4

Computation as Experiment

In this chapter definition based systems for geometrical modelling are reviewed before the work on applying the Definitive principle to design is described. The historical development of definition based systems reflects the concern of users to have geometrical models to help to shape design ideas. The emphasis of most systems seems to be more on actually displaying geometry than the shaping of ideas. Even recent systems that encourage experimentation do not exploit the rich potential of definitive methods. It is shown that the process of describing the state of a geometrical object enforced by the Definitive method is more in accord with the conceptual design process. The abstractions of shape required by the definitive method tend to have great flexibility and a potential that extends far beyond the solution of the particular problem for which they were built.

4.1 Definition-based Geometrical Modelling

Geometrical modelling began very early in the history of computing. Geometrical models that are used for design have a 'content' well outside their visual impact. In that respect they resemble graphs, circuit diagrams and so on. The designer often uses the displayed image as a guide to the state of the underlying design and the stage reached in the interaction. So it is important to be able to represent geometry. Curiously, one major early form of geometrical modeller based on definitions, APT, was not aimed at design but at manufacture, where geometry is already well defined. The reason was probably because it was text based, antedating computer graphics. APT (Automatic Programming of the Tool) was developed at MIT in 1956 for producing machine tool control instructions for the

shaping of aircraft structural sections: wings, fuselage, *etc.* It was structured around algebraic definitions of shapes of surfaces. Graphics based geometrical modelling came in as Computer-Aided Design systems were developed. Initially Computer-aided *Design* was a misnomer as systems were oriented towards draughting. Such draughting systems remain popular, still primarily oriented to 2D, although 3D is gaining ground. Geometrical modellers in 3D are based either on constructive solid geometry (CSG) or boundary representation (B-Rep). The core of many commercial systems owes much to another definition based system PADL-2.

The user interface of most CAD systems is command-based and can be programmed to create standardised command sets often called parametrics. The latter has recently become important to vendors with the growing realisation that customisation is vital to most applications to make real savings in productivity. We need to examine these developments with the definitive method in mind.

4.11 APT

APT began as a library of FORTRAN routines to solve algebraic equations. Its function was to define an object in terms of surface shape characteristics in such a way that a set of 3D coordinates (in any coordinate form) could be output. Those coordinates would be input to a machine tool control system that would cause a tool to move from one programmed point to the next so as to produce the surfaces. Tool movement could only be controlled initially by point to point in a straight line. Later, interpolators were introduced; linear, circular and exponential interpolation allowed continuous path profiles very close to that desired. Accuracy of form is achieved by setting a suitable interpolation step size in the coordinate output

The definition basis of APT is illustrated in the following program example from [Koren, 1983]

The Part program for the geometry is as follows.

```
10 SETPT = POINT/0,30,25                    $ Start position of tool
20 CNTR = POINT/110,90                      $ Centre point (100,90)
30 CRCL = CIRCLE/CENTER,CNTR,RADIUS,30      $ Circle at CNTR radius 30
40 LFTSID = LINE/(POINT/80,40),LEFT,TANTO,CRCL
                                            $ Line (80,40) to left tangent of CRCL
50 PTB = POINT/140,40                       $ Point (140,40)
60 BASLIN = LINE/(POINT/80,40),PTB          $ line from (80,40) to PTB
```

```
70 PTM = POINT/140,90                          $ Point (140,90)
80 RITSID = LINE/PTB,PTM                       $ Line from PTB to PTM
90 TOPLIN = LINE/PTM,(POINT/110,120)           $ Line from PTM to (110,120)
100 AUXLIN = LINE/(POINT/140,120),RIGHT,TANTO,CRCL
                                               $ Line (140,120) to right tangent of CRCL
110 XYPLN = PLANE/CNTR,PTB,PTM                 $ Plane through 3 points
120 PSURF = PLANE/PARLEL,XYPLN,ZSMALL,15       $ Plane parallel to XYPLN 15 mm below
```

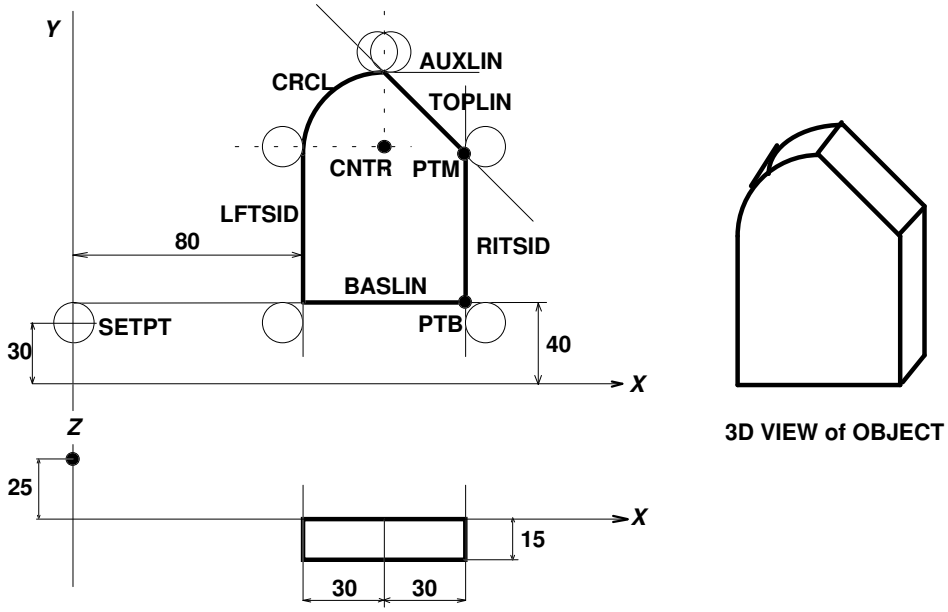


Fig. 4.1 Geometry of Program Component [from Koren 1983]

Language words denote geometry, *e.g.* `POINT`, `LINE`, `CIRCLE`, `RADIUS`, `RIGHT`, `TANTO`. These words may carry modifiers, following a slash, to define particular values. Modifiers may be expressions with references to other variables or definitions. Definitions are not typed: users are expected to check that by inspection.

User names are in FORTRAN style: upper case alphanumerics of up to six characters. Names used in the program are shown in *fig. 4.1.* The part program is submitted to the APT processor and the output is a list of coordinates marking end points or turning points. In the case of circular arcs the output lists the end point as *(x,y,z)* and the centre offset from the start point of the arc as *(i,j,k).*

By itself the coordinate list is incomplete for machine tool purposes. Geometrical statements need to be turned into machine motions. That is done by the addition of 'technological' commands to the Part Program. For example the above program continues as follows.

```
210 CUTTER/20.0                    $ Cutter Diameter
220 TOLER/.005                     $ Tolerance 0.005 mm
```

```
230 SPINDL/1740,CLW                      $ Spindle Speed 1740 rev/min clockwise
240 FEDRAT/2500                          $ Rapid Feedrate 2500 mm/min
250 FROM/SETPT                           $ Move From Setpoint
260 GO/TO,BASLIN,TO,PSURF,TO,LFTSID      $ Go to BASLIN drop to part surface at LFTSID
270 FEDRAT/500                           $ Change feed to 500 mm/min
280 GO FWD/BASLIN,PAST,RITSID            $ Commence cutting along BASLIN to RITSID
290 GO LFT/RITSID,PAST,TOPLIN            $ Machine up RITSID
    . . . . .                            $ Program continues ...
```

The addition of these statements gives an output from the APT processor that includes machining codes. Those codes are still unusable since each machine tool is different. The codes must be input to a post-processor appropriate to the local machine tool settings.

APT has historical importance; in modern CAD/CAM systems there is no need for the geometry to be defined this way as the coordinate information is already in the CAD data. For our purposes it is interesting to note the definitional and command forms that the input takes. Computation is of course on a static model - changes in geometry have to be recomputed.

4.12 PADL-2

The shape of the part program of APT could well have influenced one of the earliest forms of solid modellers: PADL. Its first experimental system it goes back to 1975. It handles objects describable as combinations of orthogonally positioned blocks and cylinders. It is important in the present work as its form at first sight is quite close to the definitive method, although differing fundamentally in significant aspects.

PADL-2 was introduced as an industrially viable "core implementation" of CSG, funded by the American National Science Foundation and ten industrial sponsors including Boeing, DEC, Eastman Kodak, McDonald Douglas and Tektronics. The core has been put to diverse use apart from straight solid modelling systems: CNC machining simulation, [*c.f.* Tan, *et al,* 1987]; verification and programming; simulation of industrial robots; representation of dimensions and tolerances; automatic feature extraction; machine process planning and automatic adaptive finite element mesh generation and analysis. Several commercial systems incorporate PADL-2, including Unisolids (McDonald Douglas), Cynergy (Westinghouse), Series 7000 (Autotrol) and AutoSolids (AutoDesk). The user manual gives the following introduction [Hartquist & Marisa, 1983].

PADL is an acronym for Part & Assembly Description Language. More generally, PADL has come to designate a family of languages and geometric (solid) modelling systems developed by the Production Automation Project at the University of Rochester. The primary representational medium in all is CSG.

The PADL language is text or keyboard oriented media with two types of statement

- declarative (definitional) statements, e.g. for creating and editing geometry, as illustrated by all but the last of the statements in the program below, and
- imperative (command) statements for evoking actions, e.g. generating displays via the `DISP` commands.

The following is a sample program in PADL-2. Line numbers are not part of PADL-2.

```
 1.    GENERIC EXAMPLE (FINAL)
 2.    A = 5
 3.    B = 10
 4.    C = A + B
 5.    D = CYL(H=A,R=C)
 6.    E = BLO(X=1,Y=B,Z=C)
 7.    CS1 = MOVX=2,MOVZ=3,MOVY=10
 8.    CS2 = CD1,MOVX=1
 9.    OBJECT1 = E MOVEDBY CS1
10.    OBJECT2 = D MOVEDBY CS2
11.    FINAL = OBJECT1 UN OBJECT2
12.    DISP PART
```

Commands tell PADL-2 to do something immediately. For example, `DISP` causes the interpreter to run through all the definitional statements and do the calculations, and then to display the current set of defined objects; `SHOW` lists all the statements input into PADL in the current session; other commands such as `SET ACCURACY = 7` will change the internal settings of PADL-2. None of these commands is remembered at the conclusion of a session.

Definitional statements are the basis of the PADL-2 language. They define user parameters (e.g. names of objects) and then assign them to the different primitives, coordinate systems and movements. The form of definitional statements is

```
username = <definitional expression>
```

and assigns a meaning to the username. Username can be any alphanumerics not commencing with a digit and not otherwise defined as linguistic terminals. Letters are always translated to upper case internally. There are three types of definitional expression corresponding to the three data types it supports:

- real expressions (may also be Boolean expressions) (e.g. lines 2, 3 and 4)

- solid expressions (e.g. lines 5 and 6)
- parameter lists (may be coordinate systems or motions)

Statements are stored within PADL-2 and are only calculated if some sort of display command is issued. Their position in the statement listing, or whether they are needed to display the particular object asked for, does not affect whether they are calculated or not. Redefinition of any of the parameters can be done by just typing in the new value: entering `A=6, B=12` discards the previous values of `A` and `B`. Any future displays or calculation on the object is done with the new values unless names are self-referential or cyclic in which case an error is indicated. A definition sequence defines "static" entities, not the "dynamic" entities that one can create with programming languages. A sequence is therefore represented as an acyclic directed graph. [Hartquist *et al.*, *op cit.*]. PADL-2 statements are not saved in the processor as entered. Only definitional statements are stored: they are translated into graph format and reconstructed if requested (*e.g.* by the `SHOW` command) but without redundant parentheses, white spaces or comments. Commands are executed as soon as entered: they are not stored.

Primitives are the usual CSG ones such as CYLinder and BLOck in the example. Derived primitives, called meta-primitives, may be constructed by employing infinite planes, half planes and existing primitives. Modifiers are set UNion, INTersection and DIFference of CSG objects. Coordinate systems are Cartesian with translations `MOVX`, `MOVY`, `MOVZ` along, and rotations `ROTX`, *etc.* about the coordinate directions.

An object in PADL-2 is formed by a set of definitional statements. That set may be identified by using the command `GENERIC <FILENAME> (<FINAL OBJECT NAME>)` at the head of the set of statements. In that case the command `DISP` without the generic name will assume the object name in the latest generic. Superficially a `GENERIC` resembles the definitive part of the 'entity' described in section 3.4 above. 'Actions' in the definitive context are effectively alternative definitions that depend upon current values of other variables. In PADL-2 one could perhaps model a simple 'action' by means of the conditional expression. For example the statements

```
Z = 10
A = IF Z GT 10 THEN 1 ELSE IF Z=10 THEN 2 ELSE IF Z LT 10 THEN 3
```

set `Z` to the value `10` and then `A` to `2`. If subsequently `Z` is redefined to `15`, say, then `A` resets to `1` after the next recalculation. A definition sequence may be reset by the guards in the conditions. That allows some variations in generic objects. The system is still essentially static as the evaluation of `A` depends upon the current value of `Z` and is only evaluated when a display is requested. Thus the `if` statement is not really an action since the definition itself remains unaltered after the action

In a recent examination of PADL-2 by Helen Butchard [Butchard & Cartwright, 1993] she found most of the difficulties were associated with the inability of the system to store commands executed during a session. Each time a geometrical object is to be produced, or PADL-2 is initiated, all the set-up commands have to be re-input; *e.g.* for the display: colours, accuracy and view type; and for output formatting: type of output file, *e.g.* Postscript. Whilst commands could be included in an input file they were only actioned at the time they were actually input. Subsequent calls on commands, to display for example, had to be re-input when required; the command `SHOW` does not list any commands.

NOTE Other practical problems occurred using PADL-2. The definition of coordinate systems quickly becomes confusing. It proved better to define systems in terms of one another rather than absolutely. Computational time is a serious problem in PADL-2: Display time is great and Boolean operations seem to take a great deal of time to compute. [Brown, 1982] gives a more thorough technical review of PADL-2.

4.13 Parametrics

Commercial CAD systems generally have an interface language that allows users and developers to write macro programs and functions in a high level form that accesses the command structure indigenous to the use of the CAD system itself. These languages are well formed in that they correspond closely to popular programming notations such as Lisp, Forth, Basic and C. The most common way of using these languages is to create sets of functions to do tasks that customise the basic CAD system to local needs. Straight customisation might be to adapt the menu structure or make specialised functions available on menu. The use of these languages to create abstract objects is called parametric modelling. The user creates an object on the CAD system and then proceeds to assign variable names to particular dimensions or shape features. That enables a ‘parametric’ to be written to produce the drawing with different dimensions. As the given program is run it requests values for the parameters required to construct the drawing.

The form of interface languages is generally complex and requires a good knowledge of the CAD system itself. Because the languages sit 'on top' of the CAD system they are computationally expensive and inefficient. Nevertheless they have proved a very strong spur to progress in object representation and a substantial commercial industry has grown up alongside all popular CAD systems.

Parametrics serve the need implicit in PADL: being able to specify objects in ways that are flexible in their design features. In concept they also share the limitation of being static objects inasmuch as they describe an object rather than create an object with state. However more recent parametrics, on ComputerVision for example, allow animation and state changes that are more akin to the definitive approach. These owe their development to work on standalone systems of which Design View is typical.

4.14 DesignView

The limitation of not being able to use parameter changes to change the object representation automatically is clearly surmountable. DesignView® [DesignView Manual, 1991] is such a solution. A quotation from the manual gives the flavour.

> DesignView is based on dimension-driven variational geometry, a technology that makes drawing much easier and more flexible than ever before. When creating a drawing using DesignView, you do not need to be concerned with the initial sizes of geometric objects - you need only sketch in the basic shape. Later on, you specify the exact dimensions and DesignView automatically reshapes the geometry for you. DesignView' s dimension-driven variational geometry maintains complex relationships between drawing elements, so circles stay tangent and lines stay connected.
>
> DesignView' s powerful analytic capabilities can solve inequalities and simultaneous non-linear equations, calculate mass properties, simulate dynamic systems, and more. It is ideal for synthesising linkages, doing tolerance stack-ups analysing forces and solving complex geometric problems such as belt-pulley configurations.

DesignView is written to suit a graphics system so can sit easily on top of an existing CAD system. The PC version sits on a variation of the Windows simple line drawing package, extended to allow greater graphics manipulation (*e.g.* the creation of splines). Another version sits on CADDS5 from ComputerVision. A sample picture from the manual illustrates the method and is quoted as *Fig. 4.2* overleaf.

The problem illustrated in the figure is to design a wall mounted crane using force polygons. The user draws in the shape of the crane with some particular dimensions and also constructs the stress diagram. The system provides the graphics facilities to do the drawing. Labels are added to the geometrical features that are to be constrained. Relationships in the form of equations on those labels are then written using DesignView facilities. The constraint system ensures that the geometry then reflects the current values of the variables in the equations. In the case of the crane the variations are the dimensions associated with the geometry and the load it must support. If, for example, the vertical dimension of element 5 on the crane is increased to 50 then the crane diagram is redrawn and the stress diagram is automatically updated to reflect the new loading regime. Similarly the results table is updated to note the new loading values. Successive changes may be input by means of iteration over a user-specified range. Those changes may then be animated to give the illusion of a mechanism in operation.

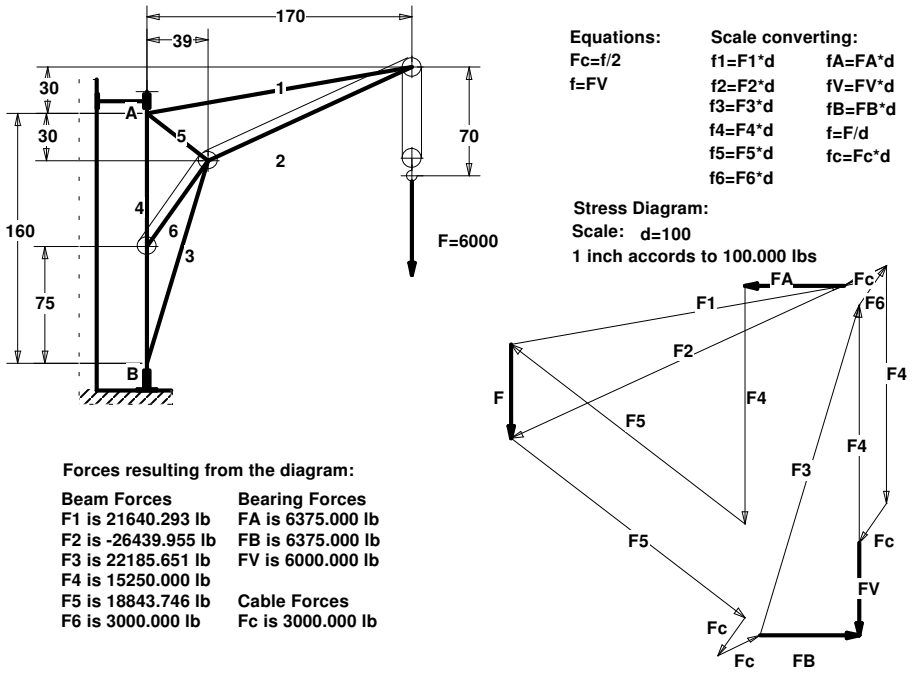


Fig. 4.2 Wall mounted Crane - file ex-crane.dv [DesignView Manual, 1991]

Because the program is constraint based it is not as flexible as it appears at first sight. Only pre-conceived parameters may be varied. One cannot for example change the appearance of the screen display through the system, nor can one change the dimensions to SI metric as neither of these was conceived when formulating the problem. Also the attributes that can be added to objects are

merely recorded. Only relationships resulting in a geometrical change will be automatically updated. DesignView may be linked to external programs such a spreadsheet using a special connection that allows parameters to be manipulated in the spreadsheet and then transferred into DesignView. In that respect the system is the nearest to the definitive method. It also points up the need for this type of facility

4.2 Definitive Notations for Geometrical Modelling

4.21 Comparison with other systems

Of the definitional methods we have described, DesignView is nearest to the definitive philosophy that we advocate, especially as it has an explicit link with spreadsheets. As an environment for design it has some desirable aspects. However it does bear the marks of a pragmatic approach to design of objects, essentially extending an idea incrementally. It is natural to try to strengthen existing commercial systems. Engineering applications are very demanding of computers - and ever more so as product life cycles are dropping so rapidly. The problem is that there is great inertia in the commercial systems because of the huge investment in programming. For example the decision by software developers to move from FORTRAN to C was made with great reluctance even though it yielded great gains in software development. Furthermore the success of approaches such as ProEngineer show that developers willing to try new methods can benefit. A new programming should be commercially attractive if it can be shown that it can be used to deal effectively with relating form and content in CAD and with problems of integration and interaction.

Definitive methods appear to hold out feasible solutions to those problems. The value of an interactive system to an engineer is not whether it can produce a solution to one current design problem: rather it is whether it can maintain selected relationships established in earlier attempts at a solution, whilst allowing the designer to try quite different approaches. That implies that the system must make all the current relationships available to the user, so that they can be modified and particular entities already designed can be referenced for future interaction.

Take PADL-2 for example. Referencing objects already designed may be done via previously defined names and composite objects by means of the GENERIC device. Modifications are relatively easy provided they use the primitives built into the

system (or meta-primitives derived from those and bounding planes). As abstractions these primitives are simple, with default values (usually of unity) for all defining dimensions. The underlying algebra is of reals, solids and parameter lists. If we wish for example to reference vertices on an object that must be done indirectly since they are unlabelled, although it is possible to 'cheat' by accessing the system' s own internal record of the object. To be able to have interaction of any kind whatsoever, *e.g.* changing a solid representation to a B-rep or wire frame or a 2D drawing or changing the display to suit a particular representation, we need a programming method that is much more open.

It needs to be shown that definitive methods are applicable to those areas. We start with the representation of shape. My work demonstrates that definitive methods enable shape definition of great abstraction and flexibility, permitting the design of computational objects with potential that extends far beyond the solution of the particular problem for which they were built.

4.22 Representation of Shape

In order to be as abstract as possible in thinking about shape, geometrical models are first defined in terms of reference and construction points, labels, skeletal structure and geometrical operations that are typically used to synthesise complex objects from simple components. By that means use may be made of many different computer representations: whether it be wireframe, CSG, B-Rep or any other.

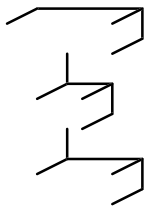


Fig. 4.3 Form suggesting "E"

Wireframe constructs are easiest to model in abstract terms. The structure can be represented by labels, connectivities, linetypes and coordinate information. They are powerful models inasmuch as they can suggest a content far beyond the form represented, as the example in *fig. 4.3* shows. The letter E emphasised in three dimensions can be perceived in the form shown even though there is no connectivity to show a solid shape. Similarly "solids" may be indicated simply by shading or selective line removal.

CSG models are constructed from primitives that are defined by inclusion of all points within the solid, *e.g.* the criterion for membership of a solid sphere is expressed as $|\mathbf{x}-\mathbf{c}| \leq \mathbf{r}$; assigning specific values to **c** and **r** determines a specific sphere. Planar faced objects are defined by use of infinite half spaces. The primitives of CSG are then assembled by Boolean operations.

Boundary representation (B-Rep) is more related to wireframe than CSG. B-Rep models are based on face-edge-vertex graphs with data structures for surface geometry, curve geometry, and coordinates: analogous to that for the wireframe. That structure is usually represented by the winged edge topological structure face-edge, edge-curve, vertex-point shown in *fig. 4.4.* Surfaces do not have to be

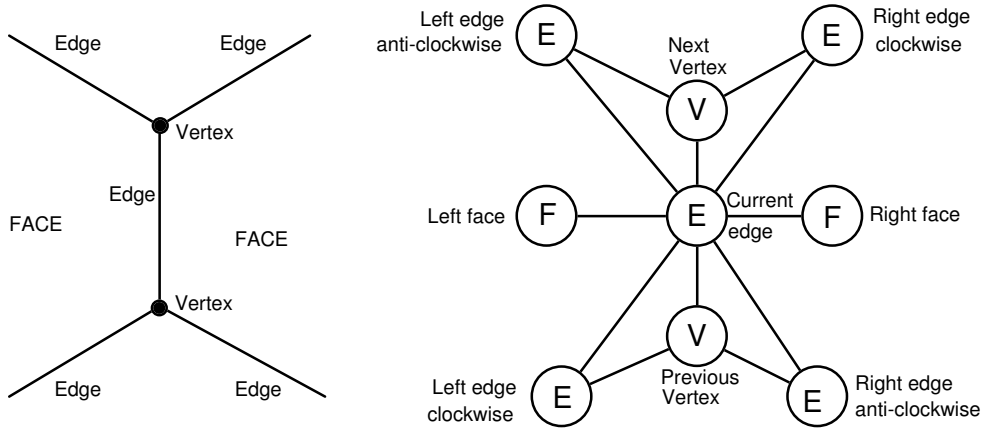


Fig 4.4 Winged Edge data structure for B-Rep
[Rooney & Steadman, 1987]

defined by analytic half-spaces, B-rep modellers can use a wide range of surface descriptions including free form or sculptured surfaces on shapes of limited extent. See for example [Woo, 1985]. The basic relationship on the face-edge-point graph is the Euler-Poincaré formula

$$V - E + F - H = 2(M - G)$$

where an object has V vertices, E edges, F faces, H hole loops (complete intersection of a section on a face, such as a cylinder creates a circular hole loop), M disjoint pieces, and G 'handles' or through holes. Faces in graphs obeying that formula may be multiply connected. The formula, having six variables all of which are integers, defines a 6-dimensional integral grid . A change of any variable (called an Euler operation) may be represented as a transition between points on that 6D grid. Thus any solid may be built up from single Euler operations. In practice these operations are combined into user-friendly groups to do things like Boolean operations, sweeping and swinging, and various tweaking operations.

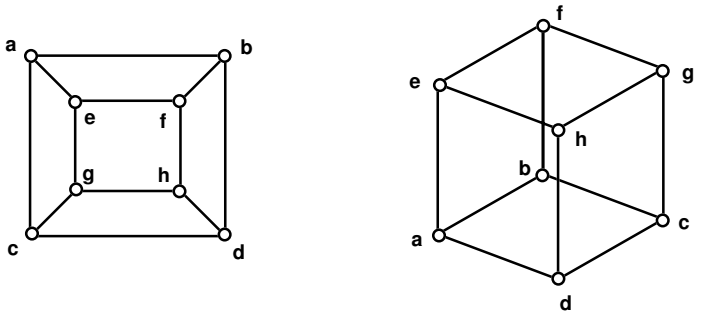
Transforming from wireframe to B-Rep is fairly straightforward as the topological structure of wireframe is a subset of B-Rep. However transforming CSG models to B-Rep or a common form is difficult since labels are very different and in general common vertices between the two representations must be derived. It is indeed an issue whether it is actually possible to have a transformation without additional information. It is also difficult to integrate the many different characteristics of geometrical models into a single unifying framework, a factor observed by [Smithers, 1987].

Definitive Shape Representation

If we specify shape sufficiently abstractly we should be able to produce a common frame of information from which many forms can be derived. One common feature of all shape definitions is the need for labels, whether to denote vertices, edges and surfaces, or to provide parameters to define analytic functions. A second need is to use graph theory [*c.f.* Wilson, 1987] to construct graphs that describe combinations of vertices on the one hand or connections between components on the other. We therefore build the algebra for our shape definition on labels and graphs. The shape model that was initially conceived was reported in [Beynon & Cartwright, 1989] but has seen some development since then. In essence, the underlying algebra incorporates three distinct sorts for describing geometric objects: complex, frame and object.

A **complex** comprises a list of labels, collected into a list of subsets of the set of labels. This is the normal way of representing abstract graphs. Labels may be thought of as nodes on a graph structure with the edge structure specified by the list of subsets. Alternatively they can be viewed as references to abstract points that lie in a Euclidean space of dimension **d** ≥ 1; this means in particular that labels may refer simply to abstract scalars. The non-negative integer **d** is the dimension of the complex although that is not specified as part of the data type. The complex interpreted topologically is designed to capture the combinatorial ingredients of the object: reference points, dimensions and incidence information expressing the way the object is synthesised from simpler components. There is no coordinate information in a complex.

The complex allows us to represent structural information concerning geometrical objects in a graph form for B-Rep and Wire-frame model or to relate analytic variables in the CSG or indeed to cover the relationship of components in a multiple-feature object. It incidentally allows us to specify families of objects or



BOX = ((a b) (b c) (c d) (d a) (e f) (f g) (g h) (h e) (a e) (b f) (c g) (d h))

Fig 4.5 Topological representations of the complex BOX

assemblies with the same abstract structure (isomorphism). *Fig 4.5* illustrates two ways of showing a complex BOX on the labels a .. h with the same subset structure. The 2D representations convey very different geometrical shapes to the beholder.

One way to give geometrical substance to a complex is to link labels with locations in space. For this we use the **frame** that consists of a complex together with a list of coordinate vectors all of the same dimension d, whose role is to supply locations for the vertices. Even in this form a frame remains an abstraction from an **object**, though there is a canonical way to realise a frame as an object, the intention is that a frame supplies a finite set of parameters that can be used to represent essential reference and construction points from which an object may be synthesised. For example the primitives in CSG may be represented by frames with default coordinates: a block would be synthesised from coordinates corresponding to a unit sided box realised in straight line segments.

The **object** is specified not by its extent (*i.e.* the set of points deemed to be within the object) alone, but by the ingredients from which the extent is in general determined, obtained from the complex and frame information. Other information may be added to give the object position and shape in terms of scaling and isometries (where it is to be located in world coordinates and what scale factors apply in each axis). That would be similar to what one might do with an object imported into a drawing in a CAD system. An object may be defined on a single frame; in general it is determined by a list of frames, together with a function that takes the parameters of these frames as arguments and returns the extent of the object. Thus the combination of the whole set of frames constitutes the object. By further functions that state how the structure is to be expressed graphically we can instantiate the object for display purposes

4.23 Operations on Definitive Shape Types

The idea behind the choice of sorts (complex, frame and object) is that an object is to be viewed as the realisation of a combinatorial structure, as represented by an underlying list of complexes. Effectively we can use these sorts to represent "parametrised objects", although these will be more generic than ordinary parametrics.

Given the three basic sorts described we can create **types** to describe each. That will enable us to make suitable **operators** on these sorts. For example we may wish to have operators to synthesise new complexes from old, geometric operators that assist in the specification of coordinates for frames, or operators that combine objects as in CSG, or to specify objects, frames and complexes in terms of each other.

Since a *complex* is built up of labels, which are atoms or strings, the algebra will include operators for list and string processing respectively. From a list of labels list operations can be used to build standard subsets such as that shown in *fig.4.5* for constructing a box. We can combine complexes by set operations such as union and intersection or perform graph property extraction such as lattice properties, paths and cycles.

The *frame* associates coordinate information with the vertex labels, so operators may be classified according to their effect on sorts.

(1) Constructors and selectors able to construct and unpick frames
(2) Operators for accepting complexes and lists of reals to realise a frame
(3) Operators on frames for making new frames
(4) Operators on frames that return an object

Under (3) we can have all the ordinary vector operations (addition, subtractions, dot and cross products, isometries, scaling and shearing). More generally we might want operations such as forming the complex hull of vertices on their coordinates, finding the mid point of point pairs, building the union of two frames or a frame comprising the boundary of several frames.

Operators under (4) deal with realisation of objects. Edges may be realised as line segments, arcs or splines. (*c.f. Fig. 4.6*). A spline, for instance, is determined by a wire-frame together with an appropriate set of boundary elements. The wire frame

has two ingredients: a combinatorial structure, consisting of an array of labelled points, and an associated array of coordinates. By specifying how the spline is abstractly defined in terms of the frame and the boundary elements, without regard for their specific coordinates and scalar values, the spline can be specified as an abstract object. By subsequently supplying parameters for an appropriate function, *i.e.* specifying a suitable explicit list of frames, a spline is derived as an explicit object.

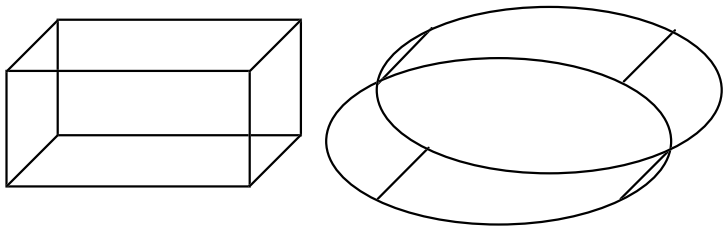


Fig. 4.6 Two realisations from the same combinatorial structure

Some of the operators attached to objects are associated with display representations. It may make little difference to the display picture whether the object appears as a 'polyline' rather than line segments, but it makes the possibilities of other operations for manipulation very different.

The representation of shape and operations have clear implementation dependent issues and are considered again in chapter 5 in connection with EdenLisp. Meanwhile we need to consider more profound issues in relation to interaction and the advantages that definitive notations introduce.

4.3 Extending Interaction

Interaction can be thought of in a number of ways. In computation the user and computer interact via inputs and outputs. In a physical assembly, such as a mechanism, the components interact via interfaces such as connections and bearings. Analogies can be made between these different notions of interaction though agent oriented ideas. The components of a mechanism are agents with a self behaviour and an interface with the rest of the world. In the computational sense we can extend the idea of ' interaction with the user' to ' interaction with agents' , some agents being human, others being autonomously acting computational objects. This is the basis of the thinking in agent oriented programming in the ADM described earlier.

Before dealing with agents we need identify who or what are the agents in a designed object or system, or in the design process itself. We begin by following up the discussion in chapter 2 on the design process. According to that discussion, design is best done by breaking the problem into relatively independent sub-problems in a hierarchical way. That enables us not only to discover the functions and shapes of components but also how they link with one another. We can then define components so that they can be dealt with as independent entities while not considering interfaces with other components. That provides one way of identifying those aspects of a design that can be accorded to one agent. We can then consider the nature of possible interfaces with other components to see how agents interact. Thus the first way of extending interaction is to be able to identify and structure potential agent elements in the design.

A second, related, issue has to do with constraints placed upon the user' s interaction with the computer, for example in the way that a design is displayed and input is made. It can be awkward if the display can only show aspects of the design in a predetermined way. This is an issue on conventional CAD systems where the user interface, usually via a screen display, has elements of format that cannot be changed very easily by the user.

The third aspect of interaction is to do with changing state in a sequential way, such as in animation. We may wish, for example, show a series of positions in a locus as emulation of a moving mechanism.

These issues are addressed in the Abstract Definitive Machine (ADM) outlined in section 3.42 above. In order to develop the methods for design purposes it was felt necessary to examine the advantages and deficiencies of DoNaLD in particular, given that this notation is nearest to CAD.

4.31 Hierarchies in Design

The concept of environment is attractive from the point of view of design. We should like to be able to structure a design problem hierarchically so that the sub-problems can be put to one side for later development. DoNaLD has the ability to create that hierarchy though its notation

```
openshape ...
  within ... {
```

However the interactive element has some disadvantages. Consider the following script of definitions written by the author in DoNaLD for illustrating Bow' s

Notation for beams in bending. *Fig. 4.7* shows a sample output from Bow.DoNaLD. A beam is modelled from straight lines using the notation described in §3.4. The structure of the program is as follows.

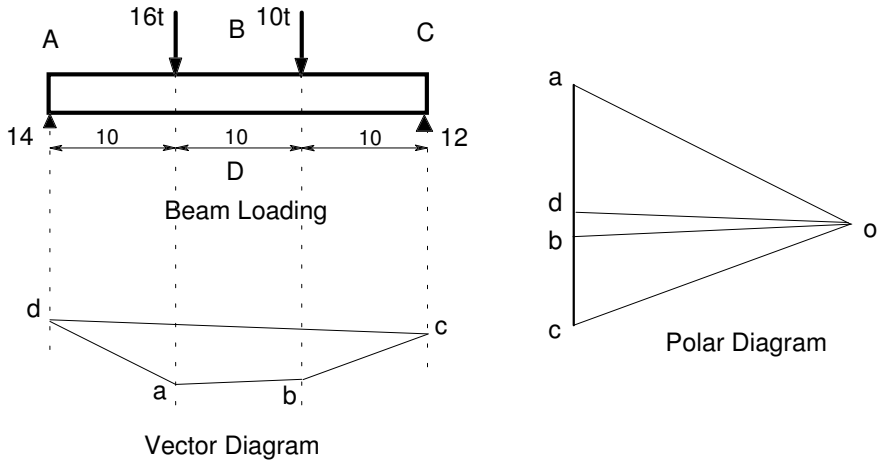


Fig. 4.7 Bow's Notation Method

Bow.DoNaLD Features
  Beam Loading diagram
    beam, load arrows, reaction supports, dimensions, leader lines
    labels: Bow's notation
    Definitions: position of beam, position of loads, scale factors
  Polar Diagram
    Position of Pole,
    length scale for loads,
    position of load vectors
    Labelling of forces **ab bc** and Pole **O**
    Lines from **O** to **a**: **Oa** and **Ob, Oc**
  Vector diagram
    Extension of lines from Beam Loading diagram
    Construction method:
    Copy polar line **Oa** and place in region **A** between the vertical extension lines from the end support and 16t load: trim to fit. Label as **da.**
    Copy polar line **Ob** and place in region **B** at the end of **da;** trim to form vector **ab.** Repeat for **Oc** in region **C** to form vector **bc.**
    The vector joining **c** to **d** is the required resultant.
  Result
    Copy the resultant vector **cd** back to the polar diagram as indicated in the *animation* with end **d** at the Pole point. Where it intersects the vertical line **abc** is the location of **d** on the polar diagram. **cd** and **da** are the magnitudes and directions (*i.e.* **c** to **d**) of the support reactions

The following is a segment of the DoNaLD code from the program script Bow.DoNaLD that illustrates some of the points under discussion.

```
### Bow's Notation for Beams in bending
point A, B                          # End points of the beam
```

```
line  AB                                # the beam
int   scx, scy                          # scale factors for the display
real  LB                                # beam length

AB  = [A, B]                            # define line with A and B ends
B   = A + [LB,0]*scx                    # position of B
scx = 20,  scy = 1                      # default scale factors
A   = (55,800)                          # origin of display beam
LB  = 30.0                              # default beam length

openshape forces                        ## beam forces and reactions
within forces {
  point C, D                            # Points of action of forces
  line RA, FC, FD, RB                   # Lines of action of
                                        #     forces and reactions
  real Ra, Fc, Fd, Rb, LC, LD           # Labels for
                                        #  reactions/forces/distances

  Rb = -(Fc*LC + Fd*LD) div ~/LB
  Ra = -(Rb + Fc + Fd)
  RA = [~/A,~/A+{0,~/scy}*Ra]           # Draw lines proportional
  FC = [C,C+{0,~/scy}*Fc]               #    to force magnitude
  FD = [D,D+{0,~/scy}*FD]
  RB = [~/B,~/B+{0,/scy}*Rb]

  C  = ~/A + {LC,0}*~/scx
  D  = ~/A + {LD,0}*~/scx

  LC = 10.0, LD = 20.0                  # Values for force positions
  Fc = -16.0, Fd = -10.0                # Values of forces
}
```

Addressing the first of the interaction issues: in Bow.DoNaLD we divide the problem. The polar diagram, beam elements diagram and force vector drawing are, for drawing purposes, separate problems and have their own definitions. For example, the code above shows definitions for the beam forces and reactions. The interlinking of the sub-problems requires access to local variables within the drawings In DoNaLD we localise variables using *within shape*. Access to variables is by having a labelling method that carries the address as well as the name of the appropriate variable. The scaling factors used in the display need to be common for the vector diagram to look correct. Access to those factors is by the symbol **~/**, a method reminiscent of the directory structure of Unix or DOS, *i.e.* **scx** is accessed by the name **~/scx** just as Unix directories address parent directories. In an OOP system that kind of addressing is difficult as interfaces need to be anticipated in order for messages to be enabled. New ways of linking are also impossible in OOP without accessing the code directly - and that would be against the spirit of information hiding.

Once entities are written interaction may be done directly by editing the definitions at the keyboard. Also we can use previously stored definition sets. Various entities in Bow.DoNaLD have to be defined for routine components such as arrow heads and labels. These can be 'library definitions' easily adapted to the case in hand. It is worth noting that labelling is a particularly powerful aspect of the definitive interaction. Stored or explicit strings can be attached to labels and displayed according to the values of their definitions so it is very easy to link both the content and position of a label with any other entity. That makes updating the display extremely easy: labels automatically get moved to new positions and/or get new strings without further definition each time any item is changed. Indeed we have shown that it is possible to arrange that the labels are always positioned where the fewest line intersections with the text occur. Labels can also have values that change according to other interactions, *e.g.* constraints can have warning labels whose value is a string that contains information appertaining to the interaction just performed. These kinds of facilities lift the restrictions noted in Design View and illustrate the potential for definitive methods.

4.32 Indirect Interaction

The second issue is mainly to do with the presentation of the current design to the user. The set of definitions that form an object represents the current state of the interaction very effectively as the user can readily determine the current state of the dialogue at any time, and can predict the effect of any dialogue actions. However, the use of definitions can also be unnatural, since it requires an acyclic system of functional dependencies between variables. If the criterion for a good representation of the state of an interaction is 'predictability of response to dialogue actions', the restriction to acyclic systems of functional dependencies is superficially unnecessary. For example in DoNaLD a square, defined by points **a**, **b**, **c** and **d** where

```
b = a + [0,1], c = a + [1,1],  d = a + [1,0],
```

has dependencies on **a** that allow the square to be translated by a simple redefinition of the value of **a**. However those dependencies are not obvious from inspection of the graphical display and suggest that it may be better to think of the definitive notation as an intermediate code that can be automatically updated by the computer under certain circumstances. The definition of **a** could be changed by the computer if the user used a mouse or similar to drag the square. In that sense we could hide the value of **a** from the user (although, unlike OOP, that definition and value would be accessible if wanted). That device is in the Abstract Machine

(ADM) where we called it an *action.* The action would be to update the definition of `a` depending on the current state of the user interface.

The idea that interaction from the user interface causes indirect redefinition can be extended. There is no reason in principle why the user interface itself should not be controlled by definitive methods. The current representation on a display should equally deal with the idea - its shape and state - and its current description in terms of what windows are open, what menu options are available and what forms of responses are required to inputs. With such complex tasks the computer is better able to cope than the user with the tedious task of redefinition. Given an appropriate method of updating (the *guards* and *actions* of the ADM) we can envisage that the state of the interaction can be changed by triggered actions.

4.33 Animation

To show, for example, the vector **cd** in *fig 4.7* 'move' from the force vector diagram to the Polar diagram we need to draw the vector in a number of intermediate positions. If that is done by redefinition there is no need to worry about deleting the entities when the next position is defined as the display is automatically updated when a redefinition is accepted. However the action of creating intermediate values is strictly a procedural one. DoNaLD shares the problem of PADL in that it is not possible to include procedural actions. Thus for interaction of this first type we are forced either into recording the intermediate positions as explicit definitions, or going beneath DoNaLD to the definitive interpreter. This is a serious disadvantage and ways of dealing with time related issues such as these are still a problem.

The definitive interpreter EDEN, an 'evaluator for definitive notations' is based upon a mixed programming paradigm [Beynon & Yung, 1988]. The EDEN interpreter has built-in support for a definitive notation based upon list processing, but can also be programmed to perform traditional procedural actions that may be synchronised with changes in the dialogue state using triggering mechanisms resembling those used in OOP. By translating definitions into the internal definitive notation it is easy to represent the state of the dialogue over any definitive notation. By using triggered actions it is easy to make responses contingent upon the current state of the dialogue. In effect EDEN makes it possible to link complex procedural actions and intricate systems of definitions: a very powerful programming paradigm but one that can prove difficult to use and analyse. It is possible to program directly in EDEN from within DoNaLD in order to

carry out the procedural tasks described. In an experimental system that is permissible although a 'proper' notation would of course have those tools built in. Experiments in 'mixed' environments have been most revealing in the investigation of state. We turn to one of these experiments now.

4.4 A Computational Experiment

4.41 Background

The issues raised in the last section and in the implementation of the Abstract Definitive Machine led us to carry out an important experiment. In design work it is common to have to write short programs to solve particular problems, or to employ a commercial package such as a spreadsheet or MathCAD. For this experiment I took such a program, written in Pascal, to investigate precisely what interactions and states were implicit there. The program was designed to solve shaft deflection problems frequently required in machine design, presenting the output in graphs similar to those shown in the diagram shown below, *Fig 4.8.*

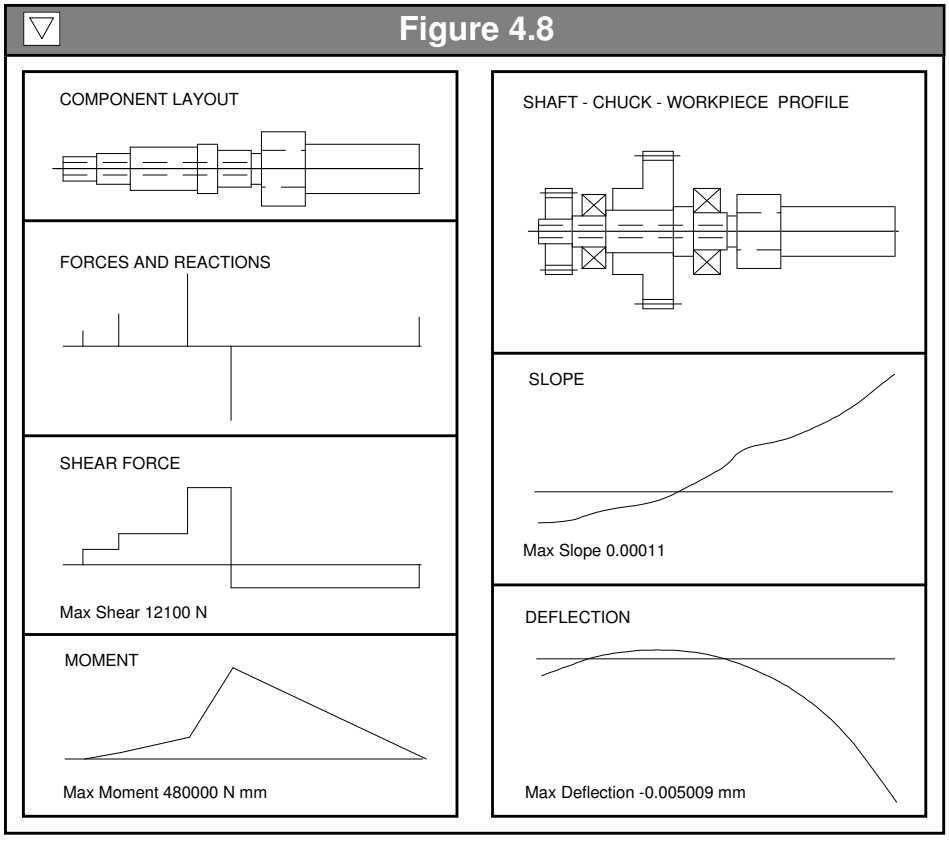


Fig. 4.8 Screen from Prototype Design Environment

The input allowed for a number of bearings and loading conditions and configurations, but the form of the output was always the same and the method of describing the solution was fixed into the procedures. Rewriting the program into DONALD revealed that the hierarchies assumed in the Pascal program were designed to carry out the computation of specific problems required by the output rather than modelling the problem in terms of abstractions of shaft arrangements, about which information could be obtained for particular cases. The input shaft was divided into segments of constant cross-section and the section properties under stress were computed by the transfer matrix method. Computation involved several independent trawls of the input in order to set up the matrices and solve them for deflection and other properties. Procedures were then set up to enable the sequence of interactions required for articulating the design.

The hierarchies for a proper modelling of the shaft should be designed to show up the functionalities and features that may at any time be altered so that any kind of experiment may be performed on the model. For example we may wish

- to devise a representation of the drive shaft/workpiece configuration from a set of parameters
- to display engineering data for different configurations
- to experiment with geometry, *e.g.* additional segments or tapered segments
- to vary the disposition of components
- to monitor the deflection as a function of manufacturing cost
- to simulate the lathe in use
- to adapt the system to new uses such considerations of strength or appearance

4.42 Method of Approach

We can simplify the tasks by applying a definitive method with its intrinsic ability to deal with 'what if?' mode of analysis. A 'what if?' scenario is defined by a state and a set of latent transformations of that state: it expresses our expectations about how the various ingredients of the shaft model are interrelated. Those relationships are very different from those used in the procedural approach as part of the hierarchical structure in *fig.4.9* shows. However, segmenting the shaft and isolating the relationships that are peculiar to each segment or ends of segments show that it is possible to get at and use variables such as segment length, material density and section modulus in order to compute cost properties.

We can extend that structure in all kinds of ways. For instance we can model the fact that if the dimensions or location of the gears on the lathe are changed that will affect the distribution of load and alter the engineering data on display. Warning texts can be values of labels related to constraints.

Modelling fundamental data dependencies enables us to represent information about the design object so that it can be appreciated through experiment. The dependencies reflect the essential nature of the object as we choose to observe it. Their representation does not commit us to a particular strategy for transforming or interacting with the design object; it models those aspects of the observed behaviour of the design object that we wish to take for granted. This makes it much easier to describe procedural activities associated with the design process, such as enhancing the design model, developing the design environment, or simulating the design object.

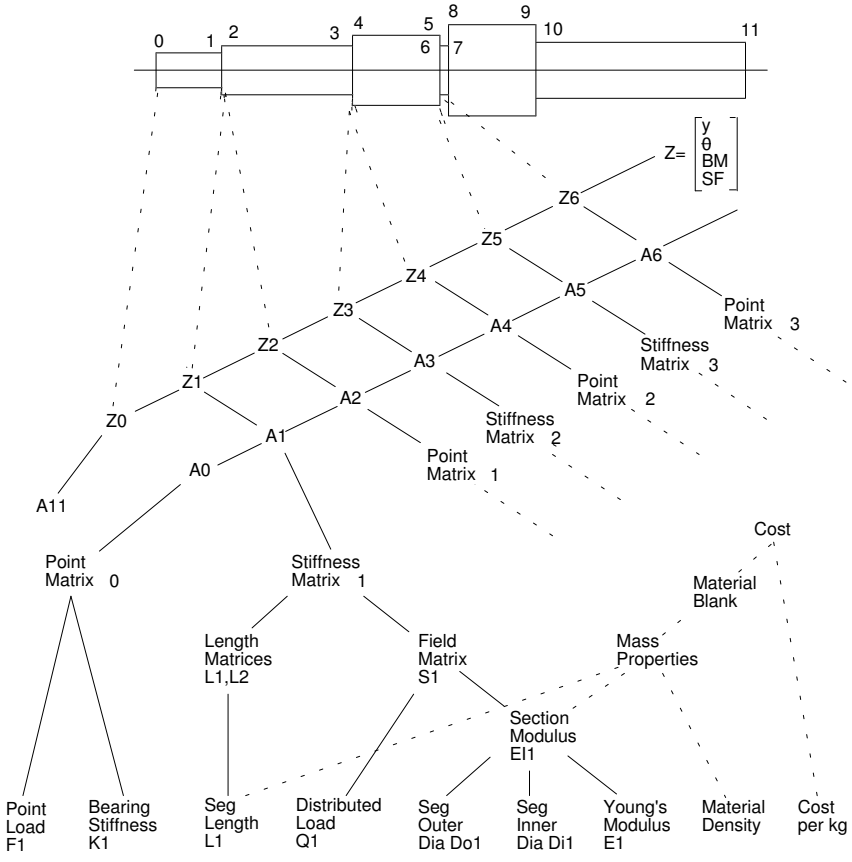


Fig. 4.9 Hierarchical Relationships in Shaft Analysis.

4.43 Implementation

The Implementation of 'shaft' was attempted in the philosophy of the ADM described in §3.42. The hierarchies described above may be defined with sets of definitions. The difficulty is implementing the way that actions can be done, as was noted in trying to construct Bow.Donald. Rather than use EDEN the idea of a *script* was developed. A script of definitions corresponds to the STORE 'D' in *fig. 3.2* as part of the ADM model. The whole shaft analysis display shown in *Fig. 4.8* is described by a script of definitions specifying relationships between windows, geometric components and textual annotations. Many kinds of interaction with the computer model correspond to simple redefinitions or extensions of the script. An important feature is that each interaction with the system leads to an incremental or wholesale change in the script: either to enrich the design model or to add functionality to the design environment. The significance of design environments will be examined in chapter 5. In this context the important feature is the ability to change the pattern of the script itself by actions.

The problem of implementing actions is apparent if we attempt to add further shaft segments. It will then be necessary to make wholesale change to the definitions that control the graphs. Sections of the script will have to be rewritten with new points, lines and shapes being declared and defined. This is a fearsome task to do manually and ought to be hidden. One way to illustrate that re-writing is to carry it out explicitly. A high level text generation program was implemented in EDEN to create the new script that is to replace that the previous state. The action in the original state triggers a call to the text generation program with the data required for the new definitions. The new script is then generated and passed to EDEN interpreter to be implemented. Since this is a form of self modifying code (using EDEN to create EDEN code that is then passed back to EDEN) the process was carried out by means of separate files that were piped in the appropriate way. This rather clumsy method showed that the principle of the ADM could work. It remained for developments described in the following chapters to see how a cleaner environment for actions might be structured.

4.44 Results

We have seen that the computer can act as an agent in the process of design by changing a script of definitions to describe a new state. It is not difficult to see that different actions in the original script can cause different scripts to be produced. In that sense a set of definitions plus actions has not only a computational state but the ability to move to an infinite number of other states according to the combination of actions that is triggered. If we now structure the

scripts themselves into the different aspects of the shaft design we can discern the following aspects.

- The topology and geometry of the shaft
- The database of relationships on the components and shaft mechanics
- The user interface: ways of accessing the prototype: mouse, keyboard, disc, etc.
- The display or output device showing some representation of the prototype

These aspects are illustrated in *fig. 4.10*.

Using the arrangement a number of trial designs illustrated the versatility of the

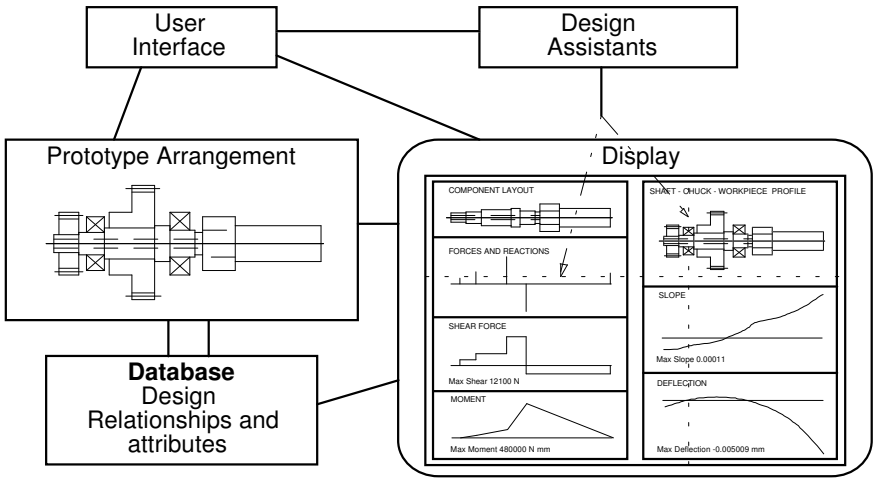


Fig. 4.10 Components of the Shaft Design

script:

- to relocate or change the dimensions of components on the shaft, redefine a parameter; the corresponding distribution of load will be recomputed automatically and engineering data updated.
- to rearrange the display, or to relate window locations, redefine the opposite corners of appropriate windows
- to take account of a through bore in the shaft introduce a definition for the inner radius and redefine the function for second moment of area
- to monitor the effect of changing bore size, set up a textual window with a warning message that is displayed if the bore exceeds a critical size and is empty otherwise; then redefine the bore.
- to introduce a sweep line or pointer on the diagram define a line or shape whose location is determined by a sweep parameter or pointer identifier. Such a line can be regarded as a "design assistant" enabling individual components of the display to be examined in more detail.

The important feature of these examples is the power of redefinition. Using that ability to reprogram on the fly enables one to consider the whole design environment as a part of the design process. As the model maker sets up the model materials, scale and environment for a physical prototype, so the designer should be able to set up an appropriate design environment in a computational model.

A issue that was explored briefly is that scripts within these groups can be considered to be independent and interactions between them would be the subject of structured interchange. A change made on the display would trigger changes on the object script. Conversely a textually based redefinition of a bore diameter would be reflected in the display of the shaft. Different parts of the design may be considered to be agents in the design process.

For further details of the work on shafts see [Cartwright & Beynon, 1992].