# Environments for Mathematical Research: a project report

Meurig Beynon

Dept of Computer Science, University of Warwick, Coventry CV4 7AL

Abstract

This paper reviews principles and software prototypes developed over the last three years in connection with the design and implementation of computer environments such as might support mathematical research in the Automatic Groups Project. It takes the form of a summary of the principal theoretical ideas developed together with an account of the current status of their practical implementation. An illustrative example, as yet unimplemented, but similar to examples that have been defined using existing software prototypes, is included. Major practical contributions to this research programme have been made (not directly under the auspices of the project) by two postgraduate research students in Computer Science: Messrs Yun Wai and Yun Pui Yung; seminal ideas have also been developed in collaboration with Mr Alan Cartwright, Dept of Engineering, and Dr Steve Russ, Dept of Computer Science.

## 1. Introduction

### 1.1. History of the Project

The research carried out under the direction of the author in connection with the Automatic Groups Project has been complementary to the mathematical research and the development of problem-specific algorithms and computer programs. Its initial focus was upon the extension of an existing prototype system based upon the ARCA notation for the description of groups and Cayley graphs [1,2]. The scope for programming support for this aspect of the research programme was limited because of the demands made by the essential tasks of algorithm design and implementation directly relevant to the study of automatic groups. The research summarised here was carried out using equipment provided by the project but by personnel funded from other sources. For this reason, the principles and the software prototypes we have developed have so far been applied to problems ostensibly more closely connected with engineering than pure mathematics. As this report will clarify, the choice of application is insignificant inasmuch as we have identified generic methods to assist the design and implementation of user interfaces and environments. The potential for applying the principles underlying the ARCA notation to creating environments for mathematical research has been clearly demonstrated. The computer representations and computational tools appropriate for the study of automatic groups are now sufficiently mature for the adaptation of our methods to this specific application to be realistic.

### 1.2. Context and Motivation for the Research

The primary computer tools to support mathematical research can be viewed as sophisticated calculators. That is to say, they are in essence programs that compute the value of an algebraic expression in which the operators are chosen to suit the application and the operands are supplied by the user. In the Automatic Groups Project, for instance, a fundamental role is played by a program that accepts a group presentation as an input and returns a set of finite state automata as output. A mathematical researcher may wish to use such a program in conjunction with other programs that generate graphical displays of finite automata or Cayley diagrams. To allow such programs to be integrated within a single package, it is essential to design the computer representations and programs with integration in mind. The primary programming task in the Automatic Groups Project has been to encode the basic operators so that they make use of standard representations and can be conveniently combined and adapted. The research described in this report also reflects a computational perspective, but addresses more abstract issues with significant practical implications in the longer-term. These include:

- What kind of environment is required to allow the best practical use of the programming tools developed?
- What form of interface and mode of interaction is most appropriate?
- What is an appropriate abstract computational framework within which to interpret user-computer interaction in such an environment?

The partial solutions to these problems proposed are generic in nature, and have not been developed with specific reference to Automatic Groups.

## 2. Developing Computer Environments for Mathematical Research

### 2.1. From Calculators to Environments

A plausible model for an appropriate environment for specialised mathematical research is perhaps a special-purpose calculator with a sophisticated user-interface. If we adopt this model, there are numerous problems in designing and implementing a suitable calculator. It is exceedingly difficult to meet the needs of all possible users. By choosing the set of primitive operators appropriately, we can make the set of possible expressions richer and more versatile; by introducing additional parameters, we can give the user greater control over input and output formats; by developing special-purpose interfaces to existing calculators, we can enhance the scope and expressive power of function evaluations. Useful as these techniques are, they fail to address a fundamental problem: not all the information about how a user will wish to interact with and exploit the calculator can be known in advance. The end-result of the design exercise is typically a complex package with far more options than any single user requires, that has to interpreted via an encyclopaedic manual. Even then, there remain problematic issues that cannot be addressed by preconceived built-in extensions to the calculator. The user may need to use the calculator in conjunction with a utility not previously considered, or to exercise dynamic control over the output format – perhaps based upon aesthetic judgements. It may also be technically impossible to specify all the essential operators so

that their evaluation is fully automatic – for instance, the task of realising a finite automaton as a graph may require a form of cooperation between user and program.

## 2.2 User vs Programmer?

The limitations of computing environments based upon the calculator paradigm are well-recognised – the issues raised above are frequently debated in connection with engineering design support systems. A pertinent question in this debate is the extent to which the user can participate in the role of the system programmer. The choice of options by the user can be viewed as a simple - though still preconceived - form of reprogramming of the system. A more radical form of system reprogramming would permit the user to modify the system code directly. In effect, this is the activity practised during the development phase of a support system, where the system programmers modify their program in the light of what they perceive as the users' perspective.

The "User as Programmer" dilemma has very significant implications. If the users' role is confined to the evaluation of prespecified functions (albeit complex higher–order functions representing operators whose action can be modified by supplying different parameters), the environment is essentially specified by a functional program. That is to say, the operators available to the user can be defined in a functional programming system[1], with the attendant advantages of transparent mathematical semantics that such a system provides [19]. Once the users' role encompasses unpredictable extension or modification of the operators as initially supplied, the procedural component of the interaction between the user and the system can no longer be disguised. In practical terms, this means that a computational paradigm with provision for modelling changes of state is essential for a satisfactory support environment. The object-oriented description of the programming tools developed in the Automatic Groups Project is an acknowledgment of this need; it does not simply specify the basic operators as abstract functions, but describes the objects that generate these functions in procedural terms. This specification may be primarily intended to assist system programmers in the modification and development of the tools, but might also serve to enhance the prospects for user reprogramming.

## 2.3 From Calculator to Spreadsheet

A calculator is most appropriate when the user simply wishes to determine the value of an algebraic expression. The result of such an evaluation is either committed to memory, recorded on a memo, or used directly to make a decision, and discarded. In typical problem-solving use, the manner in which the results of evaluation are processed and recorded by the user plays a most significant role. Access to a calculator is no recipe for routine problem-solving. The need to record intermediate results, and to establish mechanisms by which these can be subsequently referenced by the user, provides additional motivation for a state-based computational paradigm. This is to acknowledge that some functions that the user wishes to evaluate require cooperation between the user

[1] subject to there being appropriate types to represent graphical objects etc

and the computer.

A naive solution to the problem of recording intermediate results is to provide a simple data-base of explicit values. It has many drawbacks. There are typically data dependencies between the values recorded, so that if one value is subsequently altered all dependent values must also be recomputed. For most effective use, the manner in which the intermediate results are to be accessed and presented to the user has to be considered. If an explicit value is used to record the coordinates of a geometric entity such as the point of intersection of two lines, problems of numerical approximation may necessitate recomputation if the resolution of the screen display is altered, whilst anomalous conditions may arise in storage if the lines turn out to be parallel. In the users' conceptual view of a problem, the data dependencies are exceedingly important in determining the strategy for evaluation and the style of presentation.

These considerations help to explain why, in many applications, a spreadsheet is preferable to a calculator. In a spreadsheet, the process of re-evaluation to take account of data dependency is automated and the presentation of results can be specified[2]. Computational experiment in mathematics research resembles the 'what if?' activity involved in using a spreadsheet more closely than it resembles pure function evaluation. A conventional method of devising new theorems involves detecting common patterns or identifying exceptional conditions in a class of particular instances; this demands an environment in which relationships between diverse mathematical objects (such as group presentations, finite state automata and graphs) can be established and explored.

### 2.4. Computational Paradigms for Mathematical Environments

Devising an appropriate computational paradigm is a crucial problem in developing good environments to support mathematical research. The object of the above examination of the computational issues underlying the development of mathematical environments is to show that conventional programming paradigms provide inadequate models, and – in typical use – conflict with each other in a most confusing fashion. A brief recapitulation of salient points emphasises this:

• A functional programming perspective best suits the image of the user for whom problem-solving is equivalent to evaluating a complex function from a preconceived and prespecified class. Such a user will resemble a mathematician who uses a calculator in conjunction with a sheet of paper. If the computational paradigm is to take account of how the sheet of paper is used, some method of encoding state information is required. Unless the user has the privileges of the designer of the functional program, viz the power to edit the definitions of the basic functions, the only way to describe the possible interactions between the user and the system is to preconceive the form of this interaction and to exploit lazy evaluation and explicit representations of dialogue state information to encapsulate the interactive process (cf [19]). It is questionable whether such a degree of precognition about the functions required and their mode of use can be attained without

[2] in this context, a 'spreadsheet user' is privileged to assign and modify the formulae attached to cells

comprising the power and utility of the system. On the other hand, giving the user the privileges of the designer of the functional program invokes a procedural programmig paradigm, thereby forfeiting the formal merits of a declarative programming style.

• If the user is to be able to modify the functions offered by the system and to customise the modes of data presentation and access, an object-oriented programming paradigm is arguably the best technique that is currently available. That is to say, if the mathematical environment offered to the user is implemented by abstractly specifying the objects and methods that generate the environment, the technical problems of modifying its behaviour are reduced. The potential implications as far as the user view is concerned are obscure. Whilst the system programmer can readily take advantage of the nature of the underlying implementation during system development, the knowledge the user would require to perform similar tasks is apparently difficult to express in any way consistent with domain-oriented knowledge. This may suggest that the only satisfactory method of designing an interface to the user is to presume the same expert mathematical and programming skill that is appropriate for the system designer, and allow the user to interact with the environment and to edit and recompile the objects that define its implementation. In this analysis, there is no indication that an object-oriented paradigm, despite its procedural nature, is an appropriate way of modelling the process in which the user participates when engaged in problem-solving.

• The spreadsheet does not in itself represent a powerful computational paradigm. Despite this, its method of representing state information has significant advantages over other procedural models. Characteristic of this method is the combination of state information determined by the explicit values associated with display cells and information about the data dependencies governing transition between states[3]. Information of this nature arguably plays a fundamental role in formally describing how a user can conveniently conceive and manipulate objects within the application. It also helps to formalise that part of the problem-solving process that is not captured by expression evaluation, viz the evaluation strategy, and the storage, retrieval and presentation of intermediate results.

## 3. Principles behind our Approach

### 3.1. A Computational Paradigm for User-Computer Interaction

The computational paradigm for user-computer interaction proposed in this paper rests on a basic premise. If the user is to have control over some aspect of the system behaviour, the mode of interaction must be such that the user acts within a formal framework in which this aspect of the state is faithfully modelled with respect to

- what privileges the user has to change the state
- what consequent change of states occur when a privilege is exercised.

The primary modelling technique that will be used to guarantee this is the principle underlying the spreadsheet[4], viz the representation of state by sets of variables whose

[3] in contrast to (pure) constraints, which only specify the abstract relationships between values, not the procedural mechanisms that must be invoked to maintain them in the transition to another state

values are specified by defining formulae. It should be emphasised that this technique addresses a concern entirely separate from the development of efficient or ingenious algorithms to evaluate defining formulae.

General principles have been developed to formulate state information using sets of definitions. These are illustrated in their simplest form in the abstract operation of a spreadsheet. Each cell of the spreadsheet can be viewed as a variable whose value is either explicitly defined or specified by a formula expressing its value in terms of that of other variables. The data dependency between variable values established by the set of variable definitions will be assumed to be acyclic. The nature of the defining formulae is determined by an underlying algebra of values and operators. In the case of a conventional spreadsheet, this might a 2-sorted algebra including scalars and arithmetic operators together with alphanumeric strings and associated operators on strings.

### 3.2. Definitive Notations

A definitive (definition-based) notation is a simple programming medium in which sets of definitions resembling those underlying a spreadsheet can be formulated. The precise syntactic form of the notation is influenced by the values and operators that appear in the defining formulae, as determined by the underlying algebra. The choice of underlying algebra in turn reflects the nature of the interactive application. Several definitive notations have been developed in connection with our research into environments for mathematical research. These include ARCA – for displaying and manipulating combinatorial diagrams, DoNaLD – for line drawing, and SCOUT – for screen layout.

ARCA was the first definitive notation to be developed[5]. The data types in its underlying algebra were designed to represent n-dimensional realisations of combinatorial graphs. Because the model adopted for such graphs is based upon the Cayley diagram, the edges may have associated colours and directions. ARCA is well-adapted for expressing the kind of information that is centrally relevant to the Automatic Groups Project. For instance, it can be used both to specify abstract finite automata and to describe a layout for its realisation as a graph. Defining formulae in ARCA can be used to establish rich data dependencies such as are needed to express symmetries between component parts of a Cayley diagram. For instance, if x and y are the generators of a finite group and a and b are elements such that $b = a.x.y.x^{-1}$, it is possible to assert that the node of the associated Cayley diagram that represents the group element b is defined by rotating the node representing a through $2\pi/3$ about the origin. Further details of the notation, and examples of its use, can be found in [1,2,5].

The edges of a combinatorial graph such as a Cayley diagram are abstractly defined by adjacency relationships between vertices. In ARCA, the directed edges of a particular colour within such a diagram are specified by a (partial) permutation of the indices of its

[4] the principles discussed here apply to a spreadsheet stripped of its tabular interface

[5] though similar principles had earlier been applied to graphics by B Wyvill [24]

vertices. DoNaLD – a definitive notation for line drawing, is complementary to ARCA. Its basic data types are points and lines in the plane. Whereas the use of ARCA is appropriate when the abstract vertices and edges of a graph have an interpretation independent of their geometric realisation, DoNaLD is adapted for describing aggregates of points and lines whose interpretation is rooted in their geometry alone. Further details of DoNaLD, and examples of its use, can be found in [4,5,6,14].

The SCOUT notation is designed for describing screen layout [17,12][6]. A set of SCOUT definitions abstractly specifies the nature, content and location of a set of windows and the way in which they are composed to make up a display. The specification is abstract in the same sense that a window manager delegates control of particular windows to other application programs rather than specifying their content directly. For instance, a SCOUT specification may stipulate that a display is made up of three windows, one containing a geometrical figure specified in DoNaLD, another an ARCA diagram, and the other a textual commentary to be displayed in a specified format. Much of the expressive power of this method of representing the state of the screen display derives from being able to describe relationships between information represented in many different forms. This is illustrated in [12], in which the role of SCOUT in specifying interfaces is discussed in detail.

## 4. The Application of Definitive Notations

### 4.1. An Illustrative Example

The issues surrounding the use of definitive notations to define computer environments for mathematical research will be discussed with reference to an illustrative example. This example cannot as yet be implemented with existing prototypes, but the essential principles and techniques required have been practically demonstrated in other application areas [12]. The mathematical concepts underlying the example are similar to those being investigated in the Automatic Groups Project. The example itself is motivated by collaborative research currently in progress with Prof Mike Atkinson of the Department of Computer Science at Carleton University, Ottawa [18].

Consider the Cayley diagram $\Gamma_n$ of the symmetric group $S_n$ relative to its standard presentation as generated by the transpositions of adjacent symbols $\tau_i = (i,i+1)$ for $1 \leq i < n$. The vertices of the graph $\Gamma_n$ can be ordered in such a way that v precedes w if there is a minimal representation for w as a product of generators with a prefix that defines a similar representation for v. The order on $S_n$ is known as the weak ordering [22]; when realised appropriately, the combinatorial graph underlying $\Gamma_n$ defines the associated Hasse diagram.

In the weak ordering, the least element is the identity I and the greatest R: the permutation mapping i to n-i+1 for $1 \leq i < n$. Each shortest path from I to R in $\Gamma_n$ defines a

[6] the design and implementation of SCOUT is the work of Y P Yung

sequence of of N=n(n-1)/2 transpositions $\tau_{i(1)},\tau_{i(2)}, ...,\tau_{i(N)}$, where $\tau_{i(1)}\tau_{i(2)} \cdots \tau_{i(N)} = R$. This shortest path can be represented by the sequence of indices i(1),i(2), ..., i(N). Two shortest paths will be deemed equivalent if one can be derived from the other by exploiting the fact that $\tau_j\tau_k = \tau_k\tau_j$ when |j-k|>1. To determine the equivalence class of shortest paths containing the sequence of indices i(1),i(2), ..., i(N), it suffices to construct a poset with N elements 1,2,...,N ordered by the relation $\langle$, where j$\langle$k if there is a sequence $j= j_0<j_1< ...<j_r=k$ such that $|i(j_t)-i(j_{t+1})|=1$ for $0\le t<r$. It is then easy to verify that i(σ(1)),i(σ(2)), ..., i(σ(N)) is a shortest path equivalent to i(1),i(2), ..., i(N) if and only if σ(1),σ(2), ..., σ(N) is a linear extension of the poset $< \{1,2,...,N\}, \langle >$.

An alternative method of generating shortest paths is to consider a configuration of n lines in general position in the plane, i.e. such that each pair of lines intersects and no triple of intersection points is collinear. Such a configuration, appropriately scaled, can be realised by choosing suitable ranges of n points on the pair of parallel lines $x=\pm1$, labelling the points in each range by the indices 1,2,...,n in the order in which they are encountered by a line rotating clockwise about the origin, and connecting pairs of points with the same index. A representative of a uniquely defined equivalence class of shortest paths – to be specified as a sequence of indices i(1),i(2), ..., i(N) – can now be derived from the configuration. To this end, each of the N points of intersection is assigned a distinct label from the set {1,2,...,N} in such a way that their ordering by label is consistent with their ordering by x-coordinate. The index i(k) of the k-th point of intersection (X,Y) is then defined to be 1+z, where z is the number of lines whose y coordinate exceeds Y at x=X.

One objective of the mathematical research referenced above is to understand the relationship between equivalence classes of shortest paths and configurations of lines. The significance of this relationship and the status of our understanding is outside the scope of this report. In the present context, it is the nature of the computational environment that would be most helpful in exploring this relationship that is of interest.

Figure 1 below depicts a simple scheme of interrelated diagrams that might be used to animate the correspondence formally outlined above. On the left is a window in which a configuration of 4 lines is displayed [C]; on the right a window in which the corresponding equivalence class of shortest paths in the weak ordering of $\Gamma_4$ is indicated; in the middle a window in which the associated poset is depicted in two different ways [P and P']. The graphical windows are complemented by a textual windows in which the sequences of indices associated with the chosen equivalence class of shortest paths are displayed, together with other data, such as the number of minimal triangular regions defined by the configuration C.

The significance of Figure 1 is not fully conveyed by a static picture. The ranges of points that define the configuration C are to be defined by parameters that are under the control of the user: when these parameters are modified, the other elements of the display, functionally dependent upon the configuration, also change (cf Figure 2). In

principle, Figure 1 could be specified so that such a dependency is established by formulating appropriate sets of definitions in DoNaLD, ARCA and SCOUT. The precise form of these definitions is not important: for more insight into the technicalities, the reader should consult [12] for an illustrative example of comparable complexity that has been implemented using our current software prototypes. The principles behind the computational paradigm for interaction can be analysed without detailed specification of the set of definitions.

The nature of the interface associated with Figures 1 and 2 depends in a radical way upon the status of the user with respect to the set of definitions. If the set of definitions is presented in such a way that the user can do nothing other than modify the parameters determining the configuration C, the paradigm for interaction resembles conventional spreadsheet use, where the defining formulae for cells are beyond user control. In practice, it is more reasonable to imagine that the user has unrestricted privileges to change the definitions, and operates in the role of both programmer of the system and user. After all, the functional dependency that underlies the interface is highly problem-specific. Figure 1 could not be usefully generalised to much larger values of n, because of the infeasibility of displaying $\Gamma_n$, and would typically be conceived by the user at a particular stage in the problem-solving process. For instance, at a later stage in the development of the mathematical research, a user might be able to implement a complementary functional dependency, whereby the choice of an equivalence class of shortest paths in $\Gamma_4$ determined a new configuration C.

Some compromise may be preferable to absolute control over all definitions. Access to the SCOUT definitions enables the user to reconfigure the display, but it is not to be expected that a mathematical specialist should have such detailed knowledge of attributes of windows as is required in using SCOUT for page layout in desk-top publishing. A menu interface that permits selective forms of window specification and parametrisation may be sufficient. The need for user-control over the state of the display may be data-specific, however. For example, to study the change in the geometric form of the diagram P' as the configuration C approaches singular states in which points of intersection become coincident, it is necessary to increase the vertical scale in the presentation of P'.

Experience with our present prototypes suggests that the benefits of a definitive representation of state are best realised when the user enjoys many of the privileges of the system designer. This concept has also been developed in other systems (such as Hypercard), but our choice of state representation offers particular advantages. There is consistency between the representations of application-oriented and system-oriented knowledge: the same principles are used to define the interrelationship of windows as the underlying mathematical dependencies between diagrams. Customisation and experimentation by the user is assisted because state information is encoded in such a way that the consequences of redefinition are transparent to the user, and previous states are easy to recover. A typical strategy for system development is the refinement of a set of definitions through additional parametrisation: this has the advantage of guaranteeing

that the capabilities of the new version subsume those of the old; the user can safely and conveniently use similar techniques for enhancement and adaptation. Sets of definitions are also easy to integrate; for instance, it would be possible to use a pre-existing ARCA specification of $\Gamma_4$ to construct the interface represented by Figures 1 and 2 efficiently. Such speed of prototyping is very significant in determining whether a mathematician chooses to operate with pen and pencil rather than use a computer system.

## 5. Current Status of the Research

There are three principal aspects to our work on the application of definitive principles to the development of environments for mathematical research. The future development of the computational paradigm for interaction depends upon a fuller understanding of

- the philosophy of programming that it represents
- the practical techniques that can be used to support its implementation
- the status of its foundations in formal mathematical terms.

Each of these issues is briefly discussed below. For further discussion and additional references, the interested reader should consult [13].

### 5.1. A Philosophy for Definitive Programming[7]

#### 5.1.1. Definitive State Representations and Informal Semantics

A mathematician making use of a computing environment for mathematical research has to manipulate many different kinds of information. Mathematical knowledge, knowledge about system response and perhaps even knowledge about the operation of the computer itself may be required. The use of a calculator relieves a mathematician of few problems in organising and recording this knowledge. Intermediate results can be stored electronically, but this does not assist their interpretation, upon which the computational strategies adopted by the user crucially depend. To be precise, the calculator only encapsulates generic knowledge (e.g. about arithmetic relationships and operators, such as "what is the result of multiplying x by y") : in no way can it be adapted by the user to reflect problem-specific knowledge (e.g. "what it means to say that p denotes the profit from a transaction").

The spreadsheet has greater potential in this respect. By introducing a formulae that defines the profit (p) in terms of manufacturing cost (mc) and selling price (sp), a spreadsheet user creates a simple state-based model of a perceived relationship between values. The claim that this mode of representation assists the user in interpreting the computations performed in application-oriented terms can be justified: a third-party, having no knowledge of the intended meaning of the spreadsheet cells that display the values of p, mc and sc can verify that this is an appropriate interpretation either by referring to the defining formula for p, or – with less confidence – by observing the results of experimenting with the values assigned to mc and sc.

---

[7] this term is used as an abbreviation for "programming using definitive representations for state"

Definitive state representations derive their power from the simple principle illustrated in the spreadsheet. A set of definitions can be used to represent knowledge about functional relationships between data that persist throughout transition from one state to another. Subject to the usual limitations of the experimental method, this knowledge can be gained through experiment within the application (e.g. as when a mathematician studies many configurations of lines in Figure 1 to discover how to construct a configuration of 4 lines to realise any given shortest path in the weak ordering of $S_4$), then encapsulated through the formulation of appropriate sets of definitions (e.g. formulae to express the parameters of a configuration of lines in terms of a choice of shortest path).

Our underlying thesis is that the informal semantics of a conceptual or computer model of an external system is determined by the set of transformations that can be performed on it, and that definitive representations of state play a fundamental part in formally specifying this set (cf [14]). This link between the informal semantics of variables and the use of definitive state-transition models helps to explain the special qualities of the spreadsheet and its relatives. Within such environments the user can dynamically construct faithful models of external state-transition systems, representing knowledge that is specific to the problem and the context, rather than generic and preconceived. These external systems may be based upon abstract objects (e.g. the relationships between mathematical entities as in Figure 1) or physical objects (e.g. the relationships between objects in a room, as modelled in [6]); they may also refer to the computer environment itself (e.g. the current state of the machine display). The limitations of this technique are determined solely by whether the underlying data types and operators can conveniently represent the perceived state of an external system.

5.1.2. Comparison with other Paradigms

The computational paradigm represented by definitive programming, as illustrated above, is essentially state-oriented. This characteristic differentiates it totally from functional programming, despite strong superficial resemblances. Functional programming systems such as Miranda commonly make use of a script [19] comprising sets of definitions ostensibly similar to a specification in a definitive notation. The interpretation of these sets of definitions is fundamentally different. A DoNaLD specification is to be interpreted with reference to state; its informal semantics depends upon acknowledging a role for procedural abstractions. In contrast, a MIRANDA script is formally static; it is to be interpreted as an abstract description of the function computed by a procedure that is in general unspecified. In neither context does the interpretation of a set of definitions depend upon knowing what procedures are used to evaluate defining formulae, but whereas in a functional program the computation is represented by the functional abstraction expressed in the definitions, in a definitive program the set of definitions simply determines a state of the computation.

In this paper, as in the Automatic Groups Project, only the application of object-oriented methods to the specification of underlying evaluation programs has so far been considered. It is invidious to compare definitive models of states of knowledge with

object-oriented models of program states. However, the construction and simulation of application-oriented models was the original motivation for object-oriented programming [20]. It is then appropriate to compare object-oriented and definitive methods of specifying state-based models of the external system.

In an object-oriented approach, system state is modelled by identifying generic pieces of local state. Each such piece is represented by an object within an abstractly specified class. The global state of the system is determined by the interaction between its constituent objects; this is modelled by message passing[8]. If an object-oriented model is to emulate a definitive model of an external system, it is in general necessary to represent indivisible state-transitions that propagate across object boundaries. Such representation is problematical since it entails specification of the synchronisation between message passing and internal state-transition of objects. For this reason alone, an object-oriented representation of state cannot readily serve the same role as a definitive model in representing in what ways (as in which redefinitions are possible) and with what implications (as in what data dependencies must be respected) the state of the external system can be transformed.

### 5.1.3. Programming as Modelling

The use of definitive state representations, as illustrated in §4, describes only a part of the computational activity involved in interaction between a mathematician and the computer. The fact that aspects of the state of the computer system itself, such as the contents of the display, can be abstractly specified by sets of definitions under the direct control of the user, allows the mathematician to carry out some tasks that might conventionally involve reprogramming the system. There remain aspects of the computer's activity beyond the control if not the concern of the user, such as the procedures that are performed by the computer in evaluating formulae and maintaining the values of variables[9].

The significance of introducing definitive notations such as ARCA, DoNaLD and SCOUT can only be fully appreciated by looking in more detail at the relationship between programming and modelling. As explained above, sets of definitions are introduced to establish a correspondence between the state of external objects, as perceived or conceived, and the state of a model internal to the computer. The nature of this correspondence is inextricably connected with experience: its validity can only be confirmed by appeal to how an external system behaves or seems to behave. A major part of the design of a system based upon definitive state representation is deciding where to establish the interface between internal models and external experience.

There are two aspects to establishing this interface. One is concerned with devising

[8] the modern concept of object-oriented programming, motivated in part by the need for 'information hiding' in modular program development, represents a departure from the original idea behind Simula, where the objective was faithful application-oriented modelling [21]

[9] though there are prospects for applying similar principles to bring these aspects under user control

internal representations that can conveniently be interpreted in terms of the external application. This involves the choice of underlying algebra for the definitive notations to be used. The other addresses the complementary problem: ensuring that the computer system animates the internal representations effectively. In both aspects, there is an essential need to identify where assumptions about the behaviour of the system, outside the scope of the formal model, will be made. Amongst other things, these may take account of knowledge about physical laws, constraints placed upon user response, and the utilities available on the machine.

The adoption of definitive state representations greatly simplifies the transformation from one data representation to another. In the process of data refinement, power to describe the indivisible propagation of change of state can be exploited to ensure that the informal semantics of a set of high-level definitions is respected. This accounts for an emphasis – evident in our research – on cognitive aspects of the modelling process concerned with interpreting the relationship between the values of variables in the internal model and corresponding states of the external system.

### 5.2. Mathematical Foundations

Computational paradigms that introduce state are challenging to describe in formal mathematical terms. The procedural variable differs from the mathematical variable [19]. Procedural programs violate the principles of referential transparency advocated by the declarative programming school. The status of procedural abstractions in definitive programming is such that the idea of ascribing a formal semantics within a conventional logical framework is perplexing[10]. The clarity of the computational paradigm is most evident if the concept of an object with an identity but capable of assuming many states is admitted as a primitive abstraction. When viewed in this light, definitive notations appear to offer a useful theory of reference that avoids the problems of associating symbolic and actual values that is evident in many areas of computer science [23].

Definitive state representations provide a simple and expressive way to associate symbolic and actual values. The association between symbol and value is neither fixed and permanent, as in a declarative programming framework, nor as transient and quixotic as a procedural variable that is meaningful only during the execution of a program. These qualities have been exploited in a number of interesting ways in our existing prototypes: in ARCA, where it is used to provide a flexible association for between abstract variable templates and conformal families of definitions [5], in the design of DoNaLD operators, where reference plays an essential role in the specification of operators of complex type [9], and in the definitive notation for geometric modelling CADNO, where it is used to attach a realisation to an abstract simplicial complex [7]. Different kinds of abstraction can also be formally represented using definitive state representations [5].

[10] it is difficult to believe that our intuition about procedural objects, whose possible changes of state are apparently subject to no absolute restriction, can be expressed via orthodox mathematical variables whose values are not subject to change

An important area for future investigation concerns the direct use of definitive state representations in applications where explicit state-transition modelling is currently employed. Such representations can describe complex state-transitions in a succint way, but the principles have so far been applied to modelling at high-levels of abstraction. For instance, it may be possible to develop techniques for describing the finite-state machines introduced in lexical analysis and parsing, or those that arise in the Automatic Groups Project itself.

### 5.3. Implementation Issues

Generic methods can be devised for manipulating definitive state representations. They depend upon automatically monitoring data dependency and introducing efficient strategies for maintaining current values through selective re-evaluation. The essential mechanism required for implementation on conventional architectures is the invocation of procedures in response to changes to the values of variables. Such procedures can perform all the processing necessary to maintain the computer model: updating variables and manipulating the display.

The DoNaLD, SCOUT and CADNO prototypes are all implemented using a common interpreter for the special-purpose programming language EDEN [16,6][11]. EDEN was primarily designed for implementing definitive notations in the UNIX/C environment. Its features include definitive variables of basic C types (viz. lists, strings and scalars), conventional procedures, and actions in the form of procedures whose execution is triggered by changes to the values of specified variables. Generic methods of interfacing with system utilities such as SUNCORE or X-windows have also been developed.

The standard method of implementing a definitive notation involves devising EDEN representations for the data types and operators in the underlying algebra. Input definitions can then be translated into EDEN definitions at a lower-level of abstraction. Where the value of a variable represents the state of a displayed entity, such as a geometric point or shape, EDEN actions to maintain the display must also be generated by the translator. This technique has been used to create a DoNaLD interpreter, and can be adapted to encompass additional features, such as the definition of predicates to be interpreted as monitors – displaying messages as and when particular boolean conditions prevail, or as imposed constraints – revoking any user definition that leads to a violation.

At present, the practical integration of several definitive notations within a single SCOUT interface, as demonstrated in [12], depends upon using EDEN as a common form of intermediate code in conjunction with an interface to X-Windows. EDEN then provides a uniform internal representation of definitions allowing data dependencies between different types of windows to be established. The major obstacle to incorporating ARCA windows in such a framework, as envisaged in §4, is the fact that its prototype

[11] the design and implementation of EDEN is the work of Y W Yung

implementation, though adapted for X-Windows, is an interpreter written directly in C.

Conclusions and Future Prospects

General principles that can be used as the basis for designing and implementing computer environments to support mathematical research have been developed and demonstrated. The use of state representations based upon sets of interrelated definitions of variables provides an effective means of combining symbolic and actual data. Its investigation gives insight into the semantics of user-computer interaction, non-procedural programming paradigms and issues concerning reference with significance for the foundations of computer science. This has practical implications for many aspects of data representation and presentation, as illustrated by a variety of software prototypes.

The essential groundwork required to address the needs of a specific area (such as the study of Automatic Groups) has been successfully completed. The immediate technical objectives can be met by the re-implementation of ARCA in EDEN to take account of features of the notation that cannot realistically be handled by modifying the current implementation and through the development of an interface to allow the assimilation of the objects and procedures that have now been developed in the Automatic Groups Project. An appropriate method of carrying out both these tasks has been devised and demonstrated in existing prototypes.

The future development of this work is closely linked with the study of general-purpose programming based on definitive state representations. Much research has already been done in this direction [13,15]: its potential implications for the implementation of environments for design applications is described in [7]. In the short-term, there are more effective ways of developing useful practical systems than investigating the prospects for the universal application of a new programming paradigm. Further work should address a detailed study of existing mathematical software with a view to integration. The principles described in this paper may also be useful as a development tool for creating customised systems: this entails specifying the protocols for user-computer interaction so that the code for the restricted system can be compiled.

## References

1. A definition of the ARCA notation
Theory of Computation Report 54, University of Warwick 1983

2. ARCA - a notation for displaying and manipulating combinatorial diagrams
Computer Science Research Report 78, University of Warwick 1986

3. Paradigms for programming
Report on University Stirling Workshop on Functional and Logic Programming,
Alvey Software Engineering Mailshot, December 1986

4. (with David Angier, Tim Bissell and Steve Hunt)
DoNaLD: a line drawing notation based on definitive principles
Computer Science Research Report 86, University of Warwick 1986

5. Definitive principles for interactive graphics
NATO ASI Series F, Vol 40, Springer-Verlag 1988, 1083-1097

6. (with Y W Yung) Implementing a definitive notation for interactive graphics
New Trends in Computer Graphics, Springer-Verlag 1988, 456-468
[An example extracted from this paper is included in the Appendix]

7. A definitive programming approach to the implementation of CAD software
Intelligent CAD Systems II: Implementation Issues, Springer-Verlag 1989, 126-145
(Appendix with A J Cartwright) A definitive notation for geometric modelling
Proc 2nd Eurographics ICAD Workshop, CWI Amsterdam, April 1988

8. (with A G Cohn)
Representing design knowledge in a definitive programming framework
IFIP WG 5.2 on Intelligent CAD, Cambridge, Sept 1988

9. Evaluating definitive principles for interactive graphics
New Advances in Computer Graphics, Springer-Verlag 1989, 291-303

10. (with S B Russ) The Development and Use of Variables in Mathematics and Computer Science
Proc IMA conference: "The Mathematical Revolution Inspired by Computing" April 89,
to appear

11. Definitions as a framework for design
Proc 3rd Eurographics ICAD Workshop, CWI Amsterdam 1989

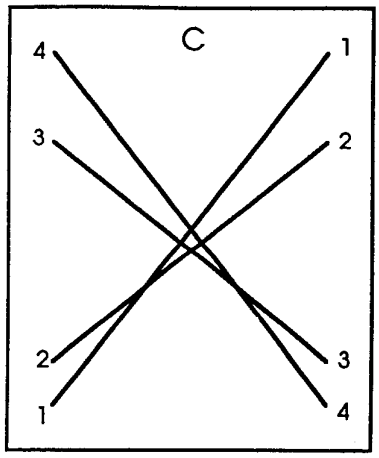
12. (with Y P Yung) Definitive Interfaces as a Visualisation Mechanism
Proc Graphics Interface '90, Canadian Information Processing Society, 1990, 285-292
[An example extracted from this paper is included in the Appendix]

13. (with S B Russ, Y P Yung) Programming as Modelling: New Concepts and Techniques
Proc ISLIP'90, Computing & Info Science Dept, Queen's University, Kingston, Canada 1990

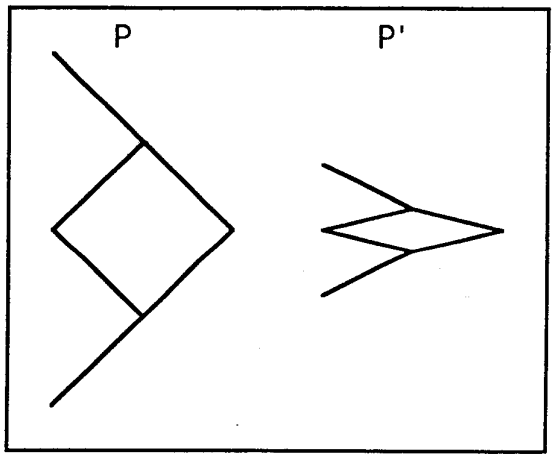
14. (with A J Cartwright, S B Russ, Y P Yung)
Programming Paradigms and the Semantics of Geometric Symbols
Proc W/S "Visual Interfaces to Geometry" in conjunction with CHI'90, Seattle, April 1990
15. M D Slade, Parallel Definitive Programming, MSc Thesis, Univ of Warwick 1990
16. Y W Yung, EDEN: An Engine for Definitive Notations, MSc Thesis, Univ of Warwick 1990
17. Y P Yung, Programming as Modelling, MSc Thesis (in preparation)

**Additional References**

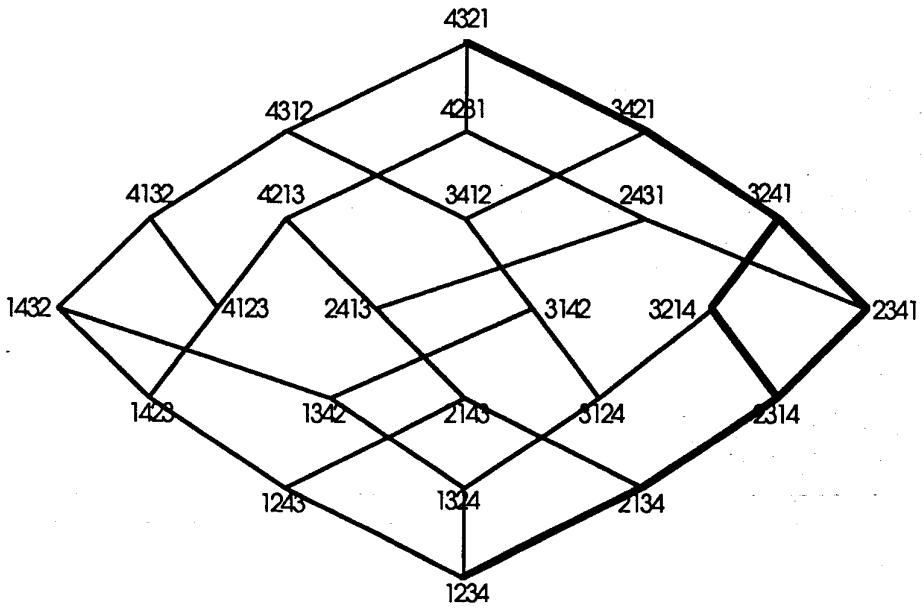
18. M D Atkinson, W M Beynon (in preparation)
19. R Bird, P Wadler, Introduction to Functional Programming, Prentice-Hall, 1989
20. G M Birtwistle, O J Dahl, B Myhrhaug, K Nygaard, Simula Begin,
Studentlitteratur, Lund, Sweden
21. G M Birtwistle, personal communication
22. A Bjorner, Orderings of Coxeter Groups, Combinatorics & Algebra, Boulder, June 1983, AMS
23. W Kent, Data and Reality, North-Holland 1978
24. B Wyvill, An Interactive Graphics Language, Ph Thesis, Univ of Bradford, 1975




2 minimal triangular regions

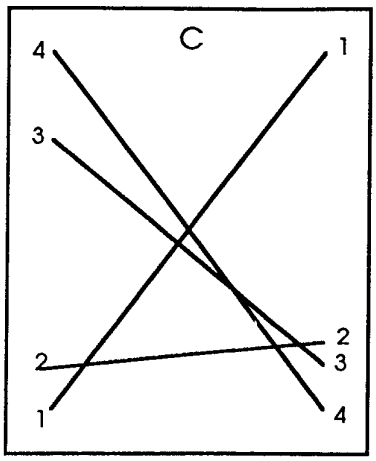



6 covering edges

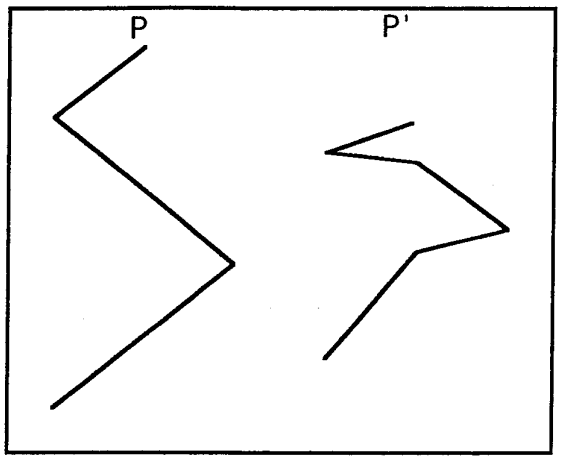



Equivalence class of shortest paths = <121321,123121>

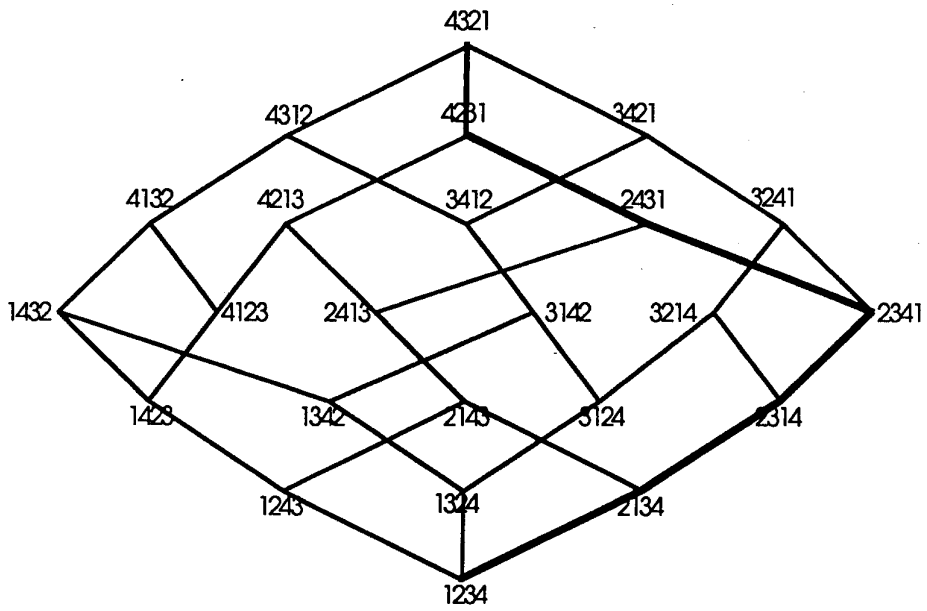
FIGURE 1




2 minimal triangular regions




5 covering edges




Equivalence class of shortest paths = <123212>

FIGURE 2

Appendix

Figures 1 and 2 are extracted from reference [12]: they illustrate the use of SCOUT for user-interface prototyping. Figure 3 is extracted from reference [6]: it illustrates the use of DoNaLD for the specification of a room layout. Both illustrative examples have been demonstrated using software prototypes implemented by Y P Yung and Y W Yung.

```
# Windows Showing DoNaLD Pictures
point p2 = {275, 100};                          # top left corner of the window
point q2 = {475, 300};                          # bottom right corner
point zoomPos = {500, 500};                     # zooming position in DoNaLD coordinates
point zoomSize = 500;                           # size of the picture in DoNaLD coordinates
window don2 = {
        type:    DONALD,                        # it is a DoNaLD picture
        box:     [p2, q2],                      # location of the picture
        pict:    "view",                        # name of the DoNaLD picture
        xmin:    zoomPos.1 - zoomSize/2,        # defining the portion of the DoNaLD
        ymin:    zoomPos.2 - zoomSize/2,        #    picture to be displayed in the box
        xmax:    zoomPos.1 + zoomSize/2,        #       defined above; zoomPos.1 returns the
        ymax:    zoomPos.2 + zoomSize/2,        #          first coordinate of zoomPos
        border:  ON                             # there is a border around the box
};
...


# Error Message Windows
integer DoorHitsTable = DONALD boolean DoorHitsTable;          # a bridging defintion
window monDoor = {
        frame:   ([monDoorPos, 1, strlen(monDoorStr)]),
                 # A string will be put into a frame containing only one box that has top left corner
                 # monDoorPos and is 1 row by strlen(monDoorStr) columns
        string:  monDoorStr
};      # by default, windows are of type TEXT
string monDoorStr = if DoorHitsTable then "Table obstructs door" else "" endif;
point monDoorPos = {25, 50};
...


# Menu Buttons
point tblMenuRef = {100, 400};                                  # the centre of the table-menu
point tblUpPos = tblMenuRef – {(strlen(tblUpMenu)/2).c, 2.r};
        # .c and .r serve as units, they can be read as 'columns' and 'rows' respectively
window tblUp = {
        frame:   ([tblUpPos, 1, strlen(tblUpMenu)]),
        string:  tblUpMenu,
        border:  ON
};
string tblUpMenu = "UP";

integer DoorIsOpen = DONALD boolean door/open;          # a bridging definition
point miscMenuRef = {250, 400};
point doorButtonPos = miscMenuRef + {strlen(plugMenu).c /2, 1.r};
window doorButton = {
        frame:   ([doorButtonPos, 1, strlen(doorMenu)]),
        string:  doorMenu,
        border:  ON
};
string doorMenu = if DoorIsOpen then "Close Door" else "Open Door" endif;
...


# Forming Display
display basicScreen = <
        tblHeader / tblUp / tblDown / tblLeft / tblRight / zoomHeader / zoomUp / zoomDown /
        zoomLeft / zoomRight / plugButton / doorButton / don1 / don2
        # if windows overlap,  tblHeader overlays tblUp etc.
>;
display scr = if DoorHitsTable then insert(basicScreen, 1, monDoor) else basicScreen endif;
display screen = if CableIsShort then insert(scr, 1, monCable) else scr endif;
        # 'screen' is a special variable reflecting the actual screen
```

**Figure 1: Extracts from the SCOUT specification for Figure 2a**

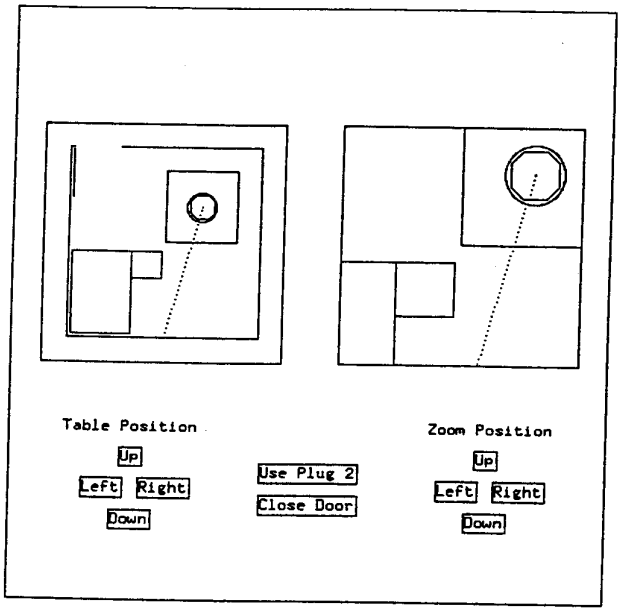



Figure 2a: The display associated with Figure 1

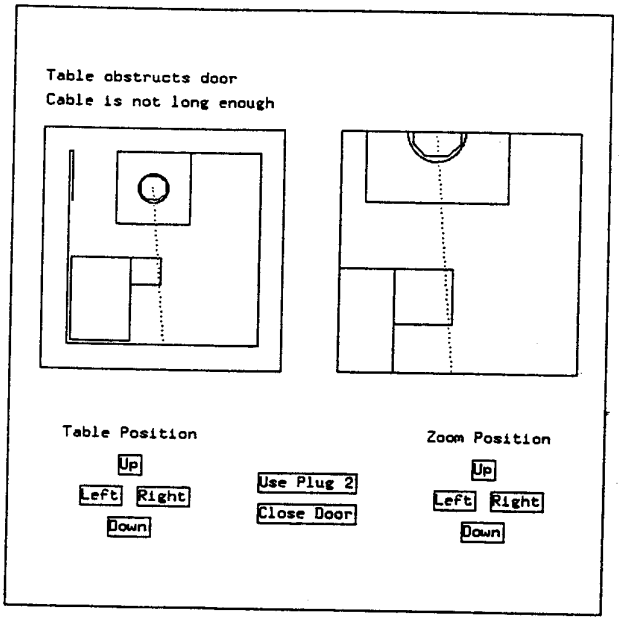



Figure 2b: The display after the table position has been redefined

**Figure 2: Two sample screen displays**

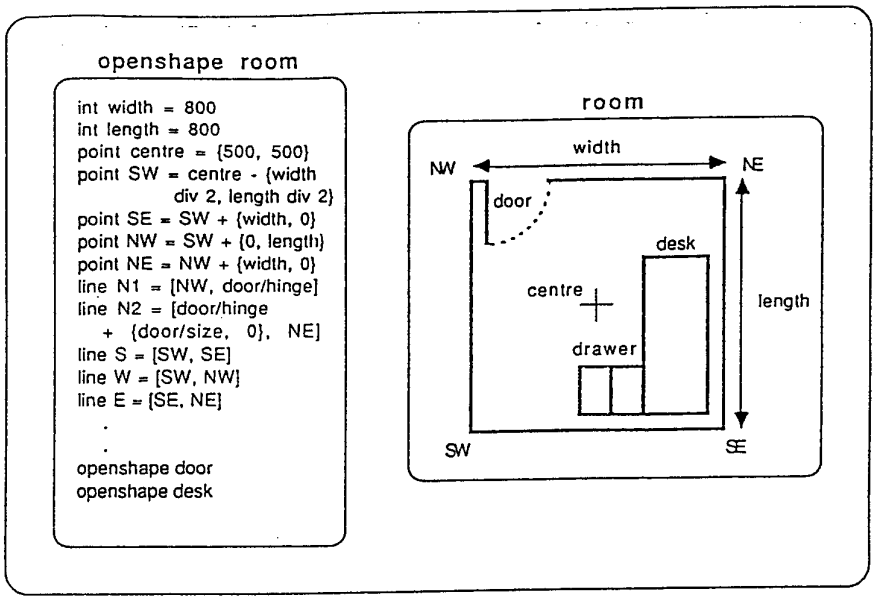



Figure 2: A context window and the corresponding display.

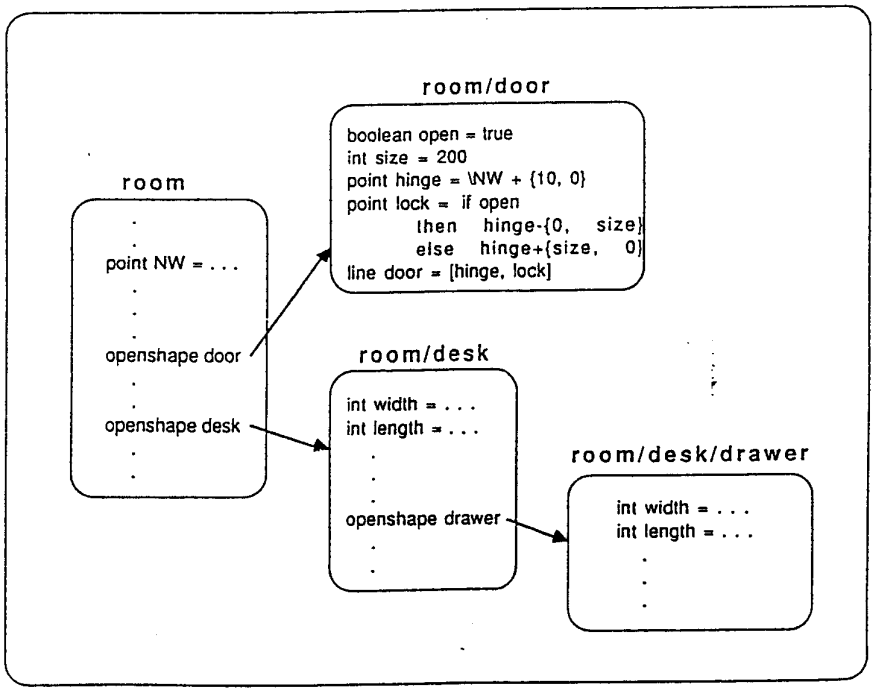



Figure 3: Context windows associated with Figure 1