necessary to provide input to manipulation routines, since these routines will now accept a wider variety of data structures without modification.

These choices are complicated by the existence of extension facilities in the high level language. These provide a substitute for the implicit construction of compiler defined data structures during compilation or execution. They are certainly necessary, since no language can supply all possible data structuress but we would certianly prefer the compiler's built-in structures where feasible.

The data structure facilities should include the following data types: integers, characters, pointers, code (machine instructions), floating point representation, double

precision and complex numbers. This list covers the data types usually supported by hardware and those commonly supplied by software. It should be possible to add new data types to this list. Using these data types as nodes or leaves, the following structures should be provided as a mininum: vectors, arrays, lists, stacks, and hash coding. Any structure should allow any other structures as an item; for example, an array whose first item is a list, and whose second item is a stack. These structures should be dynamic, so that an array, for example, could be shortened or lengthened, or an item of an array could be first a list and later a complex number. Structures for associative access might also be provided.

There are several approaches which can be taken to improve the isolation between a logical data structure and its physical representation. The ultimate solution is to let the compiler make the decision about the best physical representation based on an evaluation of the program [suggested by H. B. Baskin]. The logical data structures must still be provided, but the programmer is free to change from one logical structure to another as his needs dictate, letting the compiler provide the proper access to the physical structure. A difficulty is that the compiler does not know which other programs also access the data structure, and so it cannot do a truly global optimization.

A less difficult scheme for the separation of logical and

physical access is to declare the physical structure beforehand for those variables which need it. This scheme provides almost as many problems as compiler selection of data structure, but it allows global optimization by the programmer and removes the most difficult decision from the domain of the compiler.

The situation is much simpler, if we require the programmer to declare his data structures within the program and to use the corresponding notation throughout. This corresponds to current language implementation, except that we assume that a wider variety of data structures are offered.

An interim scheme could be implemented with current languages by using a preprocessor to implement data references as explicit function calls where necessary.

4.3.2 Analysis Techniques

The analysis routines form the bulk of the CAD system. They require substantial amounts of coding, testing, and documentation. Fortunately, these routines are most easily borrowed from other installations. A facility could expect to import as much as 90% of its routines initially and 50% subsequently. Recognizing that there are requirements for accuracy, speed, documentation, and error control, nevertheless the administrator's most pressing need is for an overview of analysis modules which will allow him to make an

intelligent initial selection for his facility, and which will allow him to evaluate what is available at other facilities.

At the lowest level are the elementary operations ranging from square root to the hyperbolic functions of complex and double precision arguments. These should be part of the run-time library of the language processor.

The next level will form the major part of the library of analysis routines. This level would include (1) mathematical functions and operators, (2) mathematical approximations, (3) simulations, and (4) symbolic manipulations.

The mathematical functions and operations are responsible for solving some mathematical equations and computing well defined operations. This class is characterized by the fact that the amount of computation can be predicted (at least bounded) a priori. Thus, matrix manipulations, Fourier transforms, and the statistical packages are part of this class. The solution of some differential equations and of algebraic equations of order 4 or less would be in this class. The solution of a quintic degree algebraic equation would not be included because there is no mathematical formula for the result in closed form.

Mathematical approximations are iterative or approximate techniques such as numerical integration, relaxation techniques, approximate solutions to differential and algebraic equations, and function minimization. The user must

be aware of the limitations of these methods, and take care that he obtains a valid answer. Where possible and appropriate, these routines should provide some measure of errors.

Simulation techniques compute the results of some process by iteratively computing the state of the process at subsequent points as a function of time (or other independent variable). These techniques are applicable where a problem is less well known or more difficult, such that a global solution by mathematical functions or approximations is not known. Since only a local knowledge of each aspect of a problem must be known, simulation techniques are more widely applicable than exact or approximate solution techniques. Discrete simulation can be used when it is known that the state of variables changes only at discrete times. Continuous simulation attempts to monitor the variables continuously in time. On a digital computer, continuous simulation is implemented as discrete simultion with very short time intervals. The variables are assumed to change an insignificant amount or in a known way between time steps. Simulation techniques are often combined with mathematical approximations such as numerical integration.

Symbolic manipulations are those which operate on the symbolic definition of a problem rather than the numeric aspects. For example, some integration problems can be solved by performing a symbolic integration on the formula and then

computing the answer exactly with the mathematical functions, rather than using a mathematical approximation technique. Examples of routines available are those for manipulating algebraic expressions, polynomial arithmetic, symbolic integration, and the predicate calculus. Routines for the manipulation of Boolean logic are in this class, as are routines for the synthesis of electronic logic from Boolean equations or truth tables. Routines which examine or manipulate graph structures are also included, such as finding the spanning tree of an electronic circuit, or finding loops in a flow chart.

Observation of a large computer installation suggests that careful attention to the library of analysis routines will be amply repaid. Programmers will almost always use the installation library for low level routines, e.g., sine, cosine, and they will even think twice before they conjure up their own version of a higher level routine, for example, a Fourier transform routine. When new routines are developed, the administration should take the following steps to ensure maximum utility fromthe routine: (1) a senior programmer/analyst should select the mathematical basis for the algorithm and should design the interface between the module and other routines; (2) a committee of users should ensure that the proposed design meets all their requirements, and (3) the routine can be coded and test. These steps are designed to ensure that the module will satisfy as many users

as possible and will be flexible and easy to use. The purpose of the committee is to ensure that any slight variations in requirements can be included in a single routine without leading to a proliferation of similar routines. The actual coding of the routine is of less importance compared to the interface since it can be changed at any time, while the innerface will become rapidly frozen as users include it in their programs.

4.3.3 User Interface

The user interface to an interactive system is, in some sense, the total set of capabilities which are available to the user. A good user interface will make all of these capabilities easy to use, while a poor one will make them difficult or even impossible to use. In this section, we will concentrate on the control aspects of the user interface. Control is a function which is complimentary to data structures and algorithms. If we consider a computer program as a model of some real situation, then we can identify the data structure as providng the objects of the model, the algorithms as providing the actions which manipulate the objects, and the user interface as providing the control and decision-making which guide the actions. It should also be clear that the elements of control do not all appear in the same place or at the same level, but rather, they are spread out and appear throughout the program.

This notion of layers of control is most important to the development of good user interfaces. These must provide flexible means for building these layers of program and control structures. These ideas are clearly parallel to those guiding the development of data structures and a similar flexibility should be provided. The principle tool should be a uniform and flexible framework for control structures at all levels including the operating system. Several alternate notations can be provided to most nearly correspond with the user's thought patterns. For example, one important class of control structures includes structures for aggregation of activities. Assembly languages provide this capability in the form of macros; higher level languages provide subroutines or procedures, and operating systems often provide cataloged procedures. The well designed user interface should provide all of these schemes at all levels, and as suggested in the section on data structures, perhaps the notation and the implementation should not correspond one to one, but rather the user interface should select the most appropriate implementation regardless of the notatin. Again, this may not be immediately possible, but it should be the goal, and as with data structures, there are reasonable interim steps. One obvious interim scheme is to have the user specify the implementation for each instance of aggregation (with defaults, of course).

Having adopted many levels of control, the user interface

must provide for moving easily through the control structure for purposes of examinaion or modification. This facility should include text editors, (hyper-text editors?) and automatic tree diagrams (or flow charts) of control structures.

A second aspect of the user interface has been described as habitability [DEFA75], or livability. This refers to the ease or naturalness of use of the interface. Several features are important to this naturalness, but the two key ideas are that the user must know what is going on, and he must be in control. There are a multitude of details which contribute to this feeling of ease.

The control section should be self-explanatory in operation with at least these features: a list of commands, acceptable and default values given for parameters, and well-labeled input and output.

Another area which is basic to an interactive program is the availability of tutorial commands. In a GMS, the help commands are not needed so much for the functioning of specific commands (since all the commands are easily understood) but rather to give a sense of direction when the user gets bogged down in detail. With this end in mind, the help commands need to be far more intelligent than the usual canned explanations found in interactive programs. While the prototype GMS was originally written with the commands

structured so as to leave the initiative with the operator, it is clear that this was overdone, especially for novice usrs. One scheme for regaining the initiative is to have more fixed interrogative sequences in which the user is told what to do or asked questions by the program. The user no longer needs to specify the normal command sequence; explicit action is needed only to escape from the normal sequence.

Another scheme for providng a sense of direction is the use of intelligent "help" commands. These commands would be used when the user is not sure what to do next in the sense of solving a problem, not in the sense of what command does what. They should be aware of the status of the program, and they could tailor their response according to the current activity in a global sense. These help commands could provide an entry to program directed activity. Such a sequence might

1. Explain the purpose of the program. Introduce elements as building blocks of the system, including symbols and definitions. Give models as diagrams to be analyzed or used as definitions.

2. Suggest reviewing existing libraries of elements for similar applications. Assist in viewing these libraries.

3. Ask what new elements are needed. Automatically cycle each element through a symbol drawing and text definition phase. Ensure that symbols are all compatible in size; give unsolicited advice about attacher points and other

matters.

4. Guide the user through the model construction phase. Suggest he start with a sketch, then place essential or critical elements, connect and label them, give numerical or constant parameters, then place non-essential (for example, I/O or optional) elements and connect them.

5. Guide the user through the use of the translator. Provide queries to set up the table of templates and provide a check-list for the job control language.

These possibilities can be achieved without redesigning the entire GMS. A more powerful tutorial facility would bring the program more into the domain of Computer Aided Instruction rather than just a tool for symbolic diagrams. Nevertheless, the reader is referred to SOPHIE [BROW74] as an example of the power of unsolicited prompting.

The control section should also provide an interrupt facility so that the user can stop a process and see if it is progressing satisfactorily. While many decision points will be explicitly included in the control section as programmed user interaction, it is also useful to have a mechanism for stopping at nearly any point and allowing the user to scan the state of the process. Of course, we cannot expect this mechanism to be as well-developed as the explicit user interactions provided by the programmer. The control module should also provide operating system status information about

the job. With this information and the interrupt facility, the user can take timely steps to stop a job which has run away or become otherwise uncooperative.

System aids which are commonly needed for control of programs include the automatic compilation of programs from decision tables, menu programs for command selection, and formatting and questionnaire routines for simplified parameter input.

4.3.4 Implementation Aids

There are many tradeoffs in the choice of a high level language. Among these are the ease of implementation, the varieties of data structure provided, and the software available for the language. The ease of implementation depends on whether the language is new or existing and whether a version exists for the hardware selected. If a version exists then that is the easiest choice, with the implementation of a known language being much easier than the design and implementation of a new language. As a personal decision, I would avoid creating a new language.

To clarify this matter, I have chosen two representative languages: one old, PL/1, and one new, EL1 [WEGB71]. PL/1 is implemented on the IBM 360 series and EL1 is implemented on the DEC PDP-10. PL/1 offers a wide variety of existing software while EL1 offers a wider variety of data structures.

ELl is an extensible language which allows the programmer to add new data types and structures, and new operators to the language. Once the language has been extended, the programmer can use a simple notation to efficiently describe data structures and manipulations in each particular problem domain. On an IBM 360 or the PDP-10 the choice would be in favor of the implemented language. On machines with neither language, I would favor ELl as more useful in the long run. This choice of language might affect my choice of hardware as well.

After choosing a language, there is a collection of text editors, loaders, subroutine libraries, and so forth which must be created. An interpreter would be most valuable as a program development and debugging aid. It allows the creation of the proper environment for a procedure under test, and by executing one statement at a time the operation of the procedure may be observed at any level of detail. The procedure may be modified and execution resumed without disturbing the environment. This is a valuable improvement over checkpoint-restart systems which may not allow a change in the procedure between a checkpoint and a restart. The interpreter offers the features of several assembly language debugging packages with the advantage of providing these features for a high level language. The ability to modify a procedure and continue may save a great deal of time when the environment of a fault in a procedure occurs only after

lengthy computing. The alternative is for the programmer to save and restore his own status; but this is generally impossible without more system knowledge than the programmer generally has.

The design and construction of the interpreter, compiler, and the rest of the environment is an admittedly complex task, but one which need not be any more difficult than the creation of the disorganized collection of editors, interpreters, compilers, loaders, subroutine libraries, and so forth which are available at any major computing center. What is necessary is an overall view which puts each piece in its place and specifies the interfaces between pieces.

Appendix A. Response to the Users Questionnaire

NAME: Richard LaPierre - Assisted by Don Austin

1. What was your overall impression of PICASSO?

   I think it has great possibilities;
   I'm just sorry it wasn't funded.

2. Did you find the system generally useful?

   Yes

3. How extensively did you use the system? Hours? Days?

   4 hours (although D. M. Austin spent two days working on this particular problem).

4. What important problem did you solve?

   A digital logic timing problem.

5. Could you have solved it another way?

   Yes, by building the hardware.

6. How would the two costs compare?

   A breadboard device could have been built in 3 days. PICASSO required three days to create a library of logic elements and two days work on this particular problem.

7. Do you plan to use the system again? Why? Why not?

Maybe, if the system is cost effective. To be cost effective, it needs good accessibility - the engineer must have constant access (in his work area); it needs simple language and more reliable software (operating system) and hardware.

8. Would you recommend the system to your colleagues?

I think they should look into it.

9. Did you recommend the system to your colleagues? Who?

No.

10. What revisions or extensions would you recommend?

Good accessibiliity, simple language, reliable hardware and software.

11. Do you know of any better system? What are they?

No

NAME: Dan Maeder

(Answered by D. M. Austin who assisted Dan Maeder)

1. What was your overall impression of PICASSO?

   It is very hard to use, the computer system is unreliable, and the lightpen is very hard to use. (P.S. I am right-handed.)

2. Did you find the system generally useful?

   Yes, very useful, in fact, we solved a problem of two weeks Fortran programming in one hour.

3. How extensively did you use the system? Hours? Days?

   An hour a day for a couple of weeks (until the problem was solved).

4. What important problem did you solve?

   The hardware design of a delay line, varying the parameters to get the proper waveform.

5. Could you have solved it another way?

   Yes, build it and use an oscilliscope.

6. How would the two costs compare?

   Using PICASSO was much, much cheaper. Building the hardware is impractical.

7. Do you plan to use the system again? Why? Why not?

No, not available in Geneva (where I have moved).

8. Would you recommend the system to your colleagues?

Yes.

9. Did you recommend the system to your colleagues? Who?

Probably (D. M. Austin is not sure).

10. What revisions or extensions would you recommend?

Some changes to MIMIC for optimization would be helpful.
You should also make it transportable to small terminals.

11. Do you know of any better system? What are they?

No

NAME: Nancy McDonald

1. What was your overall impression of PICASSO?

   Good, interesting, flexible, powerful, a little cryptic to learn to use.

2. Did you find the system generally useful?

   Yes

3. How extensively did you use the system? Hours? Days?

   I used 65% of PICASSO's facilities. I worked with it for three months.

4. What important problem did you solve?

   I used PICASSO as the basis for a picture query language.

5. Could you have solved it another way?

   Yes

6. How would the two costs compare?

   It would have been three times as much work without PICASSO.

7. Do you plan to use the system again? Why? Why not?

   Yes, I am still using it.

8. Would you recommend the system to your colleagues?

Yes.

9. Did you recommend the system to your colleagues? Who?

No, my colleagues have no need for such a system.

10. What revisions or extensions would you recommend?

-

11. Do you know of any better system? What are they?

No.

NAME: Peter Levine

1. What was your overall impression of PICASSO?

   Fantastic, pretty far out, but it had lightpen and hardware problems. I was awed, confused.

2. Did you find the system generally useful?

   Never got to that part.

3. How extensively did you use the system? Hours? Days?

   I spend a lot of hours fiddeling around.

4. What important problem did you solve?

   None

5. Could you have solved it another way?

   Yes. This problem involved differential equations for a complex feedback path in biological simulation. I formulated it as a diagram, then wrote out the equations from the diagram. Then I used a simulation system that I was familiar with.

6. How would the two costs compare?

   It was much easier without PICASSO.

7. Do you plan to use the system again? Why? Why not?

   No. Too clumsy.

8. Would you recommend the system to your colleagues?

   For certain problems.

9. Did you recommend the system to your colleagues? Who?

   No.

10. What revisions or extensions would you recommend?

    More reliable hardware, software. Also I did not have a complete set of documentation.

11. Do you know of any better system? What are they?

    No.

NAME: John S. Colonias

1. What was your overall impression of PICASSO?

   It is a well documented and structured computer program.

2. Did you find the system generally useful?

   Yes. It fulfills a definite need.

3. How extensively did you use the system? Hours? Days?

   Approximately one month - on and off.

4. What important problem did you solve?

   I did not solve any problem. I was trying to see whether it could be used effectively in circuit design applications.

5. Could you have solved it another way?

   Perhaps. But I have not given it a thought.

6. How would the two costs compare?

   PICASSO would be less costly to operate.

7. Do you plan to use the system again? Why? Why not?

   When a definite need arises, yes.

8. Would you recommend the system to your colleagues?

Yes, I would (and I have).

9. Did you recommend the system to your colleagues? Who?

I have discussed PICASSO with people at the Argonne National Laboratory and Lawrence Livermore Laboratory.

10. What revisions or extensions would you recommend?

-

11. Do you know of any better system? What are they?

I have not taken the time to investigate other systems.

NAME: Andrew E. Allen

1. What was your overall impression of PICASSO?

   Very professionally finished product.
   Much flexibiliity. Impressive.

2. Did you find the system generally useful?

   Yes.

3. How extensively did you use the system? Hours? Days?

   A total of a couple of weeks, about two years ago.

4. What important problem did you solve?

   None; just getting familiar with it.

5. Could you have solved it another way?

   NA

6. How would the two costs compare?

   NA - but not terribly expensive. In fact, for the work it does, reasonably inexpensive.

7. Do you plan to use the system again? Why? Why not?

   Probably not - not quite in my area of application (text editing and character graphics).

8. Would you recommend the system to your colleagues?

Unquestionably, if I thought they would have a use for it.

9. Did you recommend the system to your colleagues? Who?

No.

10. What revisions or extensions would you recommend?

(a) More general commands for processing polygons - i.e., shrink by factor, to fit another, rotate by degrees, etc.)

(b) Better user's manual (although the one I have is four years old and may already have been supplanted)

(c) 3-D

11. Do you know of any better system? What are they?

No.

NAME: Peter Wood

(Simulation application; see separate comments on PICASSO as a starting point for a mapping project.)

1. What was your overall impression of PICASSO?

   Good.

2. Did you find the system generally useful?

   Yes, simulation, simple drawing and mapping, and structured mapping.

3. How extensively did you use the system? Hours? Days?

   Two weeks for simulation.

4. What important problem did you solve?

   Class assignments for Engineering 111 at University of California at Berkeley.

5. Could you have solved it another way?

   Yes, using the campus META 4 - CSMP system.

6. How would the two costs compare?

   The META 4 was too crowded.

7. Do you plan to use the system again? Why? Why not?

   Yes, if applicable. I want to use GPSS which is not yet on our system.

8. Would you recommend the system to your colleagues?

Yes.

9. Did you recommend the system to your colleagues? Who?

Yes, Betty Seasonwein.

10. What revisions or extensions would you recommend?

Add GPSS to available analysis routines.

Add more analysis routines.

Better documentation for the analysis interface.

11. Do you know of any better system? What are they?

No. CSMP - too limited - fixed library,

no hard copy.

NAME: Peter Wood

1. What was your overall impression of PICASSO?

PICASSO is a well designed and debugged system.

2. Did you find the system generally useful?

I found the system useful for continuous simulation, where the symbols drawn have text definitions, for explorations of mapping where the symbol placement on the screen is significant, and for simple drawing (expecially with quantum = 0).

3. How extensively did you use the system? Hours? Days?

Initially about two or three days a week for three to six months and occasionally thereafter.

4. What important problem did you solve?

Through the USERCMD feature developed the capacity to search a data structure of symbols nested to many levels for all elements within an arbitrary closed polygon.

5. Could you have solved it another way?

Not without duplicating a large part of PICASSO. The text definitions and analysis routines

could have been bypassed. Of course, it could have been done in batch mode.

6. How would the two costs compare?

The cost in man-hours would have been more and progress would have been slower without PICASSO.

7. Do you plan to use the system again? Why? Why not?

Yes, when the occasion arises. PICASSO is easy to use, convenient, and reliable.

8. Would you recommend the system to your colleagues?

Yes

9. Did you recommend the system to your colleagues? Who?

They recommended it to me.

10. What revisions or extensions would you recommend?

-

11. Do you know of any better system? What are they?

No.

NAME: Horace Warnock

1. What was your overall impression of PICASSO?

   Great.

2. Did you find the system generally useful?

   Yes.

3. How extensively did you use the system? Hours? Days?

   Approximately 1000 hours starting in
   May of 1971.

4. What important problem did you solve?

   Scratchpad (sketching to scale) of
   printed circuit layouts.

5. Could you have solved it another way?

   Yes - work up printed circuit layouts by
   hand. Sketch to scale - tape and retape, etc.

6. How would the two costs compare?

   We found that we saved approximately 20-25%
   using PICASSO for sketching to scale and
   using Xerox as a guide or underlay.

7. Do you plan to use the system again? Why? Why not?

   Yes.

8. Would you recommend the system to your colleagues?

Yes.

9. Did you recommend the system to your colleagues? Who?

Yes. The Electronics Engineering
Department - Lawrence Berkeley Laboratory.

10. What revisions or extensions would you recommend?

Program - none
Terminal - higher resolution screens
Systems - stand alone very desirable

11. Do you know of any better system? What are they?

No.

NAME: Bill Benson

1. What was your overall impression of PICASSO?

   Sexy. A powerful tool to build and manipulate pictures. Have had no experience analyzing models.

2. Did you find the system generally useful?

   It was easy to add a user command to do animation.

3. How extensively did you use the system? Hours? Days?

   Briefly - a few days with animation. We used it extensively about a year as a specialized map editing program, but this was heavily modified and all the analysis routines were removed.

4. What important problem did you solve?

   As above.

5. Could you have solved it another way?

   Yep! but this was a fairly quick way to get going and we got experience with the problem of map editing.

6. How would the two costs compare?

Much cheaper not using PICASSO.

7. Do you plan to use the system again? Why? Why not?

No immediate application in mind, but would certainly use it on an appropriate problem.

8. Would you recommend the system to your colleagues?

Sure

9. Did you recommend the system to your colleagues? Who?

Richard Friedman, Jerry Knight.

10. What revisions or extensions would you recommend?

Use device independent graphics. Make zoom more convenient. Perhaps define symbols with relative points. Rewrite code for clarity.

11. Do you know of any better system? What are they?

No.

Appendix B. Program Documentation

Program documentation is available from the author at the following address:

Harvard Holmes
Building 50B, Room 3238
Lawrence Berkeley Laboratory
Berkeley, California 94720

Documentation and access to the prorotype GMS (known as PICASSO at LBL) is available in the following forms:

1. A short paper describing the system.
2. A users guide to the graphics section with many examples.
3. A users guide to the translator.
4. Source code and listings of the program in Fortran and assembly language (on tape and microfiche).
5. ARPANET access to LBL whereby the program can be executed for a few devices (DEC GT40, Tektronix 4010 - 4015 terminals).

References and Bibliography

AUS71 Aus, H. M., Korn, G. A., The Future On-Line Continuous-System Simulation, Proceedings of the FJCC, 39, AFIPS Press, Montvale, New Jersey (1971) pp. 379-386

AUST72 Austin, D. M., Holmes, H. H., PICASSO: A General Interactive Graphics Modeling Program, LBL-580, University of California, Lawrence Berkeley Laboratory, Berkeley, California (January 1972)

BASK68 Baskin, H. B and Morse, S. P., A Multi-level Modeling Structure for Interactive Graphic Design. IBM Systems Journal, 7, 3 & 4 (1968) pp. 218-229

BASK69 Baskin, H. B., A Comprehensive Applications Methodology for Symbolic Computer Graphics, in Pertinent Concepts in Computer Graphics, University of Illinois Press, Urbana, Illinois (1969) pp 414-428

BELA71 Belady, L. A., Blasgen, M. W., Evangelisti, C. J., and Tennison, R. D., A Computer Graphics System for Block Diagram Problems, IBM Systems Journal, 10, 2 (1971) pp 143-161

BROW74 Brown, J. S., and Burton, R. R., SOPHIE: A Pragmatic use of Artificial Intelligence in CAI, Proceedings of the ACM Annual Conference 1974, San Diego, California (November, 1974)

CONT68 Control Data MIMIC; A Digital Simulation Language, Reference Manual, Publication Number 44610400,

Control Data Corporation, Special Systems Publications, St. Paul, Minnesota (April 1968)

DEFA75 DeFanti, T. A., Sandin, D. J., and Nelson T. H., Computer Graphics as a Way of Life, Computers and Graphics, 1, 1, Pergammon Press, Great Britian (May 1975) pp 9-15

DERT65 Dertouzos, M. L. and Santos, P. J., Jr., CADD: On-line Synthesis of Logic Circuits, Electronic Systems Laboratory, Massachusetts Institute of Technology, Report ESL-R-253, Cambridge, Massachusetts (December 1965)

DERT67 Dertouzos, M. L., CIRCAL: On-line Circuit Design, Proceedings of the IEEE, 55, 5 (May 1967) pp 637-654

DIJK72 Dijkstra, E. W., Notes on Structured Programming, in Structured Programming edited by O. J. Dah, E. W. Dijkstra and C. A. R. Hoare, Academic Press, London and New York (1972)

ELLI69 Ellis, T. O., Heafner, J. F. and Sibley, W. L., The Grail Project: An Experiment in Man-Machine Communications, The RAND Corporation, RM-5999-ARPA (September 1969)

EVAN69 Evantelisti, C. J., and Morse, S. P., Graphical Modelling Using Contextually Implied Functions. Personal Communication (1969)

FRAN73 Franklin, J. L., Dean, E. B., Interactive Graphics for Computer Aided Network Design, Proceedings of the NCC 42, AFIPS Press, Montvale, New Jersey (June

1973) pp 677-683

GEAR70 Gear, C. W., Hyde, C., Lewin, H., Michel, M. J., Ratliff, K., Wilkins, S., The Simulation and Modeling System -- A Snapshot View, Department of Computer Sciences File No. 824, University of Illinois, Urbana, Illinois (1970)

GRON71 Groner, G. F., Clark, R. L., Berman, R. A., Deland, E. C., BIOMOD - An Interactive Computer Graphics System for Modeling, Proceedings of the FJCC, 39 AFIPS Press, Montvale, New Jersey (1972) pp 369-378

HELD75 Held, G. D., Stonebraker, M., and Wong, E., INGRES - A Relational Data Base Management System, Proceedings of the 1975 NCC, AFIPS Press, Montvale, New Jersey (1975)

HOGS67 Hogsett, G. R., Nisewanger, D. A., and O'Hara, A. C., Jr., An Application Experiment with On-line Graphics-Aided ECAP, in Confg. Digest 1967 International Solid-State Circuits Conference (1967) pp 72-73

HOLM72 Holmes, H. H. and Austin, D. M., PICASSO: A General Graphics Modeling Program, Proceedings of the ACM SIGPLAN Symposium on Two-Dimensional Man-Machine Communication, Los Alamos, New Mexico (October 1972)

HORO72 Horovitz, M. W., Austin, D. M., and Holmes, H. H., Symbolic Computer Graphics and Biological

Models, Proceedings of the ACM/SIGGRAPH Symposium, Pittsburgh, Pennsylvania (March 1972)

KAIS69 Kaiser Engineers, San Francisco Bay-Delta Water Quality Control Program (March 1969)

MAGN67 Magnuson, W. G., Jr., Kuo, F. F., Walsh, W. J., On-line Graphical Circuit Design, UCRL-70796, University of California, Lawrence Radiation Laboratory (1967)

MCDO75 McDonald, N., CUPID: A Graphics Facility for Support of Non-Programmer Interactions with a Data Base (Ph.D. Thesis) University of California, Berkeley, California (1975)

MARO17 Marovac, N., A Method for Defining General Networks for CAD, Using Interactive Computer Graphics. The Computer Journal, 17, 4, pp 332-336

MEIS74 Meissner, L. P., talk on Structured Programming/Local Aspects, Berkeley, California (March 1974)

MERR71 Merritt, M. J., Sinclair, R., INSIGHT - An Interactive Graphic Instructional Aid for Systems Analysis, Proceedings of the FJCC, 39, AFIPS Press, Montvale, New Jersey (November 1971) pp 351-356

NAGE73 Nagel, L. W., and Pederson, D. O., SPICE: Simulation Program with Integrated Circuit Emphasis, Memorandum ERL-M382, Electronics Research Laboratory, College of Engineering, University of California, Berkeley (April 1973)

NEWM73 Newman, W., and Sproull, R., Principles of Interactive Computer Graphics, McGraw Hill, New York (1973)

PRES65 Preston, F. S., et al, Development of Techniques for Automatic Manufacture of Integrated Circuits, Technical Report AFML-TR-65-386, Volumes I and II, Electronics Branch, Air Force Materials Laboratory, Wright-Patterson AFB, Ohio (November 1965)

PROJ72 Project MAC Progress Report IX, Massachusetts Institute of Technology, Cambridge, Massachusetts (July 1972)

RENA69 Renaud, R. G., Walters, R. F., The Interactive Creation, Execution and Analysis of Biological Simulation using MIMIC on a Graphic Terminal, Proceedings of the Conference on Applications of Continuous System Simulation Languages, San Francisco, California (1969) pp 185-191

RIEK67 Riekert, R. H., and Lieberman, D. V., DIM - A Low Level Modeling System for Conversational Graphics, IBM Research Report RC-1981, IBM T. J. Watson Research Center, Yorktown, New York (October 1967)

ROBB70 Robbins, M. F., Beyer, J. D., An Interactive Computer System using Graphical Flowchart Input, Communications of the ACM, 13, 2 (February 1970) p 115

SUTH63 Sutherland, I. E., SKETCHPAD: A Man-Machine

Graphical Communication System, Proceedings SJCC 23, Spartan Books (1963) pp 329-346

SYN68 Syn, W. M., Turner, N. N., and Wyman, D. G., DSL/360: Digital Simulation Language User Manual, Engineering and Scientific Computation Laboratory, IBM Corporation, San Jose, California (1968)

WEGB71 Wegbreit, B., The ECL Programming System, Proceedings of the FJCC, 39, AFIPS Press, Montvale, New Jersey (1971) pp 253-262

WIRT1 Wirth, N., The Programming Language PASCAL, Acta Information 1, 1, pp 35-63

TECHNICAL INFORMATION DIVISION
LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720