3

Computational Modelling for Design

In this chapter the problems of making computer models are discussed. The principal programming tools are reviewed in some detail in order to expose the difficulties of using object description, as opposed to representation of objects with state. Finally it is argued that the Definitive Method can be adapted to provide a fundamentally different approach that is inherently better suited to modelling the design process.

3.1. Object and Process Models for Design

3.11 Background

Early computer based tools for aiding design, many still in use, were developed around mathematical or logical descriptions of particular aspects of a product. 'Design' problems were then largely analytical, extracted from the design concepts and requiring only the right formulation for their solution. Interactive design was understood in terms of enabling designers to specify a series of analytical problems correctly, guiding the computer system through the various 'islands of automation'. Consider the following quotation from a CAD/CAM handbook of the early ' 80s.

> 'Interactive process design is an application of CAD/CAM to the primary manufacturing process. The designer can work with polymers, ceramics and metals in processes such as moulding, casting, extrusion, and drawing. This involves three principal elements: process physics, simulation and computer graphics.
>
> First a fundamental understanding of the process physics involved is essential. Process actions and reactions must be reduced to mathematical expressions.

> The second element, simulation, takes the mathematical representations of the component geometry or shape and combines them with the process physics in computer simulations. Interaction with the computer model allows .. optimisation, .. and parametric studies. ... A common database allows design of related tooling.
>
> Third is visualisation, allowing synthesis of information from complex pictures and so effecting vital decisions. ... Thus the designer can interactively conceptualise, design and manufacture with greater control.
>
> Analytical computer programs for stress, deformation, flow and heat transfer [need to be tailored] so that they can be linked to CAD/CAM systems for interactive process design' [Miller, *et al*, 1980]

The target alluded to in Miller's last paragraph is *integration*. And a difficult one it has proved to be. A number of mathematical models are made to solve different problems on the same design. These often give divergent partial descriptions of the design. Their contents may overlap but with different notations and mathematical foundations. Early computer models were initially associated primarily with functional properties of the design, but with progress in computer science and hardware, models have become more wide ranging, more sophisticated and more diverse *(table 3.1)*. Developments in geometrical modelling have brought pictorial description to the level where there is much talk of 'virtual realities' and indeed some of the graphics available now deserves that epithet. Nevertheless all these different models could be regarded as digital descriptions of parts of the same real-world objects. Integration has become a moving target.

- *Functional and Performance*
    - Computational methods for solving physical & mathematical relationships e.g. Finite Element Analysis
- *Relationships: intra and extra*
    - Heuristics: problem solving, optimisation methods
- *Geometrical form*
    - Graphical Processor à la Word Processor: 2D/3D draughting system
    - Geometrical model creation and manipulation: solid modeller
    - Geometrical descriptive models: APT, PADL2
    - Feature editors
    - Shape grammars
- *Appearance and Attributes*
    - Computer graphics
    - Data bases

Table 3.1 Examples of Object Description Software

Consider for example the design of a single arm robot with upper and lower arm and central axis rotation. Here geometrical models are needed of the form of the components and assembly. Various analytical areas can be dealt with using rather unrelated models - kinematics, control system, finite element analysis. The transformation of one model to another is therefore difficult or impossible. In the thinking of the ' 80s, getting information from one computer program to another was a matter of having intermediate or common forms that would be in a neutral format to enable transfer.

Common or exchange formats operate at different levels. The lowest level is the bitmap: a list of the actual bits composing the pixels in a picture generated on a raster, carrying information about colour and intensity in one or two bytes per pixel. That format is expensive on memory. A simple compacting method is to replace identical pixels in a raster row by a single pixel reference plus a byte to list the number of repeats. Bitmaps are the exchange formats used for transferring pictures to word processors for example (*e.g.* TIFF). A more sophisticated data transfer technique is to code graphic elements to make up line drawings and, more recently, solid and boundary represented geometry. That was used in the data exchange formats such as Initial Graphics Exchange Specification (IGES), Experimental Boundary File, and proprietary AutoCAD format (DXF). A third level of commonalty is the actual software used to generate graphic and other elements. One way to do that is to have graphics libraries that cover common elements in a way that is independent of computer hardware. Examples of such software are the Graphics Kernel System and the Programmer' s Hierarchical Graphics Standard. Despite valiant efforts to push these as international standards, they are unpopular with commercial vendors because of speed problems. IGES is the most popular for CAD although valiant attempts are being made towards an ISO standard in STEP. In other areas the bitmap variants and the developments in X-windows graphics both point to a growing popularity of object oriented program objects.

Intermediate forms remain essential in a market of so many different proprietary software houses. But they do not tackle the basic requirement of seamless integration. An obvious alternative is for one vendor to combine a complete set of CAD/CAM software such as that by Computer Vision, or by SDRC (I-DEAS). That is not always satisfactory because data transfer is still implicit even if it is not seen by the user. It is not integration: simply a common and better user interface for the separate packages lurking behind the screen.

But a more serious charge can be brought. Intermediate forms simply record computer based information, most of which is not actually part of the physical object but is needed to reconstruct the representation. Examination of an IGES file may disclose the current representation of the design - it gives few clues as to where the design came from, or the designer' s plan for its development. Integration really requires a more fundamental approach, bringing computer-based intelligence into the Design System. That is what has prompted the international efforts in research into interactive, integrated and intelligent CAD systems (prompting the name 'III-CAD' of the Dutch group of Akman, ten Hagen and Veerkamp).

3.12 Intelligent Integration of Models

[Tomiyama, 1989*a*] discusses the principle of intelligent integration. He introduces the idea of *meta-model* by which is comprehended those properties of models that are independent of what is modelled. Defining a *model* as a *theory-based* set of descriptions about the object world, he suggests that *modelling* is the process whereby object facts are filtered by the theory to formulate a 'world that is complete in terms of the theory'. It is important therefore to identify what is common to different models of the same system. He reduces that to a single meta-modelling precept: that all models of a given 'world' should relate to one another. An intelligent system built on that precept means that all descriptions of the design product are interrelated and built on a single set of information.

The construction of models based on a common view of the physical world requires more of a "real-world view" of design. That is itself a matter for conceptual design, and one is faced with the issue of 'content' d iscussed in chapter 2. (It also presupposes that we know what the "real-world view" actually is - a problem recently being tackled in terms of what is known as naive physics [*c.f.* Akman & ten Hagen, 1989].) The problem to be programmed has to be refined such that it not only isolates those characteristics that are amenable to examination but also allows for alteration and development of the evolving design. In other words not only must the models describe the objects, the system must deal with the design process itself. Attempts to use computer models themselves as models of the design process or to provide a computer based design support system raise philosophical issues relating to 'modelling' modelling systems, or 'meta-modelling' in an even broader context than Tomiyama' s.*Fig. 3.1*, (overleaf) picks up the robot example discussed above, but with some significant additions. ‘Content’ in the real world is larger than can ever be contained by any theory. Some possible items of ‘content’ are indicated in *fig. 3.1.* For example the problems arising from the breakage of a designed object are rarely considered at the design stage. (A classic example of that omission occurred on a machine tool gear-box designed with a shear-pin that was supposed to shear when the load was too great. Indeed it did, but the pieces of the pin fell into other moving elements of the gear-box with catastrophic results! Similarly a British Gas maintenance survey found 50% of call-outs were directly attributable to faults that could have been foreseen and dealt with at the design stage). Other designations on *fig. 3.1* relate to the lay person' s mental pictures of robots, fed mainly by science fiction. Although such pictures seem inconsequential it is interesting that on a number of occasions fiction has influenced design.

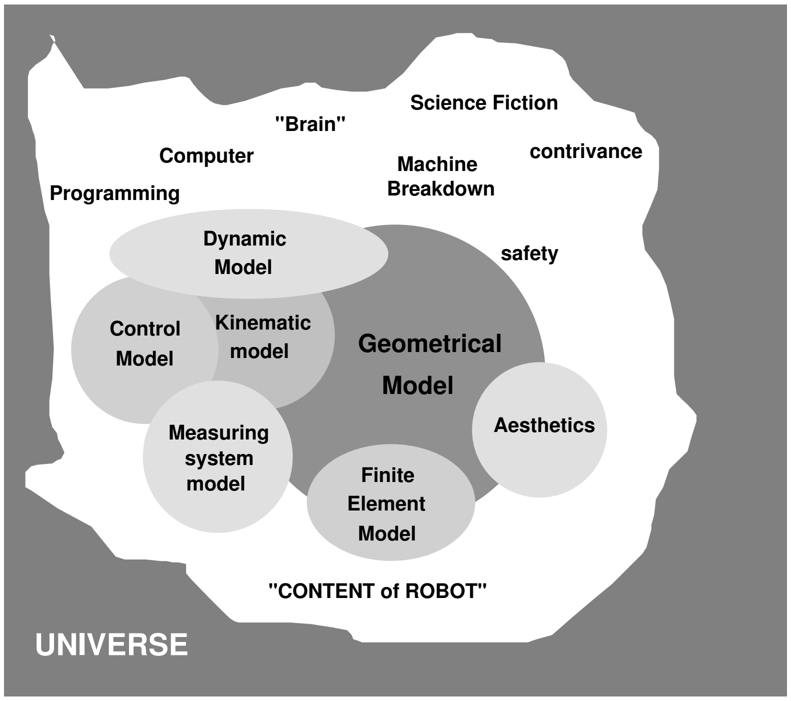

Fig 3.1 Aspects of Robot Design

One way out of the difficulty is to construct an artificial world, large but limited, basing meta-models on that universe of discourse, for example using a large number of shape features and a feature editor. Design support systems based on such ideas have been developed and form environments for design that enable track to be kept of both the process and the current stage of description of objects being designed. The discussion of those systems is deferred to chapter 4. Meanwhile we return to the fundamental problem of ‘content’. Can one make a computer environment that contains observations about reality on which one can experiment, like a physical prototype? Alternatively, can one construct a computer model that contains Platonic Forms is such a way that discoveries about the model will map into the physical world? At a practical level, in [Tomiyama and ten Hagen, 1987], the authors analyse the problem like this: an Intelligent CAD system should not only provide a medium for expressing, interrogating and modifying the current state of the model, it must represent and process *intensional* information (that which captures the functional requirements in the form of relations and other abstract notions). In the standard CAD model intensional information is not recorded, the designer may only represent the functional attributes of the product by selections from a pre-programmed set of preconceived operations. Progressing the design relies upon the designer being able to compare the current (*extensional*) computer model with the designer' s intention and modifying the model until a satisfactory match is reached.

A confusion in modelling the modifying process has to do with the idea of ‘state’: a slippery relative of ‘content’ and equally difficult to capture in a computer system. It seems that rather than constructing massive edifices to model design and design state we need to go even further back than Tomiyama in our thinking. His valuable contribution points up the problem; his answer was to look to particular computer programming methods to construct a viable solution. Before we examine his solution it is necessary to look at computer programming generally to see how state and state change can be dealt with.

3.2 State and Computer Programming

According to [Abelson *et al,* 1985] state may be defined as follows.

> "An object is said to have state if its behaviour is influenced by its history. We can characterise an object' s state by state variables, which among them maintain enough information about history to determine the object' s current behaviour. "

Many Computer Scientists, particularly functional programmers, argue that state need not exist in computing. The contention is that computer models may be constructed using computer programming methods that are built on ‘logical" principles: principles tied ultimately to the Turing machine, the basis of all modern computers. Before examining computer programming languages, it is worth a digression to examine the point about the Turing machine and statelessness.

3.21 Automata

A Turing Machine is a finite state machine in which a transition prints a symbol on a tape. The tape head may move in either direction, allowing the machine to read and manipulate the input as many times as desired. A brief description of Turing Machines is quoted from [Sudkamp, 1988].

> 'The Turing Machine is abstractly a quintuple $M = (Q, \Sigma, \Gamma, \delta, q_o)$ where Q is the finite set of states, Γ is a finite set called the tape alphabet containing a special symbol B that represents a Blank, Σ is a subset of Γ without the blank called the input alphabet, δ is a partial function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$ and $q_o \in Q$ is a distinguished state called the start state. A machine configuration consists of the state, the tape, and the position of the tape head. At any step in the computation only a finite segment of the tape is non-blank. If a configuration is denoted by uq_ivB where uv is the string of items on the tape from left to right then uq_ivB indicates the machine is in state q_i scanning the first symbol of v. This representation of machine configurations can be used to trace the computations of a Turing Machine. [*op cit. section* 9.1]
>
> A sequence of elementary transitions represents a computation. Computations read and manipulate the symbols on the tape. The result of a computation can be defined in terms of the state in which computation terminates or the configuration of the tape at the end of the computation. [9.2]
>
> A Turing machine that computes a *function* has two distinguished states: initial state q_o and final state q_f. A computation begins with a transition from q_o that positions the head over the beginning of the input string. State q_o is never re-entered. Its sole purpose is to initiate computation. All computations that terminate do so in q_f. Upon termination the value of the function is written to tape. [12.1]

Even from that very brief introduction it appears that if we can translate any problem into a finite set of symbols to form the input string to a Turing machine it is possible to formulate that problem with only one end state. Essentially that system is stateless, since only one outcome is allowed. Now a hypothesis known as the Church-Turing Thesis ensures the success of the Turing machine when it is modelling 'any problem that is computable' by coding from a finite set of symbols: whether those symbols be {0,1} or the ASCII set. So it is possible to formulate any problem based on mathematical foundations in terms of stateless abstractions.

We turn now to examine different types of programming languages and how they relate to state. *Fig 3.2* shows a very approximate taxonomy of the computer language types and examples of those types that are considered in this chapter. The hierarchy is intended to indicate the level of complexity and the implementational dependencies of the languages. Generally the more complex languages may be implemented in either functional or procedural languages.

	Automata	
	Turing Machine	
Procedural	**Declarative**	
	Functional	**Logic**
FORTRAN	Miranda	Prolog
Pascal	Lisp (Scheme)	
C		
Expert Systems	**Object Oriented**	**Definitive**
Feature editors	SmallTalk	EDEN
		DoNaLD
		EdenLisp

Fig 3.2 Some examples of Programming Languages
Levels denote increasing complexity

3.22 Functional Programming

Functional Programming languages (such as *Miranda* [Turner, 1987] and subsets of *Lisp* such as *Scheme* [Abelson & Sussman, 1985]) provide tools that emphasise the 'stateless' nature of computing. In those languages there are no changes because there is no concept of variables changing values.

> 'The question of whether the use of a Name in an Expression means the Name itself or the value to which the Name refers is meaningful in a language that has named memory cells (*i.e.* variables) because the value in a memory cell might change. In a [functional] language with no memory cells the result is the same whether we define new Names to replace Expressions or replace Names by the Expressions to which they refer. This is known as "referential transparency".' [Glaser, Hankin & Till, 1984]

The virtue of functional languages is the lack of explicit sequence of control flow in the program, which relieves the user of the burden of specifying the control flow. All that is required in order to specify a functional programming (FP) system, according to [Backus, 1978], is to specify the following sets.

- Atoms (*e.g.* digits, characters)
- Objects, derived from Atoms (*e.g.* `<YES, NO, 123>`)
- Primitive Functions over Objects (*e.g.* `null, id, eq, >, <,transpose`)
- Combining Forms (*e.g.* `composition, choice, insertion`)
- Definable Functions derived from the Combining Forms,

These sets should suffice to specify a problem completely, but unless there is some way of remembering Definitions the problem fixes the choice of primitive functions and Combining Forms. So the FP domain is extended to include Applications, in order that a function can be applied to an atom explicitly. That then raises a further difficulty: how do we remember Applications? A facility is required for allowing one to fetch a function from a library by giving the name by which the library would know that function; one would also need to be able store such functions in the library. However in accessing the library an output may not be solely dependent upon its inputs (e.g. time of arrival of inputs may be delayed indefinitely). That addition of non-determinism appears to invalidate the property of referential transparency. It is the issue John Backus called "history sensitivity" and is a serious problem in functional programming, for it also suggests state in a stateless system. One suggestion, according to Glaser *et al,* [*op cit.*], is to introduce special non-deterministic operators to deal with the particular issue they call "fairness" whereby if several inputs potentially demand infinite resources none of the objects get held up waiting for resources: there is sharing of some kind. The problem of state remains an area of research in pure functional programming.

3.23 Logic Programming

An alternative approach in the same declarative mould as functional programming (FP) is Logic or Relational Programming (RP), of which Prolog is the most popular instance. The fundamental difference is that whereas FP functions are many-to-one, RP specifies many-to-many transforms. In RP there is a set of solutions to a particular application rather than a single solution produced from a function application. So the statements

```
Joe is the brother of Jack
Jack is the brother of Fred
Joe is the brother of Fred
```

allows Joe to be mapped to two different siblings; that would not be allowed in FP.

Logic programming treats relations without regard to direction - *i.e.* which is computed from which. Programming is driven by queries about relations. The user supplies the facts and rules, the language used deduction to compute answers to queries. Thus we can write

algorithm = logic + control

'Logic' refers to the facts and rules for the algorithms, 'control' to how the algorithm can be implemented by applying the rules in a particular order. The user provides the logic, the language the control. Control is characterised by two decisions: goal order - choose the left most subgoal - and rule order - select the first applicable rule.

Both FP and RP are non-procedural and involve programming without side effects. (A side effect is programmed if some element of the system gets altered in the course of running a function but is not itself the returned value of the function; e.g. a global variable gets altered or a screen change is made). In practice it is difficult to make a pure declarative language of practical use with-out some side effect. The very acts of interaction with a screen, printer or disc are really side effects. The 'value' of `print x` is undefined, although its side effect is not! In [Clocksin & Mellish, 1987] the issue is mentioned in passing in the discussion of "built-in predicates". Those facilities have side effects. "Satisfying a goal involving it [a built-in predicate] may cause changes apart from the instantiation of the arguments. This of course [*sic*] cannot happen with a predicate defined in pure Prolog". One fundamental built-in predicate that runs through the whole of Prolog logic is the *cut*, a pragmatic programming action to break infinite loops and chop useless searches in logic. The programmer's action inserting a cut is interesting - is it a state-based action?

The point seems to be that 'state' links computational logic and functional programming to the real world. It implies that computation cannot usefully exist without some reference to state. That accords with our earlier philosophical discussion that objects in the physical world 'exist' not just abstractly but also according to perception, and that perception is influenced by its history.

3.24 Procedural Programming

We introduce this section by endorsing prefatory remarks in [Abelson & Sussman, 1985]

> 'The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called *procedural epistemology* - the study of the structure of knowledge from the imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of "what is". Computation provides a framework for dealing with notions of "how to". '

Procedural programming can be thought of as writing a recipe for solving a problem. Where it differs from pure functional programming is that variables can be assigned values and those values get changed during the program. From that point of view the program can be said to have state. However the nature of that state is not so intimately bound up with the state of the problem being programmed that any instantaneous state of the computation tells something about the state of the model.

Consider a common procedural programming language such as Pascal or C: a procedure is written as an algorithm that produces required outputs from a given input set. If the procedure is halted during execution and internal variables examined, their instantaneous values do not necessarily have any relation with the final result. For example, suppose there is a loop programmed in Pascal in the form

for *i* :=1 **to** 100 **do begin ... end;**

At a point of interruption the value of integer *i* is of interest purely to see how many times the algorithm has looped. The internal 'state' of *i* in the procedure is irrelevant as a state of the output. (Similar observations can be made about data storage references such as addressing array elements.) Procedural methods at this level simply provide a recipe for describing a resulting state that then has 'meaning'.

A further difficulty comes with any slight change in the real system being modelled. Often a new abstraction is required. Detailed examination of a problem frequently means rewriting the system, something disapproved of by commercial vendors of software. What is generally done is to have a static (stateless) system that is capable of describing states via data, and standard or preconceived ways of manipulating that data. In relation to design we shall see in chapter 5 that many established design support systems are programmed employing computer methods of that type. That means that "experiment" is not really part of the programmed system. The programmer of the system may be able set up environments in which experiment may be carried out, but those experiments have to some extent to be anticipated in order for the environment to allow them. For the shortcomings of that approach one merely needs to glance at the huge market in "enhancements" and "customisations" that are parasitic on most popular software.

3.3 State in higher level Programming

3.31 Data Modelling and Rule Based Systems

One approach to the difficulty of anticipating user requirements is to increase the size of the 'problem universe' by intelligent investigation of user requirements. The question then is what limits apply to that increase. There are two independent constraints that can be applied to obtain computable sets from descriptions of objects and their infinite number of possible states, namely *time to compute* and *space* (memory or storage). Even if certain algorithmic descriptions are theoretically computable, in practice many turn out to require either time or space which for practical purposes is too large to be feasible. If we place restrictions we can define an important class of representations that are both time-feasible and space feasible, and solutions to problems in those categories can be implemented by trading off storage against time. A useful restriction is to specify a domain by knowledge type. We can group and store knowledge according to methods that humans use, *i.e.* by rules of thumb: we must then store the methods of structuring the domain too. Programs of that type are called expert systems. The rules of thumb are known as *heuristics*: rules for structuring based upon human percepts, such as those suggested by the structure of Minsky' s Frames.

[Michie, 1986] examines the knowledge content of expert programs. He shows that in a space and time limited computer system the amount of knowledge depends upon what other things must be stored and/or use computational time. At its simplest, knowledge may be thought of as a finite function $f{:}X \rightarrow Y$, i.e. of a look-up table of pairs of the form $(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots, (x_n, y_n)$ where N is the size of the domain of *f*. Each pair represents the smallest possible transition that adds to the knowledge store. If we structure the transitions by heuristic rules, patterns, descriptions, etc. then those structures Michie calls advice. His figure, reproduced as *fig. 3.3,* shows how the increase of advice and the need for a control program effectively reduces the knowledge content in a fixed store. This is the fundamental problem for expert systems.

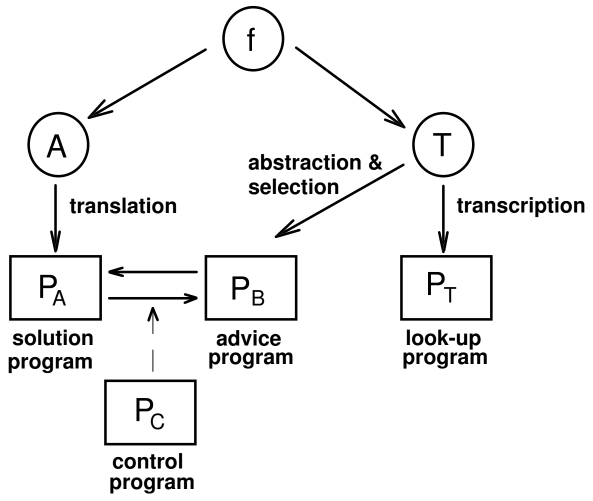

Fig 3.3 Relationships among various kinds of mathematical and computational objects. Abstract objects are ringed, concrete are boxed. A and T denote two contrasted abstract representations of the function f, namely as an evaluation algorithm and as a function table (ordered set of pairs), respectively.

[Reproduced from Michie, 1986, Ch. 18]

In [Koegal, 1989] this point is endorsed: 'Although expert systems have been developed to perform design in specific domains, such systems are difficult to adapt to other domains.'. The way round that suggested by Koegal is to make the advice section structure the domain in hierarchical chunks: abstracting at a higher level, *e.g.* to separate 'planning' from 'planning this'. Thus a design gets developed in stages of decreasing abstraction.

'Advice' in the context of object or real world description consists of heuristics structuring object and state descriptions, but not actually creating objects with state. A system must therefore be 'content' limited, using content in the form described earlier. Indeed Koegal' s major point is that expert systems are at their best when the form of the design is already in the designer' s mind*[op cit.]*. If *form* can be delineated then the task becomes that of abstraction on a data or knowledge base. Initially much work was done implementing specific domains for helping design, mainly using declarative programming, which is a form well suited to formal rule-based systems. Implementation meant providing a suitable vehicle for structuring large data bases that form the backbone of editors for aiding design. [Dym, 1987] provides a useful introduction to the problems of engineering applications of expert systems. Shape grammars [*e.g.* Leyton, 1988] have helped in the development of shape editors that are now big business, usually called Feature Editors, implemented to reduce input time in repetitive modelling activities. Examples of feature editors are many. [Gu, El Maraghy & Hamid, 1989] give a useful introduction with a good reference list as well as an example of the genre.

Using form features many common geometrical shapes may be generated and placed in libraries together with appropriate structuring heuristics, rather like a thesaurus in word processing. Whilst it is easy to decry the use of shape grammars and feature editors as "trying to write an essay using only a thesaurus and a dictionary" there is a large set of routine design tasks that, like ente ring data base material, is fairly standard. However as a way of modelling objects with state the approach is clumsy and requires sophisticated tools for adding to the 'advice' box. And advice that is useless is worse than useless as it displaces useful knowledge in a finite system.

In more recent work using rule-based methods modelling the modelling task itself forms the basis of design support tools. [Krause, Vosgerau, & Yara-manoglu, 1989]. Hierarchical dissociation is a feature of most design tasks so product data models with suitable abstractions of structure will enable the delineation of the form of the design to be systematised. The Leeds Structure Editor is an example of that approach, where a generalised software tool is used to create product data models with semantic data model characteristics [McKay, 1988, Shaw, Bloor & de Pennington, 1989]. Examples are quoted there of characteristics that are fundamental to semantic data models: unstructured objects, relationships, abstractions (under which is listed classification, generalisation, aggregation and association), networks of hierarchies, derivation/inheritance, editing of constraints and dynamic modelling. The Leeds work, along with similar rule based systems deal with static data models. Dynamics and the problems of state transition are problems still.

3.32 Object Oriented Programming

One way to characterise an object state is by having one or more *state variables* to maintain enough information to determine the current behaviour of the object. If those variables can be *encapsulated* into a procedure then we have procedures that have history, procedures with state: a computational object that has its own state variables that can change over time, *i.e.* as the program runs. To do that in a declarative language we need to introduce an assignment operator that enables us to change the value associated with a name. (For example in Lisp we have the assignment operator *setq* that a 'pure' functional version of Lisp would not have.) Associating the assignment operator with local variables in a procedure allows us to create computational objects with local state. Effectively each time a procedure is run it changes its state. For example a procedure modelling a bank balance simply subtracts an amount withdrawn from the current value in the account. The procedure alters its own internal variables, *i.e.* it is self modifying. Those variables are not generally accessible from outside the procedure. That is the principle of encapsulation. Methods for accessing the variables have to be specially written into the procedure.

Computational objects form the basis for the higher level languages called *Object Oriented Programming* (OOP). In this style of programming, objects are created that have local state because they have *encapsulated* or hidden variables. Neither those variables nor their values are directly accessible by other objects or procedures. At its simplest such an object may consist of a single procedure but they usually have associated other properties called "methods" described below. Computational objects may be used to model rather than simply describe physical objects. An abstract object group such as ‘circle' may be defined generically in terms of user-defined general words, and general properties of the ‘class' of objects specified by the user. At a second level instances of objects may be given attributes that are more particular (e.g. circleA of radius 4 mm). A class of object is associated with a specific set of operations called ‘methods' . Methods are procedures that have well-defined inputs and outputs. The way that the Methods are written is irrelevant - indeed the user is not intended to know how they are written. Each item of data within a program is regarded as an attribute of some object and only accessed by invoking one of the methods defined for the class of that object.

OOP languages have been developing since the introduction of Simula 67. The seminal OOP language is Smalltalk-80, [Goldberg & Robson, 1983]. Since that time most popular computer languages have developed the form, including C++, ADA, Modula-2, and the declarative language Lisp.

Information hiding (encapsulation) is a crucial aspect of OOP. If a Method has to compute the value of a useful physical variable within its algorithm in order to complete its output, that intermediately computed variable is inaccessible to the user. (For example a method for a generic beam-in-bending class may involve calculating a section modulus; but if it is not a ‘message' that section modulus cannot be obtained from the ‘method'). However, if information hiding is essential to OOP, it also proves to be a limitation. Tomiyama identifies the problem of extending its application where there are objects that share a common connection.

'Let us consider the design of a robot. In Smalltalk-80 it is reasonably natural to say;

- a_Robot is an instance of the class Robot
- a_Robot has 6 arms, arm1 to arm6, as instance variables.
- arm1 is an instance of the class Arm
- arm has instance variables, end_Point1, end_Point2, length, transformation Matrix, *etc.*
- end_Point1 is a_Point which has x-, y- and z-coordinate as instance variables.

Problem

How do we define the fact that end_Point1 of arm2 is the same as end_Point2 of arm1? Smalltalk-80 does not allow sharing instance variables among two objects because of information hiding.' [Tomiyama, 1989]

Despite the problems described here some very profitable work on design environment has been done and those will be discussed in chapter 5. Basically the methods create artificial environments by storage of large amounts of information and relationships on that data that are based upon observation by the users

3.4 A New Programming Paradigm for State

3.41 Background to Definitive Notations

The issues that have been discussed in this chapter relate to the computational context of this thesis. It has been shown that while state and state change are crucial to design, they are difficult to have using conventional programming methods. Definitive methods on the other hand have distinct properties of state that provide great promise in application to design.

Early work on definitive notations was orientated towards "the state of the interaction". This is in contrast to traditional procedural and declarative programming methods that were developed originally for interacting with the computer in a batch mode. In the latter methods, acts of input and output were simply regarded as necessary evils to get into the stateless mode described above.

To exemplify these properties two early definitive notations, called ARCA and DoNaLD, are reviewed. The first is named after Arthur Cayley and is for animating Cayley Diagrams. [Beynon, 1986]. Graphs and similar constructs such as circuit diagrams and control system block diagrams carry a content that is a function of the incidences on a diagram, but can only be inspected after the diagram is constructed. The visual image has little value without the underlying conceptual model, so it is necessary for the user to be able to specify the models underlying the images systematically and simply. In such a task interactions are highly important and need to be mediated through both graphical and textual interfaces. The text screen (or window) is used to develop the program code, whilst the graphical display shows the current state of the model. The code consists of a sequence of definitions. A definitive notation includes variables that denote implicitly or explicitly defined values in the underlying algebra. Values of variables are determined by definitions, each of which either assigns a formula or a specific value to a variable. Circular or recursive definitions are trapped as semantic errors.

To the user a definitive notation appears similar to a spread-sheet, without the cell-based interface normal to spread-sheets. Actions are essentially a dialogue with the user consisting of declaration of variables (if variables are typed), definition or redefinition of a variable, and evaluation of a variable. Writing a definition is like putting a value or formula in a cell of a spreadsheet. Redefinition simply over-writes the old definition in that cell. Another important similarity is the way that computation proceeds to update the system after a definition is input. Computation that takes place with a definitive notation has the following properties.

a) It is hidden from the user,
b) It occurs after every definition and updates values of variables in every previous definition affected by the latest (current) definition and
c) It can be carried out in any order whilst updating previous definitions.

An interesting feature of definitive methods is that it is not ‘pure’ (purely procedural or declarative). In ARCA, declarative features (in the form of constraints) and procedural forms (means of updating coordinate and incidence information) are mixed. This is a characteristic which the method shares with a number of programming forms such as Sketchpad and PopLog. The advantages of that are great and have been increasingly exploited as the method has been developed.

DoNaLD is a geometrical implementation of definitive methods: a 2D line drawing notation [Beynon, Angier, Bissell & Hunt, 1986]. It is a typed notation. The underlying algebra is based upon **real, integer, point, line** and **shape** variables, where **point** values are pairs representing Cartesian or Polar forms and **line** values are line segments in a plane. A **shape** value is a line drawing consisting of a set of lines and points. Objects designed in DoNaLD consist of shapes defined by a set of definitions. Its power lies in the ability to construct abstract shapes since values of variables can be specified by algebraic expressions. So if the user inputs particular values as input to those expressions the realisation of the shape reflects those assignments. An example of DoNaLD script [*ibid*] follows.

```
openshape cabinet
within cabinet {
        int    width, length        # declaration of variables
        point  NW, NE, SW, SE
        line   N, S, E, W

        N = [NW,NE]                        # abstract definitions
        S = [SW,SE]
        E = [NE,SE]
        W = [NW,SW]

        width, length = 300, 300           # value assignment

        SW = {100,100}                     # abstract definitions
        SE = SW + {width,0}
        NW = SW + {0,length}
        NE = NW + {width,0}
}
```

In the example a simple square is defined in terms of points for corners and line segments for edges. Data types are interpreted as indicated. Variables defined within **openshape** are deemed to be local. We thus have an object with local state defined in terms of the values of length and width. Since both are defined in the example the square computes and the object appears on the graphical interface. If either `width` or `length` are redefined then the square recomputes and a new object replaces the old. Notice that although a square is actually drawn it has a general object label `cabinet`. Within `cabinet` other attributes may be added, so `cabinet` gives the underlying definition of the object - the graphics are simply the representation.

3.42 State in Definitive Notations

How then can the definitive method be applied to design? A primary issue has to do with state. A family (or *script*) of definitions shows that the state of interaction is linked with the concepts of *object state* and *process state* discussed above. Indeed we have a method that captures computational state. As such it can be a vehicle for representing physical state. A script of definitions establishes relationships independently of the data required to define a particular state fully (provided there are no duplicate or cyclic definitions). Thus the script has *abstract* state rather like OOP objects. Unlike OOP, state variables are not encapsulated, rather they are a visible part of the user' s interaction. Changing variables (modelling state transition) is not done by message passing but directly by redefinition.

Redefinition is like trying something out in design - trying an alternative approach to an already tried scheme. It is something that is typical of user interaction. However the designer wants more than to be the sole source of redefinition. When a particular action is taken the user may want to be able to see the result in different ways. We have already seen above the role of side-effect in displaying on screen the state of the dialogue. The representation shown on the screen may not be necessary for the computational model. It may only be like a photograph, a snapshot of particular aspects of the model. If the model changes then the role of the computer in updating the representation on screen is in a sense independent of the user. The screen display should not only be able to reflect the new state but also to present that new state in the most advantageous manner. Thus traditional user-interface management issues come to mind: the computer should be the source of dialogue though appropriate use of windows, menus, graphical displays and the use of analogue rather than textual input, [Foley]. Of equal importance are issues such as monitoring and maintenance of constraints, an activity in which the computer must itself participate in changing the state of the dialogue. Those desires imply that the computer should not have the passive role of simply responding to user redefinition, it should have a more general framework.

In understanding how the computer can have a more active role in redefinition, the idea of scripts being computational objects is a useful device. A definitive script may be thought of as an object that gets changed either by the user or by redefinition by the computer. It would therefore be best if the same method can be used regardless of the source of the redefinitions. The method proposed is based upon an Abstract Definitive Machine model (ADM) described in *Fig.3.4.* The model consists of an overall *Program store P* containing *Entities* similar to objects in OOP. Each entity has two stores: *Store D* of *variable definitions*, and *Store A*, which contains *Actions.*

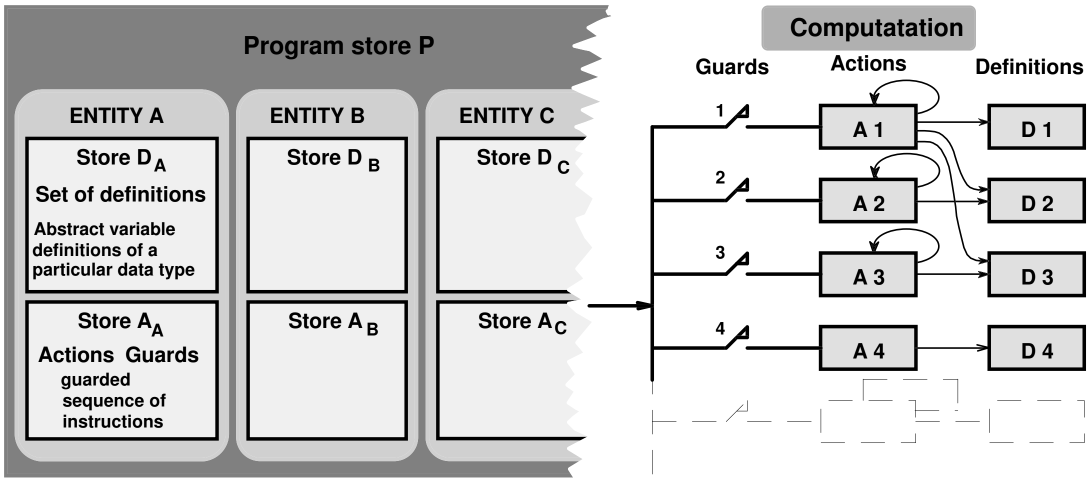

Fig 3.4 Abstract Definitive Machine Model for Object Definition

The definitions in store D are sets of definitions similar to those in the DoNaLD example - namely variable declarations of appropriate types and definitions consisting mainly of abstract formulae, but may also include assignments of values to variables. The new addition to the definitive notation is the idea of an *action*. An action is a sequence of instructions that may do one or more of the following

- redefine a variable
- introduce a block of new definitions sitting somewhere else outside the current entity
- delete a block of one or more definitions

Actions affect the contents of Store D and Store A. The triggering of actions has to be carefully guarded to prevent infinite loops and actions interfering with one another. These guards may be regarded as Boolean switches, shown as such in the right hand of *fig.3.4.* A computation consists of a sequence of parallel executions of appropriate actions. All the guards on the actions are evaluated and those that evaluate to *true* allow the actions behind them to occur. The effect of an action described above is to modify the contents of either or both of stores D and A within an entity. Actions may also affect other stores (as indicated by the dotted output of A1 in the right hand diagram).

The Abstract Definitive Machine (ADM) model provides a framework that retains the characteristic features of definitive programming, *viz.* the representation of the current state of the user-computer interaction by means of a system of variable definitions, but allows both the user and the computer to initiate dialogue actions to change the state.

These important issues in the design and implementation of CAD systems are discussed further in [Beynon & Cartwright, 1989].

The abstract machine model is not itself a programming language or notation: it is just an idea for structuring a definitive programming system. It does have features relating to the state of objects that are similar to many described in this chapter. However some clarification of the abstract machine is required as a step towards the longer term objective of developing a CAD support system. In particular the way that actions operate is not at all clear, and the problems of constraint maintenance may prove to be damaging in an engineering context. After all if one has many agents acting in a system constraints could easily be circumvented, modified or be just too complex to be implemented in a reasonable time frame.

In the next chapter we review some programming methods and design support systems used in the engineering scene that are definition-based before embarking on the application of definitive methods to geometrical modelling and interactive design.