A Computing Environment for CAAD Education

Christopher Tweed

EdCAAD, Department of Architecture

University of Edinburgh 20 Chambers Street, Edinburgh EH1 1JZ, Scotland

This paper describes a modelling system, MOLE (Modelling Objects with Logic Expressions), and its use as a computing environment for teaching architectural undergraduates. The paper also sketches the background to MOLE's development as a medium for research, and identifies benefits conferred of research and teaching through their common interest in MOLE.

Teaching at EdCAAD is conducted in what is chiefly a research milieu. Hence our teaching methods exploit the products and experience of research. But the partnership is mutually rewarding, because teaching informs future research efforts through the experience gained from using MOLE. At present, our teaching concentrates on a ten-week elective course for fourth year architectural undergraduates. Tbe main component of the course requires each student to program a simple application related to architectural design. Applications normally require a programming language with access to graphics routines, and in previous years we have used C or, more recently, Prolog with their graphics extensions. For the past two years MOLE has fulfilled this need.

The paper begins by explaining the evolution of our approach to CAAD, leading to the development of the description system, MOLE. Section two outlines the main features of the version of MOLE which has been extended to provide a comprehensive computing environment for programming simple architectural applications. MOLE in use is the subject of section three which is illustrated with examples drawn from students' coursework projects and exercises. This is followed by a discussion of the lessons learned from teaching which highlight areas of MOLE's development that need more study. A concluding section summarises what has been learned, and poses vital questions that require answers before we can expect widespread acceptance of CAAD in practice.

An Approach to Computer-Aided Architectural Design

As computer technology becomes increasingly evident in today's work places, so there is a need for greater understanding of its implications for traditional - and often cherished - work practices. For unlike natural growth, technological growth acknowledges few limits; this is its attraction - and its threat. To apply technology beyond its remit will serve only to confirm what some people already fear. In Schumacher's terms, we need 'appropriate technology' together with a clear, informed vision of 'ends' rather than a multiplicity of 'means' (Schumacher, 1978). It is these needs which fuel our interest in CAAD.

Our approach to CAAD is guided by:

- an understanding of problems in man-machine interaction distilled from observations and experience of existing CAAD systems;

- promise of declarative access to computer resources from techniques emerging from Artificial Intelligence (AI) and natural language processing

- a view of design as something people do which is manifested in the *description* of design states and solutions.

Existing CAAD systems encapsulate fixed notions of design descriptions and tasks, and thus impose models of design - decided by their creators - on the user. Users' models of design must map onto prescribed data structures which define the decomposition of designs as assemblies of discrete components. Formulations of design tasks are similarly unalterable, and may also incorporate assumptions and methods of calculation which, since they are often hidden from the user, are unquestionable and implicitly infallible. Because data structures and functions are inaccessible to the user, changes require programmer intervention and are, therefore, expensive. In short, the greatest criticism of these systems is their inability to accommodate varied and transient perceptions of design. But we should not criticise these earlier systems too harshly, as they are products of a different era of computing and merely reflect the technology of that dnie. Recent advances - notably Prolog and logic programming (Kowalski, 1979) - in the fields of AI and natural language processing offer an alternative to the prescriptive paradigm, such that users tell machines *what* they want rather than *how* to get it (Feigenbaum, 1979). These advances make reflective, responsive systems a realistic goal for the future.

A popular view of design is as a problem-solving activity, but for many this view presumes that the knowledge brought to bear on all design problems can be made overt and represented in some formal system (Bijl, 1985) . A more plausible view is Pye's, which states that design embraces problem-solving and overt knowledge within a much wider field of intuition (Pye, 1978). His argument is that overt problem constraints can never determine the aesthetic properties of solutions, and extends to fields normally treated as straightforward engineering design tasks, and therefore amenable to problem-solving techniques. Perhaps a good architectural example is thermal design, which calls on numerical methods of evaluating a building's thermal and energy performance to inform energy conscious and environmental design. How can we relate the results of these evaluations to the subtle properties of architecture, such as 'sacredness' so cloquendy described by Lisa Heschong (Heschong, 1980)?

However, to move to the other extreme. and assert that design is a 'black box' which is indivisible and inexplicable, is equally untenable and, as Heath points out (Heath, 1984), breeds arrogance and an easy escape route for designers when confronted with design failures. Our view of design is as an activity that calls on intuitive and formal knowledge to greater or lesser degrees depending on the domain, specific task, and not least the designer. Note that this view applies equally to many other fields of human activity, for example speech. What does this view of design tell us about the needs of designers and CAAD systems? To answer this we must first examine how designers express their knowledge of design at present.

Description, as a means of externalising design states and solutions, is central to all

design activity. In architectural design, two modes of description are used: drawings, and text (including numbers and formulae). The two modes complement one another as in, for example, annotated drawings and diagrams. And we observe a near equivalence in their authority, though in practice the written word generally carries more weight legally. A description system must therefore cater for drawings and text, and recognise their interdependence.

The above description of shortcomings of existing CAAD systems, new descriptive techniques, and view of design which embraces both intuitive and formal knowledge suggests criteria that a system capable of accommodating individual designer's models of design must meet. Bijl (Bijl, 1985) has identified these as:

- wholeness: the system should not see a whole as different to a part, nor a separate status for a root part, and any part may become a component of any other part;

- discreteness: there should be no notion of boundaries to parts or to assemblies of parts such that other parts are excluded, and changes to a description of a part should propagate to any other part that it describes;

- prior typing: the system should not depend on any prior definitions of types that have to correspond to types perceived in the user's world, and any part may inherit partial descriptions of any other part;

- correctness: the system should not imply any notion of an independent or absolute arbiter of correctness, it should reflect only what is has been told by users, plus what it can infer from what it has been told.

Our theory of descriptions underpins MOLE's development which serves as a vehicle for exploring these ideas. The main features of the implementation developed specifically for teaching are described in the next section.

MOLE

MOLE is aimed at enabling designers to describe their perceptions of design in the manner they choose, and using any combination of text and graphics. The system rests on a very simple notion of descriptions which can be represented as relations between entities known as kinds, slots, and fillets.

A kind (from "any kind of thing") is the starting point for all descriptions in MOLE. All examples in this paper show kind-names in capital letters. Attached to kinds are slots which are normally used to indicate some property or part of the thing being described. A kind may have any number of slots, and again the user decides what names to give slots. Names of kinds and slots have no meaning beyond those which the user ascribes to them. Slots may have fillers which may be kinds, numbers, character strings, or lists of any of these.

Dialogue between MOLE and a user is in the form of expressions that convey the user's intentions to the system, and to which the system responds. Every expression is interpreted as a query *which* the system evaluates to produce a response. Responses vary ac-

cording to the type of expression used, but in general they reflect what the user has told the system. For example if a user types a kind name the system responds with the description of that kind - it displays its immediate slots and fillers. An expression that changes the description of a kind produces a response that is the name of the updated kind. Expressions usually consist of descriptions and predefined operators. Infix binary operators establish relationships between descriptions on either side of the operator. Thus the expression

$$K_1 <+ [\ s_1=F_1, \ldots s_n=F_n\ ]$$

updates the description of kind K_1 with the description on the left of the '<+' operator consisting of pairs of slots and fillers.

When kinds are used as fillers to slots in other kinds an implied structure is created, even though each kind is accessible directly - the database contains only a collection of facts about kinds, slots and fillers, not structures.

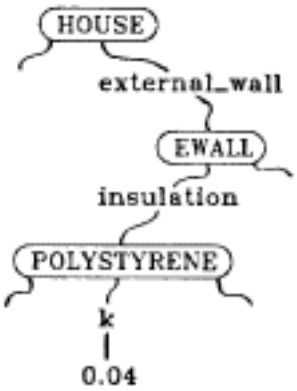

Figure 1: explicit structure of HOUSE

Figure 1 shows the explicit structure of HOUSE constructed from the following kind-slotfiller relationships:

HOUSE external_wall EWALL

EWALL insulation POLYSTYRENE

POLYSTYRENE k 0.04

This is an example of a structured view. The user may ask to see the *virtual* structure of a kind which is similar to the *explicit* view, but omits intervening kinds. All structured views are assembled only as requested and are not stored in the database.

Implicit structure also allows us to refer to parts of a description beyond its immediate slots and fillers by using part-names. Thus in figure 1 the part-name

HOUSE: external_wall:insulation

refers to the filler of the insulation slot of EWALL, which is filled by POLYSTYRENE.

Part-names can be used in expressions in place of kind-names, but the results of evaluation are different. A part-name alone evaluates to the name of the filler it points to. But preceded by the 'value' operator, it produces the description of the filler it points to.

But what happens when a part-name is used to after a description? If that part, and only that part, is to change, the system must have some mechanism for limiting change to the context of the part-name. What the system does is to create instances along the path defined by the part-name, and can best be explained by example.

Consider this example: the descriptions of all houses (HOUSE _1, HOUSE _2, and so on) have a slot called back_door which is filled by DOOR which is a kind. The description of DOOR can be changed by updating it directly via the statement:

DOOR <+ [material = PLYWOOD]

Here, all descriptions which contain DOOR will see the change. But if the change is to be limited to the back_door slot of HOUSE _1, then an instance of DOOR, with the change, must replace DOOR in HOUSE_1's back-door slot.

Figure 2 shows the new structure of HOUSE after the expression

HOUSE:extemal_wall:insulation <+ [material = WOOD-WOOL]

has been evaluated. Instances are created for each kind along the path traced by the partname to ensure that other descriptions remain unaltered. Arrows denote the direction of inheritance.

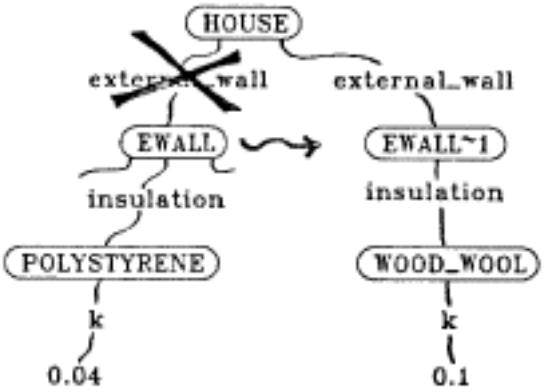

Figure 2: instances in the explicit structure of HOUSE

New instances will not be created if the update expression is prefixed by the 'all' operator, and the result of the update is the same as though the kind had been altered directly.

Figure 2 also shows that instances 'inherit' unaltered parts from their original descriptions. Additions and changes to these continue to be seen in instances unless they are explicitly altered.

MOLE offers another inheritance mechanism which operates through variants. Variants are kinds that inherit all or part of their descriptions from other kinds, which are referred to as super-kinds. Inherited parts have compound names consisting of a slot-name which is 'tagged' with the name of the kind the part is inherited from. Thus a variant may already have its own, or inherit from other kinds, slots of the same name, without conflict.

A variant continues to inherit parts till the inheritance chain is explicitly broken by the user, or inheritance of specific slots is blocked. When a relationship between a variant and one of its super-kinds is dissolved, the variant keeps parts inherited from the super-kind, but does not inherit any future changes to them or die super-kind.

Wildcards and Selective Matching

Though not strictly part of the theory, wildcards and selective matching provide convenient ways of collecting, and operating on, several descriptions at the same time. For example, the name WALL* matches all kinds whose names begin with 'WALL', and '*' matches all kinds in the database. Wildcards can also be used in part-names. The operator, 'match', filters wildcard expressions, and can be more selective. For example, the expression

DOOR* match [height=$v, $v<l800]

recalls only those kinds that have a height slot containing a filler which is numerically less than 1800. These expressions can appear as part of a larger expression, for example to selectively examine or modify descriptions with shared properties.

Graphics

Drawings are an essential part of design descriptions, and therefore must be accommodated in any description system. Descriptions of drawings in MOLE use the same elements as non-graphical descriptions - kinds, slots, and fillers. A graphical interface to the database lets users create representations for graphical objects by drawing them on a device. The resulting description in MOLE may be altered by further drawing operations or by textual expressions.

The graphical interface can be thought of as another user who translates the real user's drawing operations into MOLE expressions that build descriptions of drawings in the database, and who can later reproduce the drawings from their MOLE representations. Unfortunately, for the graphics system to be able to recognise and extract information about drawings from their descriptions, it must follow conventions when creating kinds, slots, and fillers to describe drawings and their line segments. Since the graphical interface shares the database, descriptions of drawings are available to the user. Thus a user can examine and modify descriptions of drawings using MOLE expressions.

The representation of drawings is based on the work of Steel and Szalapaj (Steel., 1983) but departs from their representation in that end-point coordinates map directly onto screen space and are therefore dependent on a real-world coordinate system. Szalapaj uses the notion of construction lines - analogous to pencil lines - to define intersections or construction points as stopping points for line segments - analogous to ink- lines in other media

subjected to multiple transformations, and then recalled to create new instances.

Functions

The final component of the system discussed here is the functional interpreter. This was added to enable users to write programs that operate on descriptions held in the database.

At present functions are treated quite separately from other descriptions and must be stored outside the database. This is far from ideal but is intended as an interim measure while work continues on extending our theory of description to include functions. Ultimately we hope to be able to describe functions as we now describe objects. However, the application of functions to descriptions is still feasible even with this temporary interpreter, and has provided useful information for the current research project. The functional language used here is a hybrid of Prolog and LISP. Every function returns a single result, and may receive any number of arguments. As in LISP the arguments may be calls to other functions, and the evaluator permits recursive calls. Variables are local to functions and serve as temporary storage during a function's evaluation. Functions communicate with the database through the built-in 'eval' function which accepts MOLE expressions and returns the system's response as a result.

The library of built-in functions is large, and many are provided simply for the user's convenience and are not strictly necessary. Built-in graphical functions give the user control over graphical objects and some aspects of their display. Other built-in functions perform computations, input and output operations, and manipulate lists, strings and tokens.

The description of MOLE given here is brief owing to lack of space, but is supported by other papers: a theory of descriptions is presented in "Representing Design Knowledge" (Krishnamurti, forthcoming); and features of this implementation are documented in the MOLE User's Manual (Tweed, 1986).

MOLE as a Teaching Environment

To date, MOLE has sustained a variety of applications, mainly in its teaching capacity. These include: a kitchen space planner; a simple shape grammar interpreter; and a system which completes partially defined room shapes. Despite these rather grandiose project titles it should be remembered that these projects were carried out over ten weeks. and count for only a part of the total coursework. Invariably, therefore, students focused on single aspects of their projects in detail. However, we are less interested in producing sophisticated programs than in exploring the field of man-machine interaction, its limitations and benefits. An outline of each of these selected projects illustrates the potential of the system.

Project A - Spatial Planning

This project was a simple kitchen planner which allows a user to draw a schematic plan of a kitchen, select furniture and appliances from a catalogue, and place these within the floor plan. The user may then test an arrangement for compliance with known criteria for preferred relationships between specific objects, doors, and windows. The catalogue of furniture and appliances could be extended, or modified by the user.

Preferred relationships between plan objects were expressed through slots attached to objects' descriptions. If, for example, the object SINK was preferred to be close to the object COOKER, then this could be expressed as by attaching a slot, close_to, with a filler, COOKER, to the description of SINK. This is an illustration of a different use of slots from that discussed above: here, the slot does not denote a part, but a preferred relationship between two objects. In other words, slots mean what we want them to, and are not confined to any one interpretation.

The finished project tackled only a few of these relationships, but these were enough to demonstrate that it could easily be extended to deal with others, given more time.

Project B - A Shape Grammar Interpreter

This project was based on work done by Flemming at Carnegie-Mellon using a fixed grammar (at present) based on existing house types found in the Shadyside district of Pittsburgh (Flemming, 1995). The objective of the Shadyside project is to generate a pattern book for new housing in this district, which will be sympathetic to existing house types. This student's project replicated some of Flemming's work in MOLE, but lack of time reduced the size of the student's grammar to seven rules of Flemming's original eleven.

The user draws an initial shape which is a plan of the hall. The hall must be rectangular, but its proportions and size are not constrained other than by the physical size of the graphics coordinate space. After the hall is completed, the program guides the user through the selection of rules which generate a complete arrangement of spaces. At each stage the user is presented with options that allow him or her to select the next rule, undo the previous rule, consult a manual page, or exit from the program.

Project C - Spatial Inferencing System

The goal of this project was to develop a system that could recognise and extract bounded spaces from plan drawings consisting of lines. Lines representing party walls could be added as the user created a plan. The interactive nature of the program meant that the student's program need only cope with two spaces at any one time. The student began by considering additive design, such that additional spaces were created by connecting a minimum number of new lines to the outside of an initial space. However, he found that the system could also cope with internal divisions of the initial shape. 'me project was an excellent illustration of the assumptions we, as people, make when reading drawings.

minimum number of new lines to the outside of an initial space. However, he found that the system could also cope with internal divisions of the initial shape. The project was an excellent illustration of the assumptions we, as people, make when reading drawings.

Discussion

From our experience of using MOLE in teaching we can identify two main groups of problems. The first relates specifically to MOLE and includes technical problems of implementation and unresolved matters of theory such as the integration of functional systems with other descriptions. The second group, however, describes problems that are relevant to CAAD as a whole and that are, therefore, more important and less easily solved. Before considering these let us look at the first group.

Students complained that MOLE was too slow. This is a valid criticism, though not one that should trouble us too much. MOLE is slow because it is written in Prolog, though we have succeeded in speeding it up by delegating routine tasks to functions written in C which are called via a dynamic loading package developed for C-Prolog . But this is seen as a temporary solution since new implementations of Prolog are beginning to appear on the market, though their current price tags put them beyond the reach of most academic institutions.

The problem of speed has a bearing on other areas by dictating what is feasible in terms of demands placed on machine resources. The representation of drawings is a casualty of such a pragmatic decision. The decision to use numerical coordinates instead of the more elegant 'conline' representation was a compromise between elegance and practicality. Clearly advances in implementing Prolog will allow us to reverse those decisions.

To some extent we can also discuss criticisms of functions because this is an area which is currently being investigated, and the current implementation of functions is seen as temporary. Comments from students, however, guide our research into how we might ultimately represent functions.

We cannot do justice to the wider issues and problems identified, merely recognise and note them. In reducing the degree of prescription, the task of programming the system effectively moves from the programmer to the user. Ultimately, this may weaken the appeal of CAAD to designers. Designers may now be able to represent their own models of design, but whether they are prepared to invest the time and effort needed to explicate these models is debatable. A compromise seems likely. Krishnamurti offers one possible solution (Krishnamurti, 1986), where the system is supplied with an initial default database containing frames or scripts that the user alters according to need. However, this still demands that the designer examine these defaults meticulously, and then compare them with his or her own mental models. Pressures on designers being as they are, the probable outcome is that designers will let the defaults shape their models of design till a threshold of tolerance is reached.

A further complication arises from the focus on individual's perceptions of design. Students work alone on specific tasks and exercise the freedom from convention that

MOLE encourages, but in practice design is rarely the responsibility of a single designer.

Conclusions

A logic modelling environment which is the central component of a teaching course at EdCAAD has been described, and illustrated with examples of coursework. From practical experience of using MOLE, we have identified areas which require more detailed study, and questions which need answers before CAAD is accepted by designers. From students' written coursework we see that they are critically aware of these issues, and are well equipped to meet the challenges and promises of new technology in architectural practice.

Acknowledgments

I am greatly indebted to Aart Bijl for many of the ideas expounded here, and to Ramesh Krishnamurti for his support and enthusiasm. The development of the logical modelling system, MOLE, has been supported by the UK Science and Engineering Research Council.

Rererences

Schumacher, 1978. E F Schumacher, *A Guide for the Perplexed,* ABACUS edition, Sphere Books Ltd, London (1978).

Kowalski, 1979. R Kowalski, *Logic for Problem Solving,* Elsevier North-Holland (1979).

Feigenbaum, 1979. E A Feigenbaum, "Themes and Case Studies of Knowledge Engineering," *in Expert Systerns in the Micro Electronic Age (ed. D. Michie),* Edinburgh University Press, (1979).

Bijl, 1985. A Bijl, "An Approach to Design Theory," *proc. IFIP WG 5.2 Working Conference on Design Theory for CAD,* (Oct 1985).

Pye,1978. D Pye, *The Nature and Aesthetics of Design: A Design Handbook,* Herbert Press Ltd, London (1978).

Heschong, 1980. L Heschong, *Thermal Delight in Architecture,* MIT Press, Cambridge, Massachusetts (October 1980).

Heath, 1984. T Heath, *Method in Architecture,* John Wiley and Sons, Chichester (1984).

Steel, 1983. S W D Steel and P J Szalapaj, Pictures Without Numbers: Graphical Realisation of Logical Models," *proc. PARC'83,* (1983).

Krishnamurti, 1986. R Krishnamurti, "Representing Design Knowledge," *Planning and Design* 13(forthcoming 1986).

Tweed, 1986. A C Tweed, "MOLE User's Manual," *EdCAAD Working Paper,* (Aug. 1985, revised Mar. 1986).

Flemming, 1985. U Flemming, A *Pattern Book for Shadyside,* draft version (Nov. 1985).

Tweed, 1985. A C Tweed, "Dynamic Loading in C-Prolog," *EdCAAD Technical Report,* (1985).

Krishnamurti, 1986. R Krishnamurti "The MOLE Picture Book," *Design Computing,* (in press 1986).