A Note on the logic of design

William Fawcett

This paper was published in Design Studies, vol.8, no.2, pp.82-87, 1987

This is the author’s preprint. Not for circulation. You are advised to refer to the publisher’s version if you wish to cite the publication.

william.fawcett@carltd.com
A note on the logic of design

William Fawcett

Centre for Configurational Studies, Design Discipline, The Open University, Milton Keynes MK7 6AA, UK

It has been proposed that design is characterized by distinct modes of inference. This proposition is related to logic programming, and considered with a design-like problem. The quantitative aspects of design problems are emphasized. The relevance for knowledge-based systems in design is discussed.

Keywords: knowledge based systems, logic of design, design theory

It is always interesting to consider whether design is different from other activities. Does design deal with different subject matter, or does it use distinctive methods? March has attempted to study the question from first principles rather than introspection; this note offers some exploration and discussion of March’s model.

In his paper “The Logic of design and the question of value” March proposes a model based on the work of C S Peirce. Peirce identifies three distinct types of inference, called deduction, induction, and abduction; the first two are familiar, the third is introduced by Peirce. The modes of inference are themselves derived from permutations of three types of knowledge, namely the data describing a case of interest, general laws, and the result of applying the laws to the case. March uses these categories analogously for his model of inference in design, although he prefers to use the term ‘production’ rather than Peirce’s ‘abduction’ (Table 1).

March seems to come close to identifying design with ‘production’; but he also shows how deduction and induction enter into the design process. Similarly it would be difficult to identify ‘production’ and design as Peirce indicates that production/abduction is a type of inference of general application. Let us consider how this logic of design might apply to the knowledge and inference found in design problems.

TYPES OF KNOWLEDGE AND INFERENCE

Knowledge

In a design context the types of knowledge are:

- the design itself,
- the laws which apply in the context,
- information about the design which comes from laws or rules.

Laws relate a design and information about it. The following ways of doing so can be informally recognized:

- by computing a derived attribute of the design, e.g. its weight, cost, heat loss, etc., through laws of physics or some other well-defined formulae;
- by identifying features within the design, e.g. the number of holes, a room with no external walls, etc.;
- by classifying the design as an instance of a type, e.g. a modern piece of music, a hi-tech office block, etc.

It is not important how knowledge about a relation is held – it could be in a look-up table, or in a derivation formula².
Table 1. The three modes of inference which result from permuting three categories of knowledge, as applied in a general case (above) and in a design context (below)

<table>
<thead>
<tr>
<th>General context</th>
<th>Deduction</th>
<th>Induction</th>
<th>Abduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Known Design context</td>
<td>Case, Laws</td>
<td>Case, Result</td>
<td>Result, Laws</td>
</tr>
<tr>
<td>Unknown Design context</td>
<td>Result</td>
<td>Laws</td>
<td>Case</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Known Design context</th>
<th>Deduction</th>
<th>Induction</th>
<th>Production/abduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Known Design context</td>
<td>Design, Rules</td>
<td>Design, Description</td>
<td>Description, Rules</td>
</tr>
<tr>
<td>Unknown Design context</td>
<td>Description</td>
<td>Rules</td>
<td>Design</td>
</tr>
</tbody>
</table>

Let us call any item of knowledge that can be inferred from a design, a description of the design. Thus a description can be a derived attribute, a feature, or a classification of a design. The laws, which will be called design rules, relate designs and descriptions.

Inference

In this discussion it is assumed that laws, or design rules, are known, and we will not deal with induction.

With given design rules and a given design, descriptions can then be inferred by deduction; in a design context this occurs in analysis, or appraisal, or criticism.

The interesting situation is when some descriptions are given, the rules are known, and one wants to get a design by production/abduction. The descriptions are in fact the design programme or specification or brief – they may be derived attributes, features, or classifications of the wished-for design. For example, the programme for designing a house might include the descriptions that it should:

- cost not more than £50 000 to build
- have two bathrooms
- be in a ‘vernacular’ style

It is known that in general design problems like this are solvable, because trained experts do solve them – architects in the case of the design programme for a house. A designer, like any problem solver or decision maker, possesses specialist knowledge that enables him or her to respond to a problem statement and come up with a solution, or decision, or design (Figure 1). The problem and its solution, i.e. the descriptions and the design, are necessarily related, and their relationship is embedded in the knowledge used by the expert. Taken together, the problem and solution, or programme and design, complete a relationship that is incomplete in the problem statement.

Logic programming

The idea of completing a pattern finds a direct counterpart in declarative programming languages, like Prolog³. A program in such a language is made up of statements declaring relationships between terms. When executing a program the Prolog interpreter attempts to find values for any terms which are input as variables. The way it does so is hidden from the programmer, who is not concerned with the procedures used. This is the difference from procedural languages, where specifying the procedure of execution is one of the programmer’s main tasks. Thus in micro-PROLOG⁴, TIMES expresses the relationship between numbers in multiplication, and

\[(\text{TIMES} \ x \ y \ z)\]

is a true statement when \(x \times y = z\). Thus if we make a statement

\[(\text{TIMES} \ 2 \ 3 \ x)\]

\(x\) will be instantiated by micro-PROLOG to the value 6; and

\[(\text{TIMES} \ 2 \times 6)\]

instantiates \(x\) to 3; and

\[(\text{TIMES} \ 3 \times 6)\]

instantiates \(x\) to 2. In all cases the relationship (\(\text{TIMES} \ 2 \ 3 \ 6\)) is completed. A statement in a declaration language can in general be used to instantiate the value of any of the terms in the relationship.

Suppose we use declarative statements to express design rules, taking the form

\[\text{(<DESIGN RULE> <design> <description>)}\]

Then for the inference involved in deducing descriptions

Vol 8 No 2 April 1987
from designs, one invokes the rule with <design> having its known value, and <description> as a variable which is instantiated to a true value; and for the productive/abductive inference involved in generating designs one invokes precisely the same rule with <description> having a known value and <design> as a variable – <design> is instantiated to a valid design.

In this logic programming environment it is hard to see any logical distinction between the two operations – they use the same declarative design rule, and the same logic programming interpreter.

A DESIGN-LIKE PROBLEM

An example of a relationship which is sometimes of interest in design is symmetry. The rules of symmetry relate a design to a description of it, i.e. whether it has any symmetry properties or not. We should be able to construct a rule

\[(\text{SYMMETRY} <\text{arrangement}> <\text{symmetry class}> )\]

in Prolog, and invoke it either in the form

\[(\text{SYMMETRY ARRANGEMENT}1 \text{ x})\]

to find the symmetry properties of ARRANGEMENT1, or in the form

\[(\text{SYMMETRY} \text{ x} 4\text{m})\]

to find a design with the symmetry property 4m (in international notation, or C4 in traditional notation). Consideration is restricted to the symmetry of arrangements of square tiles on a 2-D square grid. This means there are just five possible symmetry classes other than an arrangement could exhibit (m, 2m, 4m, 2, 4; or D1, D2, D4, C2, C4), or it could be completely asymmetrical.

A further restriction is made on the arrangements to edge-connected shapes, so the shapes are polyominoes. The number of polyominoes is finite, so the number of symmetrical ones is finite also. However, the numbers are large and no catalogue of polyominoes classified by symmetry properties exists. So the declarative rule has to examine explicitly the arrangements of tiles under consideration.

The declarative language micro-PROLOG was used to represent the rules of symmetry (see Appendix 1), using the relation

\[(\text{SYMMETRY} <\text{arrangement}> <\text{symmetry class}> )\]

A shape is described as a list of coordinate pairs, indicating the cells on the square grid that are occupied by tiles, e.g.

\[(<2 1)(<2 2)(<2 3)(<3 3)(<3 4)(<3 5>)\]

For a given shape the micro-PROLOG statement to be completed is, for example

\[(\text{SYMMETRY} ((<2 1)(<2 2)(<2 3)(<3 3)(<3 4)(<3 5>) x))\]

In executing this query the micro-PROLOG interpreter attempts to instantiate the variable x to each of the five possible symmetry classes, until one is found which makes the relation true, when execution succeeds. If none does, the shape has no symmetry. In this example (Figure 2) the variable x is instantiated to the value 2 (or C2 in traditional notation), completing the relation

\[(\text{SYMMETRY} ((<2 1)(<2 2)(<2 3)(<3 3)(<3 4)(<3 5>) 2))\]

This is the deductive case – using the relation to find a description from a design.

In the productive/abductive case, a particular symmetry class is given and the arrangement of tiles is a variable. If there are to be, say, six tiles in the arrangement and the given symmetry class is 2m (or D2), the micro-PROLOG relation to be completed is

\[(\text{SYMMETRY} ((x1 y1)(<x2 y2)(<x3 y3)(<x4 y4)(<x5 y5)(<x6 y6>) 2m))\]

The micro-PROLOG interpreter searches all possible instantiations of the 12 variables until it finds one that succeeds. But whereas there are five symmetry classes to consider in the deductive case, there are as many instantiations of these 12 variables as there are polyominoes on the square grid – a number which is combinatorially explosive. This makes execution catastrophically slow, and the method is ineffective even though it is guaranteed to find a solution eventually.

Of course it is easy to see ways to make the productive/abductive case more efficient (see Appendix 2). One can supplement the interpreter’s automatic ‘brute force’ instantiation by entering rules (it can be

Figure 2. Finite square grid with arrangement of six tiles represented by ((<2 1)(<2 2)(<2 3)(<3 3)(<3 4)(<3 5)). Grid has two bisectors, on top edge of row 2 and on centre line of column 3; centre point at intersection of bisectors; rotational symmetry about centre point. Relationship (Symmetry ((<2 1)(<2 2)(<2 3)(<3 3)(<3 4)(<3 5)) 2) is true
done in the declarative language) that say where to begin instantiation and how to add tiles successively in feasible locations. Instead of generating polyominoes by the thousand or million and then testing and rejecting them, these rules define procedures which generate a single arrangement with the desired symmetry class.

These are two findings

- With a declarative programming language, precisely the same relation can be used to execute deductive and productive/abductive inference.
- When a variable in a declarative relation can be instantiated to a very large number of possible values, the declarative relation must be supplemented by generating procedures.

**DISCUSSION: THE QUESTION OF NUMBERS**

In this experiment one discovers, or rediscovers, that the design of new objects is indeed something more than analysing given objects. But where exactly does the distinction lie? The author believes the simplest answer, and therefore the preferred one, is the question of numbers. In general a design programme consisting of description designs must be compared with an extremely large number of alternative candidate designs. It was noted that the descriptions may be based on computation, feature identification, or classification. Now, in computation there are often many ways of getting the same result (1 + 6 = 7, 2 + 5 = 7 . . . 117 − 110 = 7 . . . ; see also Reference 7). And in systems of interest in design there are usually fewer features of objects and fewer classes of objects than there are objects themselves. In systems where the opposite were true, and there were, for example, more classes than objects, then production/abduction would be easier than deduction.

It is the imbalance of numbers between possible descriptions and possible designs that makes design harder than analysis, even when the relationships between descriptions and designs are precisely known as in the symmetry experiment.

Is the symmetry example actually representative of design problems? There seem to be two possible objections. First, that laws, or design rules, are not always clearly known. A poorly understood relationship between descriptions and designs is just as troublesome for deduction as for production/abduction. The empirical proposition about the effect of the numbers should not be any less important in this context. The second objection is to suggest that realistic design constraints prevent the onset of a combinatorial explosion, and that a design programme may lead to only a reasonably small number of possible designs. By seeming to remove the question of numbers this might reinforce the equivalence of deduction and production/abduction. But actually the author thinks it is only making the same point as the symmetry experiment, because the way of arriving at few possible designs from a highly constraining programme is by using generating procedures to avoid unwanted alternatives.

**DESCRIPTIONS AND PROCEDURES**

What has been learned about how to model design expertise? This is the reason for studying the logic of design. It is the key to implementing knowledge-based systems for design applications.

There are two approaches to the representation of design knowledge. One is declarative, and was used for the symmetry example. It relates designs and descriptions of designs, but has the problem that a declared relationship may not constitute an effective means of finding designs that satisfy it. The other approach is procedural, and involves explicit generating rules for creating designs. Shape grammars are of this type. Here the drawback is that generating rules themselves do not directly specify what descriptions the designs they generate will possess – one finds out by using the rules. This makes it difficult to use generating rules in a goal-seeking way to satisfy a design programme consisting of descriptions. So each approach, declarative and procedural, is problematic if used on its own.

In the symmetry experiment it was found that effective design requires that knowledge of the relations between designs and descriptions must be supplemented with knowledge of procedures to generate designs. When given a design programme is stated as a set of descriptions, a designer or design system must be able to invoke some appropriate generating procedure or procedures. Thus, in addition to knowledge about descriptions and procedures, there must be meta-knowledge relating descriptions and procedures. Sometimes this is trivial, e.g. procedures to generate specific symmetry classes. In general, however, such knowledge is something distinct from the generating rules themselves.

The extent to which design inference relies on knowledge of descriptions, knowledge of procedures, and meta-knowledge relating them, cannot be established from the symmetry experiment.

**CONCLUSIONS**

So what is the distinctive 'logic of design'? The author prefers the point of view that the identity of design lies in empirical considerations, rather than in modes of inference; and in purely empirical terms it has been confirmed that design is a more difficult task than analysis. The question put at the beginning of this paper was whether design deals with distinct subject matter, or uses distinct methods. The author argues that it is distinct subject matter that makes design what it is: design is defined by the types of problem worked on, not by the way that they are worked on. 'Design methods' need not be unique to design problems, and are probably found in all creative problem solving or decision making. Equally, one should
be prepared to carry over into design any expertise in creative problem solving or decision making that has been developed in other disciplines. Design is not a private world.

REFERENCES


2 Michie, D 'A Theory of advice' in Elcock, E W and Michie, D (eds.) Machine Intelligence 8 Ellis Horwood, Chichester (1977)

3 Kowalski, R Logic for Problem-solving North-Holland, New York (1979)


7 Bennett, C H and Landauer, R 'The Fundamental physical limits of computation' Scientific American Vol 253 No 1 (July 1985) pp 38–46

APPENDIX I: A DECLARATIVE REPRESENTATION OF SYMMETRY USING MICRO-PROLOG

The top-level relationship is

(SYMMETRY <arrangement> <symmetry class>)

where <symmetry class> can take the value of the five relevant symmetry classes (m, 2m, 4m, 2, 4), and <arrangement> is a list describing the tiles making up a shape. The different symmetry classes are individually defined in terms of second-level relationships

(MIRROR <arrangement> <mirror line>)

where there is a mirror line through the arrangement,

(HALF TURN <arrangement> <rotation point>)

where there is a rotational symmetry when the shape is rotated about the <rotation point> by 180°, and

(QUARTER TURN <arrangement> <rotation point>)

where there is symmetry when the shape is rotated through 90 degrees. For example, symmetry class 2m is defined by

(SYMMETRY <arrangement> 2m)
IF (MIRROR <arrangement> <vertical line>)
AND (MIRROR <arrangement> <horizontal line>)
AND (HALF TURN <arrangement> <rotation point>)

There are similar definitions for each symmetry class. The second-level relationships are themselves defined in terms of third-level relationships:

(REFLECT <arrangement> <reflection> <mirror line>)
(ROTATE <arrangement> <rotation> half-turn <rotation point>)
(ROTATE <arrangement> <rotation> quarter-turn <rotation point>)
(BISECTOR <arrangement> <bisector line>)
(CENTRE POINT <arrangement> <centre point>)
(DIFFERENCE <first shape> <second shape> <difference between shapes>)
(UNION <first shape> <second shape> <union of shapes>)

where <reflection> is the outcome of reflecting <arrangement> in <mirror line>; <rotation> is the outcome of rotating <arrangement> by a half-turn or quarter-turn about <rotation point>; <bisector line> is a line with an equal number of tiles on either side; <centre point> is the intersection of horizontal and vertical bisectors; DIFFERENCE and UNION are self-explanatory.

Thus, for example a second-level relationship is defined by

(MIRROR <arrangement> <mirror line>)
IF (BISECTOR <arrangement> <mirror line>)
AND (REFLECT <arrangement> <reflection> <mirror line>)
AND (DIFFERENCE <arrangement> <reflection> 0)

The third-level relationships are defined in terms of bottom-level manipulation of the list of tiles in an arrangement.

When seeking the symmetry properties of a given arrangement the query

?(SYMMETRY <arrangement> x)

is submitted, and the matching value of x is produced, or if none matches then <arrangement> has no symmetry. When seeking an arrangement, the query is

?(SYMMETRY x <symmetry class>)

which causes the Prolog interpreter to instantiate x to
lists of cells, of which there are very many, until one list is found that satisfies the definitions of the specified <symmetry class>.

APPENDIX 2: A DECLARATIVE REPRESENTATION OF GENERATING RULES FOR SYMMETRICAL ARRANGEMENTS

Effective generation of symmetrical arrangements requires generating procedures. A declarative definition of generating procedures has a top-level relationship

(SYMGEN <arrangement> <symmetry class>)

and when the query

?(SYMGEN x <symmetry class>)

is submitted, a sequence of actions begins.

First, ascertain how many tiles are to be used.

Second, assume values for <bisector line> in both vertical and horizontal directions – usually the central lines of the grid being used.

Third, derive the value of <centre point> from the intersection of the two bisectors.

Fourth, derive the value of <domain> – the area of the grid around the centre point where it is feasible to place a tile and still achieve a connected arrangement.

Fifth, assume a value for <new tile> within the <domain> – this is the next tile to be placed.

Sixth, use REFLECT and ROTATE on <new tile> to place other new tiles as necessary to maintain the desired symmetry class.

Seventh, use UNION to add the new tiles to the tiles previously placed.

Repeat fourth to seventh steps until all the tiles are placed.

The exact set of actions, including the rule for deriving the domain, are defined for each symmetry class.

These SYMGEN procedures rely on exactly the same understanding of symmetry properties as the SYMME-
TRY relationships, and use the same third-level relationships. But to avoid a combinatorial explosion the knowledge of symmetry is used differently. It is a constrained generation strategy rather than a generate and test strategy.