A method for defining general networks for CAD, using interactive computer graphics

N. Marovac

Department of Electrical Engineering, Imperial College of Science and Technology, Exhibition Road, London SW7 2BT

The use of interactive computer graphics in CAD applications requires a great deal of programming effort. To avoid this effort many general display handlers and graphic languages have been written. They enable programming on a higher level as well as calls in a higher level programming language but they suffer from three disadvantages: they are oriented too much towards graphics and geometry, the handlers and languages are procedure oriented as opposed to problem oriented and modification in picture primitives requires re-programming and re-compiling. This paper proposes a method for defining general networks which overcomes these problems.

(Received February 1973)

1. Introduction
The use of interactive computer graphics in CAD applications requires a great deal of programming effort. This is particularly true if the programming is done in a basic display assembler level. Even after the initial stage of programming has been completed and the programming system is working any further modifications, which are mainly problem-oriented, require the presence of a basic level specialist and a considerable extent of reprogramming. To avoid this, many general display handlers and graphic languages have been written (for example, Schwinn, 1967; Hurwitz and others, 1967; Rully, 1968; Butlin, 1970; Elliot-Automation; Shearing, 1970; Bracchi and Ferrari, 1970).

They enable programming on a higher level as well as the use of graphic calls in higher-level programming languages, such as FORTRAN. However, these display handlers, and the majority of graphic languages, are oriented too much towards graphics. They provide the user with a tool with which to build up a picture on the screen out of basic geometrical elements, such as points, lines, curves, and so on. The picture is sequentially built up from these elements. It can be structured into parts called picture primitives. These primitives can then be separately moved, copied in a number of instances, deleted, identified as separate entities within the picture when needed, and so on.

However, the use of these general display handlers and the majority of graphic languages present the following three disadvantages.

Firstly, they are oriented too much towards graphics and geometry; they lack the power to include in the specification of a physical process and its parts properties which are relevant to this process and to our interest in it. For example, topological and constitutive properties where parts of a picture on the screen are the visual representation of parts of a network.

Secondly, the handlers and languages are mainly procedural, i.e. procedure-oriented as opposed to problem-oriented, the latter being more suitable for CAD applications.

Lastly, any modification in picture primitives requires the re-programming and recompiling of the part of the software system in which the modification is made, and after that rebuilding of the software system if it is made up of a number of segments or overlays.

2. Objectives for a network definition mechanism and language for a software system for analysis and design of general networks using interactive computer graphics

Bearing in mind the aforementioned disadvantages in the use of general display handlers and graphic languages, objectives were set to develop a network oriented defining language and mechanism. The language and mechanism would provide the network designer with the means to specify the characteristics of networks and the properties of parts of networks which he wishes to explore by CAD programming systems using interactive computer graphics.

The language and mechanism should comply with the following requirements:

1. The language should be based on elements which are familiar to the network designer, who may not be a specialist in computer languages or computers themselves. In other words, it should be based on a natural language, such as English, incorporating a network type 'jargon'.

2. The user of the language should be able to specify the class of networks as well as the topological, constitutive and pictorial properties of parts of networks of any nature which he wishes to explore. In these specifications only pictorial properties are relevant to graphics and they are reflected in the visual representation of networks on the screen.

3. The language should be problem-oriented, rather than procedure-oriented, i.e. it should enable the user to request actions in CAD programs which are based on networks and not on sequences of pictorial actions which are 'meaningless' in network semantics. An example of this difference would be a call 'delete a node in a network' instead of a sequence of calls to delete separate lines representing connections forming the node.

4. The mechanism should enable a network designer to change the specifications of properties of any part of a network in a class of networks, or specifications of the whole class of networks itself without needing to recompile any part of the programming system or resegment and link it together if composed of a number of segments.

3. Network-based considerations in the development of the language and mechanism

A network is an entity comprising objects and physical relationships. The term object refers to an item in a network, which can be either a physical one such as a network element, or an abstract one, such as a node. The term physical relationships refers to a set of interactions existing between objects in a network, like the influence of one element on another, interconnections between elements in the topological sense, and so on.

The types of objects and physical relationships present in a network depend upon the nature of the network. They will be different in an electrical network from those in a PERT network. Even in networks of the same nature but of different classes the subsets of objects and relationships will differ, as for example objects and relationships in the study of small signal response of linear electrical networks will differ from...
those in the study of non-linear sampled-data electrical networks.

The model of a network in a computer memory must adequately describe the pictorial, topological and constitutive properties of objects in the network as well as the relationships amongst the objects in the network.

A software system which is designed as a tool for exploring networks of a general nature must possess at the right time specifications of the right objects and relationships relevant to the particular class of networks currently under consideration.

However, it would be impractical to keep a library in the software system comprising a logical union of descriptions of all possible objects and relationships in all types of networks. Even so, a specification of an object can change from application to application and from user to user. To illustrate this point let us consider a resistor in an electrical network. For an application, i.e. for one type of investigation of electrical networks under the programming system, the resistor can be specified as

Resistor is an object having coordinates relative to the corresponding network; a determined visual representation; terminal 1; terminal 2; input parameter—resistance; output parameter—current through the resistor.

However, in another application the same resistor can be specified as

Resistor is an object having coordinates relative to the corresponding network; a determined visual representation (which may differ from the previous one); terminal 1; terminal 2; input parameter—resistance; input parameter—tolerance; output parameter—current through the resistor; output parameter—power dissipated by the resistor; output parameter—a sensitivity coefficient through which the resistor participates in the change of a certain network variable.

The actual specifications for the resistor as used by the software system for each of these applications would be, in the first case,

resistor is an object having:
relative coordinates, pictorial representation, terminal 1, terminal 2, parameter.

and in the second case,
resistor is an object having:
relative coordinates, pictorial representation, terminal 1, terminal 2, parameter, parameter, parameter.

The number of parameters in the specification is given by Number of specified parameters = max (number of required input parameters, number of required output parameters).

It is immediately apparent that these two specifications of the resistor differ in the number of parameters. The first specification for the resistor would not be suitable for the second application because it would not allow for the storage of all the parameters required in the second application. Also the second specification would waste a considerable amount of computer memory space if used for the first application.

Furthermore, a general programming system would have to have exact information as to the number of parameters a resistor has and the way in which they are stored (which may vary from application to application). This information is required by the system when the network designer is assigning values to input parameters, examining output parameters and also when the system processes the model of a network to produce various network matrices used for the design of the network.

The example presented was a very simple problem upon which a compromise solution could be found without the use of the definition mechanism.

However, in general, changes from one application to another usually involve principal changes and changes in specifications which imply changes in the nature of the networks, in which case a simple definition mechanism which is easy to use and does not involve any changes in the programming system, would be a convenient method of specifying networks prior to their investigations by the system. For example, if the designer of a data communication network wishes to use the programming system for the analysis and design of communication networks; the specification of the objects and relationships for that class of networks would be considerably, if not completely different, from the specifications of electrical lumped networks.

Communication lines comprised in data communication networks, have similar graphical and somewhat different topological properties as do the connections in electrical lumped networks, but the communication lines also have constitutive properties which the connections in the aforementioned class of electrical networks do not possess.

The constitutural properties of the communication lines are similar to those of some network elements.

One can use the definition mechanism very conveniently to define communication lines and their properties, as well as properties of all other objects in the class of data communication networks, and so use the programming system for their design without any changes in the system.

Furthermore, the semantics of some actions of the designer using the programming system can change depending on the specifications of some objects in a class of networks. Let us explain this statement using the same example: i.e. the difference between communication lines in a class of communication networks and connections in electrical lumped networks.

When a network designer draws an electrical lumped network on the screen, if he points the light pen at a line on the screen representing a connection, he usually wishes to start drawing another line representing another connection forming a junction with the first one at that point.

However, when a designer draws a data communication network and he points the light pen at a line representing a communication line, he wishes to assign values to the attributes of the communication lines such as capacity of the line, load on it and so on, instead of starting another communication line from it by forming a junction. Such a junction in the case of a communication network would be meaningless.

If the communication lines are appropriately specified, the programming system would be able to understand the designer’s action properly and translate it into a meaningful request.

4. The network defining mechanism and the network defining language

With this in mind a network defining mechanism and a network defining language were designed and incorporated in a software system for analysis and design of networks of a general nature using interactive computer graphics. They provide the network designer with the means of specifying the characteristics of the networks which he wishes to explore, and the properties of parts of these networks. These are not only pictorial properties but also network-oriented properties relevant to the desired analysis and design of the networks.

The definition mechanism is based on a problem-defining type of language, called NEDLAN (NETwork Defining LANguage), and the use of a software driving table, called a reference and
The network-defining language is used by the network designer to define the type of problems he wishes to investigate by the software system, i.e. to define the class of networks in which he is currently interested.

To date the only known problem-defining language is STRESS (Fenves, 1965). The NEDLAN differs from STRESS on two principal points. NEDLAN is not used to define each particular problem separately as is STRESS, corresponding to the specification of each particular network for analysis or design. NEDLAN is used to specify the whole class of networks whose members are later investigated by the system. In other words it is used to specify the syntax of networks. Each particular network in the class is then defined by the use of interactive graphics on the basis of the class specifications made with NEDLAN.

To illustrate this point let us assume that the network designer is exploring linear electrical networks. He uses the language to specify the characteristics of these networks. He specifies every element in the networks which may be resistors, capacitors, voltage sources and so on, by properties of these elements. He also specifies the relationships which are likely to take place between elements in linear electrical networks. These may be interconnections, controlling influences of some elements on controlled voltage and current sources, etc. After specifying networks in such a manner, the system is set for the designer to draw any particular network in this class on the screen and apply to it any analysis of design routine.

The other difference is that the use of NEDLAN is not based on a set of programming calls through which specifications are made by assigning values to various arguments in the calls, but it has all the characteristics of a programming language such as FORTRAN, for example. NEDLAN possesses alphabet and syntax. The notation used in NEDLAN is a modified version of the BACKUS NAUR notation used to specify the ALGOL language (Backus, 1963).

It is beyond the scope of this paper to present a description of the structure and syntax of NEDLAN, this will be done elsewhere (Marovac, 1973; Marovac, 1973a).

An example of a specification in NEDLAN is that of the resistor as required for the previously mentioned second application.

\[
\langle \text{resistor}\rangle \gets (\text{relative position} / \text{dx} + 0, \text{dy} + 0) /, \text{display intensity on} /, \langle \text{resistor image} \rangle, \langle \text{displacement function 1} \rangle, \langle \text{terminal 1} \rangle, \langle \text{displacement function 2} \rangle, \langle \text{displacement function 2} \rangle, \langle \text{terminal 2} \rangle, \langle \text{displacement function 1} \rangle, \text{display intensity off} /, \langle \text{parameter} \rangle / \text{real} /, \text{display intensity off} /, \langle \text{parameter} \rangle / \text{real} / \rangle #
\]

\[
[\text{object, semantic type} = 3];
\]

In this specification of the resistor, all features on the right side of the horizontal arrow ‘\(\gets\)’ enclosed in the angular brackets ‘\(\langle \rangle\)’, have to be further defined in the same way as the resistor was. However if the designer or user of the programming system wishes to use the definitions of any features which were defined by somebody else or himself previously, he can do so simply by indicating this in the corresponding statement. For example, in the specification given above the designer can use already existing definitions of features terminal 1 and terminal 2, if he is satisfied with the existing definitions of the terminals.

The expression in brackets ‘\(//\)’ represents parameters used either for graphical purposes such as intensity, or to specify the nature of the parameter of the resistor, as for example ‘real’ which states that the resistor parameters represent real numbers. The terms in the rectangular brackets declare the resistor to be an object in a network of the semantic type, value ‘3’ which is used to distinguish the resistor amongst objects of other types in networks. A further description of the parts featuring in this specification can be found in the previously mentioned references to the author’s papers.

A network designer using NEDLAN specifies networks during the run of the system, prior to the exploration of the networks by the system. A complete specification of the characteristics of a class of networks and the properties and parts thereof is called a specification list.

When a specification list is completed it is compiled by part of the programming system into a reference and driving table. The table represents the computer memory form of the specification list as understood and used by the system.

It is used by the system for two purposes:

1. It is used by the network creation part of the software system in the process of building a model of each particular network, in the specified class, inside the computer memory. This model is later used for the analysis and design of the network.

When a network designer makes a request to the system to create the representation of an object as part of a network, to insert it in the network model and to add the visual image of the object to the diagram of the network on the screen, this part of the software system refers to the driving table. It extracts from the table all relevant information concerning the particular object. This information is then used to create the representation of the object in accordance with the given requirements and specifications and draws an image of the object on the screen. By changing the information in the table a different object can be created and inserted into the network model, and added to the picture of the network on the screen.

Furthermore, when the designer wishes to remove the object on the screen or to change any value of its parameter, as for example, the parameter of the resistor, the creative software part refers to the driving table and finds which parts in the representation of the object in the network model need to be amended, and in which way, to fulfil the designer’s request.

2. The table is used by the interface, i.e. the part of the software system which analyses the built-up model of a network. When the representation of an object in the model of a network is encountered the interface goes to the same table. It finds the specification for the object in question.

The specification assists this part of the system in processing the representation of the object in the model and enables it to extract the relevant information required to make corresponding entries to various network matrices. These matrices are general and independent of the class of networks and are passed to different routines used in the analysis and design of the network.

The use of the table by the creative and interface part of the software system corresponds to the use of the driving table in a table-driven universal computer language interpreter, as illustrated in Fig. 1.

![Fig. 1](http://comjnl.oxfordjournals.org)
Fig. 1 shows a general higher level language processor which is driven by a reference and driving table. The processor accepts a statement in a higher level language, such as FORTRAN, interprets the statement and produces the result. The driving table contains a specification for the higher level language in its metalanguage. By changing the meta specification for the language it can then interpret any other higher level language such as ALGOL.

Analogous to this, the NEDLAN language is a metalanguage used for specification and description of the syntax of networks, and the driving table contains this meta specification for the corresponding class of networks.

The creative and interface parts of the software system represent universal interpreters. The creative part accepts input statements in the form of interactive user requests, interprets each request and produces an amendment in the network model. The interface part accepts an input statement in the form of a representation of part of the network in the model, and as a result updates various network matrices with the entries corresponding to that part of the network.

Changing the meta description of the class of the network both software parts would produce different results in accordance with the new specifications which would correspond to the new class of networks.

Use of the driving table ensures the universality of both the network creative part and the interface part of the software system, as well as that of the software system as a whole. This is because all application dependent information is concentrated in the driving table which is external to the system, thus removing the need for re-programming and recompiling any part of the system each time a modification is made in the specification of networks or when specifications are changed entirely, as would have been necessary were they embedded in the system.

When commencing the run of the system the network designer has three options regarding the use of the driving table:

1. He can create a completely new driving table. As already mentioned, this is done by typing in a complete specification list which is then compiled by the system into the new driving table.

2. He can use an already existing driving table, created by him or any other user of the system and saved on any computer storage medium. This is done by simply declaring to the system which particular table he wishes to use and on which medium it is stored.

3. He can also choose an already existing specification list recall it from the medium on which it is stored and modify any specification which he wishes to. Thus the modified specification list is recompiled again into a driving table ready to be used by the system.

A specification list has the form of a sequence of definition statements written in the language NEDLAN. The statements resemble declarative and data statements in a higher level language, such as FORTRAN, and in fact they serve a similar purpose. The principal difference is that declarative and data statements define types, structures and values of alpha-numerical data which is used by a FORTRAN program during its run. Statements in a specification list define the class and structure of networks as well as the properties of constituents of networks which are to be investigated by the system.

However, there is another more mechanical difference. The declarative and data statements are part of a FORTRAN program, and any modification of statements requires re-compiling of a part of the program (or the entire program itself in some computer systems), and reloading the whole program, whereas in the case of the specification list the programming system remains as it is and it is the system itself which recompiles the specification list and produces the driving table.

The statements in a specification list specify from top-down the entire structure of the class of networks; from every object to its actual constituents.

The definition mechanism has proved to be useful. It was used to specify objects and relationships for a class of lumped electrical networks, a class of digital filters and a class of critical path networks.

The mechanism was also used to specify the display control features of the programming system by a user.

The mechanism was particularly valuable for rapid assessment and modifications of specifications of objects and relationships in these classes of networks, as well as in experimenting to find ergonomically suitable visual representation of objects in the networks and also the representations of the mentioned control features.

Using the mechanism a large amount of programming, re-compiling and resegmenting of the system was avoided, and substantial computer time was saved.

Lately, the potential of application of the programming system to the design of data communication networks has been investigated, using the definition mechanism to specify these networks.

Conclusion

The network defining language and mechanism described in this paper present a method of defining types of networks which a network designer wishes to explore by a software system using interactive computer graphics.

The network defining language is not a graphical language but a problem oriented defining language developed for use with interactive graphics in CAD. It is used to define the structure and all relevant properties of objects in networks, and relationships between these objects within the networks. The graphical properties of these objects form only a subsidiary part of all the properties.

The network defining mechanism and language relieve a network designer, using the software system, from any constraints imposed by the system on the types of networks he is able to study. The method can be used by a wide range of network designers with no programming experience in graphics.

Acknowledgements

I wish to thank Professor W. S. Elliott, Department of Computing and Control, Imperial College, London, for his encouragement and advice. This research was sponsored by an SRC Grant.

References


Book reviews


The term ‘software engineering’ is often used to describe the field of software system development, to emphasise that the construction of a compiler or an operating system is not an art but is a process to which the well-known techniques of engineering design can usefully be applied. Engineering is usually based on a sub-structure of science, and this book is a noteworthy contribution to the scientific sub-structure of compiler writing. It is also noteworthy in that it must be the first book about compiling techniques for which ‘... previous exposure to computer programming is helpful but not necessary’.

The designer of a compiler may try to optimise the generated code at two levels. There is ‘within-statement’ optimisation that endeavours to exploit the architecture of the target machine in the code generated for individual statements, e.g. making best use of the available registers for intermediate results. Much more difficult is global optimisation which aims to improve efficiency by removing non-varying quantities from loops, recognising patterns in the array elements, and reordering the statements of the program to avoid redundant re-computation of sub-expressions. The gains to be obtained are considerable: the IBM H-level FORTRAN compiler is reported to have produced object programs occupying 25% less space and using 40% less processor time than the programs produced from the same source by the non-optimising G-level compiler. However, the dangers are equally great: after it was released to users, it was discovered that the ICL optimising FORTRAN compiler sometimes generated incorrect code. Thus the designer of an optimising compiler needs to be sure that his transformations retain the meaning of the program, and it is here that science comes in, the engineer the assurance that his artefacts will perform as expected.

The book under review provides such a foundation for global optimisation. Since such optimisation involves a detailed analysis of the flow of control in a program, the approach is to express the structure of the program as a directed graph. The transformations required for optimisation are now transformations on the graph, and the mathematics of graph theory can be used to give a rigorous proof of the validity of these transformations. The first seven chapters of the book are devoted to establishing the necessary (and considerable) body of mathematical background: there then follow a series of chapters each dealing with a particular class of optimising transformations. Although these transformations are expressed in abstract mathematical form, each chapter begins with a number of concrete examples to give the reader a feel for what is to be achieved. Finally three appendices cover optimising algorithms (written in APL/360), an overview of the phases of an optimising compiler, and a discussion of the influence of partial recompilation (‘incremental compiling’) on the global optimising process.

The book is a most valuable contribution to the literature of compilers, though the mathematical standard will make it inaccessible to many workers in the field. Those with the necessary degree of mathematical sophistication to cope with the fine detail will, but exceedingly rewarding. For the rest it is a potted of the shape of things to come, and a valuable counter-example to the proposition that the computer scientist need have no mathematical ability.

D. W. BARRON (Southampton)


As well as editing the collection, Pushkin is either an author or has his work referred to in rather more than half of the nineteen papers. A heuristic process is seen as one which constructs a new action aimed at the achievement of some goal in a situation which is new to the system performing the process. Thus by heuristics is meant the science which studies the laws governing the design of new actions in new situations. In ‘Toward a definition of heuristics’, Pospelov, Pushkin and Sadovskii deny that a computer program can be heuristic. It is clear that this view is strongly held but they fail to make an adequate case to support it. They argue, for example, that a potential infinity of languages for the formation of a model is needed by a system that is to be heuristic, but it is not obvious to the reviewer that this must be so in a sense that makes it impossible in a computer system.

The main thesis of the book, insofar as there is one, is that the simple maze view of heuristic search is not adequate and that we must look at systems which can both construct internal models and can also radically revise them in the course of problem solving. This leads to the most important achievement of the book, its insistence on a close relationship between theory and psychological fact. For example, one paper puts forward the view that complex tasks can be automated most successfully by first conducting a psychological study of the human methods of solution.

By far the longest of the four sections is Experimental Studies of Heuristic Processes. This contains studies of human problem solving, mostly using protocol analysis, but in one interesting case, by using recordings of the eye movements of chess players whilst they decide upon a move.

The book ends with a short section on reflexive control, i.e. on attempts to win in human and man-machine conflict situations.

**Problems of Heuristics** is a lively collection containing within it a range of differing, and at times contradictory, views. An editorial guide to those various attitudes would have been a useful addition to this interesting book.

E. A. EDMONDS (Leicester)