Prescribing designs
Christopher Tweed

Much of the debate and argument among CAAD researchers has turned on the degree to which CAAD systems limit the ways in which designers can express themselves. By defining representations for design objects and design functions, systems determine what it is possible to describe. Aart Bijl used the term ‘prescriptiveness’ to refer to this property of systems, and the need to overcome it was a major preoccupation of research at EdCAAD during the 1980s, including the development of the MOLE (Modelling Objects with Logic Expressions) system. But in trying to offer designers the freedom that was judged to be essential to evolving design practices, MOLE transferred much of the burden of programming from system developers to end-users — you can have any design objects you want, as long as you write the code. Close examination of MOLE’s logic reveals that it too had to rely on fundamental definitions that, even if not domain-specific, are certainly historically contingent.

This paper will return to the issue of prescriptiveness, summarising the lessons learned from the MOLE ‘experiment,’ and identifying new prescriptions that are deciding what designs can be. Looking beyond computer representations, we find that designs are shaped by much larger, and arguably more powerful, historical, social and cultural forces surrounding design practice. These forces are shaping the way CAAD is used and how new systems are conceived and developed.

Keywords: Bijl, prescriptiveness.

Introduction

For all the volatility attributed to computer systems, they are remarkably rigid. Once a definition—of an object, a type, a procedure—is enshrined in a computer program it assumes a permanence that belies its existence as a magnetic field on some medium. To alter a definition requires access to the source codes, specialised skills and, usually, significant time, effort and money. For all practical purposes, most users must accept the definitions which have been captured in the program’s code, as prescribed by the system’s developers.

Aart Bijl coined the term ‘prescriptiveness’ to describe how CAAD systems determine the way designs can be represented and manipulated (Bijl, 1987). The argument against prescriptiveness is simple: CAAD systems accommodate a finite number of possible design descriptions which, no matter how large, will sooner or later prevent a designer from representing a design in the way he or she wishes. The difficulties of maintaining the integrated CAAD system developed for the Scottish Special Housing Association (SSHA) persuaded EdCAAD (Edinburgh Computer Aided Architectural Design) to embark on a programme of research centred on the MOLE...
(Modelling Objects with Logic Expressions) system which aimed to overcome the problems of prescriptiveness which were so acutely felt in the development of the SSHA system.

This paper describes the approach adopted in the MOLE ‘experiment,’ summarises its main achievements and shortcomings, outlines some successors to MOLE and concludes with a general discussion of prescriptiveness, particularly in relation to new developments in CAAD and the construction industry.

MOLE

The work on MOLE grew out of previous experiences in developing a CAAD system for the Scottish Special Housing Association (SSHA) and observations of similar CAAD systems developed by others, such as OXSYS (Hoskins, 1977), which were summarised in a report commissioned by the Department of the Environment (Bijl et al, 1979). The problem that emerged from the development of these systems was that they encapsulated data structures and processing functions that expected designs to be presented and operated on in predefined ways.

At the same time (early 1980s) interest was growing in the declarative programming language, Prolog, which appeared to overcome many of the difficulties associated with conventional procedurally oriented languages, such as FORTRAN. The shift has been characterised as moving from telling computers how to do something as a series of machine-dependent instructions, to saying what you want. It was argued that by using Prolog’s approximation to first order logic, programmers could concentrate on the logic of the task in hand without having to tell the computer how to arrive at the result. One of the main benefits was the reduction of the sheer amount of program code needed to perform a task. Comparisons between Prolog and C, for example, showed that to achieve the same result in C required more than twice as many lines of code (Swinson, 1983). Prolog programs were also said to be much easier to read and understand, an important advantage when several people were working on the same piece of code.

MOLE, then, was seen as a means of avoiding the prescriptiveness of earlier systems, relying instead on the declarative methods promoted by Prolog. Specifically, MOLE set out to avoid problems associated with earlier systems by following four main maxims:

**There should be no prior typing of descriptions**

Many systems had incorporated pre-defined types which users had to conform to. These established the system properties of various building elements (walls, roofs, spaces, etc.) and the type of operations that the system could perform on instances of these elements (join, tilt, calculate area, etc.). There was a recognition that such definitions were often arbitrary and subject to revision, e.g. in building regulations that a wall is any element that makes an angle of 70° or greater with the horizontal. Prior typing is a powerful way in which system developers can determine what can be described to a system. It effectively determines what (say) a wall can be, within a CAAD system.

**There should be no conditions of completeness**

As with typing, setting criteria for the completeness of descriptions restricts what users can do. Conventional approaches require that a data structure be complete before the system can do anything useful with it. The MOLE strategy relaxed this requirement to allow users to expand or shrink descriptions when they wanted rather than when the system dictated.

**There should be no predefined model of correctness on users’ descriptions**

Designing sometimes requires that we are “wrong” before we are “right”, and the value of getting something wrong should not be overlooked. Being wrong at some point often helps us move in a different
direction. Suspension of judgement is an essential ingredient for most design. Systems, therefore, should not seek to judge our descriptions as to do so would block opportunities for creativity.

**Provide only those functions that are independent of any particular domain**

If we observe the practices of designers inevitably we can distill certain ‘universal’ functions they routinely invoke as part of their designing. However, to embed these in a system is almost always premature since observing a larger sample or observing the same designers at a later date will usually suggest revision of the earlier formulation of their activities. MOLE, therefore, sought to exclude any built-in functions which were clearly specific to a particular domain—for example, architecture—tending to rely on functions which were universally valid—for example, logic and mathematics.

**MOLE descriptions**

MOLE adopted a structure for descriptions from the notion of a ‘frame’ in Artificial Intelligence. The difference between MOLE and earlier frame-based systems was in the intent, how resulting descriptions were to be treated. In frame-based systems there is often an assumption that a frame captures all of the essential components of a description, a frame is complete. As we have seen, one of MOLE’s central tenets was that any description could continue to expand, shrink or change. It should never be treated by the systems as being complete.

The basic elements of a MOLE description were *kinds, slots* and *fillers*. A kind was shorthand for “any kind of thing.” The name of a kind could be formed from any sequence of alphanumeric characters, with the usual proviso that it must start with a letter. In MOLE, kind names were always displayed in uppercase. Slots were attached to kinds, and again slot names could be any sequence of alphanumeric characters starting with a letter. Any number of slots could be assigned to a kind. The critical point to be made is that none of these names were significant to MOLE.

Fillers were treated differently. A filler was anything assigned to a slot in the description of kind. Fillers provided the details of a description. Fillers could be references to other kinds, numbers, text strings, or indirections. An indirection was simply a MOLE expression which referred to another description.

A user, therefore, was free to choose any names he or she wished without any interference from the system. Or, to put it another way, the names had no meaning for the system other than serving as names to which further names could be attached. A user could quite merrily create a kind called “TIMBER_HOUSE” and then attach slots called (say) “steel_structure”, “concrete_frame.” MOLE would not object. At the time this was considered to be refreshingly anarchic. Many other research efforts were trying to tie down the meanings of architectural and building terms and build these into their systems, such that the system could prevent inconsistencies creeping in descriptions. MOLE wanted to avoid interference from the system, because we believed that during design it is often necessary to be wrong before getting it right. The example often used to illustrate this approach was if I start to design a housing layout on the assumption that I can fit four units on to the site, the system should prevent me from describing a fifth.

These sorts of issues go right to the core of design methodology. MOLE, therefore, explicitly rejects the prevailing CAAD view that designers proceed methodically in a stepwise fashion, beginning at a high level of abstraction and gradually moving towards increasing levels of specificity and detail—top-down approach—nor did it demand that users define low-level entities before they could be referred in higher-level descriptions. In fact the concept of levels was not recognised by the system which relied on flat networks rather than hierarchies.

**Processing designs**

The work on MOLE tried to remain true to the
underlying tenets of Prolog, in which the use of procedural code was frowned upon. But the need for ‘functions’ was brought about when MOLE was to be used in earnest for actual projects. Intellectually satisfying as it may have been to stick to a purely declarative strategy, users were unable to do anything other than create and manage descriptions.

Functions in MOLE initially followed a LISP-like format and so required users to master a completely different syntax. The programming environment provided function calls that could resolve arguments using descriptions entered as kinds, slots and fillers. Gradually functions became more and more integrated with the original MOLE structure. Fillers could be functions that returned a value. LISP syntax was replaced with MOLE syntax. Any operation that was not part of the original implementation of MOLE was incorporated in an ‘external machine.’ Mathematical functions, for example, were part of a MATH machine which could be referenced using MOLE syntax, as in

\[ \text{MATH: add:}[5, 4] \]

which yields the result 9. These external machines were in fact very similar to ‘objects’ in today’s object-oriented systems. They contained class variables and methods, and used a system of inheritance.

With hindsight, it appears as though the introduction of procedural and functional components signalled the end of the MOLE project. It was no longer possible to claim that MOLE did not attach any meaning to names used for kinds and slots. Typing ‘MATH’ would invoke that particular, and most importantly, pre-defined description. The need to provide sophisticated ways of operating on descriptions tested the declarative approach to breaking point and resulted in a double-bind decision: either to retain the purity of the original conception, and live with the minimal functionality it offered, or to admit conventional forms of programming that would require users to become ‘programmers.’ Despite the abstract, mathematical elegance of MOLE descriptions they became virtually impossible to decipher, even to their original authors. Perhaps it could have been possible to devise a new way of having the best of both worlds, but since Prolog itself is unable to avoid meta-logical statements it seems unlikely that MOLE could have done the same.

### The end of MOLE

Work on MOLE ended in the early 1990s at the conclusion of the EU funded COMBINE project (Augenbroe, 1992). Why it did so can be attributed to several events. Changes in personnel at EdCAAD emptied the research unit of most who had been closely involved in the design and coding of the software. A sea change in funding from the Science and Engineering Research Council (SERC) brought an end to more speculative research with a palpable shift towards projects that should yield firmer results in much less time. The rise of proprietary CAD systems, most notably AutoCAD, in architectural practices signalled a willingness among practitioners to settle for less, in this case to use computers primarily for draughting rather than for integrated design. Ironically, AutoCAD satisfies several of MOLE’s requirements in that it is a generic system that does not provide prior definitions of architectural objects or procedures and offers a separate programming environment in which skilled users can write their own functions.

Experience with student projects based around MOLE had shown that despite the carefully considered logic of the system it remained abstract and unrefined in the features and functions it offered users who were fresh to computing in any form. MOLE, however, did pioneer some features that have now become standard in recent systems. The idea of developing objects through which users could access the common functions of a computer system has only recently become part of mainstream systems and programming environments.
After MOLE

The later stages of MOLE development at EdCAAD overlapped with the start of a new project to research and develop an information system for assisting with the authoring of building regulations (Stone and Tweed, 1991). This was a collaborative venture with the Department of Artificial Intelligence at Edinburgh and the Department of Civil Engineering at Leeds University. Most importantly it included an ‘industrial’ partner in the form of the Building Directorate of the Scottish Office. The Science and Engineering Research Council (SERC) funding came through Innovative Technology Applications (ITA) initiative, in which a heightened sense of realism was sought requiring projects to find matching funds from industrial partners, as a demonstration of commitment from outside the academic community. The change in mood is probably best captured by an incident at one of the early project meetings when the industrial partner asked for the word ‘prototype’ to be removed from one of the objectives, to be replaced with ‘working system.’

This project resulted in a system called Plinth (Platform for Intelligent HyperText). Plinth runs within a POPLOG environment employing an object-oriented design. The system provides a graphical user interface for creating, viewing and editing networks of nodes and links. At the core of the system is a set of Network Configuration Models (NCMs) that define what types of nodes and links can be used in a network of a given type. An NCM also defines what types of link can be used between different types of node.

Plinth exhibits few specific MOLE features, but recognised the problems of hard-wiring system definitions by allowing users to alter definitions of the types of nodes and links that could be used in constructing networks of information.

CrossDoc

The CrossDoc system developed in Belfast was heavily influenced by experiences with MOLE and Plinth. Unlike MOLE, CrossDoc was conceived from the very beginning as not much more than a way of
linking and organising heterogeneous information in networks that were manipulated graphically. Following Plinth, CrossDoc allows users to build, using graphical tools, networks of nodes and links that are typed according to predefined Network Object Definitions (NODs). Thus it is possible to define a NOD which, for example, would contain node and link types to support the construction of networks of argumentation as shown in Figure 1 (Tweed, 1998).

CrossDoc contravenes fundamental maxims of MOLE development. It requires prior typing of nodes and links in the way that MOLE’s kinds, slots and fillers did not. This taboo was broken for entirely (system) pragmatic reasons. It was reasoned that since users could alter the definition of node and link types there was no (logical) difference between prior typing and no typing.

CrossDoc development has never reached the stage of providing user-definable functions or methods. It was created with the modest ambition of carrying over some of the things we can do on paper, such as annotate texts and drawings, to a digital environment.

The issues

The issues surrounding prescriptiveness all hinge on freedom—freedom of thought, freedom of expression and freedom to build—and the extent to which pre-existing structures inhibit or deny freedom. There are several dimensions to the problem.

First, the difference between inhibition and denial. A system prohibits description of designs if it does not provide any mechanisms for their entry or representation; it inhibits their description if the user must resort to complicated, skill-dependent or time-consuming methods. So, while many systems today provide very flexible customisation tools, the obstacles to using these can be immense. This underlines one of the conclusions from the work on MOLE: to avoid prescriptiveness the burden of system development is transferred from the system’s programmers to the end-users.

Second, if a user is able to define his or her own structures and procedures, is he or she prescribing to themselves? Programmers will be aware of this problem. Returning to code written months or years before, they will often be flummoxed by their earlier purposes and methods unless they have been disciplined in documenting how the code works and what it is supposed to do. We may conclude, rather trivially, that it is entirely possible to limit one’s future actions by one’s own prior definitions.

Third, there is a question over whether prescriptiveness encourages designers to modify their design proposals to fit the constraints systems impose and, if so, does this have any significant effect on architecture. Perhaps it runs deeper still. The extent to which language, as a medium of expression, determines what we can think about is a long-running debate in linguistics for which there are not knock-down answers. The analogy with design is obvious. Do the structures of CAAD systems surreptitiously determine what we can conceive in design?

Fourth, since most designs are executed by builders rather than by their designers, instructions must be communicated from designer to builder. If CAAD systems are the sole medium then they can only transmit what they have previously been configured to do so. Of course, we still rely on paper, which can be annotated, as the primary medium for drawings and specifications and while that continues there is always the option of elaborating CAAD resident descriptions.

Finally, there are prescriptions that lie beyond, but influence the shape of, CAAD development. Before we even get to a computer terminal many prescriptions are in operation. Our culture, of which language is a part, is replete with all kinds of prescriptions—norms, beliefs and values—that govern what is considered to be acceptable behaviour. A generally liberal background dictates that we give reason the final authority in most decision-making and most of our economies rest on a recipe of growth (with concomitant advances in technology) accompanied by fiscal prudence which it is claimed will lead to higher
standards of living and better quality of life for all. In the UK, this political philosophy underpins compelling arguments to rationalise and improve the efficiency of construction by adapting methods from manufacturing industries for use on buildings. CAAD is heavily implicated in this strategy and one can be forgiven for thinking that because architectural design has proved to be too difficult to rationalise it has been marginalised in favour of candidly prescriptive systems in which buildings are assembled from collections of components. If that is true, then, rather oddly, we appear to have arrived back at the integrated CAAD systems in which prescriptiveness first became an issue.

References

C. Tweed, Annotated networks of urban project information and meta-information in: U. Schiavoni, ed., Information systems and processes for urban civil engineering applications (European Communities, Luxembourg, 1998), 81-93.