The MOLE Picture Book: On a Logic for Design

Ramesh Krishnamurti
*EdCAAD, University of Edinburgh,
20 Chambers Street,
Edinburgh EH1 1JZ*

Accepted for publication May 2, 1986

This paper examines the interaction between designer and machine. The view that designs can be seen as the outcome of dialogue between a designer and a machine is promoted and is illustrated by a detailed reference to the MOLE system. MOLE provides a logic modelling environment for spatial and non-spatial object descriptions constructed from a hierarchy based on kind-slot-filler relationships and two distinct forms of inheritance relationships. Design descriptions can be updated or queried by part expressions which consist of a kind together with a conjunction of slots. Lastly, the role of MOLE in a design system is considered; a schema for design systems that can be customized to reflect the intentions of individual designers is suggested.

"The possibility of all imagery, of all our pictorial modes of expression, is contained in the logic of depiction"

Ludwig Wittgenstein—*Tractatus Logico-Philosphicus*, 4.0101

COMPUTER MODELLING OF DESIGN

Many approaches to design are advocated and practiced by architects. While these approaches to designing may differ, there is one belief that the various architects and designers appear to share, namely, that their methods encapsulate the "essence" of designing. If we are to build machine-based design systems that support designers' intentions, then these systems must be based on a view of design that embodies this essence of designing. Furthermore, the view of design incorporated into these systems should not be biased towards any particular approach to designing.

This paper explores the connection between designers' perceptions of their design tasks, and the computer modelling of design activities, and suggests a paradigm for design wherein design solutions are arrived at through conversations between a designer and a design medium.

From an information processing standpoint, the behavior of a designer can be described as in Figure 1 (annotated with respect to design).[1] In the case of a traditional architect, the representation is held in the drawings that describe a design, and the reasoning that led to the "creation" of the design is kept in the thoughts of the architect. To realize a machine-based environment for design, we need to be able to transfer the intelligence that lies beneath the reasoning processes into some sort of *design support procedure* that the machine can execute. The machine must not only carry a set of design descriptions (perhaps as drawings) but it must be capable of "understanding" what each component in the design means.

The conventional CAAD approach to modelling design has been influenced by the principles of *problem solving*.[2] In problem solving, design tasks are hierarchically structured into subtasks that are, ultimately, simple to resolve. In terms of Figure 1, the knowledge base is expressed as a collection of prescribed rules for design. Indeed, Simon[3] presents a powerful case to support problem solving as a vehicle to tackle ill-structured problems as typified by design. Akin[4] illustrates how notions of *problem restructuring*, *generate-and-test*, and *heuristic search* can be applied to design problems.

Alternative formulations correspond to *generative* approaches, rooted in language theory. Stiny and March[5] describe a "design machine" based on algorithmic descriptions of designs as illustrated by shape grammars.[6,7] The shape grammar formalism has been adopted by Flemming[8,9] in his generative expert system for spatial layouts based on loosely packed rectangles, and applied by Coyne and Gero[10,11] in their design grammar and planning based method.

However, one can question whether the problem solving paradigm—in its widest sense—is sufficient to specify an "open" design environment. Attached to this question is the expectation a CAAD system must fulfill, namely, that it should mirror the process(es) that designers perceive or recognize as designing. Bijl[12] presents a holistic view of design that bears some connection with linguistic theory, and he suggests that

CCC 0884-8165 86 030171-18$04.00

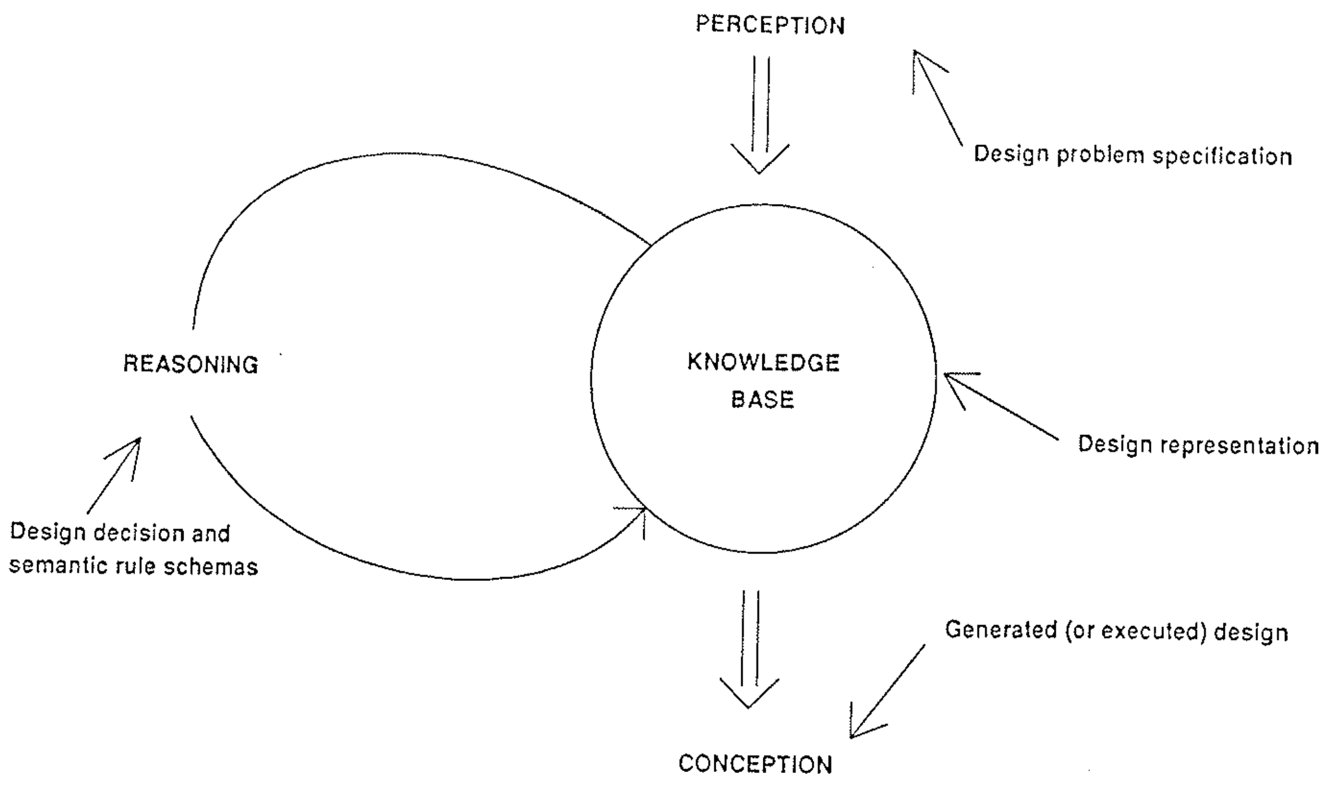

PERCEPTION
Design problem specification
REASONING
KNOWLEDGE BASE
Design representation
Design decision and semantic rule schemas
Generated (or executed) design
CONCEPTION

Figure 1. Information processing model of designer's behavior.

design objects have a syntax and a semantics that have to be bridged in order to get a clear understanding of the design process. The deeper issues raised by the question are far too complex to detail in a single paper (see Heath[13] for a treatise on Method in architectural practice).

One assertion that we can make is that designing is an activity in which the architect's intention is to give form to a loosely defined thing—to make something essentially new. That is, designing is constructive. The architect is confronted with a demand—typically, for a kind of building. Associated with this demand is a "design environment" or "problem space" that defines the constraints that the architect must resolve.

Many factors influence this design environment. Some of these factors are conditioned by the nature of the demands and expectations made on the architect. Some evolve or change during the process of designing, through interactions between the architect, the client and others whose opinions are consulted. Some call to play the intuitive "feel" of all parties involved in a design. For example, the architect does not often carry out, say, lengthy calculations for the sizing of lintels; instead, he or she resorts to intuitive "judgement" about what will or will not work. This feel for design manifests itself in the externalized description of the design. The major effort on the part of the architect is to determine whether something is to be done, what is to be done, or how it is to be done. Part of the design process is to determine what the design problem is. To sum up, the essence of architectural design can be modelled as performing *actions* based on some decisions and in then deciding whether the *decisions* taken are appropriate.

Dialogue Paradigm

No single model can capture the processes involved in architectural design, in the same way that no single characterization of buildings can be specified. However, it is possible to envisage a design system wherein the knowledge base and the reasoning processes are maintained in a fluid state; they go through a transient and, in principle, non-monotonic sequence of closed world states. Whilst each temporal state has an affect on the subsequent state during the course of a design, there is no requirement that the knowledge and reasoning rules that apply at any instance in the design process need be consistent with those at any later instance.

In such a system, design can be seen as a kind of dialogue between an "opponent" and a "proponent".[14] The problem constraints oppose or attempt to oppose what the architect proposes or attempts to propose. This "battle of wits" ends when the architect decides that a design has reached a satisfactory state of existence or when constraints, such as the client's budget cannot be further exceeded, deem it to be so.

The crucial element in this dialogue is the ability of the designer to convey (partial) descriptions to the machine, and that these descriptions can be manipulated or changed according to some criteria that the designer may wish to apply. Moreover, the machine must respond in the language of the designer, namely, using the same descriptional terms that the designer employs. These descriptions must possess a quality of "truth" in that they must reflect the (factual or otherwise) beliefs held by the designer. One way to treat these descriptions is to regard them as statements that belong to some logical framework. This is the approach adopted in the MOLE system.

THE BASICS OF MOLE

The name, MOLE, is an acronym for "modelling objects with logic expressions." As the name implies, statements about objects are modelled as expressions that are treated as logically true. Queries are expressions that become true only if the variables in any query can be unified to facts currently residing in the database.

The central concept in MOLE is that of a *description*, namely, a collection of "features" one associates with an object. Some of the features, in turn, may be further described. For example, one might describe a car in terms of its make, model, capacity, color and so on. Descriptions are not unique and reflect choices that one considers important. Clearly, a car designer working with a CAD system will employ an entirely different set of features to those used by, say an architect. Thus, descriptions are reflective with no system-imposed object types.

MOLE employs a representational structure that is akin to *frames*[15] or *semantic nets*.[16] Where MOLE essentially differs from frames or semantic nets is that there is no prescribed interpretation associated with the relational links between objects. MOLE uses three sorts of entities: *kinds*, *slots* and *fillers*. A kind corresponds to an object, the name of which is, by convention, capitalized. A filler can be any object, and represents a value for some feature that a kind may have. There are no restrictions imposed on fillers. For instance, a filler may be a kind. A slot denotes a relationship (attribute) between a kind and a filler, and its name is, again by convention, given in lower case. Kinds, slots and fillers are the building blocks from which descriptions can be created.

In principle, there is nothing novel about MOLE's structuring of object descriptions that cannot be captured within, say a *relational database*.[17] One exception is that there is no explicit notion of kind and filler types. That is, MOLE does not adopt any conventions as to what a particular slot stands for. Slots with the same name occurring in different descriptions may be associated with fillers of differing types. Whether or not this is useful in a design context remains to be seen.

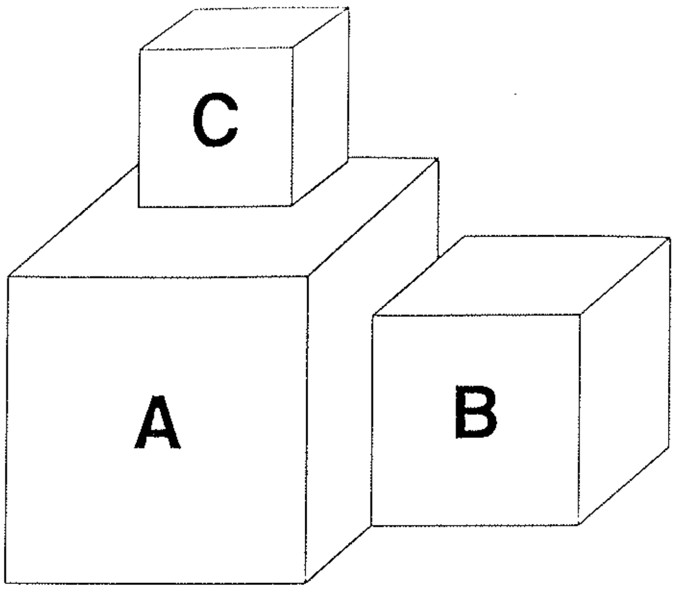


Figure 2. An arrangement of three cubes.

As an illustration, consider Figure 2 showing an arrangement of three cubes *A*, *B*, and *C*. Here cube *B* is beside cube *A* (or vice versa) and cube *C* is on top of cube *A*.

A possible MOLE description for the arrangement of cubes is as follows (indented for easy reading):

```
A:
    is_a   = (CUBE)ˇ
    beside = B
    above  = C
B:
    is_a   = (CUBE)ˇ
    beside = (CUBE)ˇ
C:
    is_a   = (CUBE)ˇ
    below  = A
```

Each filler *(CUBE)*ˇ denotes a distinct *instance* of some kind, *CUBE*, whose MOLE description will typically contain the geometry, perhaps parametrized, of a cube.

The example indicates another feature of MOLE, namely, that of *inheritance* of properties from one object by another. Here, each instance of a cube inherits the attributes of some cube albeit with different fillers. There are two forms of inheritance schemes allowed in MOLE. Instances provide one form of inheritance. A second form of inheritance, referred to as a *variant* of a kind, provides a view of an object as seen with respect to the attributes of another. The distinction between the two forms lies in the roles played by the attributes in a kind. Thus, we could say that *JOE* is a variant of some archetypal *MAN* and some archetypal *MUSICIAN*. It is possible that both man and musician may have slots with the same name, say *"preference,"* where for the former it may refer to manly

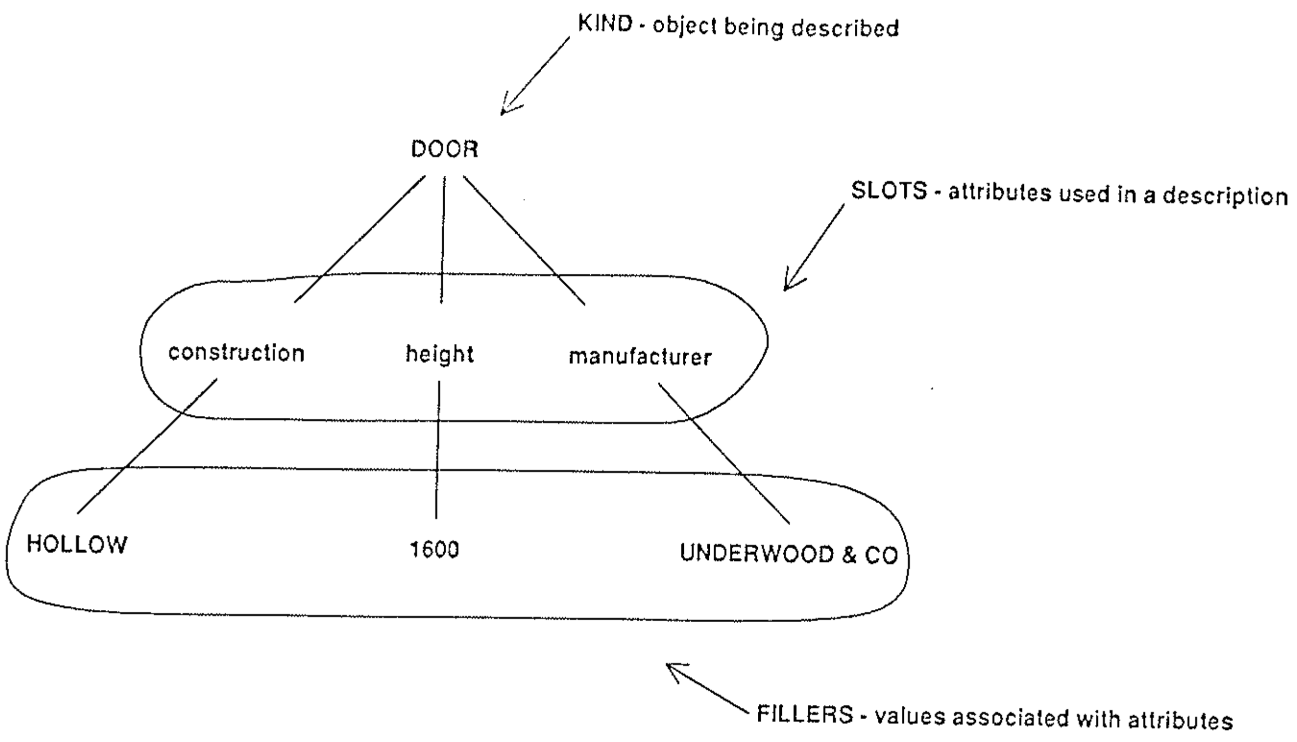

KIND - object being described
DOOR
SLOTS - attributes used in a description
construction
height
manufacturer
HOLLOW
1600
UNDERWOOD & CO
FILLERS - values associated with attributes

Figure 3. Graphical representation of the description of a kind using a labelled directed graph.

preferences and for the latter specifically to musical preferences. In the description of *JOE*, the slots are disambiguated only when the context in which they occur are specified. That is, when the context *MAN* and *MUSICIAN* are specified.

Instances are created when two kinds contain in their descriptions references to the same object, and when the description of one of the kinds is altered without changing the other. We give an example of one such situation. Before doing so we remark that a description of an object can be pictorially drawn as a directed graph wherein the *nodes* denote kinds and fillers and each *edge* represents a slot labelled with its name. Figure 3 presents a graph representing a single level description of the kind *DOOR*. An inherited slot can be unambiguously labelled with the slot name with the kind in whose description it is defined as illustrated in Figure 4. (This is technically referred to as *tagging*.) In Figure 4, the kind *BLUE_DOOR* has, in addition to its own color slot, the slots inherited from the kind *DOOR*. In principle, as it is shown later, it is not necessary to maintain a copy of the inherited slots so long as a reference to the inheritance relationship is kept in the database.

By replacing all kinds in a description by their graphical counterparts, we can build a complex graphical representation of the description of a kind. *Paths* in this graph relate a kind and a filler through a conjunction of slots.

It is convenient to think of MOLE's database as consisting of *parts* expressed by the identity, $K{:}s = F$, where s is a slot that associates filler F to kind K. Part relationships are asserted to the database by statements each of which comprise of compositions of one or more slots relating a kind and a filler. Thus, for instance the statement

$$K{:}p{:}q{:}r = F$$

is deemed to be true provided there are fillers X and Y such that identities $K{:}p = X$, $X{:}q = Y$ and $Y{:}r = F$ are derivable from the part relationships in the database. A query is a statement-like expression consisting of at least one variable which when unified to slots or to a kind or filler becomes a statement. It should be remarked that the evaluation statements (and thus queries) can pose problems since fillers are not necessarily kinds. The formal treatment for evaluating of part statements for certain types of fillers has been presented elsewhere.[18]

Updating a MOLE Database

We now consider an example of changing a MOLE database. The database initially contains the following part identities:

$$\begin{aligned} TOMS\_HOUSE{:}front\_door &= DOOR \\ DOOR{:}construction &= HOLLOW \\ DOOR{:}door\_knob &= DOOR\_KNOB \\ DOOR{:}color &= BLUE \\ DICKS\_HOUSE{:}back\_door &= DOOR \end{aligned}$$

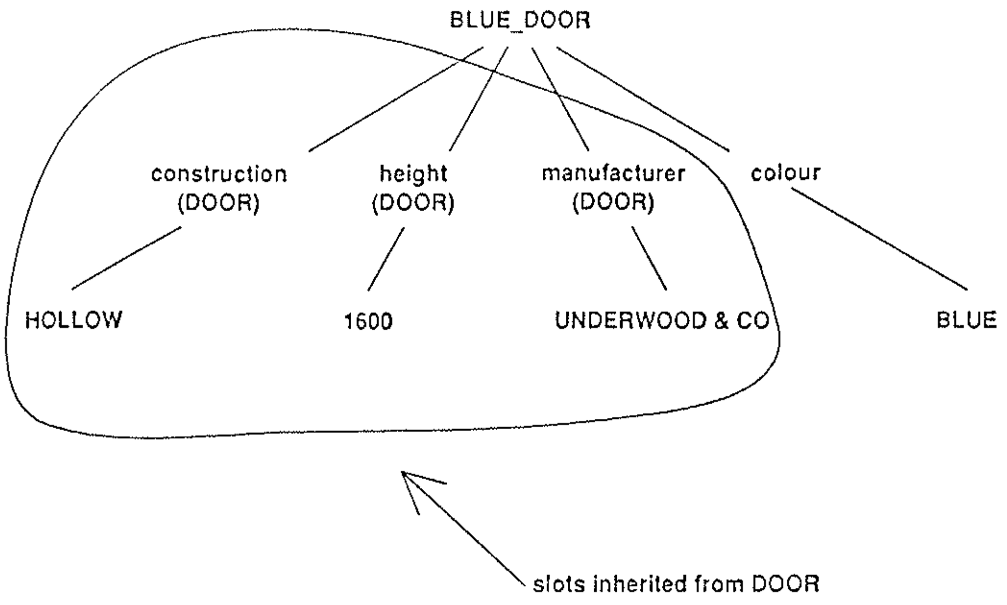


Figure 4. Graphical representation of the description of a kind indicating inherited slots.

An equivalent pictorial view of the above relationships is given in Figure 5.

Suppose we wish to state that the door knob on the back door of Dick's house is made of brass. At the same time, since we have not made any statements about the nature of the door knob on the front door of Tom's house, its description should remain unaltered. In other words we want the following statement to hold:

DICKS_HOUSE:back_door:knob:material = BRASS

but not the statement

DOOR_KNOB:material = BRASS

which if it were true would change the description of Tom's house. We do this by introducing instances of kinds. The picture of the altered database is shown in Figure 6. Here, instances *DOOR*$^{-}$ and *DOOR_KNOB*$^{-}$ have been created. In the diagram, => represents the inheritance relationship between kinds and instances. The door knob and material slots of *DOOR*$^{-}$ and *DOOR_KNOB*$^{-}$ respectively mask the fillers inherited from the corresponding slots in *DOOR* and *DOOR _KNOB*.

A similar situation would result if instead of the part identifies:

TOMS_HOUSE:front_door = DOOR
DICKS_HOUSE:back_door = DOOR

we had stated the separate doors as variants:

TOMS_HOUSE:front_door <= DOOR
DICKS_HOUSE:back_door <= DOOR

to read as: "the front door of Tom's house and the back door of Dick's house inherit attributes from the archetypal Door." In this case, the resulting database would be pictured as shown in Figure 7. Here as in

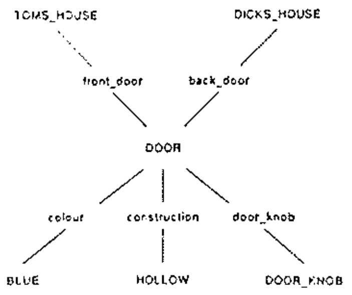


Figure 5. A snapshot of a MOLE database.

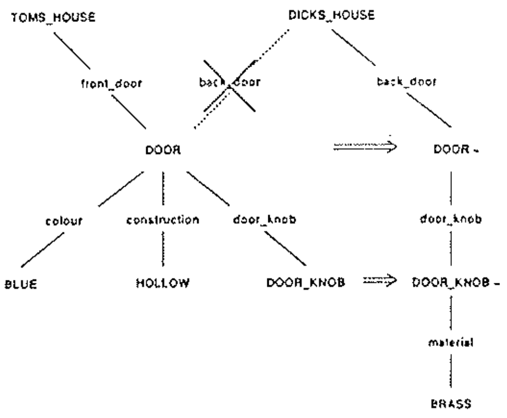


Figure 6. Updating the database shown in Figure 5. × marks the deleted slot. ⇒ indicates inheritance between a kind and its instance.

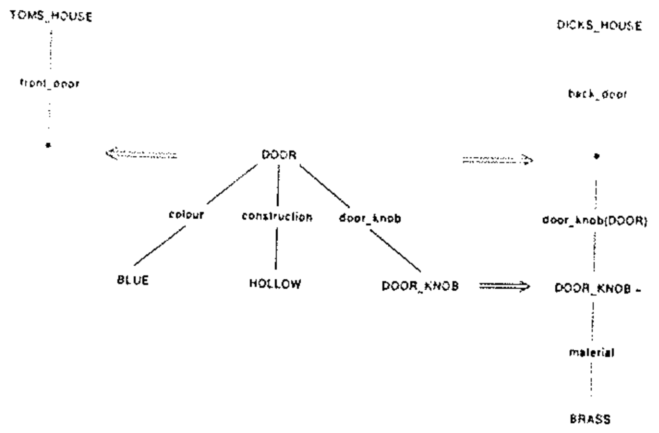


Figure 7. Updating the filler of a slot inherited from a variant. Observe that the updated door_knob slot has been tagged by the parent kind DOOR. Asterisk (*) indicates a kind or filler that is not explicitly named.

Figure 6, => denotes an inheritance relationship between a kind and another, instance or otherwise. Notice that in Figure 7, the fillers of *TOMS_HOUSE: front_door* and *DICKS_HOUSE:back_door* are denoted by an asterisk "*" which signifies that the fillers correspond to kinds that are not explicitly named. Moreover, the newly created slot *door_knob* in the new description of Dick's house is tagged by the kind *DOOR* to indicate from where the slot was originally inherited. Notice that the description of Tom's house remains unaltered.

Indirections

Consider the description of a semi-detached house. It has two houses—a left house and a right house. Moreover, the right wall of the left house is the left wall of the right house, namely the party wall of the semi-detached house. One way this can be achieved is by ensuring that the fillers of the appropriate slots refer to the same kind, say *PARTY_WALL*. That is

SEMI:left_house:right_wall = PARTY_WALL
SEMI:right_house:left_wall = PARTY_WALL
SEMI:party_wall = PARTY_WALL

However, if we were to make a change to the party wall using the part statement

SEMI:left_house:right_wall:material = BRICK

we would—as discussed in the previous section—end up with a new instance of *PARTY_WALL* which has the material slot filled with *BRICK* that is seen in the description of the left house but not in the description of the right house. What we require is that no matter what changes are effected to the semi-detached house (so long as it remains semi-detached) the description of the party wall as seen by both houses should stay intact. We achieve this by an *indirection* and is reflected in the part descriptions

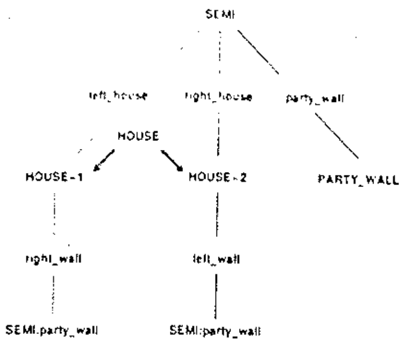


Figure 8. Indirections: the indirection SEMI:party_wall is a pointer to the kind that fills the party_wall slot in SEMI.

SEMI:left_house:right_wall = SEMI:party_wall
SEMI:right_house:left_wall = SEMI:party_wall

The expression *SEMI:party_wall* is an indirection in the description of the left and right houses, and it acts as an instruction to the description system to find the part that is identified by the part name.

Consider the database as shown in Figure 8. Here the left and right side houses are variants of some archetypal house. Suppose we wish to assert that the party wall is made of bricks. That is, either statement

SEMI:left_house:right_wall:material = BRICK
SEMI:right_house:left_wall:material = BRICK

should have the same effect. The updated database is shown in Figure 9. Here a new instance of the party wall *PARTY_WALL*, *PARTY_WALL-*, is created which contains the asserted material slot. The new instances of the party wall inherit the attributes of the original party wall description, and consequently is seen in the description of both the left and right house.

The concept of indirections can be extended. Imagine an estate of semi-detached houses say *SEMI- 1*, *SEMI- 2*, *SEMI- 3*, etc., each of which is an instance of some archetypal semi-detached house. The description of each semi-detached house should contain the fact that its left and right houses refer to the same party wall but not the party wall of any other semi-detached house. To ensure this we could laboriously

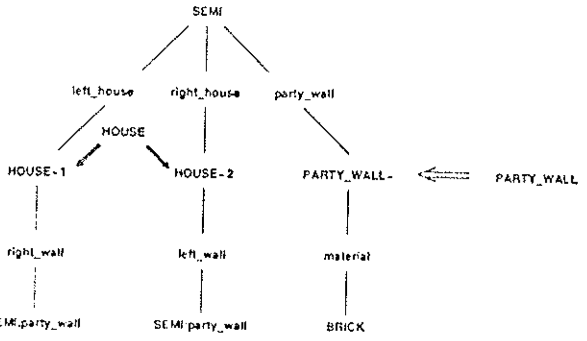


Figure 9. Updating the description of the indirected part in Figure 8. => denotes inheritance between a kind and one of its instances.

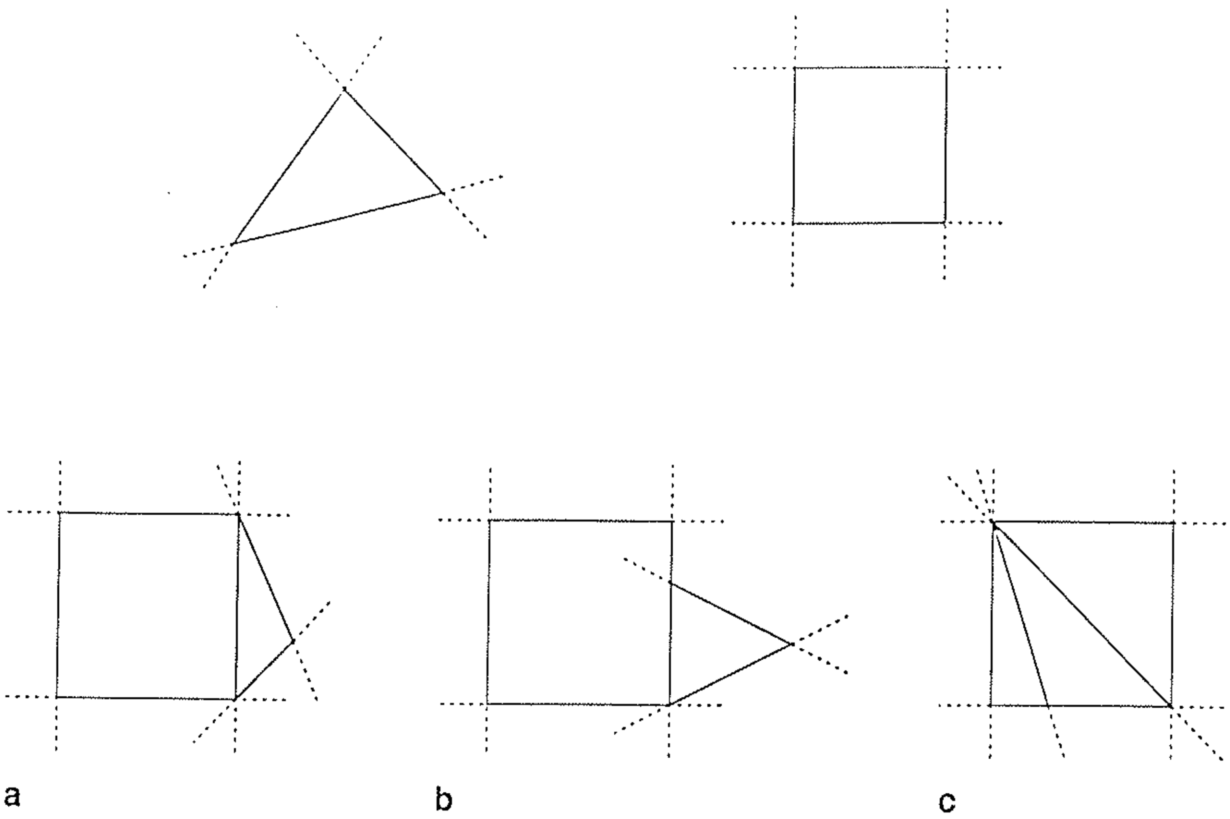


Figure 10. Examples of spatial relations between a triangle and a square.

change the indirection *SEMI:party_wall* to *SEMI-1:party_wall*, *SEMI-2:party_wall*, *SEMI-3:party_wall*, etc., in each description. Alternatively, the indirection *SEMI:party_wall* can be regarded as an instruction to find the part referred to in the particular instance of the semi-detached house. That is, the part name *SEMI-3:left_house:right_wall* which has the filler *SEMI:party_wall* should be interpreted as the instruction *SEMI-3:party_wall*. For a rigorous treatment of such complex indirection examples together with the algorithm to handle the evaluation of indirections, the reader is referred to Krishnamurti.[18]

Drawings

The major visible effort in any design activity is spent on the design drawings. The implication for MOLE is that it must provide a mechanism for carrying descriptions of the geometry of spatial objects. This mechanism—typically a graphics editor—provides a representation for graphical objects. This representation may either correspond exactly to the MOLE format or can be translated into such. For the sake of convenience we may assume the former.

Consider Figure 10 which gives arrangements of shapes formed by spatial relations between a triangle and a square.

MOLE descriptions for the triangle, square, and spatial relation b in Figure 10 are shown in Figure 11. These descriptions are based on the graphics processor elaborated in the papers by Szalapaj and Bijl.[19,20] Briefly, a shape is considered to be made up of *ink* lines placed on *pencilled* construction lines which are arbitrarily placed somewhere on a piece of paper (or screen). Shapes made up of two or more shapes in some spatial relationships are determined by relationships between the attributes of one shape and the others.

In Figure 11, the slots have fixed interpretation insofar as the graphics processor is concerned. Thus, slots with the prefix "*line*" refer to ink lines with end points given by slots with prefix "*ep*" and which lie on construction lines whose descriptions are attached to slots prefixed by "*cl*." Both ink lines and construction lines can be parametrized. Ink lines have length fillers that either determine or are determined by their end points. Construction lines may be fixed in their orientation. It should be noted that we could have given an equivalent MOLE description where fillers are typed expressions to enable interpretation by a graphics processor. For instance, a point could have been described by the filler *point(X,Y)* thereby providing the freedom to refer to points on, say a square, by slots such as *top-left-hand*. The choice depends entirely on the graphics processor employed. It is straightforward to show how

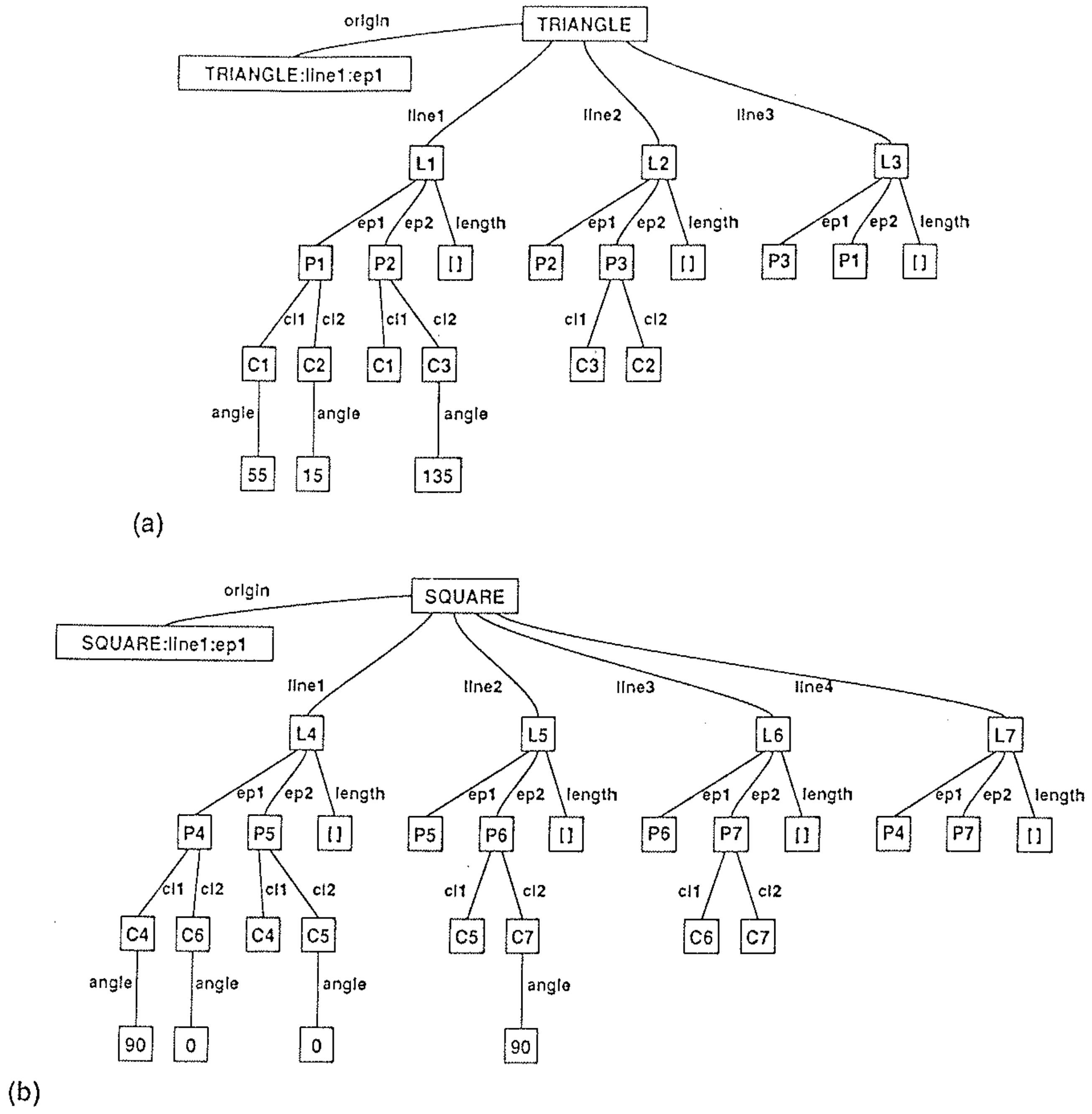


Figure 11. MOLE descriptions of the (a) triangle; (b) square; and (c) spatial relation b given in Figure 10.

spatial relationships can be constructed through sequences of MOLE statements. The expressions below define the spatial relation corresponding to the description in Figure 11.

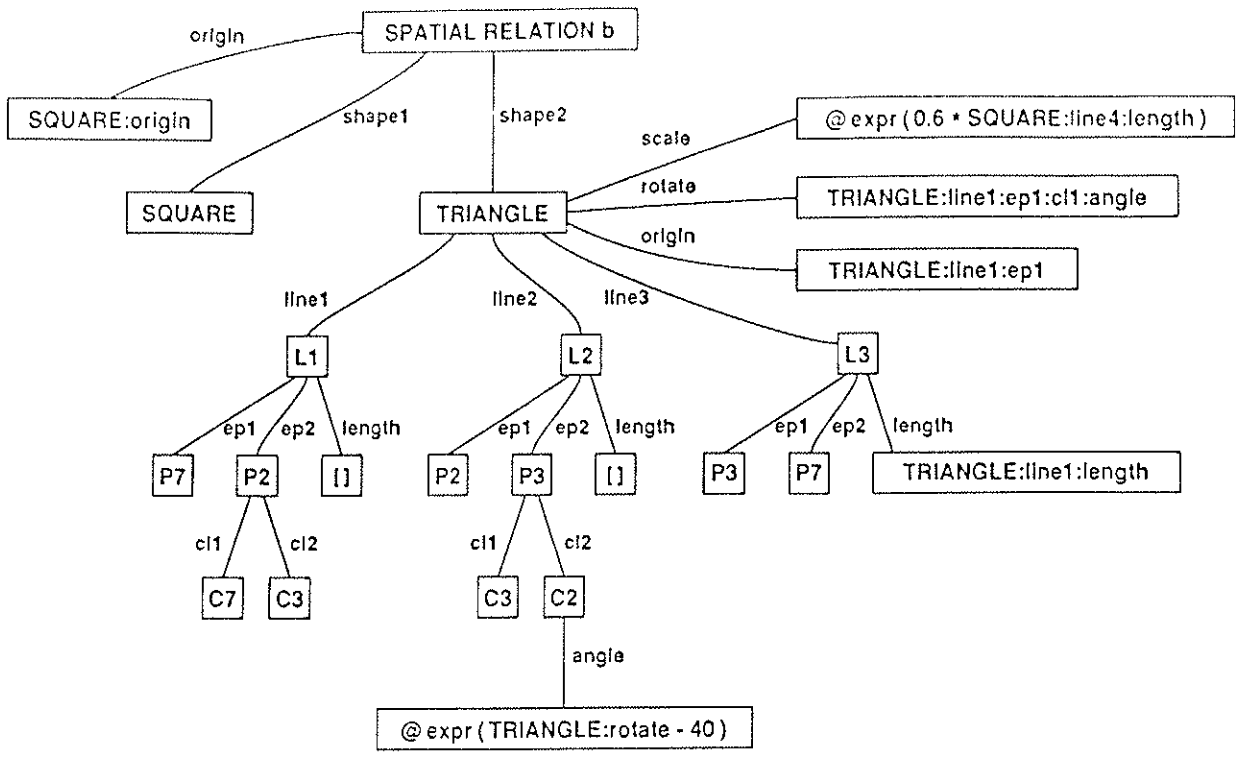
(1) *SPATIAL_RELATION_b:shape1 = SQUARE*
SPATIAL_RELATION_b:shape2 = TRIANGLE
SPATIAL_RELATION_b:origin = SQUARE:origin

(2) *TRIANGLE:line1:ep1 = SQUARE:line4:ep2*

(3) *TRIANGLE:rotate = TRIANGLE:line1:ep1:cl1:angle*
TRIANGLE:line2:ep2:cl2 = @expr(TRIANGLE:rotate - 0)

(4) *TRIANGLE:scale = @expr(0.6 * SQUARE:line4:length)*

(5) *TRIANGLE:line3:length = TRIANGLE:line2:length*



(c)

Figure 11. (*Continued*).

The expressions have been grouped into five classes labelled (1) through (5).Class (1) states that the spatial relation consists of two shapes, namely, a square and a triangle, with its origin coincident with that of the square. Class (2) translates the triangle so that a point on each shape coincides. Class (3) rotates the triangle so that one side in each shape coincides. That is, their construction lines coincide. This will rotate the other constructions lines of the triangle correspondingly. The value of θ is the angle between the construction lines that meet at the origin of the triangle. In this case, it equals 40°. The filler "*@expr*" invokes a procedure (see next section) that evaluates the indicated expression which may be parametrized. Class (4) scales the entire triangle by a factor of 0.6 so that the length of the side of the triangle coincident with the square is 0.6 times the length of the square. Class (5) moves the other vertex of the triangle so that the noncoincident sides of the triangle are equal. This will affect the description of the negatively sloped construction line. Notice that the description of the resulting spatial relation remains parametrized. A "fixed" shape is obtained when some of the points or line lengths have been specified. In other words when all filler bindings have been resolved.

Procedures

Some effort in the design task is spent on calculation or computation. A common example is the routine floor area calculation to check whether a room satisfies the statuatory minimum or maximum given in the building regulations. Computations in general can be described by *evaluable expressions* that return a value according to some criteria of evaluation (not necessarily numerical). We have seen already an example of evaluable expressions—namely, indirections. Typically, an evaluable expression corresponds to a call to a procedure or a call to a nesting of several procedures. A procedure can be thought of as consisting of a *head* together with associated parameters and a *body* that defines the procedure. For simple functions such as calculating the area of a circle, it is possible to provide a MOLE-like description. This is illustrated in Figure 12. The slots with prefix "*arg*" contain the arguments to the procedure. The slots labelled 1, 2, . . . , are the statements, in order, within the body of the procedure. The result is contained in the filler of the "*head*" slot. To distinguish between an ordinary filler and a filler that invokes a procedure, the latter is prefixed by the symbol "@".

For complicated functions, a MOLE-like description may prove difficult to formulate. It is likely that a Lisp-like or Prolog-like notation provides a more suitable format. This is not entirely desirable since the intended user is an architect with no prior programming ability. An investigation into MOLE-like descriptions for non-simple functions forms part of another research project to commence shortly.

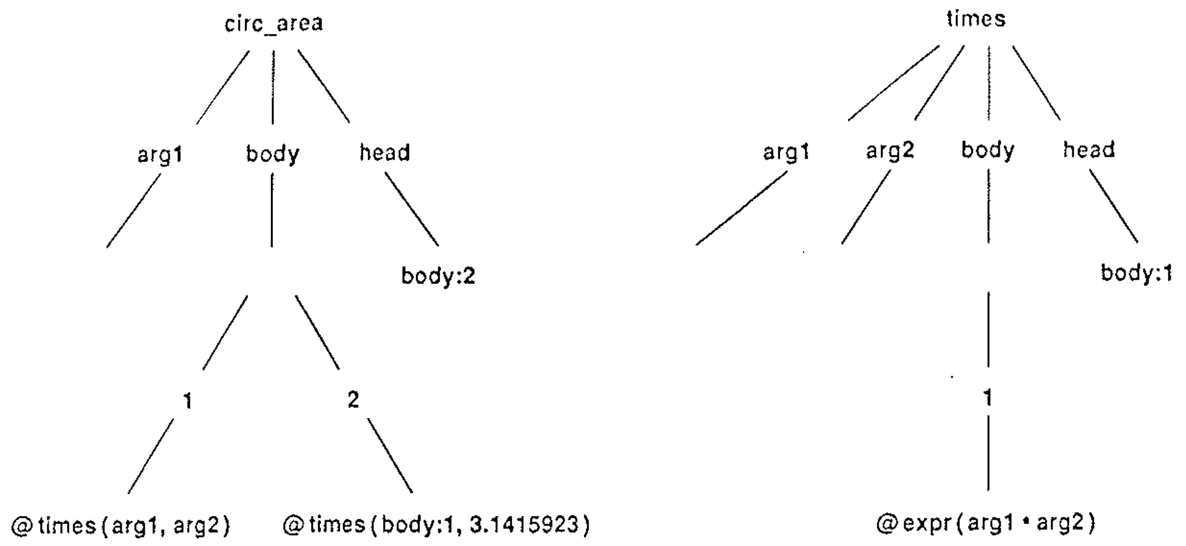


Figure 12. MOLE-like descriptions for simple evaluable functions.

The preceding remarks must be seen in light of the MOLE approach to modelling. MOLE is designed to be reflective of user intensions and descriptions. That is, the user sees only what he inputs, in the form that he inputs. The internal representations and workings of MOLE must at all times be transparent to the user.

As it now stands, MOLE is far from complete but it has the capability of accepting object descriptions, both textually and graphically. MOLE has two major components: a knowledge base manager and a graphics editor. The latter provides a representation for graphical objects similar to that described in this paper. The knowledge base manager has the capability of accepting and evaluating function descriptions using a mixture of Lisp and Prolog syntax. In addition MOLE provides facilities such as a mini Emacs-like editor and a history mechanism. MOLE is written in an enhanced version of C-Prolog.[21] The details of the current implementation can be found in Tweed.[22] In a forthcoming paper,[23] Tweed discusses some experiments that have been conducted with MOLE with applications to kitchen space planning problems; to simple shape grammar implementations; and to subshape recognition problems.

CONVERSING WITH MOLE

In the preceding section it was seen that design descriptions can be made up of relationships, each between a kind and a filler via a slot. We now briefly look at how such relationships can be presented to MOLE.

Suppose that a designer is engaged in a dialogue with MOLE. Let us assume, for ease of explanation, that MOLE accept as logically true any statement presented to it. In other words, any statement relating a part of a kind and the fillers of its slots is treated as a fact; if it is a query, its answer makes it a fact (as represented in MOLE). Thus, we can describe the dialogue using an extended form of the parts expression (the precise syntax is unimportant here). Below are given sample extracts of possible dialogues between a designer and the system. Each dialogue has three parts: (a) the meaning of the dialogue, (b) a parts expression and (c) the system response. Where necessary, additional remarks or pictures are given; otherwise each dialogue should be self-explanatory.

statement>	A red door of height 1600 mm made to BSI standard.	(1)
designer>	DOOR: [color = RED, height = 1600, made_to_BSI = yes]	
MOLE>	DOOR created	

The database is shown in Figure 13.

statement>	In addition, the door has a door knob which is made of brass, and it has a stile	(2)
designer>	DOOR: [door_knob: [material = BRASS], stile]	
MOLE>	DOOR updated DOOR:door_knob created	

The modified database is shown in Figure 14. Notice that the filler of DOOR:door_knob is not explicitly

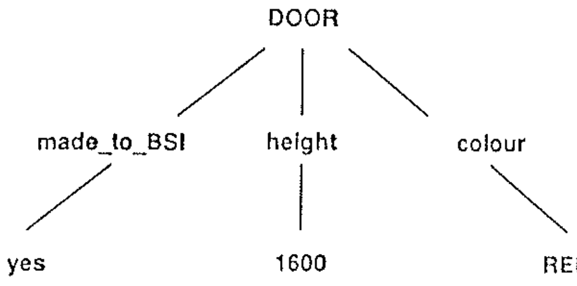


Figure 13. Dialogue 1—defining the description of DOOR.

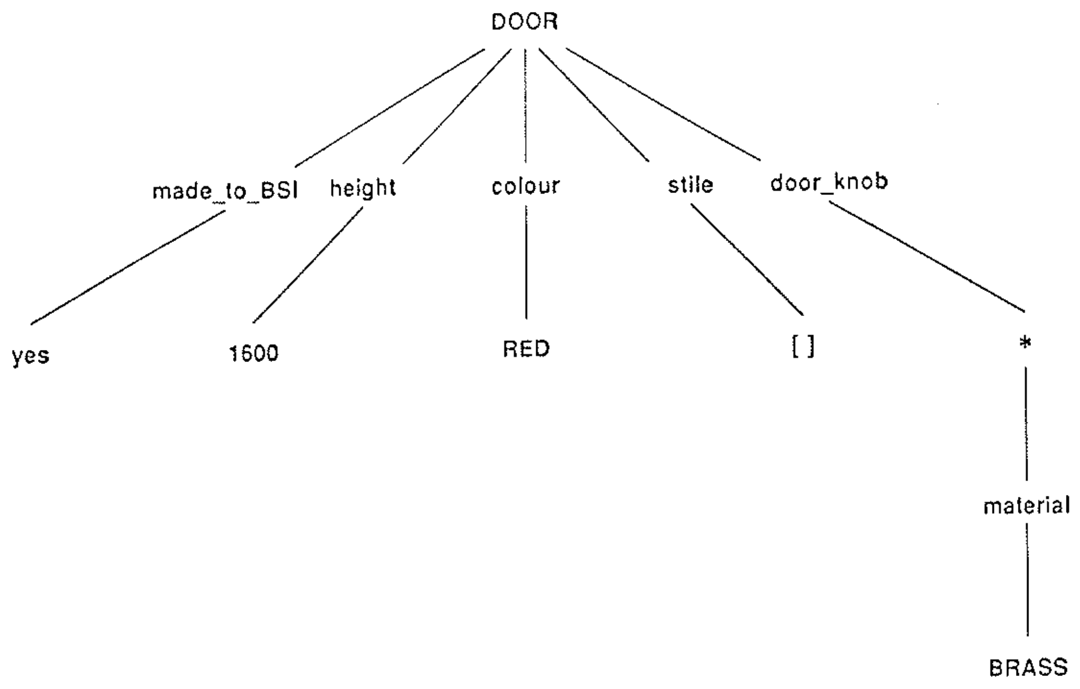


Figure 14. Dialogue 2—updating the description of DOOR [ ] denotes an uninstantiated filler.

named, and is indicated by an "*." The filler of stile is unspecified and is equivalent to the expression stile = [ ], where [ ] denotes an uninstantiated filler.

statement>	Does the door have a brass door knob ?	(3)
designer>	DOOR:door_knob:material = BRASS?	
MOLE>	yes	

Here the following convention is adopted, namely, that the presence of "?" should always denote a query. Any statement that includes "?" should not have the effect of changing MOLE's knowledge base with the exception of the "not" and "match" statements. As a later example shows, a "not" statement implies a query and negates the answer. The "match" statement allows conditional updating. The "?" may also be considered as a wild card as illustrated in the following two examples.

statement>	What is the door knob (of the door) made of ?	(4)
designer>	DOOR:door_knob:material = ?	
MOLE>	BRASS	

statement>	What objects are colored red ?	(5)
designer>	?:color = RED	
MOLE>	DOOR	

Suppose there are several doors DOOR - 1, Door - 2, . . . , etc., some but not all of which have brass door knobs. Then, the query about which doors have brass door knobs may evoke the response:

statement>	Which doors have brass door knobs ?	(6)
designer>	DOOR?:door_knob:material = BRASS	
MOLE>	DOOR - 1 DOOR - 3 DOOR - 7 (say)	

One could go further and inquire about the list of red colored doors which have brass door knobs.

statement>	Which doors colored red have brass door knobs ?	(7)
designer>	DOOR?:[door_knob:material = BRASS, color = RED]	
MOLE>	DOOR - 3 (say)	

The next example shows how inheritance relationships can be specified.

statement>	Tom has a house and its front door has the same description as DOOR	(8)
designer>	TOM: [house = TOMS _HOUSE: [front_door <= DOOR]]	
MOLE>	TOM TOMS_HOUSE created TOMS_HOUSE:front_door made a variant of DOOR	

Here, unlike in example (2), the filler of the intermediate part TOM:house—namely TOMS_HOUSE—is an explicitly named kind. Observe that the filler of TOMS_HOUSE:front_door is unnamed (see Figure 15).

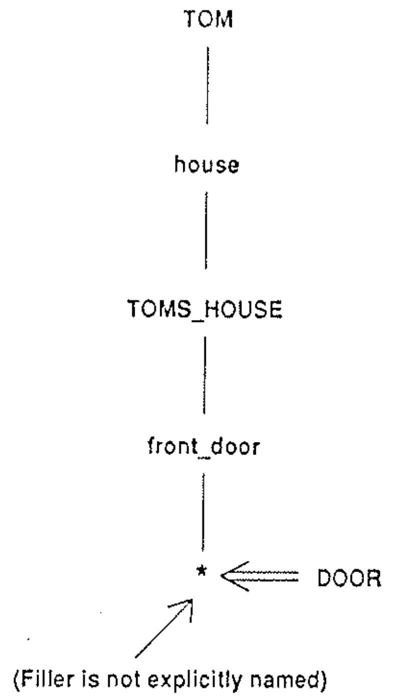


Figure 15. Dialogue 8—the description of TOM. Observe that the front_door slot is unnamed and is a variant of DOOR.

The following examples (9) and (10) are two alternative ways of presenting the same description.

```
statement>  Tom's house has a living          (9)
            room, a kitchen, two bed-
            rooms and a nursery.
designer>   TOMS_HOUSE: [ living =
            LIV, kitchen = KIT, bed_1,
            bed_2, nursery = NURS ]
MOLE>       TOMS_HOUSE updated
```

Or

```
statement>  Tom's house has a living          (10)
            room, a kitchen, two bed-
            rooms and a nursery.
designer>   TOM:house: [ living = LIV,
            kitchen = KIT, bed_1, bed_2,
            nursery = NURS ]
MOLE>       (TOMS_HOUSE)- created
```

If (9) is applied Tom's house is explicitly updated, whereas if (10) is applied the filler of the "house" slot of the kind TOM has been updated. That is, a new instance of Tom's house is created (see Figure 16). MOLE creates a new instance of the indicated part to ensure that any other reference to the indicated part is not corrupted, and thereby ensure part update consistency.

The next three examples introduce the "::" operator and which is used to denote all possible slot expressions of the form ":?: . . . :?:". That is, the variables "?" are instantiated to slots so that these slots together with the entities on either side of the "::" correspond to a valid part description derivable from the information known to MOLE.

```
statement>  Is there anything red in Tom's    (11)
            house ?
designer>   TOMS_HOUSE::? = RED
MOLE>       TOMS_HOUSE:front_door:color
            = RED
```

```
statement>  Describe Tom's house ?            (12)
designer>   TOMS_HOUSE::? = ?
MOLE>       TOMS_HOUSE:
                 front_door:
                       color = RED
                       height = 1600
                       made-to-BSI = yes
                       stile = [ ]
                       door_knob:
                             material =
                             BRASS
                 living = LIV
                 kitchen = KIT
                 bed_1 = [ ]
                 bed_2 = [ ]
                 nursery = [ ]
```

Here MOLE's response is pretty-printed for easy reading.

```
statement>  Is any part of the description    (13)
            of Tom's house inherited from
            any other description ?
designer>   TOMS_HOUSE::? <= ?
MOLE>       TOMS_HOUSE:front_door
            <= DOOR
```

Suppose Dick's house is similar to Tom's house.

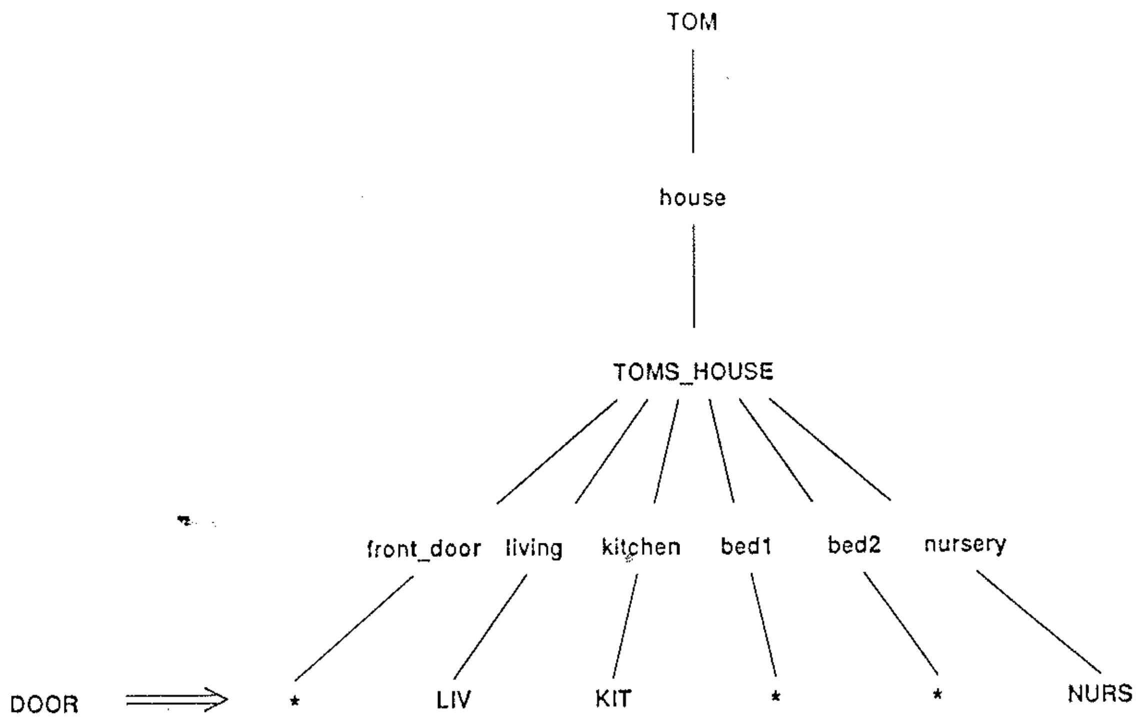
```
statement>  Dick's house is similar to        (14)
            Tom's house
designer>   DICK: [ house = DICKS
            _HOUSE <= TOMS_HOUSE ]
MOLE>       DICK created
            DICKS_HOUSE made a vari-
            ant of TOMS_HOUSE
```

The next example introduces the "not" statement.

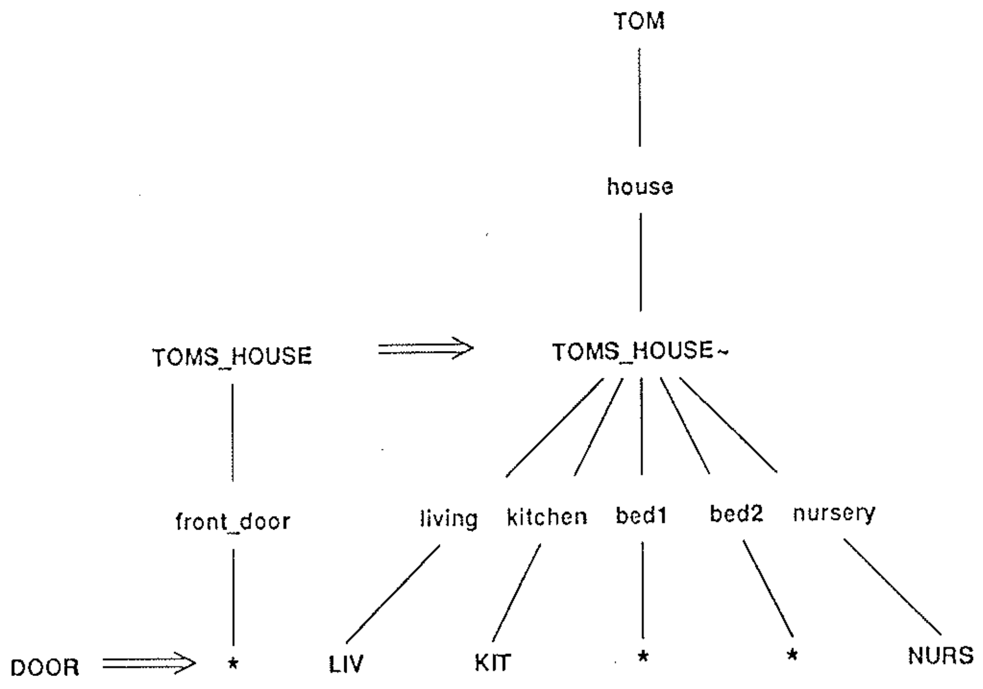
```
statement>  (However) Dick's house does       (15)
            not have a nursery
designer>   not (DICKS_HOUSE:nursery = ? )
MOLE>       DICKS_HOUSE updated
```

Notice that the "not" statement has taken the answer from the query to delete the nursery slot from Dick's house.

```
statement>  Does Dick's house have a          (16)
            nursery ?
designer>   DICKS_HOUSE:nursery = ?
MOLE>       no
```



(a)



(b)

Figure 16. Two different ways of presenting the same description. (a) corresponds to dialogue 9—the new slots are appended to the description of TOMS_HOUSE. (b) corresponds to dialogue 10—a new instance of TOMS_HOUSE is created.

The following example illustrates the situation where MOLE treats a "not" statement as a query. Since the slot to be deleted is not in the description of the kind, MOLE answers the deletion in the affirmative.

```
statement>  Dick's house does not have a      (17)
            nursery.
designer>   not (DICKS_HOUSE:nursery
            = ? )
MOLE>       yes
```

We see no conflict between the use of "not" as a statement and as a query, since the intention in both cases is to ensure that the indicated kind-slot-filler relationship is no longer in the knowledge base.

One more example to illustrate the simple use of the "?" operator:

```
statement>  Which rooms in Dick's house       (18)
            have blue ceilings ?
designer>   DICKS_HOUSE:?:ceiling:color
            = BLUE
MOLE>       living
            bed_1
```

So far we have seen how kinds can be updated and queried. Suppose we wish to update kinds only when certain conditions hold. We can do so by the "match" statement. Suppose we have a list of lintels numbered 1, 2, 3, etc.. and we wish to update those that are made of concrete. Then we can invoke the conditional match statement:

```
statement>  Lintels made of concrete are      (19)
            reinforced
designer>   LINTEL?:material = CONCRETE
            match [ reinforce = STEEL_BARS ]
MOLE>       LINTEL-1 updated
            LINTEL-4 updated
            LINTEL-5 updated (say)
```

Matching is a binary operation in which its left-hand side expression is a query that returns a kind and the right hand side expression is a part statement. It is easy to see that by matching we can carry out a global update as the example below shows.

```
statement>  The lintels have width of 900 mm  (20)
designer>   LINTEL? match [ width =
            900 ] -
MOLE>       LINTEL-1 updated
                 .
                 .
                 .
            LINTEL-5 updated (say)
```

The matching facility, indeed the simple query facility, can be extended by allowing comparisons to be made on filler values. The example below shows how only those doors which are wider than 850 mm should be panelled.

```
statement>  Doors wider than 850 mm are       (21)
            panelled
designer>   ?DOOR?:width > 850 match
            [ type = PANELLED ]
MOLE>       FRONT_DOOR updated
```

We now indicate how indirections can be specified and indirected parts updated.

```
statement>  A semi-detached house with a      (22)
            left and right house that share
            a party_wall
designer>   SEMI: [ left_house =
            HOUSE : [ right_wall =
            SEMI:party_wall ],
            right_house = HOUSE :
            [ left_wall = SEMI:party_wall ],
            party_wall = PARTY_WALL ]
MOLE>       SEMI
            HOUSE-1
            HOUSE-2 created
```

```
statement>  Define an estate of - say 7 -     (23)
            semi-detached house
designer>   ESTATE: [ dwelling(1-7) =
            SEMI ]
MOLE>       ESTATE created
```

The expression "dwelling(1–7)" is short-hand for dwelling1. . . . , dwelling7.

Now suppose the 5th dwelling in the estate has a party wall made of bricks and arranged in a flemish bond.

```
designer>   ESTATE:dwelling5:left_house:      (24)
            right wall: [ material =
            BRICK, bond = FLEMISH ]
MOLE>       (SEMI)-
            (PARTY_WALL)- created
```

Snapshots of the database before and after the description of the party_wall in the fifth dwelling is updated are shown in Figures 17 and 18.

The examples above are not exhaustive, nor are they intended to be so. They serve to demonstrate how, with a simple interface structure, one can build a powerful modelling environment. As stated earlier, fillers need not be simple entities. For instance, consider the following description of a rectangle,

```
designer>   RECT = [ length = 5, width        (25)
            = 8, area = @poly_area
            (length,width) ]
MOLE>       RECT created
```

followed by the query:

```
designer>   evaluate RECT:area = ?            (26)
MOLE>       40
```

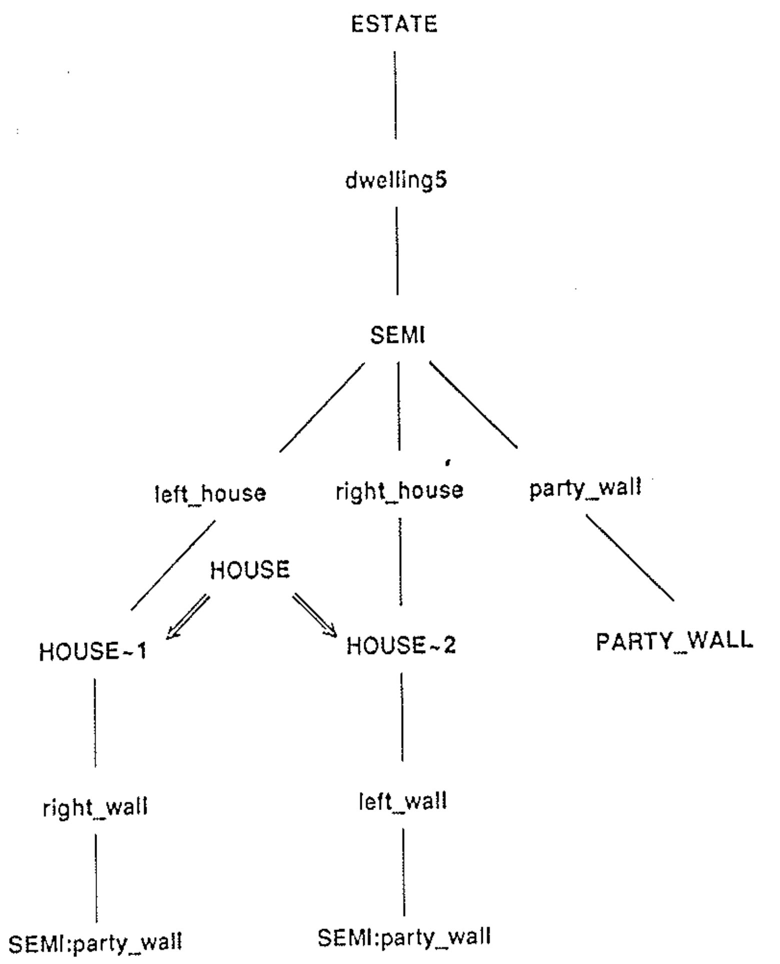


Figure 17. A snapshot of the database after dialogues 22 and 23 illustrating the presentation of indirections.

Here MOLE evaluates the procedure "poly_area" which takes as its arguments the length and width of the rectangle and returns it value. The "@" symbol denotes that the filler is a call to a procedure.

The "evaluate" function can be used with any part expression as indicated the query:

```
designer>   evaluate RECT                  (27)
MOLE>       RECT:
                   length =   8
                    width =   5
                     area =  40
```

A more complicated example is illustrated by the situation where the designer may supply with the description of a beam, the description of a bending moment analysis and a filler containing an instruction to carry out the analysis. Clearly, queries such as—are the dimensions of the beam sufficient to bear a (stated) load ?—can then be evaluated as the fragments of dialogue below indicate.

```
designer>   BEAM:moment = @proc_bm         (28)
            (LOAD, BEAM:length)
MOLE>       BEAM updated
```

followed by the query

```
designer>   evaluate BEAM:moment           (29)
MOLE>       yes
```

In the above example (29), it is natural to expect MOLE to display the bending moment diagrams on the screen. As stated in the preceding section, the current implementation of MOLE cannot provide answers to such queries in a clean manner. However, when the next phase of the MOLE project which is focused on function and procedure descriptions is completed, this will be possible.

In a similar fashion drawings may be described and queried. As a simple illustration, based on a functional description for shapes[24,25,26] one could say

```
designer>   SHAPE:drawing =                (30)
            @beside(@above(FLAT,FLAT),
            @yscale(FLAT,2))
MOLE>       SHAPE updated
```

Here the drawing of the kind SHAPE contains a functional description of a kind FLAT above itself and beside another which has been scaled in y-direction by

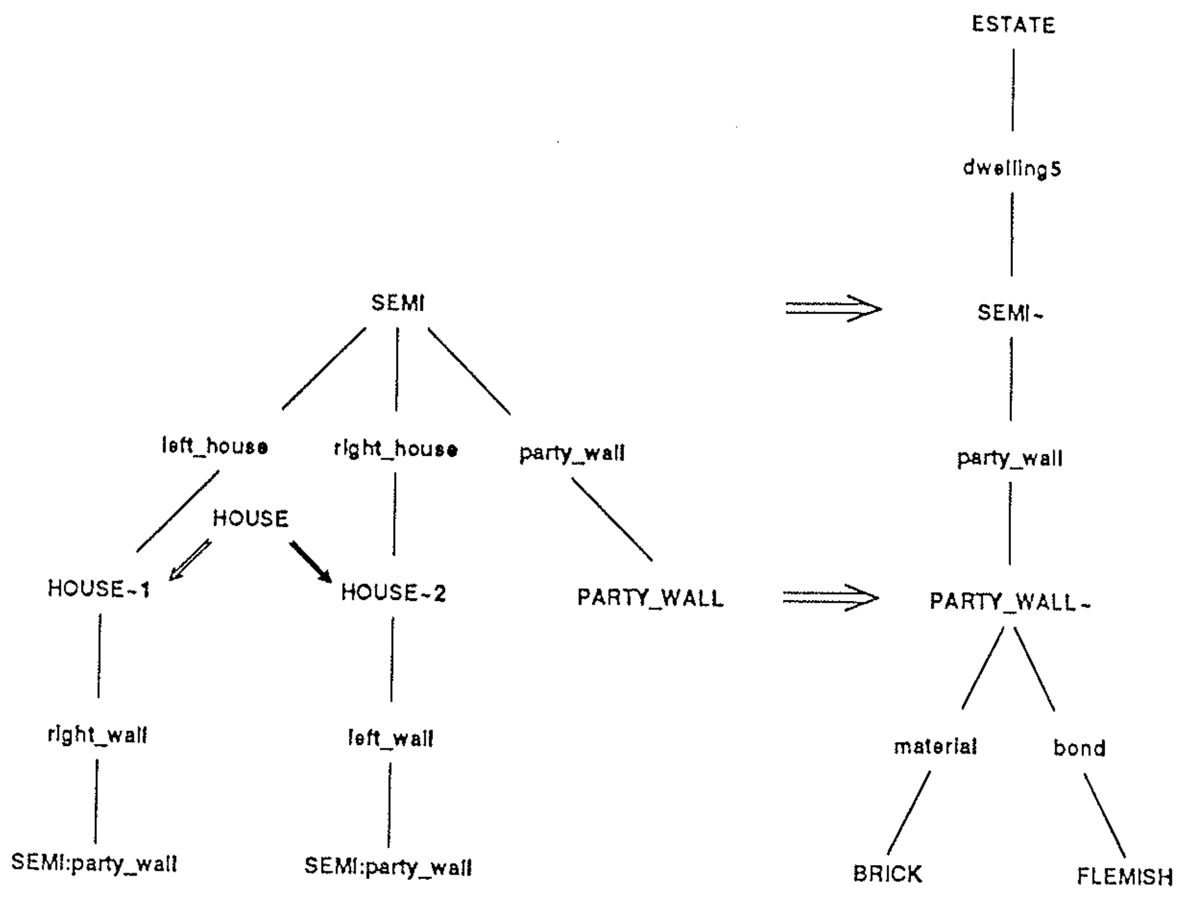


Figure 18. Dialogue 24—updating the description of the party wall in Figure 17.

a factor of 2. FLAT defines a rectangular space. Notice that the arguments to a procedure may also be calls to procedures. If the drawing is to be displayed, one could say:

designer> draw SHAPE:drawing (31)
MOLE> yes

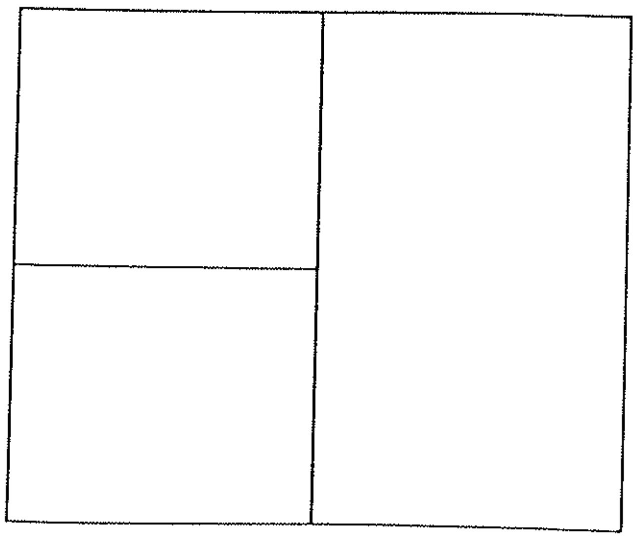


Figure 19. The displayed drawing according to the functional description given in dialogue 30.

The displayed drawing would appear as shown in Figure 19.

It is envisaged that spatial compositional rules such as those encapsulated by shape grammars[6,7] can be described in such functional terms.

TOWARDS A DESIGN SYSTEM

In this section the use of MOLE in a design system is considered.

First, a further observation about designing is made. Designing may be considered as analogous to textual composition, albeit with additional constraints. However, to produce a text composition system about—say grammatical constructs, language "style," and an understanding of the subject matter of the composed text—is unrealistic. A text composition system serves to assist the user. Likewise, the role of a design system is seen primarily as an assistant to the designer and not as an automatic generator of a design solution. Following this analogy, at the lowest level, a design system may be, simply, a dumb drawing program.

Discarding dumb drawing programs as the solution, a design system must have the machinery capable of representing both the designer's understanding of drawings and his understanding of design descriptions.

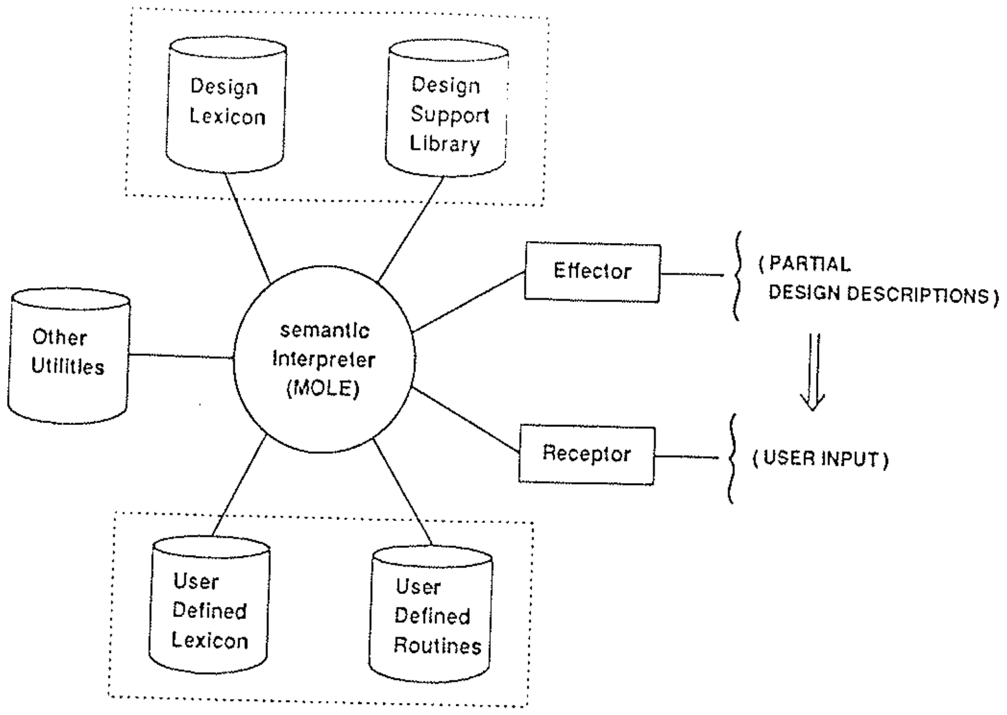


Figure 20. A schema for a reflective design system based on MOLE.

Further, it must be able to interact between the two. In practical terms, a design system must have both a graphics manager and a knowledge maintenance system and it must have procedures to link between knowledge about designs and their drawing descriptions. Design knowledge can be characterized as:

(i) reflecting the conventions and constraints that the designer *has* to adopt,

(ii) reflecting the constraints and rules that the designer *chooses* to adopt.

There is no general way of determining what category any particular design constraint falls into. For instance, in some situations, site and layout constraints may be forced on a designer. In other cases, the designer may be free to adopt or reject the need for such constraints. The only certainty is that no single design system can be prescribed to meet the needs of each and every designer (or design problem). This leads us to consider schemas for a design system that can be customized for each individual designer. A suggestion for one such schema is given in Figure 20.

At the heart of the schema in Figure 20 is the interpreter, say MOLE or equivalent, that provides both the modelling and decision support environment necessary for designing. The user communicates with the interpreter through the receptor which translates the user input, either textual or graphical, into a semantic representation. The user input may be a description that the user wants asserted to the knowledge base (after perhaps undergoing some previously defined set of consistency checks.) Or, it may be a query about some aspect(s) of the partially constructed design. The interpreter responds (feeds back) through an effector, usually by expressing partial design descriptions. The knowledge base comes in two compartments: (i) a library of design terms and rules that the designer has at his disposal, and (ii) a collection of terms and rules that apply to the specific design problem on hand.

DISCUSSION

The premise of this paper is that designs are the outcome of dialogue between the designer and a knowledge base. The gist of the argument is as follows. If the knowledge base represents statements in some first-order logic, then dialogue theory[27] for design is simply a theorem prover for deduction in that logic. However, in real design, any new assertion about a design may have the effect of invalidating any previous assertion about the design. In other words, designing is essentially nonmonotonic. Consequently, dialogue theory for design must be equipped with decision procedures based on rules that allow new assertions and invalidate old deductions.

The paper goes on to suggest that dialogue can be supported by a common representation for the description of the spatial and non-spatial elements in design. The paper describes the MOLE modelling environment wherein descriptions are given using a kind-slot-filler based hierarchy together with two distinct mechanisms for inheritance. The paper illustrates how with a uniform interface based on part expressions, MOLE's knowledge base can be updated and queried. It also indicates how design support facilities and design graphics can be incorporated through the use of procedures and other evaluable expressions as fillers in descriptions.

Design support rules are essential if we are to verify the validity of design descriptions against some model of the world. Moreover, design rules should have some form of flexibility built into them. For instance, where cost constraints are to be used as a guide for the designer and not as a maximum, a change introduced by the designer might have the effect of an "acceptable" increase in cost which would otherwise be rejected by the decision procedure that handles cost constraints. On the other hand, spatial consistency rules[28] are examples of "strong" design rules that should have the power to invalidate changes that would otherwise result in drawings that yield physically unrealizable buildings.

To complete the picture from theory to practice, the paper concludes that viable computer-based design systems for the future will be design assistants that can be tailored to the reflect the intentions of the individual designer.

I am grateful to Aart Bijl for his encouragement and support, and for his many constructive discussions during this research. I would like to thank John Lee and Chris Tweed for their critical comments on earlier drafts of this paper. The research reported in this paper has been carried out within the context of research programs at EdCAAD funded by the Science and Engineering Research Council and the ACORD project funded by the CEC Esprit program.

References

1. Moore, R. C., lecture, Workshop on Knowledge Representation, *ESPRIT'85*, Brussels, Sep. 1985.
2. Newell, A., and H. A. Simon, *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
3. Simon, H. A., "The Structure of Ill-structured Problems" *Artificial Intelligence*, Vol. 4 (1973), pp. 181–200.
4. Akin, O., "A Formalism for Problem Understanding & Resolution in Design," Proc. UCLA/NSF Workshop on Computational Foundations of Architectural Design, Los Angeles, Jan. 1985.
5. Stiny, G., and L. March, "Design Machines" *Environment and Planning B*, Vol. 7 (1981), pp. 245–255.
6. Stiny, G., "Introduction to shapes and shape grammars," *Environment and Planning B*, Vol. 7 (1980), pp. 343–351.
7. Stiny, G., "Kindergarten grammars: designing with Foebel's building gifts," *Environment and Planning B*, Vol. 7 (1980), pp. 409–462.
8. Flemming, U., "A Generative Expert System for the Design of Building Layouts," Working Paper, Department of Architecture, Carnegie-Mellon University, Sep. 1985.
9. Flemming, U., "On the representation and generation of loosely-packed arrangements of rectangles," *Planning and Design*, Vol. 13 (1986), pp. 189–205.
10. Coyne, R. D., and J. S. Gero, "Design knowledge and sequential plans" *Planning and Design*, Vol. 12 (1985), pp. 401–418.
11. Gero, J. S., and R. D. Coyne, "Knowledge-based planning as a design paradigm" in H. Yoshikawa (ed.), *Design Theory in CAD*, North-Holland, Amsterdam, 1985.
12. Bijl, A., "An approach to design theory" *Proc. IFIP WG 5.2 Working Conference on Design Theory for CAD*, Tokyo, Oct. 1985.
13. Heath, T., *Method in Architecture*, John Wiley & Sons, New York, 1984.
14. Hoepelman, J., J. Machate, "Dialogue theory, theorem proving, data base questioning and natural language," in P. Katz (ed.), *ESPRIT'85*, North-Holland, Amsterdam, 1985.
15. Minsky, M., "A framework for representing knowledge," in P. Winston (ed.), *The Psychology of Computer Vision*, McGraw-Hill, New York, 1975.
16. Woods, W. A., "What's in a link?," in D. G. Bobrow and A. Collins (eds.), *Representation and Understanding*, McGraw-Hill, New York, 1975.
17. Date, C. J., *An Introduction to Database Systems*, Addison-Wesley, Reading, Mass., 1982.
18. Krishnamurti, R., "Representing design knowledge," *Planning and Design*, Vol. 13 (1986), forthcoming.
19. Szalapaj, P. J., and A. Bijl, "Knowing where to draw the line" *Proc. IFIP Working Conference on CAD*, Budapest, Sep. 1984.
20. Bijl, A., and P. J. Szalapaj, "Saying what you want in words and pictures," *Proc. INTERACT'84*, London, Sep. 1984.
21. Pereira, F. C. N. et al, "C-Prolog version 1.5 User's Manual," Technical report, EdCAAD, University of Edinburgh, 1985.
22. Tweed, A. C., "The MOLE User's Manual," Technical report, EdCAAD, University of Edinburgh, 1985.
23. Tweed, A. C., "The MOLE Exercise Book," Technical report, EdCAAD, University of Edinburgh, 1986.
24. Arya, K., "A Functional Approach to Picture Manipulation," *Computer Graphics Forum*, Vol. 3 (1984), pp. 35–46.
25. Henderson, P., "Functional Geometry," *ACM Symposium on Lisp and Functional Programming*, Pittsburg, Aug. 1982.
26. Lee, J., "A Simple Picture Description System in PROLOG," Special session of *EUROGRAPHICS'86*, Lisbon, Aug. 1986.
27. Beth, E. W., and J. Piaget, *Mathematical Epistemology and Psychology*, Reidel, Dordrecht, 1966.
28. Borkin, H., "Spatial and Non-Spatial Consistency in Design Systems" Working paper, Architectural Research Laboratory, University of Michigan, Ann Arbor, Nov. 1985.