10

Discussion

This thesis is primarily concerned with a novel computer tool to aid the conceptual stage of engineering design. During the course of the work new insights were gained into the design process, particularly in appreciating the roles of observation and experiment, and the parts that agents play in design. Definitive methods have the property of latent state that allows them to faithfully model experiment. The implications of having a state based system to aid design have been explored during the development of EdenLisp, the Definitive programming system linked with CAD. In this chapter the benefits that EdenLisp brings to the conceptual design process are discussed. Also the stage that notation has reached is reviewed and a prognosis is presented on its further development.

10.1 The Thesis

10.11 Understanding Conceptual Design

The concept of "computation as experiment" has evolved during the period of the work carried out to support this thesis. Indeed, the investigation into applying definitive methods to the design process via EdenLisp has become a major vehicle for insight into the nature of the conceptual design process.

The link between the design and the designer is very close at the conceptual stage. It is difficult to separate “the design” from the mass of thoughts, ideas, trials, experiments and hunches. It may be impossible; the design may be inchoate at the time one is trying to disentangle it. The idea of computation reflecting that stage of “chaos into order” is very attractive. The many design and program examples scattered through this dissertation are proof of the way that “real” design can be handled by definitive method. It is not claimed that these examples could *only* be handled by definitive methods. Indeed it would be easy to assert the contrary view.

All of the examples can be written in other programming languages. The uniqueness of the method lies in its ability to help formulate, as well as solve problems: to "think aloud" with them, to test out ideas, to frame relationships and try them out, to search for pattern and abstraction and experiment with generalisations that seem to point to a richer class of solutions than solving *this* design problem.

In chapter 3 we noted that the physical world is essentially understood by people in terms of observations. The precise mechanism for combining and experimenting with observations is still under contention. Minsky's theory, discussed there, supports a kind of grouping of observations into "frames". A similar notion of "chunking" discussed in chapter 9 is indicated by psychologists as typical of the declarative approach to learning, where people link new experiences with old and make discriminations and generalisations. The abilities to record observations and discriminate between them seems to correspond to important activities going on during conceptual design. A Definitive script supports these activities in two ways: it faithfully records observations by means of definitions and it allows the incremental examination of the interaction of those observations by redefinitions.

A single definitive statement such as `Car_colour = blue` is a metaphor of an observation or characteristic property that should find a counter-part in the physical world. A selection or grouping of such statements constitutes a kind of abstract object, perhaps corresponding to Minsky's frame. It is that selection process that matters when forming concepts, rather than what the actual constituents of the object are. The latter seems to have only a passing relationship with the *process* of conceptual design. If one concentrates only upon what that object might look like, or worse, tries to encapsulate that object, then one misses the point. The object at that stage is only a provisional set of observations to be tested against the real world and constantly to be open to detailed alteration or radical reorganisation.

The testing process is the second feature mentioned for Definitive scripts. When a definitive statement is redefined (as `Car_colour = silver` for example), the effect of that change is experienced (computed automatically) in all other definitions that depend upon the changed statement. The important feature of that change is that it is not predetermined by the existing scripts of definitions. Any change in a script automatically creates a new state that mirrors the change made.

If the script faithfully records observations of the real world, by means of relationships found by experiment, and the change also is consistent with those observations then the new state will be consistent with physical reality. It is the ability to perform incremental changes that gives rise to the notion that each definition carries a set of *latent states* - states that can be obtained by redefinition in a consistent way. The way that one checks on the new state after a single change, rather than accumulating a series of "logical" changes, appears to accord with our human perception. We prefer to be constantly checking out our perceptions against reality, a procedure endowed with the name "non-monotonic reasoning" by some logicians.

In earlier chapters we explored the contrast between *form* and *content* and showed that content cannot be captured by any set of symbols. With EdenLisp one can explore the "content" of a design by recording as many observations as seem to be consistent with its specification, including perhaps observations that do not appear to be significant. Not all of the observations that are recorded may be needed in the design. Indeed some that are needed may turn out to be missing. At the conceptual stage of the design it is never really clear how the observations will interact in the product of the synthesis. What the designer is seeking is a novel combination of observations, a kind of pattern seeking or imposition of pattern that effectively groups observations. It is only in interaction that one can begin to explore the implications of linking particular patterns of observations. Inspection of the redefinition of the car colour mentioned above yields an unexpected content: "silver" can be interpreted as a colour or as a material, with very different perceptions of its meaning. That kind of perception would be impossible to automate and emphasises the importance of interaction: any computational tool that provides for that will certainly support conceptual design. Interaction becomes progressively more significant as the design task gets bigger. As the number and complexity of the observations grow so the "experiments" one can do by redefinitions within the scripts also grow.

10.12 Computational Experiment: The place of EdenLisp

The Definitive method was not originally developed with engineering design in mind. It is really a way of computer programming that differs significantly from existing forms in that it hides those aspects of computation that do not belong to the process being modelled. Computational operations such as iteration, recursion and evaluation are not important to *what* is being modelled, they belong more to *how* it is being modelled. An analogy might be seen by watching someone

crossing the road, glancing to each side to check it is safe to cross. She seems to know exactly how fast to cross and not appear to be dodging cars. If asked she would not be able to describe *how*, but would certainly say *what* she was doing. The process going on in the brain is hidden at a lower level. In a similar way the definitive script is evaluated transparently to the user, and re-evaluated automatically each time a definition is added or changed. A further property of the notation is that it does not need to sit on top of a conventional programming language. Unlike higher level languages such as so-called 5th-generation languages, there is no intrinsic need for that. It is feasible to have it definitive all the way down to processor level - indeed it is conceivable that a definitive processor could be designed.

The crucial development in definitive methods was Eden by [Yung, 1987], implemented to enable algebraic relationships to be processed and so permit experimentation that was algebraic in nature. The desire to have a tool to experiment with graphics relationships led to the line-drawing definitive notation, DoNaLD. That work demonstrated the possibility of applying the paradigm to computer graphics and thence into design. Work began on a 3D version of DoNaLD called CADNO [Stidwell, 1989]. It was then realised that to develop the graphics further would entail entering computer graphics research, and that that would severely constrain the development of definitive notations. The decision to consider the design process, without further development in graphics, was made possible because of a significant property of definitive notations: they are "impure" in the sense that both functional and procedural forms are permitted in definitions. That makes a notation able to call other programming forms that exist as procedures or functions. Thus DoNaLD makes use of existing graphics libraries; interfaces to the screen make use of graphics and screen routines under X-Windows.

Given that graphics handling systems can be linked to definitive notations, CAD systems are better candidates than simple graphics libraries, especially as many engineering design aids began life with CAD. We have seen that the choice of AutoCAD with AutoLisp has proved to be beneficial because of the declarative form of Lisp. EdenLisp simply calls appropriate AutoCAD functions, both for drawing and for revising drawings.

Setting up EdenLisp was a major task and might have got in the way of the primary aim of aiding the conceptual design process in engineering, except that the implementation issues actually helped to illuminate aspects of that process.

For example input definitions must be stored in a symbol table for evaluation purposes. That task could have been done, by analogy of "definition as observation", simply by means of a linear list. It is more useful, however, to record them hierarchically in terms of connections with such abstractions as lead to their instantiation as objects. The user should be allowed to make the connections and the symbol list should keep those connections in a way that allows for extending and editing the observations in the appropriate context. The flexibility of the Unix directory structure makes it a useful model and that has been implemented in EdenLisp, even though it has raised all kinds of questions about the links that now must be made between the "windows" or "environments" in the structure.

The ways used by the mind to select and structure observations are also likely to be used to synthesise new ideas. Groups of observations become "chunked" and then can themselves have higher level structures with other chunks. Those higher level structures will be complex, involving interactions with other, autonomous, objects. It is that thinking that led to the idea of autonomous objects being like agents and thence to agent-oriented programming. Programming of that nature is still in process of development and the Abstract Definitive Machine (ADM) described in chapter 3 (§3.42) is being elaborated to cope, aided by the ideas decried in this work [*c.f.* Adzhiev, Beynon, Cartwright, Yung, 1994*a*]. The ADM was originally devised as a methodology to structure definitive scripts [Beynon, 1990]. It is not itself a programming language; it is more like a pseudo-code. As the ADM sorts out the interactions of agents, it helped to structure the symbol table in analogy with Minsky' s frames. Abstract "objects" within scripts written in EdenLisp can be developed interactively as the design evolves. Scripts can even be generated by autonomous agents provided the interactions with other agents are covered by appropriate constraining definitions.

The requirements for constraints to be clear and for progress in the design to be monitored inspired the notion of the ‘virtual prototype’, a computational model that structures definitive scripts in a way that gives behavioural characteristics similar to an engineering. That is the state of the set of scripts is the state that is currently reached in the design. That idea still needs some working out, but it has the potential to support some very important features of the design process, including interaction of different agents exploring separate substates for the prototype simultaneously. The most significant issues needing to be addressed with concurrency are associated with decision-making and constraint management,

but while these remain subjects of further work there is little doubt that the principle is a sound one.

10.2 Achievements

10.21 Interaction

It became clear during the course of this work that the inability of computational tools to model the "content" of symbols can be offset by a more explicit partnership with the user. For that to happen interaction has to be a fundamental part of the tool being created. Definitive methods differ in that important respect from other programming paradigms. Many standard programming methods have by their nature a static or predetermined set of states which the user activates by appropriate inputs. With a definitive notation the program plus data has current state, but with an infinite number of other states. Proceeding to another state is by incremental action by redefinition; it is not pre-programmed. The current state of the script of definitions *is* the design rather than being a *description* of the design.

The way that EdenLisp has been constructed constrains the user to modes of interaction that actually support the engineering design process. The need to declare the type of variables encourages the user to think carefully about what is being attempted. For example, it may cause one to consider topology before geometry, so stimulating the formation of more abstract representations of the design problem or sub-problem. That, in turn, may suggest more useful ways of tackling the basic design, or it may cause a whole family of possible design scenarios to be considered. EdenLisp' s use of strings to generate abstract labels is increasingly recognised as a useful device for creating new or generic variables. (It is noteworthy that MatLab uses just such a method for creating its object environment.) EdenLisp is able to deal very effectively with the creation of character based lists. Some of the string handling ideas derive from trying to translate DoNaLD statements into Eden in such a way that Eden can be employed both to create and to interpret the statements being created. Lisp is better structured to deal with that kind of translation.

Interaction via EdenLisp may be at different levels. At the top level it is with the set of definitions; at the second level it is by writing or generating functions and procedures to be used in those definitions. Writing new functions involves some knowledge of AutoLisp, which is a disadvantage since most engineers are unfamiliar with Lisp. To counter that the functions can be constructed to be as generic as possible and so interaction can be by means of libraries of functions that can be built up to the level where few new ones are required in a particular

case. That would mean that an engineering user would need to create few if any special functions to carry out a particular design task.

10.22 EdenLisp CAD Environment

The most significant achievement in this work is the implementation of a definitive notation within a CAD system. The CAD and EdenLisp environments together provided a platform for the development of the definitive method itself. The link between design and geometrical modelling has been complemented by linking the specification of the design with the initial thinking about that design. The formation of the design specification can be done using the EdenLisp notation. That means that as flesh is put onto the design geometrical models can be formed and checked at every stage. The idea of "what if?" that is inherent in the method is very apparent by the way the display gets updated at each redefinition. It is most illuminating to see the display of the dental arrangement (described in chapter 7) showing the tooth forms as they get modified by a single change of variable. The animation enhances the effect and stimulates the designer to try other possibilities, so increasing the likelihood that the design is the result of a decent search though the feasible solution space.

The exercise of creating library functions for topology and geometry showed that it is possible to use AutoCAD commands embedded in the function. The user can call precisely the geometry required but then play with the definition whilst retaining the design constraints. If a polyline is to be modified into a spline then the symbol table retains the connectivities that the node data of the spline has with other entities. The same data can in principle be used to convert the display from wireframe to a solid model - a step that in conventional CAD is quite difficult. A bonus that comes with the use of a CAD system is that the geometrical data is stored in two ways, one in the symbol table and another in the CAD database. Although that creates some redundancy, unnecessary duplication is avoided by connecting the two forms. EdenLisp takes advantage of the references (handles) AutoCAD has between its database and its display. Display information is processed in AutoCAD whereas structural or design information is in EdenLisp - a useful division of labour that also emphasises the role of EdenLisp as a design tool.

10.23 EdenLisp Implementation

A number of issues were recognised in the implementation of EdenLisp. Apart from the symbol table discussed above, the main issues were speed, size of code,

the need for an intermediate code (as between DoNaLD and Eden), the typing of variables and the links with CAD.

Speed and Memory

Problems of speed and memory size were addressed by trying to write all Lisp functions as simply as possible. In designing a function, a recursive form was used to start with as that usually keeps the length of program text to a minimum. Where recursion caused problems by too deep nesting then a tail-recursive form was tried. Tail-recursion keeps the stack depth to one and is also the form that is most easily re-implemented in iterative form if speed tests showed that to be desirable. The code is therefore as compact and as fast as the AutoLisp allows. Tests also showed that a significant improvement was always gained by compiling the AutoLisp code.

Typing

Typing creates a discipline in programming that forces the user to think in a manner that is tied to an EdenLisp way of thinking. The types constructed in EdenLisp are ‘character’, ‘integer’ and ‘real ’together with list of each and list-of-list of each. The latter allows for the development of topological and geometrical operators. More generic types such as ‘symbol ’and ‘list ’may be required at times to deal with odd functions and conversions but these are not usual.

The implementation of typing of variables in an untyped language like Lisp proved exceptionally difficult. Several particular problems can be highlighted. The first concerns functions that are created by the user when constructing definitions in EdenLisp. There is no separate EdenLisp function generator in the way that Eden has Eden functions without recourse to the underlying C language. Functions have to be written in AutoLisp and pre-declared and typed, to establish them as legal EdenLisp functions, before they can be called. A library of many useful functions has been built up and pre-declared in EdenLisp, but a new user function has to be entered into the EdenLisp structure so that the parser recognises it. Initially, the user had to actually enter the type data into the EdenLisp program code. That unsatisfactory solution that has now been rectified by a function that does the task in a more transparent way.

A second problem arose in dealing with what is called "quoted" atoms and statements. Those have a peculiar but extremely useful status in Lisp. A quoted atom or statement is not evaluated by the Lisp interpreter, so a quoted atom

returns its name, not its value. Lisp can store any list, including what amounts to program text, as an unchangeable object until such time as it is wished to use that content that is quoted. When EdenLisp program text input by the user contains quoted statements as part of a definition it is desirable that they are dealt with in the same way as Lisp does. In these cases parsing is difficult to do as the lexical analyser wants to identify all that is in the input text, but that device is implemented as part of EdenLisp.

Notational Consistency

It is perhaps inevitable that the work on constructing EdenLisp commenced before the conceptual ideas had been fully sorted out. Ideas were implemented and then changed. It often proved difficult to reconcile the old and the new. Where possible the samples of code that have been discussed in the text have been cast into the most recent notation. Earlier versions of EdenLisp contain significant differences, some of which remain in the program code as possible re-implementation areas should there be a need. Some of the changes may seem rather cosmetic. For example it may be better to have a different symbol for "=" in the definition assignment. That is because the assignment of a variable to a definition is rather stronger than the "=" sign implies. Eden uses the symbol "is" to indicate that the definition is one that gets updates automatically if any dependent variable gets amended. The Eden notation allows the use of "=" when assigning a constant (as an indication that it never needs to be re-calculated). In EdenLisp it was decided that the use of "=" for all types of definition was justified since all definitions are scanned during evaluation, whatever form they have.

Some other ideas tried in the early stages have left their mark. The methods used to structure the symbol table for the windowing system deal with the programming problems of global and local variables rather than properly addressing the relationships an object might have with its constraints. That problem remains as a subject for further work.

Most recent developments in EdenLisp are attempts to address the problems of agent interactions. Those are what has led to the changes in the ADM in order to express how the different agents in a system relate to one another. Proper programming of these ideas in EdenLisp still lies in the future

10.3 Comparisons

10.31 Conventional Approaches

An important issue with any new method or tool is its potential. Is the method actually going to do more than existing methods in terms of engineering design? Is it worth the bother of learning arcane methods of construction? It does seem from these investigations that the Definitive concept differs significantly from existing methods, particularly with respect to the way it expresses state. It is difficult to define what is meant by "state" in computational terms, but there is no doubt that "the current state of the interaction" is a notion that expresses a design idea in the construction of a prototype.

At its heart conceptual design appears to be concerned with making new connections within the set of mental conceptions that the designer has obtained by experience. That experience is fed by other people' s experiences but ultimately all experience is based upon experimental observations of the real world. "Conceptual design is a kind of negotiation between *what we believe to be true* and *what we observe to be true"* [Adzhiev, Beynon, Cartwright, Yung, 1994*b*]. In generalising from experience one tries to create *integrity* in what is expected. Objects will have patterns of behaviour and that behaviour will be constrained in ways that are believed to be typical or expected. The characteristic of experiment is *immediacy:* the experimenter is concerned with what is observable now, and what incremental changes might be possible. Changes are not preconceived: one can perform whatever experiments one likes on real objects and observe what happens.

In computational terms, integrity is well covered. Objects are represented in Oriented Programming and patterns of behaviour by formal specifications. State transitions are pre-arranged by providing the ability to change values of variables. Traditional Computer Science gives well specified, precisely circumscribed behaviours of reliable computing devices. Definitive methods give a direct correspondence between values in the computer model and observables in the external world.

One can go further. Computer Science tends to work by Logical Specification. In order to construct models of the real world certain groups of observations are selected from the infinity of observations that can be made. Those models are then constrained to behave in a particular way so that one can know all about them. It is not necessary to know what happens in reality in order to know what happens in the model: once constructed the model can be divorced from reality.

If we look at the definitive script we see that the modelling is *not* independent of the context. Indeed we can say it is *situated modelling*. The modelling is constantly being situated in its real world context as it is being formulated. In order to have a realistic model of what I observe, any incremental change in my model must be consistent with what I observe, otherwise I need to change the model to make it so. A script in that sense is tentative, as our beliefs about the world are. We believe that the world behaves thus because of experience, but we are ready to amend our beliefs should the experience subsequently require.

We can contrast this idea of immediacy by comparing design with analysis. Analysis has to do with understanding what is, whereas design is to do with what might be. Synthesis is to design what analysis is to research. Conventional computational approaches suit analysis, differentiating and discriminating what is. Definitive methods are to do with integrating and generalising - more typical of design processes.

These ideas do not appear to be as explicit in non-definitive systems. For example it is perhaps an open question whether these notions are apparent (or could be constructed) in such programming paradigms as APT, PADL-2 and application programs like Design View. On the other hand the power of these methods and the growing developments of the parametric approach to CAD systems generally bear witness to the perceived need for more "situated modelling" techniques.

10.32 Other Definitive Methods

EdenLisp derives from other definitive notations based on Eden, but it is significantly different in its conception and implementation. Unlike Eden, EdenLisp emphasises the definitive nature of the code by deliberately separating the delineation of functions from the script of definitions that call those functions. In that respect it has more in common with derivatives of Eden such as DoNaLD and CADNO. However, those notations translate to Eden whereas EdenLisp is the basic notation. In Eden, procedures and functions tend to be the principal elements of scripts in terms of the bulk of the programming effort, so the definitive nature of what is being coded is difficult to see. Indeed some of the code written by others in Eden looks more like C-program code. By contrast, the process of constructing functions separately in EdenLisp has proved a strong discipline. Since all functions have to be typed and declared, much thought has to go into their construction. The user is thus encouraged to create functions that are as generic and useful as possible so they can be added to libraries for use by other

scripts. Examples of such functions abound in the function libraries of EdenLisp such as EdenUtil, EdenTop, EdenGeom, and EdenDisp. Those contain functions that are respectively general utilities such as sorting, replacing items in a list, set operations; topological and geometrical relationships and finally display calls to AutoCAD. Many functions are structured so as to make it easy to call another function, *e.g.* display functions always need a call to the AutoCAD "command" functions, so the name of the command can be passed via a string, using various pseudonyms to enable access to different ways of constructing particular shapes.

It might be argued that the use of AutoLisp rather than EdenLisp to generate EdenLisp functions is "impure" since Eden functions are generated by Eden statements. That is true and some guarding is done by imposing the requirement to declare and type the functions. (Eden does allow calls to C functions. DoNaLD too has the ability to call Eden directly rather in the manner that one might include Assembler code "in line").

10.4 The Future

10.41 User interface

The Engineering designer wants computer tools that will do the job without having to worry about how they operate, and preferably that they do it better and faster than anything else. From that point of view objections can be raised concerning the practicality of definitive methods. The user interface is textual. It requires a sophisticated programmer with a good understanding of the definitive method to make any headway with the system. Typically a new user needs prior knowledge of a language like C to work with Eden, and to use EdenLisp the user needs both to be able to use AutoCAD reasonably well, and to be able to program in AutoLisp at the level expected of a good programmer in Lisp.

Those issues are important, but it was thought that definitive principles themselves needed to be clarified and implemented before considering the user interface in detail. Some work has been done on definitive user interfaces. A number of experiments by other workers in the research group have shown the feasibility of using text windows and graphical buttons and so on in a graphically based definitive system, *e.g.* SCOUT, [Beynon, Yung & Hogan, 1992]. It would not be difficult to write a similar interface using EdenLisp and AutoCAD facilities that will itself be definitive in form. That would mean that the display and the means of interaction would itself be a function of that which is being designed. Different views of the design demand different interfaces so it makes sense to

work in that direction. Although it is difficult to imagine a totally non-textual interface for EdenLisp in the formation of definitions, one can envisage many short-cuts that can be used in creating geometrical forms without the need for much typing. The objections relating to the user interface are not therefore insuperable. Whilst not trying to defend the current situation, one can perhaps reflect that many interface languages to popular software are also rather obscure. Most CAD systems have macro or language facilities that defy easy entry. The popular Matlab provides another example of the difficulty in making specifications.

10.42 Actions

Actions are crucial to the definitive system. Without them the system is only a "graphical spreadsheet". Whilst an extremely useful tool, it would be a mere extension of parametrics, somewhat in the manner of Design View. The significance of actions is that they cause redefinition indirectly. An incremental change in a script is easy to do by entering a new definition in place of an existing one, but to make a massive change that might be required to achieve a desired state might require a significant amount of redefinition, much of it of a tedious and repetitive character. The role of an action is to carry out redefinition, activated by means of a trigger (the guard in a concurrent system). The trigger is usually a variable that when changed by a redefinition input by the user (or an agent) causes the action to take place. Thus a single change of variable might be responsible for setting off wholesale changes in a script; indeed it might even cause groups of scripts to change! Such changes are obviously dangerous. The constraints explicitly set by an agent are under the control of that agent, but actions unwittingly set off by triggers of which the agent is unaware could cause chaos. These are the problems of what are called autonomous agents: those set up by the computer to do things like change the screen layout. Clearly much will need to be done to keep tabs on such autonomous actions!

In EdenLisp the approach taken to that problem of actions is to make the triggers explicit. Redefinition is actioned by calling specific EdenLisp functions. The user is therefore always aware of the action being taken. However it is not at all clear at this stage whether that will continue to be the case when concurrency is implemented. There will have to be some further work to deal with that.

10.43 Multi-Agent systems and Concurrency

The problems of the windowing or agent oriented environments discussed in chapter 6 need to be addressed in order to develop the method for multi-agent scenarios. In the paper [Adzhiev, Beynon, Cartwright, Yung, 1994*a*] the issues that are identified relate to the development of the Abstract Definitive Machine (ADM): to the ways that agents must interact, how to structure the kinds of changes that are permitted within and between scripts so as to allow for the following.

- Parallel redefinition among a group of scripts to permit several agent actions to occur simultaneously,
- Automatic detection of conflicts, such as when two agents attempt to redefine the same variable the ADM prompts the user to resolve the situation.
- Exceptional privilege of super user, allowing interaction to direct the computation as an independent and unspecified agent within the ADM.

In practice the structure will need to be very complex. Agents may work in several different modes: they may be free to develop their own designs independently or one may have a leading role and constrain others.

It is the responsibility of a Design Co-ordinator as Super-User to make decisions at all levels. Milestones in a design would then correspond to high level decision points, by analogy with a design conference of all agents operating at that stage. In principle different sets of agents can be involved at different milestones. After such decisions, the specification becomes fixed and all affected variables inherit that information.

Between milestones, agents would be autonomous; they could experiment with their own scripts. Those independent experiments, corresponding to a kind of design folio on the part of the experimenters, must eventually yield a proposal for the modification of the virtual prototype that furthers the actual design. The Co-ordinator' s Role in this experimental stage would be to

- direct patterns of communication to settle ways that agents relate common variables to one another between the milestones (e.g. what material to use could vary)
- identify what is to be fixed, establishing goals at milestones that select from the variant scripts generated in each agent' s design folio.
- estimate progress and degree of consensus

- impose constraints and schedules to build up the components so as to synthesise the design in a "bottom up" manner, and hence progress the design of the product.

It is hoped that the development of a multi-agent system would realise the dream of a truly concurrent design approach, providing a single unifying model (the "virtual prototype") for investigating many views, and using computing metaphors that are consistent between different modelling needs.

10.5 Conclusions

1. *Engineering Design is based upon observation and experiment. Conception is associated with heuristics, abstracting on the basis of incremental behaviour. Invention appears to require the ability to accept the common experience of others only provisionally, being prepared to make new relationships that lead to novel designs.*

2. *The servicing of conceptual design has been done historically by experiment on prototypes - typically physical models that capture information in such a way that new observations can be made.*

3. *Computational models are symbolically based and therefore have a form that cannot capture content except in a preconceived way. Symbols have the power to suggest content well beyond their internal application and that is the basis of much computer aided design. The user may be provided with an empty space but with pre-conceived, but suggestive, symbols around to stimulate and support.*

4. *The computational method based on definitive methods provides that empty space in the form of latent states, where the forms and symbols are invested with significance by the user on the basis of experiment, rather than being pre-conceived.*

5. *The state of a definitive script is that of the incomplete exploration of possible behaviours and relationships. Incremental changes in the script are metaphors of physical behaviour. A set of scripts can be thought of as a virtual prototype with similar properties to a physical one: experiments can be performed, autonomous agents can act concurrently on the prototype, new information may be extracted by "what if ...?" explorations.*

6. *The virtual prototype leads to a single unifying model for investigating many views, with metaphors that are consistent between different modelling needs*

7. *EdenLisp represents a definitive notation that captures the ideas described above. It illustrates in principle the feasibility of virtual prototyping by a single user; it has the potential to develop a multi-agent approach via the windowing environment.*

8. *Actions in EdenLisp enable definitive scripts to be created and edited. They can cause incremental or massive changes in scripts that reflect substantial but predictable behaviour in the object world. New objects can be created, new relationships formed and different modelling environments can be called. The power of actions is immense and checking is likely to be a significant area for future work.*

9. *EdenLisp shows that it is possible to link definitive notations to commercial and other programming approaches in a synergistic way: the designer is aided with routine work whilst freed to carry out experiments of a conceptual kind.*

10. *The experimental approach is educationally interesting. It deals with both discrimination and generalisation of declarative material, cognitive processes that are the easiest stages of learning and which stimulates procedural thinking. In the EdenLisp environment the student of design is encouraged to think in a generic way, leading to creative and fruitful products.*