5

A Design Prototyping System

5.1 Design Process Support

5.11 Specification

In the consideration of the design process the main criticism levelled against computational support for conceptual design is that ‘content’ outstrips form, so any preconceived structure on design tends to over-constrain that part of the design process. In an ideal support system the designer must be free not only to create the product in mind in as free a way as possible but also to be able to tailor the working environment similarly. Veerkamp, in probably the best exposition of these requirements, makes clear the relationship between the designer and the design, emphasising that the designer must be able to exercise his (*sic*) faculties.

> ‘I believe that the essential thing in designing is that the designer creates his own design environment and the system must give him the freedom to do so. The system must understand the designer' s commands and translate them into system tasks.
>
> ...
>
> The designer and not the system determines the way the design process is directed.
>
> The user interface must allow the designer to express his ideas in his own terminology" [Veerkamp, 1992]

Some reasons for needing control of the working environment were mentioned in the last chapter. The designer interacts with the system in order to change the state of the design. However if the system merely records descriptions of the design then a significant change in the design may involve considerable adjustment to that description. If the adjustment must be done explicitly by the user there is a

serious risk of error, but if it is looked after by the computer constraints may be invoked that are functions of the support system. We have already seen how geometrical modellers are constrained in that way.

So what do we want in a support system? [Tomiyama and ten Hagen, 1987] and others say that a design support system should be:

1. a place to describe what designers have in mind
2. a tool to verify the designer' s ideas in terms of feasibility, cost, performance, *etc*.
3. a system to store and retrieve design information and to transform it amongst various subsystems.

I would put the first of these rather differently in the light of the forgoing discussion. The system would be more useful if it not only described the object but that:

- The model of the design is as faithful to the physical world as possible
- It has relationships based upon observation and experiment on physical objects
- One is able to experiment on the model as one might do on a physical model

Tools for modelling the designer' s ideas need great flexibility. Any convenient geometrical modelling method should be allowable, preferably interchangeably and in combination (*e.g.* CSG, B-rep, wireframe). That means addressing the problem of transforming between models that may contain or be based on very different descriptions of designers' ideas

Interaction is crucial, particularly in the evolution of the design. At the design level it should enable

- *variational design :*
  given a value for a design attribute, the system should respond by
  - checking constraints
  - propagating the effect through the design
  - monitoring : to see if more changes are needed to specify the design
    to show a suitable message if a constraint is violated
    to show up conflicting constraints or design decisions.
- *design history*
  - exploring "What if ....." questions
  - able to return to an earlier design

At the user interface the screen arrangement, the input tools and the image on the display should all be definable by the user. Design assistants in the form of pointers, image tracking, moving crosshairs and so on should be definable rather than pre-conceived

Integration is usually interpreted as the ability to communicate in a consistent way with application programs or macros external to the system. In that context it is worth noting that there are two kinds of integration. Designers may embrace the development of the entire product from conception to manufacture, or they may specialise in one special phase of design for many products. Historically the latter has tended to be the practice in most countries (although, interestingly not in Sweden). [Wærn, 1986]. Where the designer is concerned with only a single phase then design support may centre on areas of specialisation, *e.g.* customisation of the CAD system, access to data bases, and linkage with analytical methods such as finite element analysis. On the other hand, integration of all product activities is becoming more common with the development of design groups or 'forward engineering' noted in section 2.5 above. With design groups more sophisticated management and development of the computer model are necessary to cope with many different agents of change. The support system should allow the concurrency of different users simultaneously interacting, or from computational agents looking after internal consistencies arising out of user actions. An integrated system also requires the ability to link different disciplines, for example electronics, chemistry, precision engineering linking downstream with manufacture, assembly and servicing.

5.12 Approaches to Process Support

Knowledge Based Systems

The historical tendency has been to construct Design support tools to aid product description rather than the design process. The design process is abstract, but knowledge about the product and its evolving description may be structured and managed in various ways; *e.g.* help can be provided to the designer in organising the thinking required; suggestions and starting points can be provided though browsing in libraries of appropriate entities. The Edinburgh Design Support System [Popplestone, 1984, Popplestone, Smithers *et al*, 1986] is an example of a knowledge based system. It uses an "Encyclopaedia" of engineering knowledge, with a method for creating a "Design Description Document" that contains knowledge appropriate to a particular design activity. Popplestone, *et al,* identified the basic requirement for retaining design knowledge in a consistent way,

particularly where a system has to operate in many disciplines as the design process continues. Process support may be assured by remembering the generated knowledge in a structured way so that later stages will be automatically constrained by earlier knowledge of the process. The difficulty here is when significant changes are requested at a late stage in the design process as fairly massive reworking will be necessary to update the relationships.

Feature Editors

In feature editors the design process is circumscribed in a preconceived way. The constraining effects of feature based systems are particularly serious when the bases of features are manufacturing methods. By definition particular manufacturing techniques can only produce a finite family of features, albeit infinitely associative. For example a feature editor based on manufacturing methods reported by [van Houten & van t' Erve, 1992] coerces the design process to be essentially design for manufacture. In [Case, 1992] the author comprehends feature technology as an integrating methodology between CAD and CAM, but he admits that the number of features that can be recognised is limited and the designer' s intent may be lost. Even so, he believes that it may be worth working within those constraints if gains in productivity are realised.

Object Oriented Programming (OOP)

Significant attempts at supporting the design process have come out of the OOP stable. The important point in the philosophy of OOP is that the user is free to model reality without having to go into the mechanics of how the model itself works (i.e. how it is programmed). That forms the background to the development of design support systems using OOP. The designer can concentrate upon the objectives of the design in hand to develop a particular product. Rossignac, *et al*, [Rossignac, Borel & Nackman 1989] identify the need for the designer to be in charge of the design process, claiming that it is up to the user to come up with an operation order that will meet the functional requirements of the required system. The system can then support the design process by providing an intelligent apprentice, able to deal with well-defined tasks encountered during that process.

Mäntylä follows the accepted notions of top-down or strategic decision-making and bottom-up detailing for the conceptual and detail design stages, [Mäntylä, 1990]. He supports the need to focus on particular aspects of the design and to capture the sequence of focus changes. He highlights the need for capturing and preserving design history and alternative elaborations from a common starting

point. Although he argues for tackling the problem of process modelling, Mäntylä' s Browsers still make the designer responsible for using the system in a structured way.

Reasoning more deeply about the design process, Veerkamp suggests that the designer thinks in a *goal-oriented* way, [Veerkamp, 1992]. He distinguishes meta-level reasoning and object-level reasoning. The former is concerned with strategic decisions on how to proceed with the design - what must be done next - and with the formulation of design goals. The latter is concerned with the current state of the design object - how to extend the model to reach the current design goal. The subdivision of such goals describing the process is a tree structure with meta-level goals as nodes and object-level goals as leaves. He concludes that it is possible to construct a descriptive model of the design process.

Veerkamp's Artefact and Design Description Language (ADDL) is designed to be interactive, working with a data-base called the fact base. The fact base contains all the literal facts currently known about the object and is extended as the design proceeds to contain more and more detailed information about the artefact, describing the structure of parts of the overall objects. The object base, a sub-set of the fact base, stores these parts as separate objects, each with its own state. The system operates on these data, via the Interpreter, with the active co-operation of the designer, via the user interface. The Interpreter uses *scenarios*. A scenario is a piece of design knowledge employed by the system to perform a design step. It consists of a set of methods and rules that query the object information state at a particular stage of the design process, depending upon the current design goal. A scenario, once activated will continue to be interpreted until it terminates. At that point the state of either the design process or the design object is updated, and a new scenario is chosen. A sequence of activated scenarios represents the design process. Back tracking allows poor design directions to be ignored and earlier stages to be points for restarting the design process.

Provision for process modelling in ADDL is made in that design objects may be modelled independently of certain contexts by means of the meta-model mechanism. Assumptions generated at the object level may be transformed to process parameters in the process information state, the method being step-wise refinement from an incomplete to a detailed description. A single design step is performed by means of a scenario. For each incomplete state of the design object a

scenario appropriate to that object is selected and executed. The design knowledge is represented as rules with a forward chaining strategy for inference

The ADDL suggested by Veerkamp give good support for the design process, but again the process is descriptive and static. Changes have to be propagated via the scenarios that encapsulate information that one may wish to access.

5.13 Agent Oriented Approaches

The agent concept is central to the way we deal with the design process. It can be interpreted in different ways according to which aspect of the design process we wish to address.

First, the components of an engineering system can themselves be regarded as agents. By analysing the relationship between them we represent knowledge about how the behaviour of the entire system is related to the components.

Second, different experts acting on the same design product are agents. An engineer may redistribute elements in a design without regard to their aesthetic appearance or cost of manufacture. A manufacturing engineer might examine features in terms of ease of manufacture without reference to the design functionality. By analysing their interrelationships we understand how to represent the design to the different participants and to identify potential conflicts between their requirements

Third, the computational elements required to construct a computer model of the engineering system and to simulate its behaviour can be regarded as agents. By representing them we both record the current state of the design and prototype the system to be constructed in software in such a way that we can perform realistic experiments on the computational model.

One approach to using agents is to consider the user as a kind of super-agent in charge of a series of autonomous computational agents. That generalises the idea of packaging independent computer programs such as the I-DEAS, CADAM, and ComputerVision systems. In commercial systems the "integration" is done by internal translation but the user is responsible for guiding the product through the separate stages. In a more agent-oriented approach the autonomous packages have specific tasks around the product design but the modelling methods are linked at a high level. The first of that type is that being developed by Tomiyama' s group in

Tokyo, based upon the ideas discussed in chapter 3 above. The latest developments of that are described in [Tomiyama, *et al,* 1994]. The core of his framework is the Metamodel system that manages the different models used by the functional modelling and external analysis programs so as to link appropriate model concepts for use in later stages. The knowledge management system also links different kinds of knowledge: catalogue, physical, contextual, abstract, assumed or observed knowledge.

There are a number of design systems being developed based on a framework of linked systems. One that explicitly explores the notion of agents is being developed at Lancaster University called Schemebuilder [Oh, Langdown, Sharpe, 1994]. The general idea of having different agents is that one may wish to carry out design tasks like simulation, component selection and layout. The principle is that of embracing a co-operative relationship between the man and machine, providing decision support that augments design creativity by guidance and suggestion rather than an expert system that tries to replace design functions. The agents are in fact a heterogeneous set of software systems (including MetaCard, based on Hypertext, a Knowledge Engineering Environment, Simulink and a CAD system) linked by a combination of Unix pipes and shared file mechanisms. The underlying philosophy follows [French, 1985] in positing that Conceptual design is a process of structured logical thinking whereas it is apparent from our earlier discussion that the process may be anything but logical. There is however some provision for browsing via hypertext that provides opportunities for experimentation.

Turning now to Agent Oriented Definitive methods the agent idea may be much more explicitly worked out using the ADM principles discussed earlier. If we link sets of scripts so that they can be modified according to the needs of the different agents who may wish to interact with the design, then we have conditions similar, for example, to the scenarios described by Veerkamp. There is an important difference though: the scripts will have linked actions according to their agent of origin. Each agent has its own script and privileges with respect to changing both its own script and those of others. The collection of all participating agents creates a system that effectively has state. The state of the interaction and possibilities of other states depend upon the actions triggered by redefinitions.

In Definitive methods the fundamental abstractions for constructing state transition models for complex interactive systems are outlined in principle in

chapter 4 above and is reported in [Beynon *et al,* 1990, and Beynon & Cartwright, 1993].

We have seen the importance of the 'experiment'. In the Abstract Definitive Machine the interaction between an agent and its environment is modelled in terms of observations, recording the agent' s view of the state of the environment. Observations are represented by variables whose values can be changed through actions by agents in the system. Observations are also of different kinds. They may be measurements that an engineer makes; they may be concerned with the behaviour of the system from the viewpoint of one of the agents. Alternatively they may be associated with computational agents and correspond to variables to which they respond or which they can conditionally change.

The design process involved in agent orientated modelling is directly analogous to that which is carried out with a physical prototype. Any changes in a prototype naturally have a knock on effect on the design. Move or lengthen a lever and the whole linkage is distorted. Change the shape of an object and its relationship with the environment (*e.g.* its wind resistance) is also affected. In simulation, define the state of the display in terms of internal variables and the display is automatically updated as they change.

The potential of agent oriented support of engineering design is greater than any previous approach because of its fundamental difference in the way that it handles state and state changes. However its practical outworking has proved to be a hard road.

5.2 Evolving a Prototyping system

5.21 Definitive Notations and EDEN

The Abstract Definitive Machine (ADM) and agent oriented programming have been slow to evolve. The idea of state and action came out of the first real Evaluator of Definitive Notations [Yung, 1987; Beynon & Yung, 1988]. EDEN is written in C under Unix and supports limited graphics, now under X-windows. EDEN allows a script of definitions to be evaluated much as a spreadsheet. More significantly it has built-in support for a definitive notation based upon list processing, and can 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.

To explain its operation we use the following dialogue with the EDEN interpreter. EDEN responses are written after the $ prompt

```
1.   x = 34
     $ 34
2.   p = writef (x)
     $ 34
3.   x = 9
     $ 9 9
```

Statements 1 and 3 are definitions in the conventional sense. Statement 1 assigns the integer **34** to the variable **x** and returns **34**. 3 redefines **x** to be the integer **9** returning **9**. Statement 2, however, defines a procedure, not having a 'value' in the strict sense. Following the input of the second statement the number **34** is displayed. After the third input the second definitive statement is invoked automatically again and because the argument of **p** has changed, the number **9** is displayed, hence the two responses of EDEN.

More subtle is a procedure that is triggered by a variable different from its arguments. The statements

```
4. b = sin(0.56)
         $ 0.5311861979209
5. p = writex : b (x)
         $ 9
6        b = 12.9
         $ 12.9 9
```

cause the procedure **p** to print the value of its argument **x** only if the variable **b** is changed. So after statement 6 **b** is redefined to have the value **12.9**, but also **p** is activated and prints **9,** the value of **x**. (**b** is simply the trigger: its value is irrelevant to the procedure in 5.)

Because it is a mixed programming environment we need to be careful to decide what is meant by the 'value' of a definition. As observed above we cannot say that the value of **p** is **x** since the definition of **p** is procedural and **x** is displayed by side effect, *i.e.* it does not change anything in the programming system or data. In EDEN we separate definitions-with-values such as statements 1,3 and 4 from procedures like 2 and 5, calling the first *definitions* and the second *actions.* In the ADM we go further in defining an action. There an action may do more than side effect. It may actually cause new definitions or redefinitions to be invoked. Possible ways of doing that are described below (section 5.4)

EDEN makes it possible to link complex procedural actions and intricate systems of definitions. It is a very powerful programming paradigm but one that can prove difficult to use and analyse. One way of using the paradigm is to treat EDEN as low-level. Realistic programming tasks often require routine detailed sets of definitions, *e.g.* for graphical entities such as points and lines. These definitions can be automatically generated using a suitable translator of higher level definitions. That is the basis of the DoNaLD notation. DoNaLD statements define lines for instance by a type declaration and end points. The necessary detailed EDEN definitions and the actions to put lines on an X-windows display under Unix are complex but normally should be transparent to the user. (Unlike OOP however the information is not encapsulated. The EDEN code can be inspected. Indeed it is not difficult to write the code in EDEN ; it is simply tedious for routine work).

Several different notations that translate into EDEN have been devised by others in the research group; each designed to enable a particular specialisation. For example CADNO [Stidwell, 1989] is an attempt to represent more abstract topological ideas than DoNaLD and to produce more complex geometrical modelling tools. SCOUT is a notation that deals with the complexities of interfacing windowing systems.

By translating definitions into the EDEN interpreter one can represent the state of the dialogue over any definitive notation. The examples cited in chapter 4 illustrate how that works out in practice. However if we wish to implement the more comprehensive underlying algebra for dealing with shape representation expounded in that chapter there are practical problems. The graphical tools available under Unix are inadequate to deal with the geometrical modelling of objects described there (CSG, B-Rep and spline constructions) when compared with specialist CAD systems. Rather than re-invent the wheel I felt that the best way of proceeding with the prototype design support was to link a definitive notation with an existing CAD system. While that meant that the definitive notation would not be pure, in that it is not definitive "all the way down", EDEN itself is also not pure, being written in C and calling procedural routines for graphics and operating system actions. A bonus of linking with a CAD system is that it shows that the definitive approach is not exclusive and can easily interface with other systems.

5.22 Development of EdenLisp specification

The idea of a definitive notation interpreter linked with a CAD system was influenced not only by the constraints of the existing graphical tools but also by pragmatic problems of implementation. For an engineer the idea may be important but once accepted the gearing problem takes over. Gearing is the ratio of the time to prove the idea, to the time to get a productive system operational. In many cases ideas never get into production because gearing is too great. I wanted to show that it is feasible to implement definitive methods so that their benefits are exposed and their commercial implications can be explored. A danger in that approach is that while research in the basic idea is going on there is a strong possibility that a particular implementation will be overtaken by events. I therefore needed a medium that would be adaptable to such changes. Since EDEN is essentially list orientated and declarative in style it made sense to use a list based language linked to a CAD system that could interpret it. The medium chosen was AutoLisp®, the interpretative language of AutoCAD®. [AutoDesk, 1987, 88, 90, 92]. AutoLisp is a superset of Lisp, consisting of a Lisp interpreter that accepts the common core of Lisp expressions together with all the AutoCAD commands, so enabling the interpretation of Lisp statements to be both computational and graphical.

The name EdenLisp was attached to the new notation to link it with its pedigree. The initial specification was for a definitive notation interpreter similar to EDEN. That meant creating a program in Lisp that could take an input in the form of definitions and carry out evaluations after the manner of a spreadsheet. Since Lisp is recursive in form and centres around lists the evaluator could be structured around trees. For example the script in fig 5.1 has the tree shown on the right.

```
1.  a=b+c
2.  b=f+2
3.  c=d+e+f
4.  e=1
5.  f=3
```

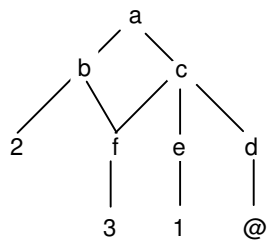


Fig 5.1 Tree form for evaluating definitions

Here **a** is related to **b** and **c**, **c** to **d**, **e**, **f** and **b** to **f** and **2**. **e** is related to **1**; **d** is undefined and given the symbol **@**. On receiving the 5th statement the evaluator would assign the value **3** to **f** and then inspect the branches of the tree above. Seeing two branches (to **b** and **c**) evaluation would proceed with each in turn: **b** would be re-evaluated by inspecting its leaves in turn. If all leaves have a value then **b** acquires a value, otherwise it keeps the undefined symbol **@**, in this case

b=5 is assigned. The same procedure applies evaluating **c**, except that if the value of **d** is undefined the value of **c** also remains undefined. Proceeding further up the tree from **b** and **c, a** acquires a new value if **b** and **c** both have values. In this case **a** remains undefined. Thus after statement 5 the values of variables **(a,b,c,d,e,f)** are respectively **(@,5,@,@,1,3)**

While EdenLisp has the EDEN engine as its core, the basic philosophy of Lisp is such that higher levels of definitions can be created by defining higher layers of functions. The underlying algebra of the notation can be implemented directly by defining appropriate Lisp functions, rather than having a translator such as DoNaLD uses. That greatly simplifies the task of exploring new ideas. However before any algebra could be implemented it was necessary to have control of the input to EdenLisp to ensure that it is in the correct form. Input statements are not Lisp code and therefore have to be correct syntactically and semantically according to EdenLisp. A lexical analyser and parser were necessary to deal with those aspects respectively. Evaluation on the other hand is carried out by the Lisp interpreter by assuming that formulae are in Lisp. The right hand side of definitions and all commands have to be in the form of Lisp statements. Whilst that is not a problem for a Lisp programmer it is somewhat clumsy to write for example **(* a b (/ (+ c d) e))** instead of the more conventional form **a*b*(c+d)/e**. An obvious addition to the parser was a translator that converted the conventional form into the Lisp form for internal computation.

The system should be able to deal with an algebra of abstract types for objects, and that requires that input is strongly typed. Typing enables us to have control of the way that an object is built up and simplifies the propagation of the computation. Lisp itself is untyped, although it will identify certain forms such as integer, real, string, atom, list, and symbol. Type declaration and checking, both on input scripts and internally generated scripts, had therefore to be part of the specification of EdenLisp. While that means that the user of EdenLisp must operate within a given set of types, anyone familiar with AutoLisp can add new types without difficulty.

The types specified for EdenLisp are intended to be those associated with the algebra outlined in chapter 4. However in its evolution EdenLisp has followed the path blazed by DoNaLD and CADNO. Emulations of those languages were constructed in order to test the system and develop the types necessary to describe topology and geometry. Models were developed in a similar way to that used in

DoNaLD with the data structure and symbol manipulation under the control of EdenLisp. The AutoCAD system is called on when it comes to the display of geometrical models. It is used in two ways: to actually display information and also to obtain display data from the AutoCAD database for use in redisplay, or to delete previously displayed but now superseded entities when a redefinition causes a display change.

The most revealing aspect of specifying a Prototyping system has been in dealing with actions. Lisp is extremely powerful in its ability to reproduce itself. Lisp code can be encapsulated within Lisp quite easily so that it is treated simply as a list. In that respect Lisp differs from languages such as C that has to use clumsy string-to-variable transforms. It makes the task of producing new scripts out of "actio ns" much more straightforward

5.3 Characteristics of EdenLisp

5.31 The Language

EdenLisp is defined syntactically by the grammar described in Appendix 1 and by lexical rules covering name construction and the use of the semi-colon and comma characters. Where alphabetic characters ("A".."Z") are allowed only uppercase are significant. They may be entered either as upper or lower case, but EdenLisp always translates them into uppercase. The following subsections describe the features of the language in an informal manner. Chapter 6 deals with the formal definitions and implementation issues.

Identifiers

EdenLisp users may specify identifier names for any variable. Valid user names are alphanumeric character strings that

1. begin with an alphabetic character ("A".."Z")
2. may contain any number of alphanumeric characters, or characters from [$,£,%,_] although names are stored in AutoLisp in batches of 6 characters, so longer names take longer to access.
3. are not identical with an AutoLisp, AutoCAD or EdenLisp linguistic terminal defined in Appendix 1. To do that will redefine those terminals with odd results. In particular the letter "T" is used in Lisp for "True" so if it gets redefined there is real trouble. (EdenLisp warns the user if a terminal name is declared)

Basic Data types

In the current EdenLisp the following data types are defined. The declaration symbols are alongside.

integer	`INT`
real	`REAL`
string	`STR`
list of integer	`LINT`
list of real	`LREAL`
list of string	`LSTR`
list of list of integer	`LLINT`
list of list of real	`LLREAL`
list of list of string	`LLSTR`
frame	`FRAME`

A further type available under Lisp is the quoted atom, parsed in EdenLisp as `QATOM`. The quoted atom is an extremely useful device. It consists of a Lisp atom (a single name or number) that is dealt with simply as an unchangeable object without being evaluated. The use of `POINT` in place of `LREAL` has been allowed and may be used interchangeably to help readability. The first three types in the table are the same as those in AutoLisp. The second three are flat lists each of which have members that are all of the appropriate type. The third three are flat lists of the respective second group. `FRAME` is a special type since it deals with lists of various types.

Statements are of three kinds:

1. *variable declaration* of the form

   `type : username`

   which sets up a variable of the given type with an undefined value or definition denoted by the cipher `@`. Further usernames of the same type can be declared by appending them to that statement with some white space between. For example

   ```
   int : a b c
   real : j k l
   str : stringA stringB
   ```

2. *definition statement* which may have of either of the forms

```
username = <expression >
username = if relational_expr then statement [else statement]
```

3. *commands* enclosed in parentheses. These may be to do with setting up object structures or they are AutoLisp functions that may invoke AutoCAD commands. For example

   `(openwin 'Swindow)`

   is similar to the DoNaLD statement: `"within { ... " .`

Expressions within definition statements may be any of the following

constant

explicit constant *e.g.* `78, "word", (5 6)`

constant arithmetic expression; *e.g.* `56.8*log(0.67)`

symbol name or list of symbol names *e.g.* `[a, b, c]`

formula

algebraic *e.g.* `a*b/8`

defined function *e.g.* `Projn[LineA, 2]`

geometrical function

a function that actually represents geometry that may be displayed

Scripts are sequences of definition statements. Whilst a single definition statement may by arbitrarily long it is better to have a sequence of definitions. Scripts containing sequences of short definitions aid clarity and make for easy editing. They also make constraint enforcement and redefinition much easier. The following example of a script uses real expressions. (Line numbers are added for reference, they are not part of EdenLisp.)

```
          Monolith.Lsp
1.    real  : th R b L E Omax pi
2.    real  : F Mmax K Kt Qmax
3.    real  : thetam lamda TorSt

4.    thetam = 4*K*R*Omax / (Kt*E*th)
5.    lamda  = E*b*th*th*th / (24*K*R*L*L)
6.    Mmax   = b*th*th*Omax / (6*Kt)
7.    K      = 0.166 + (0.565 * th / R)
8.    Kt     = 0.325 + ((2.7*th)+(5.4*R))/(8*R + th))
9.    Qmax   = thetam*L

10.   E = 72000.0  ; aluminium
11.   b = 5.0
12.   th = 0.5
13.   R = 5.0
14.   L = 55.0
15.   Omax = 100.0
```

```
16.    F = 400.0
17.    th = 0.3
```

The example illustrates characteristics mentioned earlier.

1. Entities can be used before they are defined, *e.g.* `K` and `Kt`. This is a very powerful aid to top-down design.
2. We can change the order of the sequence without affecting the values, unless a redefinition changes an existing value. Line 17 redefines `th` from its definition in line 12 from 0.5 to 0.3. In that case EdenLisp coerces a unique definition by using only the last reference to `th` in the script: line 17.
3. Self referential or cyclic definitions are not allowed

Comments are allowed in EdenLisp after any definitive statement provided the comment is preceded by a semi-colon. A comment may be on a separate line. EdenLisp does not store comments when it reads in a line of input. When line 14 is read in, EdenLisp discards the comment and semi-colon: `; aluminium`

String Reconstruction

Definitive statements are not saved in string format by the interpreter but translated into Lisp and can only be read back from EdenLisp in that form. Commands are not reconstructed or saved, they are executed and then forgotten.

Real and Integer Expressions

1. Real and integer expressions may contain real sub expressions, real variables and constants and the usual real operators. Examples of all these are given in the program.
2. Unary minus "-" may precede any expression.
3. All the other AutoLisp functions relating to reals and integers listed in Appendix 1 are usable in EdenLisp.

Lists and Functions

Lists are fundamental both to EdenLisp and AutoLisp. However some restriction is placed on the format of lists so as to enable strong typing and type checking. Lists are therefore created using the special operator [ ] together with commas to separate the list entries. Both devices are foreign to Lisp since the latter simply

uses parentheses ( ) with white space as separator. To create an EdenLisp definition as a list we write

```
a = [1,2,3,4]
```

Lists can of course contain different typed data in the same list.

```
AL = [360, Alan, "Erica", [6 7 8]]
```

defines a list with an integer constant, a symbol or variable name, a string and a list

Functions that are invoked in definitions are in a format that allows arguments to be passed in the form of lists. This follows the more usual practice where arguments are in parenthesis following the function name rather than the Lisp form. *e.g.* the definition that would in Lisp read as

```
ObjectA = (object basef size origin)
```

is entered into EdenLisp in the form:

```
ObjectA = object (basef, size, origin)
```

(Notice the commas between the arguments in normal Pascal or C style; round parentheses surrounding the arguments are also more usual than around everything.)

Conditional Expressions

We noted above that we can control the structure of expressions by the following form

If <logical> **then** <expr1> **else** <expr2>

where <logical> is a real expression interpreted as either true or false, and <expr1> and <expr2> are expressions (that may be further conditional expressions). Both <expr1> and <expr2> must evaluate to expressions of the same type as declared in the username and when the expressions are nested all the expressions that can supply meanings to the outermost condition must be of the same type.

Files

Input scripts of definitions are easiest to enter into EdenLisp by using text files since such files may be constructed using any text editor. (Text editors can co-exist with AutoCAD by means of a Windows environment such as X or MS-

Windows, or using a Terminate, Stay Resident (TSR) package.) EdenLisp provides the AutoLisp function that is written

```
(Dload "userfile")
```

from the AutoCAD COMMAND prompt. The file has to be stored in the form

```
username.lsp
```

where in MSDOS the username must have a maximum of eight characters. EdenLisp assumes the suffix `".lsp"` is there. If the command is used within a file to insert another file then the EdenLisp will expect the EdenLisp form

```
Dload ("userfile")
```

with the parentheses around the argument.

5.32 Defining Abstract Objects

The structure described so far for EdenLisp is similar to many definitional notations (*e.g.* EDEN, DoNaLD and indeed PADL2). However when we come to defining objects we have a more abstract view, following the algebra based on the complex, frame, object types developed in chapter 4 (§4.22 and §4.23) where we introduced the terms used here. We describe the structure of complex, frame, object using the following fragment of EdenLisp code called **BOX** as illustration.

```
        ; BOX.LSP
1.      ; set up combinatorial structure
2.      LLstr : box
3.      Lstr  : AB BC CD DA EF FG GH HE AE BF CG DH
4.      AB = ["a","b"]
5.      BC = ["b","c"]
6.      CD = ["c","d"]
7.      DA = ["d","a"]
8.      EF = ["e","f"]
9.      FG = ["f","g"]
10.     GH = ["g","h"]
11.     HE = ["h","e"]
12.     AE = ["a","e"]
13.     BF = ["b","f"]
14.     CG = ["c","g"]
15.     DH = ["d","h"]
16.     box = [AB,BC,CD,DA,EF,FG,GH,HE,AE,BF,CG,DH]
17.     ; set up carrier structure
18.      point : a b c d e f g h
```

```
19.     a = [0,0,0]
20.     b = [1,0,0]
21.     c = [1,1,0]
22.     d = [0,1,0]
23.     e = [0,0,1]
24.     f = [1,0,1]
25.     g = [1,1,1]
26.     h = [0,1,1]

27.     ; create generic object frame
28.     frame : boxf basef
29.     boxf  = complex (box)
30.     basef = boxf
31.     frame : baseO baseD

32.     ; create instantiation of object
33.     point : origin size
34.     origin = [0.0, 0.0, 0.0]
35.     size   = [100.0, 30.0, 20.0]
36.     baseO  = object (basef, size, origin)
37.     baseD  = Wireframe (baseO, "line")
```

We start with the most abstract form: the *complex* that is a list of subsets of labels. In the program it is a list of symbols representing the names for what could be sides of the box yet undefined. Line 16 is

```
BOX = [AB,BC,CD,DA,EF,FG,GH,HE,AE,BF,CG,DH]
```

Each of the terms `AB, BC, CD...` is defined as a pair of strings such as `AB =["a","b"]`. Elements pairs are not yet variable or identifiers, they simply indicate a combinatorial structure.

To derive a *frame* from a complex it is necessary to supply specific coordinates and scalar parameters corresponding to the abstract labels of the complex. So `boxf` needs to be supplied with definitions for the named variables `a b c d e f g h`. Rather as in PADL-2 it is convenient to have default values of unity for these values. Lines 19-26 define coordinate points of type `Lreal` (list of real) in terms of the unit box. Strictly one should specify the dimension of the space in which the complex is to be realised, but in EdenLisp the length of each coordinate list is coerced into that of the longest (by adding zeroes to the tail of the list). A warning is issued if the dimensions are not all the same, but it is allowed as it is sometimes convenient to enter a mixed 2D and 3D set in order to transform 2D into 3D.

The function

complex(box)

is the operator of the type "Operators for accepting complexes and lists of reals to realise a frame" described in §4.23. This operator accepts a complex and turns the strings (or quoted atoms) into labels and associates them with the pre-declared variables of that name. So line 29 returns the value

```
boxf   =  [(a b)(b c)(c d)(d a)
           (e f)(f g)(g h)(h e)
           (a e)(b f)(c g)(d h) ]
```

which is a frame. New frames can be constructed with the same default values by the equivalence operator, so line 30 creates the frame called `basef` that is identical with `boxf` but with a significant difference: the function **`complex`** yields the *names* of variables as its returned value, whereas the equivalence operator returns the *values* of the variables associated with those names. The distinction between the two operators is a nice one but very useful for further manipulation.

Intantiation of a frame as an *object* is done by means of the function **`object`**, viz. line 36

```
baseO  = object(basef, size, origin)
```

`object` acts as an operator on a frame plus two vectors. The first vector associates a scaling factor with each of the components of the corresponding variables. The second vector is a translation of the whole frame to a new local origin. Thus `baseO` evaluates as follows (in AutoLisp format)

```
origin  =   (0.0 0.0 0.0)
size    =   (100.0 30.0 20.0)
baseO   =   (object basef size origin) =
     ( ((0 0 0)(100.0 0 0)   (100.0 0 0)(100.0 30.0 0))
        ((100.0 30.0 0)(0 30.0 0)   (0 30.0 0)(0 0 0))
        ((0 0 20.0)(100.0 0 20.0)   (100.0 0 20.0)(100.0 30.0 20.0))
        ((100.0 30.0 20.0)(100.0 30.0 20.0)   (100.0 30.0 20.0)(0 0 20.0))
       )
```

Although this is now a set of vertices at a given position the object is only defined as a graph. We need operators on that graph to realise the object in the various ways we may wish to form the geometrical model: as a wire-frame or B-Rep, *etc*.

5.33 Operations on Sorts

The *type* `FRAME` is used for the complex and the frame depending upon the realisation of the graph structure that is employed. Complexes may start life as strings or symbols (Lisp allows either) but usually end up as lists of coordinates,

i.e. `LLreal` so the operations on the algebra frequently change the type as the sorts are transformed from complex to the object.

Operations on complexes depend upon how the labels are specified. If they are strings then they can be concatenated with other strings to create new pairs with the same combinatorial structure. One method for doing that was first explored in CADNO by John Stidwell [Stidwell, 1989]. In EdenLisp we have a function ISOMORPH that acts as an operator on a complex with a string to create an identical complex in terms of its combinatorial structure but with each element of the input complex having the input string concatenated. So the function

```
isomorph("BOX", box)
```

concatenates the string `"BOX"` to each of the elements defined by `AB BC CD...` to yield pairs like `AB =("BOXa" "BOXb"),` etc. The advantage of this operator is that new abstractions with the same structure can be created but with different labels and hence with their own 'life' independent of their pedigree. Once the combinatorial structure is defined then the function **complex** transforms the complex to a frame in the way already described. So if we attach `BOX` in the way indicated to a new variable `boxf1` then **complex** would yield the same structure with different labels:

```
boxf1  =  ( (BOXa BOXb)(BOXb BOXc)(BOXc BOXd)(BOXd BOXa)
            (BOXe BOXf)(BOXf BOXg)(BOXg BOXh)(BOXh BOXe)
            (BOXa BOXe)(BOXb BOXf)(BOXc BOXg)(BOXd BOXh) )
```

The ability of the complex to form new variables is very powerful. It allows us to localise information in a single instantiation or to pass information of a generic kind across a whole family of instantiations. The effect of redefining the basic form of `BOX = (AB BC CD DA EF FG GH HE AE BF CG DH)` would clearly change all derived objects. It would be the equivalent to redefining a primitive such a BLOck in PADL-2 during a session.

In fact with Lisp we can dispense with the need to have strings to form complexes. Using the quoted atom we can get exactly the same structure as with the string form excepting that we lose the ability to concatenate strings as before. The following amendment to the EdenLisp code for Boxes illustrates the difference.

```
      ; BOXES
1.    ; set up combinatorial structure
2.       LLreal : box
```

```
3.        Lreal  : AB BC CD DA EF FG GH HE AE BF CG DH
4.        AB  = ['a,'b], BC = ['b,'c], CD = ['c,'d], DA = ['d,'a]
5.        EF  = ['e,'f], FG = ['f,'g], GH = ['g,'h], HE = ['h,'e]
6.        AE  = ['a,'e], BF = ['b,'f], CG = ['c,'g], DH = ['d,'h]
7.        box = [AB,BC,CD,DA,EF,FG,GH,HE,AE,BF,CG,DH]
```

Since the complex is a graph structure we can have operators that perform edge creation for us. So in EdenLisp the following operators are provided

PEDGE: (poly-edge) takes a list and arranges it into pairs forming a consecutive set of edges like a polyline:

```
pedge(a,b,c,d) = ((a b) (b c) (c d))
```

CEDGE: (cycle of poly-edges) takes a list and arranges it into pairs forming a closed cycle joint first and last nodes:

```
cedge(a,b,c,d)  =  ((a b) (b c) (c d) (d a))
```

CGRAPH: (complete graph) takes a list of nodes and connects them in every possible way pairwise

```
cgraph(a,b,c,d) = ((a b) (a c) (a d) (b c) (b d) (c d))
```

Graphs may be combined by simply listing them. Other graph properties are easily added to these functions as required. For example paths and cycles could be extracted. (See implementation in chapter 6 for details of how to do that).

Frame functions are standard vector and matrix transforms on coordinates. The following functions are implemented in EdenLisp.

`Vsum (V1, V2)`	Vector sum
`Vdiff (V1, V2)`	Vector difference
`Vtrans (Rlist, Transl)`	Translate a vector Rlist through Transl
`Vscale (RRlist, Scalar)`	Scale an object by single scale factor
`Vshear (V1, V2)`	Scale vector by selective scaling
`SProd (V1, V2)`	Scalar Product
`InnerProd (vectA, vectB)`	Matrix inner product
`transpose (matrixA)`	Transpose of a matrix
`matrix-add (matA, matB)`	Matrix addition of 2 matrices of same size
`matrix-product(matA, matB)`	Product of 2 matrices of appropriate sizes
`Mshear (matA, vectS)`	Selective scaling of a matrix
`Vect-transform(matA, vectX)`	Isometry of a vector => vector
`rotV (vect, angl, axis)`	Rotation of a vector

`rotobj (lvect, angl, axis)` Rotation of a set of coordinates

Frames can be used as the basis of sweeps, extrusions and similar 2D to 3D transforms, *e.g.* the function `extrude(cedge (["a","b","c","d"]))` takes the vertices closed cycle `["a","b","c","d"]` as argument and creates a new cycle of vertices `["ea","eb","ec","ed"]` and returns both with corresponding edge connections as an extrusion.

```
extrude(cedge(["a","b","c","d"]))
            = (("a" "b") ("b" "c") ("c" "d") ("d" "a")
               ("ea" "eb") ("eb" "ec") ("ec" "ed") ("ed" "ea")
               ("a" "ea") ("b" "eb") ("c" "ec") ("d" "ed"))
```

Apart from the transform function `object` described above, the main object type operators are to do with how a skeletal structure may be realised. Using AutoCAD' s own commands one can realise a frame as line segments as in the left hand drawing in *fig. 5.2*, or it can be all circles to form a sphere. Or indeed it may be selective as in the cylinder formed on the right of *fig. 5.2.*

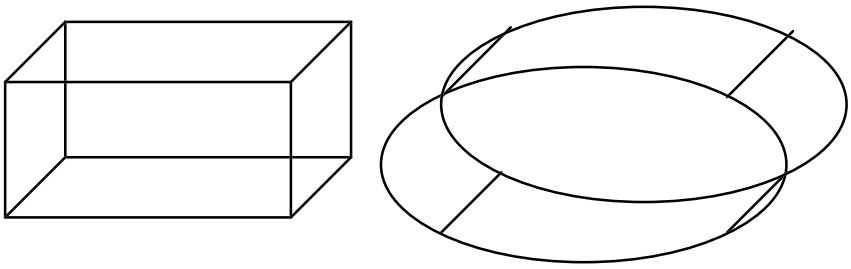


Fig 5.2 Two realisations from the same combinatorial structure

Realisation functions allow the user to specify the object in very different ways. The primary realisations are **`wireframe`** and **`Surface`** and CSG **`solid`**. **`Wireframe`** joins vertices by curves or straight lines as specified by the argument (*e.g.* `bard = Wireframe (barf, "pline"))`. The currently implemented realisations under wireframe include

```
"LINE"       . directed line segment
"SEGS"       . series of separate line segments
"PLINE"      . polyline
"SPLINE"     . splined curve through the vertices
"3DPLINE"    . 3D polyline
```

```
"ARC"         . circular arc
"POLYHEDRON"  . wireframe through the specified edges
"EXTRUSION"   . extrusion on a frame
```

`Surface` realisations follow a similar functional form, for example,

```
"3DFACE"      . 3D face using AutoCAD face command
"EXTRUSION"   . extrude to produce surfaces
```

Finally we can utilise AutoCAD's own CSG representation to display solids but with the added functionality of EdenLisp.

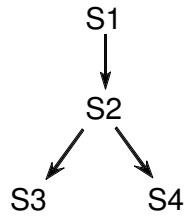
5.34 Windows

Objects made up of several components from the same root, such as assemblies of cuboids, need to be hierarchically structured so that parts can inherit properties that are above or generic whilst being able to have local properties independently of similar parts in the same family. The problem is analogous to having local variables in a procedure or function within a programming language. We may wish for example to have a box with the same labels for vertices `[a,b, .. h]` whether that box is on one part of the system or another, so saving having to remember whether labels have been used before. The labels have to be local. In EdenLisp that is achieved by means of Windows, rather in the manner of DoNaLD's `within{...}` declaration. Windows are structured as Unix directories, so that within a window, variables may be named regardless of their antecedents or descendants. Variables may be referenced from within a window structure if they not declared within the window (locally) but are declared in a parent window. If a parent variable is redeclared locally then that overrides the parent definitions whilst in that window but such variables only affect descendants, not antecedents when evaluation takes place.

A tree used to represent the window structure has nodes, each of which represents a particular context where new definitions may be constructed and that inherits all definitions on the path to the root node. Definitions created in a particular node can only be evaluated with information from that node or its antecedents on the path to the root. Redefinition or re-evaluation is not allowed from child contexts.

An instantiation of a window taken from a remote node is allowed by making the imported variables and definitions local and independent of the source node.

An object is built up in nested windows with the most abstract information at the outermost window. An abstract shape is split into components: one set gives the underlying relationships used to delineate the shape, another identifies the way in which that shape will find representation. To illustrate the structure we create two objects with similar topology and geometry using windows `(S1, ... S4)` in a tree as follows.




```
(OpenWin 'S1)          ; Create and open Window S1
   LLstr : face$
   face$ = cedge (["a", "b", "c", "d"])
                       ;complex definitions
   (OpenWin 'S2)
      point : a b c d O
      O = [0.0, 0.0, 0.0]
      a = vsum (O, [-0.5, -0.5, 0.0])        ;frame definitions
      b = vsum (a, [1.0, 0.0, 0.0])
      c = vsum (b, [0.0, 1.0, 0.0])
      d = vsum (a, [0.0, 1.0, 0.0])

          (OpenWin 'S3)                    ; Opens S3 off S2
            frame : faceF faceG faceO1     ; object definitions
            Real_vector : scale origin
            faceF  = complex (face$)
            Origin = [100.0, 100.0, 0.0]
            scale  = [20.0, 30.0, 0.0]
            faceG  = object(faceF,origin,scale)
            faceO1 = wireframe(faceG,"LINE")
          (CloseW 'S3)                     ; closes S3, moves to S2

          (OpenWin 'S4)                    ; opens S4, off S2
            frame : faceF faceG faceO2
            Real_vector : scale origin
            faceF  = complex (face$)
            Origin = [200.0, 100.0, 0.0]
            scale  = [35.0, 42.0, 0.0]
            faceG  = object(faceF,origin,scale)
            faceO2 = wireframe(faceG,"SPLINE")
          (CloseW 'S4)
      (CloseW 'S2)
   (CloseW 'S1)
```

Nodes S1 and S2 provide the topology and abstract vertex positions for the two "face" objects that have different realisations in S3 and S4. Changing the abstract definitions in S1 or S2 affects S3 and S4 in the same way, but changes in S3 are independent of those in S4. So `face01` is a wireframe, `face02` is a spline.

The window structure of EdenLisp creates a definitive notation for Object representation with the following features.

a) Object representation is by means of trees of scripts that define geometrical and technological information at increasing levels of abstraction where the current state of an object is given by a single path in the tree. This path then represents the current state of interaction.

b) An evaluator of definitions accepts definitions from a script as input, and after checking, evaluates according to current information, transparently to the user and in any order.

c) The state of the display is defined according to the state of the object representation but may also show the current state of interaction and currently suspended states.

d) The graphical interface handles anything that the modelling system may wish to display.

5.4 Design Process: State and state change

State is represented in EdenLisp by the current set of definitions. Alternative states exist by redefinition, so are infinitely variable. In practice it is often desirable to constrain the state changes that can be made. Constraints are easiest to define via the `if` statement provided in EdenLisp. Using that statement in a definition makes the value of the definitive variable conditional upon another definition in a way that can be used to trigger re-definitions that constrain the user's ability to redefine (change state). At the weakest level a warning can be issued; at stronger levels preconceived changes may be triggered that counter, or follow, the effect of a change. For example a table lamp moved from its position may trigger an action to move the flex and plug to another, nearer socket. Alternatively the user may not be "allowed" to redefine a particular variable. (Clearly that is impossible to a super-user, who has ultimate power, but it can apply to an agent who is thereby kept from accessing a particular definition.)

A more subtle way of implementing changes due to constraint violation is the automatic rewriting of definitions. As explained above definitions that do that are called *actions* and can trigger wholesale rewriting. We investigate ways of doing such manipulation.

If we divide definitions into groups, or scripts, then each script may be operated upon more or less independently. Interactions with other scripts will be taken care of transparently. In principle each script could be changed by different agents. It would only be necessary to prevent interactions that would lead to illegal dependencies such as circular definitions, and that can be taken care of by suitable constraint management. What remains is whether those other agents can perhaps be the computer itself. For example it happens that certain scripts are very repetitious, involving large numbers of definitions that are broadly similar. Creating and managing those scripts is tedious and the computer can be asked to take over the task of changing the state of a script by writing or rewriting appropriate definitions and implementing them as the new current script. That process of script manipulation then becomes a problem of definitive notations for defining definitions, a kind of meta-definitive notation. That meta-notation thus consists of state changing actions.

One way to create actions is pointed up by the experience of writing `shaft.DoNaLd`, namely by manipulation of scripts of definitions. There the scripts were managed by manipulating files of definitions, creating destination files by manipulation of text in source files. Using Lisp it is easier to manage such text manipulation. New definitions may be constructed as text or quoted lists and called into being as part of meta-definitions. In such an operation the text to be operated on could be the value of a definition of type string (or a list of strings). The action definition would have as its value a definition that was an amended version of the source text. Provided the new version was a legal EdenLisp definition and appropriate variables were correctly declared then the action would both evaluate the text changes and also send the amended text to the EdenLisp interpreter.

Another kind of action can create new variables and hand values to them. If a series of new variables is formed in sequence, each of integer value in increasing order, *e.g.*

```
p1, p2, p3, p4, p5, p6, ... ,
```

it is easy to see that an array can be constructed such as might be needed in forming a Cartesian graph. Actions would create the X and Y values that would be the plotted variables. If the string version of a formula in Y(X) is passed to an appropriate function then sets of pairs containing (Xi, Yi) can be computed. By that method individual points of the graph have separate variable name, rather than being accessed via a pointer into an array. There are times when it is useful to be able to access points in that way. For example the representations of points in different graphical format (bar, pie chart, scatter diagram) are well-known alternatives available in most computer packages. Using the actions described it is fairly trivial to use the points p1, p2, etc. to generate alternative formats for display purposes.

The power of actions is highly dangerous. It is possible to wipe out and re-write whole sections of code. However there is nothing except rule to prevent a prototype designer doing the same kind of damage to the product in either geometrical or physical form. The weapon of ' action' must be used circumspectly! And that matches the idea of constraint definitions described above. In combination with "if" statements guards can be put on actions in the way that is described in connection with the Abstract Definitive Machine described earlier (section 3.42). If an action does do damage then some notion of state history is necessary. The simplest way to do that is to record the previous states by filing static positions. That is not a satisfactory way in the long term as it relies on the user doing a fair amount of storage, maybe most of it useless. This remains an area for further research.