Design knowledge in architectural CAD

William Fawcett

This paper was published in Computer-Aided Design, vol.18, no.2, pp.83-90, 1986

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
Design knowledge in architectural CAD

William H Fawcett

This paper is concerned with computer aids in architectural design, focusing on decision-making rather than drafting or representation. It argues that the knowledge-based systems paradigm is relevant, and distinguishes low-level from high-level design knowledge. It proposes that low-level knowledge-based systems are viable in the short term. An example, concerning the specification of fixings in buildings, is described. Generalization to high-level design knowledge is discussed.

architecture, design, knowledge-based systems

There seems to be a well-established conventional wisdom about the role of computers in architecture amongst the semi-informed (I am referring primarily, but not solely, to architectural students). The proposition is that computers can deal with all technical aspects, but can never get to grips with the real issues of design.

Both aspects of this proposition are suspect. I think it will be readily acknowledged by anyone who has attempted it, that the creation of a workable computer system for any technical aspect of architecture is a far from trivial exercise, and one in which success is not guaranteed. However, the conventional wisdom indicates that such systems, once created, will be accepted and believed in — perhaps over credulously.

On the other hand, the mystique built up around ‘design’ and the belief that it cannot be computerized seem equally ill-advised. What distinguishes design from the solution of technical problems? Both require knowledge, imagination and creativity, perhaps in varying degrees; if the one is not trivial, nor is the other impossible. Pretty average people produce architectural designs every day, both students and practitioners, and not many come up with original ideas very often. They design within established standards, which may be more ephemeral and less explicit than, say, the building regulations or codes of practice, but are not so different as to justify the clean severance in the conventional wisdom between technicalities and design.

Suppose that there is a gradual progression from the simplest, most mechanical aspects of architectural design, up to the most sophisticated and highly valued. Without being too specific about what attributes are used to characterize the progression, I think it will be generally accepted. Let us say that it is a progression from low-level to high-level aspects of design. Can computer aids have a role at every level in the progression, from the lowest to the highest — not now perhaps, but in the future? Certainly there does not seem to be any obvious point which could with any confidence, be said to mark a final limit for computer applications.

The question is important, for if the conventional wisdom is accepted, then computer aids will remain peripheral to the architectural profession’s central activity. We should consider whether the conventional wisdom reflects prejudice, or the weakness of existing CAD tools.

DESIGN KNOWLEDGE

Architecture fits perfectly within Michie’s definition of a machine intelligence problem: one that computers cannot do yet, and humans can. Humans, indeed, perform successfully at every level of the progression described above, but how? This is the central question in machine intelligence, or artificial intelligence, and has been studied in many contexts. A view that is now strongly established is that human expertise in a particular domain is not based on very fast and sophisticated methods of deduction, but on large quantities of domain-specific knowledge. This seems to distinguish, for example, the way we do mental arithmetic (or used to) by memorizing multiplication tables, from the way that a calculator works by always recalculating very rapidly with a recursive algorithm.

Sometimes, for example for arithmetic, knowledge-based and deductive approaches are viable alternatives, and for many problems high-speed deduction is highly suitable, eg for most computer graphics, which are extremely calculation-intensive. However, many human skills cannot be represented at all in a deductive model, or only crudely, no matter how powerful a machine is used. In these cases, knowledge-based systems have achieved some notable success. Paradoxically perhaps, computers are perfectly capable of modelling such systems with appropriate computer languages, and the representation of human expertise in this way is called knowledge engineering. The models are called knowledge-based expert systems.

How relevant are these ideas to design problems? Many existing expert systems essentially perform tasks of classification; when data about a particular case is input to the system, the case is allocated to one of a number of pre-defined classes. This could be for:

- identification, eg of bacterial infections
- diagnosis or fault-finding, when there is a fixed number of known fault possibilities
- decision-making when there is a fixed number of decision options

Design does not fit naturally into this mould, as the final product is expected to be a new object, not a selection from established designs. However, there are expert systems which overcome this objection, notably the computer configuration expert system R1 or XCON, and this program provides strong empirical evidence that the knowledge-based expert systems paradigm is applicable to design-like problems, and that human design expertise is capable of explicit representation.
R1: evidence for expert systems’ capabilities

R1 starts with a schedule of computer components required in a particular computer order — memory units, tape-drives, etc. These components have to be arranged in three dimensions inside standard computer cabinets; their arrangement is subject to numerous functional and dimensional constraints. The task of computer configuration was performed by a number of highly skilled and specialized engineers, whose method of working was as intuitive as most designers.

The expert system for this task was built as a rule-based system, so that when engineers were questioned about their skills, their answers were expressed as rules. About 200 rules resulted from interviews together with information made explicit in manuals, etc. But the 200-rule system performed very poorly; and every time it failed, the engineers pointed out some factor that they had not mentioned at the interviews — even though they were able to identify the problems. Further rules were added to represent this implicit or intuitive knowledge. By the time the system had about 750 rules it performed well and seldom failed; further rules were added incrementally. We see that R1 was able to represent knowledge about a design task that human experts were not aware that they possessed, even though they used it regularly.

Architectural design

Returning to architectural design and the progression from low-level to high-level design tasks, we may expect that there are bodies of domain-specific knowledge corresponding to the skills of human experts at each point in the progression. We would expect knowledge for low-level tasks to be relatively simple, perhaps rather precise, and common to all experts. High-level knowledge is more difficult — obviously, since it takes so long to train an architect — and may vary in detail or substance from expert to expert, ie from architect to architect.

It should be easier to represent fully low-level knowledge, that is, to create expert systems that perform to the standard of human experts. For high-level knowledge we may only be able, as yet, to sketch knowledge in a crude form, and build systems that are well short of levels of performance. The situation can be expressed in a diagram (see Figure 1), which distinguishes low-level from high-level knowledge, and low-performance from high-performance systems.

Realistic current targets are to produce systems at point A, high-level/low-performance, and point B, low-level/high-performance. In the course of the time, by incremental development, we can expect to move closer to point C, high-level/ high-performance, by the two paths shown and no doubt by other indirect ones too.

In this paper we present in some detail a knowledge-based system for a low-level task, the specification of fixings in buildings. It is hoped that this system actually achieves a position about point B in the diagram. After describing it we will discuss knowledge-based systems for higher-level tasks in architectural design.

Notice that the representation of design knowledge in a computer-based expert system implies no value judgement about the quality of the designs it produces. There is no implication of replacing plurality of expertise with a single ‘correct’ standard. The ambition is more modest: to achieve performance as good as an expert — any expert — where ‘good’ is defined by the expert. The explicit representation of design knowledge might allow greater scrutiny, and possibly lead to a raising of standards (particularly if inconsistency is viewed as a cause of ‘bad’ design), but that is a bonus and not the main aim.

FIXER: LOW-LEVEL HIGH-PERFORMANCE DESIGN KNOWLEDGE

Fixer is an interactive expert system with knowledge about fixings in buildings; it is intended to help in the task of determining fixing specifications. It is the outcome of work done in 1983–4, and is written in the logic programming language micro-Prolog, running on a micro-computer. Prolog has been proposed as having significant advantages for architectural CAD. Development of Fixer is far from complete, but it is hoped that even in its present form Fixer’s basic principles and potential for incremental enhancement can be indicated.

In design problems, unlike classification problems, there are often a number of viable alternative solutions, and even in a low-level task like specifying fixings there is no ‘right answer’. The final specification depends on the preferences of the designer and Fixer’s approach reflects this. Unlike R1 it does not attempt to come up with finished solutions automatically, but rather it uses its knowledge to offer the user a choice of viable alternatives at each stage in the generation of a specification. The options are offered in menu form, and when the user has made a selection from a menu the system sets about generating the options for the next menu. So the form of interaction with Fixer is that the system presents a sequence of menus and the user makes a sequence of selections. A typical Fixer consultation session is reproduced in Appendix I.

Structure of Fixer

Fixer is made up of Prolog clauses in micro-Prolog format. They are divided into two sections and stored as two files,
one for control of the system and the other for the knowledge base about fixings. In the course of an interactive session a third set of clauses is generated, as a working memory with data about the case under consideration.

Control

The control section is not thought of as a general-purpose 'shell', but it could be used with other knowledge bases that are amenable to the menu-oriented strategy of Fixer. The structure of the control clauses is straightforward: it uses Prolog's built-in search strategy in a degenerate way. A Prolog clause has the following form (using the micro-Prolog format and the convention that the symbol < separates consequent and antecedents):

\[ \text{((consequent) < (antecedent1)(antecedent2) ...)} \]

When (consequent) is the current goal, Prolog tests it by treating each antecedent in turn as a new current goal. When all the antecedents succeed (consequent) succeeds. In control statements the structure is used as follows:

\[ \text{((label1) < (action1)(action2) ... (label2))} \]

When (label1) is the current goal the actions are performed in turn and in so doing 'succeed'. When (label2) is encountered it is treated as the current goal and control is passed to another clause:

\[ \text{((label2) < (action3)(action4) ... (label3))} \]

A typical sequence is shown below (listing only the labels in full)

\[ \text{((make-menu x y) <} \]
\[ \text{( ( ) ( ) ... (present-menu x y z)})\]
\[ \text{((present-menu x y z) <} \]
\[ \text{( ( ) ( ) ... (menu-selection x y z)})\]
\[ \text{((menu-selection x y z) <} \]
\[ \text{( ( ) ( ) ... (make-menu x y z))} \]

The flow of control could readily be represented in a flow chart.

The main function of the control clauses is to assemble and present menus, and thus involves searching the knowledge base. The same control clauses are used for all menus, including their explanatory text, and no variables are named in the control clauses – neither the names of menus nor the names of candidates. In addition there are control clauses to respond to the user's selections from menus, and to deal with backtracking and queries.

Knowledge base

All clauses which contain variable names are knowledge base clauses, including the program header. Some clauses in the knowledge base actually contain knowledge about control, for example:

\[ \text{((next-menu SUBSTRATE material) <} \]
\[ \text{(make-menu SUBSTRATE material)} \]
\[ \text{(next-menu FIXING fix-type))} \]

A set of 'next-menu' clauses determines the top level sequence of execution; conditionals can be included in these clauses.

But most clauses in the knowledge base contain knowledge about the fixing problem. The source of the information about the fixing problem used in Fixer is standard architects' reference materials. All items of knowledge are represented in a uniform way, as (PART attribute value) triples. The fixing problem is divided into four parts:

- COMPONENT
- SUBSTRATE
- CONTEXT
- FIXING

The attributes for each part differ; COMPONENT has the following attributes:

- material
- application
- thickness

and each attribute has a range of predefined values. A menu presents all the viable values for a given PART and attribute, so the following clause generates a menu:

\[ \text{((make-menu FIXING fix-type))} \]

Thus a typical knowledge base clause takes the following form:

\[ \text{((FIXING fix-type nails CONSIDER) <} \]
\[ \text{(IS SUBSTRATE material timber))} \]

Here the value 'nails' should be included in the 'FIXING fix-type' menu (ie be presented to the user for consideration) if the antecedent is true. The structure of the antecedent will be explained below.

Some knowledge base clauses do not include antecedents:

\[ \text{((COMPONENT material timber CONSIDER))} \]

means that 'timber' will always be considered for the 'COMPONENT material' menu.

Knowledge base clauses can contain variables:

\[ \text{((FIXING fix-type adhesive CONSIDER) <} \]
\[ \text{(IS COMPONENT material x) \]
\[ \text{(IS SUBSTRATE material x))} \]

which means that 'adhesive' should be considered for the 'FIXING fix-type' menu if the material of the COMPONENT and SUBSTRATE have the same value.

Negative knowledge can be expressed in two ways:

\[ \text{((FIXING fix-type adhesive REJECT) <} \]
\[ \text{(IS CONTEX duration removable))} \]
\[ \text{((FIXING fix-type nails CONSIDER) <} \]
\[ \text{(NOT IS CONTEX visibility exposed))} \]

The antecedents in knowledge-base clauses, as we have seen, take the following form:

\[ \text{( ... < (IS PART attribute value))} \]

There are control clauses for 'IS', and their operation is first to check the working memory to see whether a selection from the 'PART attribute' menu has already been made and what value was selected. If so the antecedent succeeds or fails. If not the 'PART attribute' menu is
generated and presented to the user. Thus there can be
nested menus within the top level 'next-menu' sequence.
This is hidden from the user, since all menus are handled
in the same way; he does not know whether a menu is at
the top level or nested.

There may be a number of different clauses for the same
(PART attribute value) triple, for example

\[
(FIXING \text{ fix-type explosive CONSIDER}) < \newline
(IS \text{ CONTEXT duration temporary}) \newline
(FIXING \text{ fix-type explosive REJECT}) < \newline
(IS \text{ SUBSTRATE application precast})
\]

The 'make-menu' clauses work in the following manner.
If any REJECT clause succeeds the 'value' is rejected.
Otherwise, if any CONSIDER clause succeeds the 'value'
goes forward to the menu. If no clause succeeds the 'value'
does not go to the menu.

When all the values for a menu have been obtained
from the knowledge base there may be:

- more than one option to consider
- one option
- no options

In the first case the menu presents the options for the user's
selection. In the second case the single option is presented
in a Boolean menu, and the user is asked to confirm or
reject it. In the third case the make-menu clause that
generated the menu fails.

Working memory

Working memory is cleared at the beginning of a consulta-
tion session, and during the session is accumulated with a record
of the user's selections from menus as well as the sequence of
menus presented to the user. Typical clauses are

\[
(WM \text{ COMPONENT material wall-board YES}) \newline
(WM \text{ FIXING sub-type masonry nails NO})
\]

Negative entries arise if a value is rejected in a Boolean
menu. The sequence of menu selection is recorded as a list of
(PART attribute) pairs, e.g.

\[
(WM \text{ menu-sequence (FIXING fix-type) (SUBSTRATE material) (COMPONENT material)})
\]

When the user asks the system to backtrack there are
control clauses that take it back through the recorded
sequence. The preceding menus are offered in turn and
earlier selections deleted from working memory; the user
makes a new selection and the system continues.

Discussion

Fixer is initiated (when the system is in micro-Prolog and
after the control and knowledge base files have been loaded
into memory) by typing the command FIXER! at the con-
sole. After that the user simply has to respond to prompts
and make menu selections. When the user has reached an
acceptable specification the working memory contains data
about the case in hand and the menu selections made, and
it can be saved as a disc file.

There are a number of points about Fixer that are of
interest, and these are discussed next.

Forward or backward chaining

There is the question of whether it is a forward or back-
ward chaining system. Prolog is inherently backward chasin-
g, since execution of a clause begins with the consequent
followed by consideration of each antecedent in turn. The
sequence of top-level menus, however, has a forward chasin-
g characteristic, moving from known data towards the
final specification. In effect, therefore, there is forward
chaining of main execution stages, with backward chaining
within stages. Such a structure is very versatile. The number
of top-level stages and the amount of inference carried out
within a stage can be varied. The format for a particular
problem is based on pragmatic considerations.

In a consultation session, just one specification is ul-
timately determined, and just enough questions should be
asked to isolate it from all other possibilities. It is desirable
to avoid asking too many, too detailed questions. If the
system was not broken down into forward-chaining stages,
execution would be treated as one backward-chaining
entity, and the system would have to be back-chained from
all possible specifications. The user would be bombarded
with questions, mostly irrelevant to the final specification.
This behaviour can be simulated in Fixer by initiating an
interaction session with ((make-menu FIXING sub-type).
When this is done working memory is empty and menu
creation cannot take account of selections already made and
recorded in working memory, as would normally be the case.
However, the system does not fail, and it could be initiated
at any menu, since the flow of constraints through the
knowledge base clauses will always lead to all antecedents
of a proposition.

The pragmatic response is to divide the execution into
stages. While the number of stages and their sequence is
a matter of choice, it should lead to the user being asked
questions in what seems to him reasonable order, with
more general questions preceding detailed ones, and with
as few apparently irrelevant questions as possible.

The desired behaviour is also affected by the number
and detail of antecedents attached to knowledge base
clauses; if too many antecedents are added in the early
clauses they will generate over-detailed questions. With
fewer antecedents, there is of course the risk that the user
will be offered and may select an option which later proves
abortive; the balance of characteristics is a matter of
judgement.

The idea here that the knowledge base determines the
characteristics of the solution arrived at, and that control
factors are purely pragmatic, is consistent with the logic
programming style of declarative programming1, and is
significant for design systems in general, as we shall con-
sider below.

Uncertain inferences

Fixer does not deal with uncertain inference, only the
extremes CONSIDER and REJECT, or YES and NO. No
doubt it could be possible to replace these with a graduat-
ed scale and introduce a mechanism for uncertain infer-
ence. This may, however, be less useful for a menu-driven
design system than for, say, a diagnostic system. If Fixer
were being asked to find its own way to the 'best' specific-
ation it would be necessary to attach weights to options
that are presently offered on menus. In the menus there
is nothing to indicate which options are weak or strong,
and options can be selected which prove abortive; but
Fixer gives the user the opportunity to use his own criteria
for selection.
Considerations of efficiency

Fixer's prime objective to date has been as a demonstration program, as expert systems in an architectural or design context are still unfamiliar. This orientation, though, has reduced the importance attached to considerations of efficiency, in storage or run time. In Fixer's present form neither causes a problem, but no doubt the system could be modified with a view to efficiency. While control efficiency is always desirable in order to speed execution, there may be legitimate reasons for preferring a discursive to a cryptic knowledge base, even though multiplication of knowledge base clauses causes penalties in both storage and execution.

This would apply at its simplest as a preference for names spelt out in full rather than abbreviations or codes, and more broadly as a desire that the clauses should be individually intelligible and self-justifying (to an expert). These issues must become more critical as the scope of the system is expanded, or less compact design problems are tackled.

User friendliness

While efficiency has not been a priority, a degree of user-friendliness has been sought, by the menu format itself and the use of fairly full descriptive texts during consultation sessions. The major topic in this respect is an explanation/justification facility, if that can be called an aspect of user-friendliness.

The ability to explain reasoning and conclusions is often stated as a necessary characteristic of expert systems. Note that R1 lacks this feature. In Fixer it is not clear how significant explanations might be. As we noted previously, the clauses in the knowledge base should be self-explanatory; their function is to generate menus from which the user makes selections. All that can be explained is why certain values are on the menu or why others are not, and that can be accomplished by reproducing the relevant knowledge base clauses. Where the clauses contain antecedents these could also be traced. This facility has not yet been implemented in Fixer, but should require no modification to its structure except for the addition of some control clauses.

Interface capability

Fixer is designed as a self-contained, stand-alone system. When developed there is no reason why it could not function as such in an architectural office. However, the specification of fixings is a very small problem from the perspective of architectural design, and a computer system would not be installed for that single purpose. The interface between Fixer-like systems and other computer aids will become an important issue; but it is not tackled in this exercise.

Modifications

Fixer has at present no facility for the user to modify the system, for example to add knowledge, correct errors, or mould the system to reflect his own personal preferences rather than the generalized standards of acceptable practice. This has to be done by modifying the program files, and cannot be done at run time. The versatility of the control and knowledge base structures is very great, and it would be desirable to put it at the control of the user. Such a facility would in principle be possible, but would involve considerable additional system development, for example to avoid problems of introducing contradictions or inconsistencies. It has not yet been tackled.

HIGHER LEVEL DESIGN KNOWLEDGE

Fixer may be within striking distance of being a useful and usable tool in its limited area of concern. It may be typical of similar tools, which could be developed for other areas. It would normally be classified with the 'technicalities' of design. What can it tell us about the potential for computers in design?

We can note some properties of Fixer. It is a knowledge-based system, and no attempt is made to deduce fixing specification from the underlying physics or chemistry. To use another distinction from Michie\(^4\), it is a performance model, not a causal one, aiming as it does to generate the 'right' answers that a human expert would also generate, without being able to explain the causal relationships that make them 'right'.

A comparison can be made with other attempts to model design problems using causal models from the operations research tradition. The classic example is to model floor-planning as an application of the quadratic assignment problem. For architectural design it has not produced very worthwhile results\(^5\), and the immense effort spent in finding better solutions to the quadratic assignment problem does not overcome the difficulties. Here, the causal model is an oversimplification of the real design problem; and it seems unrealistic at present to construct causal models of architectural design that are not oversimplified.

The trade-off gained from moving from causal to performance models is the ability to handle realistic complexity. R1 is a system that deals with a level of complexity that stretched human experts, and it uses a rule-based knowledge representation.

There are examples which point to equivalent effectiveness in the architectural context. The first is Alexander's pattern language\(^6\), introduced in the 1960s. It is an ambitious attempt to model all knowledge required for designing buildings — social, economic, institutional, as well as spatial and constructional knowledge — in a uniform format: the pattern. A pattern is structured as an 'if . . . then . . .' rule. Alexander usually adds a nonoperational text to justify the rule, but fundamentally the patterns are arbitrary, and patterns can be written to achieve design characteristics determined in advance\(^7\). It is a performance approach to modelling, and has demonstrated an ability to deal with realistic architectural complexity\(^8\).

A second example is the shape grammar formalism\(^9\). It is a method for generating complex shapes out of simple ones, by the sequential application of 'shape rules'. A shape rule has a left hand side and a right hand side, both of which are shapes. To apply a shape rule to a given shape, the left hand side of the shape rule has to be identified as a subshape of the given shape; it is replaced with the right hand side of the rule, thus transforming the shape.

A shape grammar consists of a set of shape rules. Shape grammars have no explanatory power; they simply model the manipulation of shapes. But they can do this for shapes of apparently unlimited architectural sophistication\(^9\) — the highest level productions of architectural minds.

Although modelling given shapes is not fully equivalent to design, shape grammars demonstrate a rule-based representation that does not require any simplification in the complexity of architectural form.

Both these performance models differ from R1 in an important respect: they do not possess a control mechanism that turns them into automatic design machines. Fixer's control strategy is perhaps relevant here; it does
have a control strategy, but not an automatic one like R1. Its key features are:

- giving the user the choice between viable alternatives since in design problems there is generally no 'right answer'
- a pragmatic division of the task into forward and backward chaining stages

This could be proposed as a general control strategy, applicable to many design problems. It seems to resolve a perceived conflict between 'design process' and 'design product' for primacy in design methodology. The design process is at best enigmatic, and it is an impossible condition to impose on a CAD system that the design process should be completely understood and specified for it to work. On the other hand, to focus solely on the possible products of design is problematic, as the solution space for even elementary design problems is combinatorially explosive, and presents insuperable problems of selection.

In a generalized design system, or perhaps a better term is design-aid system, all knowledge about the properties of designs is held in a knowledge base; to achieve efficiency and also naturalness and convenience to the user, the execution of the program is divided pragmatically into stages approximating to a reasonable hypothesis about the design process. The same knowledge base could be used with different sequences and numbers of stages, that is, with different hypotheses about the design process.

CONCLUSION

We thus see that a number of key elements seem to be ready for use in the creation of new CAD tools. From artificial intelligence, or more specifically from knowledge engineering, there are techniques for computer representation of complex knowledge typical of human expertise, if the relevant knowledge can be extracted from human experts. The pattern language formalism and shape grammars demonstrate possible ways of expressing quite sophisticated architectural knowledge in explicit ways. And existing architectural CAD shows that techniques for graphic representation and manipulation in computers are already highly developed.

With these CAD tools the distinction between computer applications in technicalities and in design would have to be reconsidered. Even if today's conventional wisdom is a reasonably fair appraisal of CAD tools currently available, it would become untenable in the context of powerful knowledge-based tools for handling architectural decisions, in particular decisions about architectural form. Design is often narrowly interpreted as being exclusively concerned with form, but even the specification of fixings is inescapably part of the design problems faced by architects: if an architect does not himself know an answer to a fixing problem he relies on someone who does. Knowledge about fixings is therefore design knowledge, of a low level.

This paper has attempted to show that the knowledge-based systems paradigm is highly appropriate to design problems and CAD. It argues that the example of Fixer reinforces empirical evidence that knowledge-based systems are already viable for low-level design problems, and also demonstrates certain principles that are relevant for design problems in general, including higher-level design problems.

REFERENCES

APPENDIX 1: TYPICAL FIXER CONSULTATION SESSION

A typical consultation session with FIXER. The user types in responses when requested by the system. In an interactive session the screen is cleared between menus, " & " is the micro-Prolog prompt, and the session is initiated by typing "FIXER!".

FIXER is a micro-computer based expert system with knowledge about the specification of fixings in buildings.

It is a consultative system, and will ask the user to provide data about a fixing problem. It will offer alternative options from which the user makes selections, and the final specification will follow the selections.

Press 'c' to continue (and 'return') : c

FIXER is considering the COMPONENT.
Please select one option for the material from the menu :

key number option
1 services
2 rigid-tiles
3 flexible-sheet
4 light-cladding
5 light-metal
6 structural-steel
7 insulation
8 wall-board
9 timber

A abort
D don't know
? query the system
U undo previous menu selection
N none of the options is applicable
C change answer to previous question
E end session, record data

Enter one key number and press 'return' : 9

FIXER continuing . . .

FIXER is considering the SUBSTRATE.
Please select one option for the material from the menu :

key number option
1 light-metal
2 structural-steel
3 timber
4 wall-board
5 concrete
6 masonry

A abort
D don't know
? query the system
U undo previous menu selection
N none of the options is applicable
C change answer to previous question
E end session, record data

Enter one key number and press 'return' : 5

FIXER continuing . . .

FIXER is considering the FIXING.
Please select one option for the fix-type from the menu :

key number option
1 heavy-anchors
2 structural-timber-fixings
3 screws
A abort
D don't know
? query the system
U undo previous menu selection
N none of the options is applicable
C change answer to previous question
E end session, record data

Enter one key number and press 'return': 1

FIXER continuing...

[clear screen]

FIXER is considering the FIXING, specifically the sub-type.

Do you wish to use cast-in?
Please answer by making a selection from the menu:

key number   option
  0        no
  1        yes
A abort
D don't know
? query the system
U undo previous menu selection
N none of the options is applicable
C change answer to previous question
E end session, record data

Enter one key number and press 'return': 1

Sorry — FIXER has run out of knowledge!

The user is then asked whether he wishes to save the working memory which records the selections made; and the session ends.