A KNOWLEDGE BASED SUPPORT TOOL FOR THE EARLY STAGES OF ELECTRONIC ENGINEERING DESIGN

DEAN GRANT CURTIS SCOTHERN

A thesis submitted in partial fulfilment of the
requirements for the Council for National Academic Awards
for the degree of Doctor of Philosophy

September 1991

Polytechnic South West

IN COLLABORATION WITH
THE UNIVERSITY OF READING.
and
PLESSEY SEMI-CONDUCTORS,
ROBOROUGH,
PLYMOUTH.

Acknowledgements

I would like to thank both my supervisors Professor Keith Baker and Phil Culverhouse for their sterling support and encouragement during the period of research and patience throughout the long write up.

Thanks also to all the members of the PEDA project team including Professor J.St.B.T Evans, Ian Dennis, Pat Pearce, Peter Jagodzinski, Linden Ball and the late Gill Venner. The continued advice and stimulation of these colleagues proved invaluable in creating a rich and friendly research environment.

I would also like to acknowledge the financial support given by the National Advisory Body, without whom the present programme of research would not have been possible.

Finally I would like to thank the staff of the Computing Dept of R.N.E.C Manadon for the use of a computer during this write up, and my friends the Robs, Rich and Ian, for the long suffering I have subjected them to.

Declarations

1) Whilst registered for this degree, I have not been a registered candidate for another award of the CNAA or other University.

2) The present research project was funded by a National Advisory grant originally awarded to Professor K.D.Baker and Dr G. Sullivan in 1986. On their departure to Reading University at the end of that year, the grant was taken over by Mr P.F. Culverhouse, Professor J.St.B.T. Evans, and Dr P.D. Pearce. The funding awarded to the present researcher extended over the period from 1st September 1986 to 30th December 1989.

3) The research project was one of three associated projects funded under the N.A.B grant. Whilst the three projects were motivated by the common desire to develop a software system to aid engineers in the early stages of design, each researcher's work was undertaken as a distinct and separable programme of work. The author of this thesis was concerned with the underlying functionality of the design aid. The other two were: (a) concerned with the nature of the cognitive processes in engineering design with the aim of directing the development of the design aid; and (b) the interface to the system.

4) A course of advanced study has been completed in partial fulfilment of the requirements for the degree consisting of: (a) attendance at selected lecture and seminar series run under the M.Sc. Intelligent Systems course at Polytechnic South West; and (b) attendance at a number of relevant professional conferences and workshops.

Title: A Knowledge Based Support Tool for the Early Stages of Electronic Engineering Design.

Author: Dean Grant Curtis Scothern

Abstract:

A desire to produce a design support system for the early stages of electronic engineering design, has led to the conception of the Plymouth Engineer's Design Assistant (PEDA), pulling together experience from the three fields of computing, psychology and electronic engineering. The basic emphasis of this tool has been to use psychological techniques to analyze the cognitive aspects of designers in action and then make recommendations for design tool improvement.

The results of the complementary psychological research, and other relevant literature are examined and potential avenues to realizing an improving design explored. A new idealized abstract representation of early electronic engineering is proposed, which is more in line with the cognitive needs of designers, thus enabling the production of more capable design tools. The main points of the representation are discussed, and comparisons with other approaches and tools drawn. The abstract representation is then taken and used to form a specific implementation as the core to the PEDA tool. An overview of the PEDA tool is given, followed by a discussion regarding the important aspects of the implementation. Important issues and problems raised during the course of the research are discussed, together with suggestions for future work.

Table of Contents

Chapter 1: Introduction

1. Introduction

1.1. Overall Structure of this Chapter

This introduction provides an overview of the research discussed in the rest of this dissertation. The research has been concerned with the means of producing a design tool for the early stages of electronic engineering design, which would offer improved support for the designer by addressing those parts of that activity which were shown to be important as a result of related psychological work. This process has been split into two parts. The first involves the generation of an abstract representation that captures these important aspects. The second is an implementation of this representation as a software based design support environment known as the Plymouth Engineering Design Assistant (PEDA).

The chapter begins with an outline of the main themes behind the research. This is followed by a general introduction, providing background information to the subject of electronic engineering and its support. It ends with the aims of the research, and a brief outline of the form and content of the remaining chapters.

1.2. Underlying Concerns, Aims and Goals

There are a number of concerns, aims and goals that have had a crucial effect on the work discussed in this dissertation. They are briefly mentioned now to give an insight to the intended overall context of the work.

The first concern was that a design support system should help the designer through a cooperative approach, and that the representation used at the core of such a tool should be explicitly designed with this in mind. The second, intended that cooperative support would be improved through a clearer knowledge of the needs of designers. And thirdly, these needs would in part be cognitive in nature and therefore would be best addressed through psychological analysis of designers in action. These concerns in turn promoted the view that satisfying these concerns would produce tools which were more useful and less complex than the results of many other approaches, by providing functionality which was tailored to the actual psychological needs of designers. The logical result of these considerations was the initialization of work on

the PEDA project overall, and in this case on deriving the abstract representation and its implementation (discussed in chapters 3 and 4), based on requirements derived in part from psychological studies (Ball, 1990).

1.3. General Introduction

This section provides a short summary of design in electronic engineering and its progression from a purely manual, towards an increasingly automated task. This development is shown to be the result of escalating design complexity over the years, and has resulted in a large number of support tools which are simply classified into two divisions. Overall trends in this area are discussed and are followed by a lead into the basis of the PEDA project.

A more detailed discussion of the various aspects of these tools is not reviewed here. This information regarding the specific background details of requirements, representations, models and implementation particulars, has been moved to the applicable sections of chapters 2, 3 and 4 which concentrate on these aspects individually.

1.3.1. The Field of Electronic Engineering

The field of electronic engineering is a wide and diverse domain, covering many areas, from the large to the small, and affecting much of our modern lives. In an engineering sense it has traditionally been divided into two large subdomains, known as digital and analogue electronics. These have been exemplified to the public by personal computers on the digital side, and television & audio products in the analogue field. In addition this separation has been made more marked by correspondingly specialised analogue and digital design engineers.

1.3.2. Design Activity

The process of design is a highly complex activity, that has largely resisted attempts to categorise it effectively. This is shown somewhat by the plethora of different approaches in existence in many domains (There are for example, quite a few

software design methodologies). It can be described however reasonably well in abstract terms as: "largely a process of integrating constraints imposed by the problem, the medium, and the designer" (Mostow, 1985),

For description purposes this design activity is commonly divided into a number of stages. Unfortunately the exact description and positioning of those stages is somewhat open to debate and is often a source of confusion. However a simplistic view adequate for this text, and avoiding any exact definition of any particular design activity or process, would state that a design proceeds from specification to artifact with various hierarchical levels in between, going from abstract to reality in a logical manner. Obviously the real activity is much more complicated than this, but the above description does give the references to the early stages of design mentioned later somewhat more meaning in the overall context.

1.3.3. The Need for Design Support Environments in Electronic Engineering

Design support systems have over the years been introduced to make the task of producing electronic designs easier, quicker and less error prone. In the 1980s tools were used which provided such facilities as schematic capture and logic simulation, offering assistance to design entry and validation stages of design. As designs have become more and more complex there has been a natural tendency for design tools to further aid the designer by incorporating more and more of the design process. This has been shown to be most apparent in the VLSI arena, where the improvements in lithographic techniques have made possible very complex designs involving millions of transistors. Purely manual design of such complex designs, would be very difficult and as a result design tools have been very successful in this area. Further, in the computer design industry the process seems to be accelerating, with the technology directly providing the more powerful computing platforms, which the increasingly more sophisticated design software needs to run on, and in turn the designers using the latest software to achieve reasonable design times of the next generation hardware. The net result is that there has been a rapid growth in electronic design support systems, with the newer systems covering increasingly more aspects of the design cycle, and a gradual movement towards the higher and more abstracted levels of design. The rationale behind this tactic has been the placing of design resources where their impact on design

is greatest (Bunza, Hoffman & Thompson, 1990), promoting creativity, and finding design errors early on, where the cost of correcting them is relatively low. As a result new and improved design tools have been introduced to aid the designer, as the complexity of their designs has increased. This increase in complexity and a desire to minimize the product design time has created the need for more and more sophisticated design aids.

1.3.4. Design Support Environments

A very simple but useful classification of computer based design support environments splits them into two main types (Culverhouse 1988).

The first broad category comprises the conventional toolkit, which is often the combination of low level circuit design systems including schematic capture, together with a number of functional subsystems, for example logic simulation. A large number of these tools exist covering a multitude of areas, for example logical, behavioural and mathematical simulation for both the analogue and digital domains of electronic circuit design. At other levels, support may be provided for the layout of VLSI designs as well as printed circuit board manufacture. An example of this type of tool would be the early Mentor Graphics IDEA 1000 system, commercially available in the middle 1980s.

The second class of tool makes use of Artificial Intelligence based techniques (Winston, 1984, & Harmon and King, 1985) and often incorporates embedded specific expert knowledge to assist the designer in a selected area. These systems take an active role in certain parts of design activity and in those areas can greatly improve an engineer's productivity. In recent times a great deal of effort has been centred on such tools especially in the design automation arena and prototype systems such as the "Design Automation Assistant" (Kowalski and Thomas, 1985) and "VEXED" (Mitchell et. al., 1985) have been described. Good progress has been made, and is likely that these tools will eventually automate a great deal of design activity, perhaps even from the earliest specification stage downwards.

An examination of the literature has indicated that although current examples of both types of system offer much to the engineer they seemed lacking in a number of areas:

Firstly, many of the conventional toolkits tend to be passive user directed

systems, containing little embedded knowledge about the design or the designer. As a result they can do little to automate the repetitive aspects of design, or help the designer be more creative.

Secondly the expert knowledge embedded within the second class of systems tends to be very specific in its content and has dealt with particular design problems in comparatively narrow domains, and not with the more general issues applicable to a wider range of problems. Though this may be due to the difficultly of eliciting this type of abstract knowledge from designers (Evans, 1986).

Another aspect seen often in design tools, has been the tendency to focus on the later stages of design concerning with the validation of a design through simulation or timing analysis, as opposed to the earlier and higher levels of the design process (the "what if" stages). This trend has begun to change with the development of more sophisticated tools, though developments in this area have been tentative. In this respect the knowledge based workbenches tend to cover the widest range of the design process, in an albeit narrow domain, going from behavioural, functional and physical specifications for a VLSI design, to a completed chip floor plan.

In a similar vein, there has been a drift towards completely automating whole portions of the design activity, basically adopting a replacement strategy and stepping away from a cooperative approach to design in which the best aspects of both machine and operator are effectively utilized.

However the most important consideration from the point of view of this work is that the basis of these tools appears to be formed from the desired end problem, for example a CAD based verification tool would be based on the requirements of design verification. In a similar way an automated design tool covering behavioural specification to integrated circuit floor plan, would be based on the engineering requirements of these areas. A more interesting and potentially more rewarding approach towards producing a more effective design tool, would be to derive engineers needs, from a rigorous analysis of designers at work. Such a tool should be more centred towards their needs, than those whose target is a design goal. This analysis has been performed by (Ball, 1990) a co researcher on the PEDA project, and has been used to help derive a useful representation for early design and a pilot implementation in the PEDA system.

1.4. Aims of the Research

As mentioned before, the main aim of the PEDA project was to produce a tool, based upon sound psychological principles and research, that offered support to the designer in the early stages of design, by attending to their cognitive as well as engineering needs. This would be done in part by paying close attention to the results of psychological work due to Ball, (1990) a co-researcher on the PEDA project, and discussed in chapter 2. High level requirements for a cooperative system, from that work would be used as the basis for the internal representation of early design within the PEDA environment. The primary goal of this research was therefore to formulate a representation that was a consistent and logical framework for representing important aspects of early engineering design within PEDA, directed by these requirements.

1.5. Thesis Outline

This chapter has given an outline of the basic motivation and aims of the research, and a short introductory overview of electronic engineering design. The contents of the remaining chapters are now described briefly.

Chapter 2: This chapter discusses the psychological basis of the PEDA project, and shows how the requirements for the abstract representation in chapter 3 were obtained and justified.

Chapter 3: This chapter discusses the theoretical aspects of representing the important aspects of the above design process within a tool. It is divided into a series of sections each concerned with a particular aspect of design activity, forming a link between the requirements and how they might be achieved.

Chapter 4: This chapter examines the PEDA tool and the implementation of the abstract representation discussed in chapter 3 at its core.

Chapter 5: This chapter pools together the work described in the previous chapters and provides a summary and conclusions regarding the work. A section

is devoted to possible further work on PEDA.

The appendices contain an example of the PEDA tool in use, an overview of the various tools and programming languages used in the realization of the PEDA system, and the program code relevant to the main text.

References for Chapter 1

Ball, L., "Cognitive Processes in Engineering Design," PhD Thesis, Department of Psychology, Polytechnic South West, Devon, UK, 1990.

Bunza, G., Hofman, G.and Thompson, E., "Design automation goes concurrent in the 1990s," Electronic Product Design, April 1990.

Culverhouse, P., "Design Tools for Engineers," Polytechnic South West NAB group internal report, 1986.

Evans, J. St. B. T., "Knowledge Elicitation in the Training and Assessment of High Level Cognitive Skills," Report Prepared for the Army Personnel Research Establishment, 1986.

Harmon, P. and King, D., "Expert Systems," John Wiley & Sons, 1985.

Kowalski, T. J. and Thomas D. J., "The VLSI Design Automation Assistant: What's in a Knowledge Base," Proceedings of the 22nd ACM/IEEE Design Automation Conference, 1985.

Mitchell, T., Steinberg, Louis, I., and Shulman, J. S., "A Knowledge-Based Approach to Design," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 7, no. 5, 1985.

Mostow, J., "Towards Better Models of the Design Process," The AI magazine, pp.44-57, 1985.

Winston, P. H., "Artificial Intelligence 2nd Ed," Addison-Wesley, 1984.

Chapter 2: Requirements for Early Electronic Engineering Design: The Psychological Basis of the PEDA Project

2. Requirements for Early Electronic Engineering Design: The Psychological Basis of the PEDA project.

2.1. Overall Structure of this Chapter

The aim of this chapter is to show how the set of requirements for the representation discussed in chapter 3 were obtained. The main emphasis of the discussion is on the justification for the psychological basis of the PEDA project, and some of the results of the work conducted by Ball (1990), the psychological researcher on that project, which are directly relevant to the work on the representation of early electronic engineering design discussed in chapter 3. The psychological work involving studies on the cognitive processes involved in engineering design, is outlined and then a link is formed to the work on realizing design support and the early design abstract representation discussed here in later chapters. Only those areas that are thought to be directly relevant to this dissertation are examined, as a complete account is given in Ball's thesis (Ball 1990). These areas cover a discussion on the cognitive needs of engineers and a set of requirements for a cooperative system that would begin to address them.

The chapter ends with a discussion of other work in the literature which has been found to be applicable, and the overall relevance of the final requirements towards systems which purport to assist the early stages of electronic engineering design.

2.2. Justification for Psychological Basis of the PEDA project.

The PEDA system was initially conceived as a cooperative engineering design support tool that would offer assistance to the engineer designer during the early stages of design. A brief survey by Culverhouse, (1986) had indicated that contemporary design tools offered little assistance to these earlier stages, involving the testing of ideas, comparison and selection of alternatives. Very little was known by the research group on the PEDA project on exactly how such assistance should be afforded. Unfortunately it appeared that the design of computer aided support systems was traditionally an introspective and intuitive method done by the tool creators, who were often the domain experts, ie a "designed by engineers for engineers" philosophy. This

was not viewed satisfactory, as a sound and solid basis was required on which to base the development of the tool. Without this basis the tool might hinder the designer by not taking into account some important and unthought of aspect of design activity. It was logical to reason that this would be best achieved by first understanding the way in which engineers design, a task which lends itself to psychological study. In the literature there appeared to be very little prior work on the underlying processes of design in the field of electronic engineering, but there has been some research on software design processes (eg. Jeffries et al, 1981), though the majority of human factors research has been concerned with the Human Machine Interaction aspects of computers (eg Myers, 1986 or Hutchins, 1985). As a consequence the research by Ball into the cognitive processes involved in engineering design was instigated. One especially exciting aspect of this work, was that a study of engineers solving real world problems, might provide valuable information regarding the engineering design process, and indicate the particular strengths and weaknesses in their designing. This information could then be used in the formation of a design tool, which would be targetted at these cognitive needs, providing the correct type of support that engineers need. This type of support would have the added benefit of being comparatively general in scope, covering a wider range of end problems, than systems whose intelligence was aimed at one or two narrow domains.

2.3. Results of the Related Psychological research

The related research produced a number of findings about the way in which engineers design which have important implications for design tools that address the early stages of design.

The first finding deals with the way in which the engineers initially addressed their designs. Ball found that the engineers tended to adopt a problem reduction strategy in the development of their work. Initially this would involve splitting the main problem into a set of smaller, more manageable subproblems that could be dealt with separately and with the minimum of cross interaction. Each subproblem would then be focused upon in turn, and in the case of the less experienced designers, developed in an essentially depth first manner, completing each subproblem before moving on to the next. The more experienced engineers would tend to adopt a more breadth first

approach to solving the design problems, and would only complete a subproblem to a particular level, before moving on to the next one. Only when all subproblems had been addressed in this way, did these engineers move down to the next level, splitting up the design as mentioned above.

In this aspect the results of the psychological research are in agreement with, the majority of computer based engineering tool approaches. There are a multitude of commercial CAD based tools available, that cater for the hierarchical decomposition of designs into functional modules, aligning very well with the problem reducing strategies adopted. However there is little evidence for the use of a sound psychological basis in the development of such tools, except in software engineering (Jeffries et al 1981), and the similarity between results appears to be due to a fortuitous agreement between intuitive techniques and the psychological work.

The apparent differences between experienced and novice designers, has in the instance of software development tools, been responsible for the appearance of tools which enforce a particular stratagem on the user, one that would hopefully improve the performance of novices, by adopting an idealized approach, for instance the use of breadth first, top down strategies (Jeffries et al 1981]. A problem with this type of tactic is that such techniques may hinder rather than aid the designer, in that there may be additional aspects of the design activity, hidden to casual analysis, that are not faced by the particular approach chosen. The psychological findings suggest that a design assistant should encourage, but not enforce a particular design strategy. This tactic is supported by the tentative observation that even expert designers do not always adopt a rigid regime and may for example expand parts of a design in at least partially depth first manner. In fact this overall theme appears throughout the findings, indicating that the design process is governed by trends. It therefore seems unlikely that user centred design support will be effectively achieved through the use of rigid methodologies, when the underlying activities are at present seen to be so changeable.

The second important finding was primarily concerned with the way in which designers pursue alternatives. Ball found that the electronic engineers studied spent very little of their design time in a search for different solutions to the problems, but instead concentrated on one high level solution. This was found to be true over widely differing levels of expertise, from expert to novice and so appeared to indicate a general aspect of the subjects studied. He attributed the result to the use of a "satisficing" principle by

the designers, in which possible solutions are accepted on the grounds that they are good enough or "satisfactory", instead of using a more rigorous or exhaustive search for the "best" design.

This trend is in stark contrast to our preconceived views as engineers of the way in which we design. Good design practice dictates, that designers initially examine a number of alternative ideas, before selecting a few to pursue further. Ball does admit that the engineers studied may have been performing some form of rapid and hidden comparison, or that the time constraints imposed on the designers during his studies may have influenced their design activities, away from the common engineering conception of an idealized route. This is countered by the observation that if extensive comparisons were being made, then it would be expected that this would be expressed in their verbal or "think aloud" protocols and little evidence was found for this in the early and more abstract stages of design. In the later stages of their design activity however, he did observe a trend by the more expert engineers to produce slightly improved versions upon the main design theme, though this activity was comparatively minor to that expected. (Note: Verbal protocols are a successful psychological tool used to gain access to the thoughts of subjects, by making them verbalize their thoughts whilst solving the problem.)

Ball suggests that designers may suffer from a form of cognitive overload affecting their searching strategies at the higher levels of design where changes have far reaching consequences, but cope quite well at the lower levels, with small changes or "tweaking", where the effects of change are generally far more manageable to the unaided designer. For example: a decision to use an analogue, digital or mixed technologies approach in the initial stages of design will affect all aspects of the design from then on, whilst a small change involving a gain control resistor in the final design will have comparatively little effect on the rest of the design.

The effects of time and other environmental constraints are harder to account for, although a number of different studies made by Ball (1990) with quite different environments produced similar results. The first study involved the use of undergraduate electronic engineering students making a current design log of their final year project, together with video and sound recordings of their activity reports. A second set of studies involved video and sound protocol studies of electronic engineers solving a particular set design problem, and employed a time pressing environment, as this was

the only way of reasonably collecting the verbal protocols during design. Unfortunately if the study had taken place over the normal time scales of a product, then the resulting data would have taken many man years to collate and analyze. Ball suggests that the results are still applicable though, as a time pressured environment is probably more indicative of real life design problems on projects that are working to deadlines, and incidentally where a design support tool would be used to most effect.

He follows on and states that the issue of the pursuing of alternatives is further clouded, by the problems involved in determining the relative optimality of a design, and it is quite possible, that some of the engineers could have produced the "best" design, from their "satisficing" approaches. Unfortunately whilst designs can be judged on "satisficing" grounds in that they do or do not meet the design criteria, the determination of the "best" design is somewhat subjective and based to some degree on the expert designers opinions, with the attendant biases which that might entail. Fortunately these problems were somewhat avoided, by the observation in this case, that the majority of solutions provided in the second study were quite different, and so although one design could be the most optimal, it is unlikely that all the others were as well. This suggests that the methods employed by the engineers were not geared to producing the "best" design, but rather a satisfactory one.

From a design standpoint, the lack of explicit alternative solution generation may be very important, considering the very high demands set on designers today, although it must be added that the "satisficing" approach may be the reason why designers can produce effective and reliable designs within a reasonable time scale, considering the complexity of even small problems. Ball indicates that some of probable causes of the "satisficing" tactic might be the cognitive limitations associated with the finite size of human working memory, in that we tend to have problems keeping track of a great deal of information at once, and it seems reasonable that the "satisficing" route might reduce the number of variables that the designer had to consider at any one time to a manageable level. This can be taken as a very good case for some form of computerized assistance regarding the management of alternatives, as a means of overcoming these working memory limitations. In addition the fact that engineers already possess some form of rudimentary selection system, indicates that a system that superintends this activity, by perhaps making it more explicit and accessible, may improve the performance in this important area. It can also be seen that some form of

automated comparison scheme is desirable to help compare different solutions in an unbiased and consistent manner.

The selection of alternatives is an area that contemporary design support tools do not seem to address, and whilst there are many schemes for the representing of alternative solutions or versions (Chou & Kim, 1986), (Gabbe & Subrahmanyam, 1987) and (Katz, Anwarrudin & Chang, 1986), there appears to be little effort centred on the selection of alternatives within the framework of electronic engineering based design tools. Constraint based systems exist (Chan & Paulson, 1987), but the emphasis tends to be towards design and not comparison processes. This may again be due to the way in which tool requirements have been traditionally gathered, by the use of such techniques as retrospective and intuitive analysis of the problem domain or questioning of target users. Ball draws attention to research that indicates that such methods may give unreliable results and suggests that systematic psychological techniques (eg. protocol analysis) are more scientific (Nisbett & Wilson, 1977 and Evans 1986).

Another outcome of the studies is the suggestion by Ball that there is some evidence to support the view that the subject engineers were creating and then using mental models (Johnson-Laird, 1983), to simulate the behaviour of different aspects of the designs as they evolved. For example, what would happen at the outputs if a certain set of inputs were applied to a design? Interestingly, the same type of modelling could be potentially used to evaluate the usefulness of alternative designs for a particular problem. Ball suggests a tentative theoretical model for the processes in engineering design that involves the use of a generalized high level "design schema", an entity which contains knowledge which is applicable to a wide range of similar problems. The "design schema" controls the partitioning and decomposition of the problem into subproblems, together with the use and evaluation of possible solutions at each level, and is in effect the coordinator of the designing activity. He then proposes that the basis of the "design schema" is in fact the problem reduction strategy mentioned earlier. If this is superintended with the means of efficiently providing domain specific technical knowledge, then the core of a fairly sophisticated entity, that can be used to provide reasonably expert solutions to a wide range of design problems, is produced.

This model may have some important ramifications for the generation of design aids. Firstly, if it were possible to implement the model within an aid, it could then be used to provide the basis of the assistance. It's internal state would be indicative of the

user's state and therefore could be used to provide contextually based assistance relevant to the user's need at the time. This model could also be used as part of a training system that "taught" novice engineers how to design more effectively, by comparing their design activity to its internal representation, and offering advice on what they should be doing next or possibly intervening in some subtle, but calculated manner, with the aim of altering their "design schema".

Secondly, if the user were substituted for the problem reduction strategy aspect of the model, then reasonably expert solutions could be produced, by correctly linking the required domain specific knowledge to the user, in effect producing a cooperative system where the best aspects of both machine and operator are effectively utilized. This is a desirable approach as most designers are quite capable of applying such a strategy, whereas it would be difficult and unnecessary to encode efficiently in an automated form at present. As a result engineers might be able to produce more optimal designs in an area away from their specialities. In addition the availability of a general purpose model applicable to other design situations, involving different problem domains should theoretically speed the development of design assistants in those areas, or more ideally, allow the creation of a general purpose design assistant.

A fourth major outcome of the psychological research relevant to the generation of a computer based model of design, is related to the inconsistencies, omissions and errors made by the engineers in their design activities. Errors can occur when the designer is trying to form some understanding of the problem.

An example of this may be seen where some of the engineers studied inadvertently created new incorrect relationships between mathematical parameters. Normally trigonometric functions like sine are associated with angles, but in this problem this was not the case. Unfortunately the designer was given other cues that tended to reinforce the angle association, and one of these involved the use of the irrational number pi in the equation definition for the variable that would be used in the sine function. The error that occurred, happened when the designer was deriving some other equations that involved the use of other trigonometric functions which did involve angles. An erroneous substitution was then made using one these angles in the given equation.

For example:

Now initially:
a = sin(x)
x = w.k/pi

where: w is a length, and k is a constant

now:
theta = arcsine(l/a) :- a user derivation

later:
w.k/pi = theta :- an erroneous substitution

A possible explanation for this was that the engineer had failed to build up a complete representation of the equations. This was supported by the fact the engineers involved who displayed this tendency, spent the least time in initially understanding the requirements of the design problem, and also spent the least time referring back to the specifications as the design progressed.

Errors also occurred subsequent to this representational or encoding phase, when applying design knowledge to generate, combine or even evaluate possible solutions. One example involved a designer who was performing an expansion of a partial solution, and failed to include parts of the original partial solution in the expanded version. Additionally inconsistency problems arose, with mathematical notation in a number of design solutions, where the same variable name was used in two equations at different points in the design work. This caused problems later when the wrong equation was substituted in another equation.

For example:

Now initially:

theta = arcsine(l/a)

a angle usually called theta
is the inverse sine of the
Opposite divided by the hypotenuse and later:

theta = arccosine(j/l)

another angle,
which will be called theta as well

And finally:

a = tan (theta)

Now substitute for the wrong theta,

therefore: a = tan (arccosine (j/l)),
instead of: a = tan (arcsine (l/a))

Ball attributes these types of errors to a number of causes: working memory limitations are blamed, when the designer is concentrating on several items of information, and errors occur, for example the equations; and a lack of vividness of the problem information may have caused the problems regarding incomplete formation of an internal representation of the specification.

This is an aspect of design which apparently has yet to be tackled to any great extent by design support environments. This may be due to the fact that most design tools tend to address the later stages of design, where any equations have already been formulated and checked for correctness. Simple mistakes, dependent on where they happen in the design, may cause problems when discovered, causing delays due to redesigns and may even invalidate the whole design solution, a result which would not be acceptable in the fiercely competitive design market of today. Additionally there are problems to do with the safety of critical systems, that occur when an ok, but mathematically incorrect design, is a key element in a complex system, for example an aircraft control system, where the reliability of such systems is an very important issue.

It seems, therefore, that there is a great need for some form of assistance to address this type of cognitive problem, especially in the earlier stages of design, where mathematical manipulation and equation generation is generally carried out. The benefits would hopefully involve reduced design time and greater throughput, due to the elimination of simple mistakes early on in the design history.

In conclusion, the psychological research, although tentative, has produced a set of findings regarding the design strategies and cognitive limitations of designers which were generalisable over very different tasks, time scales and skill of the designers. This work is in general agreement with recent research in mechanical engineering by Ullman, Dietterich and Stauffer, (1988), who have produced a similar set of findings regarding the activities of mechanical engineering designers. These encouraging results have enabled Ball to produce a set of very general requirements for a design support system that would be targetted at the cognitive needs of designers. The requirements, shown overleaf, are quite general, but provide valuable pointers to the creation of such a system.

2.4. Requirements for a Design Support System

The five general requirements are itemized below, together with an individual description relating their relevance to design support.
A design support system should:

(1) "Encourage the designer to consider an increased number of initial high-level solution concepts and enable the efficient formulation of alternative versions of each solution concept through levels of increasing design detail."

(2) "Assist with the choice of competing design solutions, for example, enabling evaluations of solutions to be made on the basis of comparative functional simulations."

(3) "Superintend the designer's exploratory activity, for example, helping the designer to backtrack if a path proves unpromising (i.e. by providing a record of paths taken together with the current point of exploration) or suggesting worthwhile paths of investigation (i.e. by suggesting design alternatives)."

(4) "Ensure the designer's awareness of design conflicts (e.g. if crucially important constraint requirements have been overlooked when the designer is focusing on a narrow aspect of the overall design solution)."

(5) "Ensure the designer's awareness of inconsistencies in the notation that is being used (e.g. if two different design parameters have been given the same symbolic label)."

The requirement (1) arose from the observation mentioned in the previous section, that designers tend to use a "satisficing" principle in which they focussed their

efforts on a single satisfactory high level solution path. It was therefore thought that encouraging a more explicit approach, in which options are more thoroughly investigated, could lead to improved design performance.

Requirement (2) arose from the same observations as (1), but the emphasis is on making the selection between alternatives easier, by allowing the user to apply a set of selection criteria to a whole set of alternatives. This activity would be a difficult and error prone process if done by the user unaided, and whilst it would be wrong to say that, making such a scheme available, will make the user pursue alternatives to a greater extent, it can however, make the proposition more attractive by reducing the cognitive burden in this area. In addition the problem of implicit biases affecting the choice of alternatives is somewhat alleviated with the introduction of an explicit checking scheme. The designer can now be made aware of which constraints are being applied. Similarly less time should be wasted on pursuing "dead end" designs, by making sure that all the relevant constraints are applied in a consistent manner to the selected alternatives, at an early stage of the design activity.

Requirement (3) is also designed to help with the pursuing of alternatives, by reducing the amount of information that the designer has to keep in working memory. The information held would hopefully allow the designer to retrace his or her steps, up from an unsatisfactory solution, regaining the information that was valid previously, before the unsatisfactory alternative was examined.

Both requirements (4) and (5) are associated with the mistakes that designers can make during their designs. A designer can loose consistency with various parts of a design, when concentrating upon a particular aspect, and so needs to be alerted to the fact. The designer also needs to be alerted to inconsistencies in the notation of the solution, be it mathematical or otherwise.

2.5. Other Relevant Work

The aim of this section is to outline other work which has significantly influenced the selection of the requirements stated earlier, as being suitable for the formation of the abstract representation and implementation discussed in chapters 3 and 4.

Most design systems are built to satisfy a set of explicitly or implicitly specified

requirements and so the number of potentially influential sources is quite high. However there are comparatively few which are applicable to cooperative early engineering design.

In the area of Co-operative interface management M. Smyth, (1988) outlines a set of requirements for a cooperative system, which are quite similar to Ball's in electronic engineering:

(1) Increase the number of initial design solutions.
(2) Reduce the time and cost of the design process.
(3) Increase the number of design iterations where necessary.
(4) Increase the designers awareness of potential design conflicts in the proposed solution.
(5) Move the solution range closer to the theoretical optimal solution.

A similar view is taken in the Mechanical domain by Ullman et. al. (1988) who makes a number of recommendations to improve CAD systems:

The first is that we should raise the abstraction level at which computer based tools can provide external memory aids for the designer.
Secondly, tools might also be extended by providing some means of constraint management assistance.
Thirdly, there is a general need for CAD to support the human designer's cognitive limitations.

A very useful and often cited paper by Mostow (1985) suggests the areas that a comprehensive model of design should address.

In short these are:

1) The State of the design.
2) The goal structure of the design process.
3) Design Decisions.
4) Rationales for design decisions.
5) Control of the design process.
6) The role of learning in design.

It can be seen that there is a degree of commonality between the sets of requirements. Both Ball and Smyth are in broad agreement, and Ullman makes overall suggestions that the other two detail. The last set of suggestions due to Mostow, appear to differ, but in fact are pitched at a different level, being concerned more with the structure of the internal model than the tool. The overall concern of all these approaches has been to produce better systems by taking into account more of the human design process than just the state of design.

2.6. Relevance To Design Support Systems in Electronic Engineering

Very little psychological work has previously been carried out on design in electronic engineering, and it appears that introspective and intuitive techniques have been used to formulate the specifications for many design tools in this area. Unfortunately, psychological evidence has indicated that these techniques can be ineffective (Nisbett & Wilson, 1977 and Evans 1986), and so there has arisen a real need to determine accurately the needs of designers. The research done by Ball has been successful so far in that it has produced a set of requirements, that should address some of these needs. Some aspects of design activity have also been addressed by contemporary tools. These aspects will be covered in the next chapter where the requirements are linked to the derivation of an abstract representation for the early stages of electronic engineering design, but in brief they can be roughly divided into two areas. Firstly, non intelligent tools which tend to be reasonably general, but normally support the later stages of design, for example from circuit design solution to

printed circuit board manufacture. Secondly, intelligent tools based on expert system approaches, which cover more of the design activity, but tend towards tackling a specific design problem domain, such as a digital filter, VLSI chip or mechanical linkage design, for example The VLSI Design Automation Assistant by Kowalski et al ,1985, in which the intelligence is aimed at the automatic decomposition of functional and behavioural specifications in VLSI designs. One result of this directed effort, is that the intelligence has been aimed at the target problem, instead of the cognitive problems of the designer. It was with this in mind that the PEDA system was conceived, and by basing its design on sound psychological research on the cognitive processes in design, it was hoped that it would be more able to address the general needs of designers, rather than specific problem domains.

References for Chapter 2

Ball, L., "Cognitive Processes in Engineering Design," PhD Thesis, Department of Psychology, Polytechnic South West, Devon, UK, 1990.

Chan, W. T. And Paulson Jr, B. C., "Exploratory Design Using Constraints," AI EDAM, pp 59-71, 1987.

Chou, H. and Kim, W., "A Unifying Framework for Version Control in a CAD Environment," Proceedings of the Twelfth International Conference on Very large Databases, pp. 336-344, 1986.

Culverhouse, P., "Design Tools for Engineers," Polytechnic South West NAB group internal report, 1986.

Evans, J. St. B. T., "Knowledge Elicitation in the Training and Assessment of High Level Cognitive Skills," Report Prepared for the Army Personnel Research Establishment, 1986.

Gabbe, J. D. and Subrahmanyam, P. A., "An Object-Based Representation for the Evolution of VLSI Designs," Artificial Intelligence in Engineering, vol. 2, no. 4, 1987.

Hutchings , E. L., Hollan, J. D. & Norman, D. A.,"Direct Manipulation Interfaces," Human Computer Interaction, Vol. 1, pp. 311-338, 1985

Johnson-Laird, P. N., "Mental Models," Cambridge: Cambridge University Press, 1983.

Jeffries, R., Turner, A. A, Polson P. G., and Atwood, M. G., "The Processes Involved in Designing Software," in Cognitive Skills and their acquisition, ed J. R. Anderson, Hillsdale, NJ.: Lawrence Erlbaum associates.

Katz, R. H., Anwarrudin, M., Chang, E., "A Version server for Computer-Aided Design

Data," Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986.

Mostow, J., "Towards Better Models of the Design Process," The AI magazine, pp.44-57, 1985.

Myers, B. A., "Visual Programming, Programming by Example, and Program Visualisation: A Taxonomy," Proceedings of ACM/SIGCHI, pp.59-66, 1986.

Nisbett, R. E., & Wilson,T. D. "Telling more than we can know: Verbal reports on mental processes," Psychological Review, Vol. 84, pp. 231-295, 1977.

Smyth, M., "Articulating the Designer's Mental Codes," LUTCHI Research Centre internal paper (draft) ref: HCC/L/24, 11th May 1988.

Ullman, D. G., Dietterich, T. G., and Stauffer, L. A., "A Model of the Mechanical design process Based on Empirical Data," AI EDAM, vol. 2, no. 1, pp. 33-52, 1988.

Chapter 3: An Idealised Representation for the Early Stages of Electronic Engineering

3. An Idealised Representation for the Early Stages of Electronic Engineering Design

3.1. Overall Structure of this Chapter

The aim of this chapter is to present an idealised abstracted representation for the early stages of electronic engineering design, that has been devised during the development of the Plymouth Engineer's Design Assistant (PEDA), a tool designed to aid and complement the designer in the early formative stages of design activity. The chapter begins with the rationale and arguments for the creation of the representation, and then leads on to a full description. The various parts of the representation are outlined and developed into a characteristically simple structure of alternative designs and constraining information. A critique then follows, comparing the high level idealised representation with other methods, models, approaches and environments in electronic engineering and related domains. The chapter concludes with an outline of the salient aspects of the representation in preparation for the next chapter, which describes a partial implementation as the core within the PEDA environment.

3.2. Rationale for a New Representation

The objective of this section is to provide the rationale and arguments for the creation of a new representation for the early stages of electronic engineering design, discussed in section 3.3.

This is done primarily through an examination of the psychologically derived requirements in chapter 2. These requirements are individually scrutinised generating a number of issues that need to be addressed if the requirements are to be dealt with adequately. Methods of achieving these aims are discussed, and where they exist examples from the electronic engineering design domain are taken and are shown to be generally unsuitable. Overall representational issues that affect the choice of a representation in this area of the domain are also outlined. All these points are then combined with the arguments behind the PEDA project, to state the claim for a new representation in section 3.2.3.

3.2.1. An Examination of Requirements

Each requirement from chapter 2 can now be examined in turn, to see how they can be met, and to indicate the unsuitability of some common solutions in the literature.

3.2.1.1. The First Requirement:

The first general requirement for a design tool, taken from chapter 2, is as follows:

"Encourage the designer to consider an increased number of initial high-level solution concepts and enable the efficient formulation of alternative versions of each solution concept through levels of increasing design detail."

There are a number of issues that need to be addressed here before the above requirement can be realized. The first and most basic one is that a representation for a design needs to be found. Initially this may seem a comparatively easy operation as it is not specified within the requirements and as a result there is considerable freedom on how it may be realised. The literature abounds with design representations and so there should be little difficulty in selecting a suitable candidate. Unfortunately it is in this area that a number of problems arise, which stem from the basic conceptual emphasis of trying to meet these requirements in a simple but elegant manner. Ideally a representation of design is needed which is uniform in its structure. This is not only desirable from a aesthetic point of view, but more importantly is in line with the results of the psychological studies (Ball, 1990) in which designers tended **not** to separate the domains of description (unlike the proposals by Stefic et al, 1981), but merged them in the early stages of electronic engineering design. Sadly this reduces significantly the number of candidate representations from the design automation arena. Another meaningful problem was that representations are generally discussed in either low level terms, using a language such as LISP (for example Davis & Shrobe, 1983), or in very abstract terms (for example Sinclair et al, 1989), and cause problems regarding the level of design detail. This is an important issue covering the whole of the representation and is discussed briefly in section 3.2.4 and in section 3.4 when comparing the representation with other approaches.

In essence, what is actually required, is a simple representation for the design that captures the important aspects of the design in the early stages. A hierarchical structure is desirable as this neatly captures the normal design representation by designers in their work (Ball, 1990). One credible solution would involve the structure being made up from a collection of blocks that would together form a block diagram, very similar to the visual representation on paper. These blocks would be connected in the horizontal plane by connections allowing the desired mathematical functionality to be built up and along which information or data would be seen pass by the user during simulation. The blocks themselves would contain all the information, necessary to describe what they are, and their relationships to other blocks including information would be relevant to the requirements of the design.

The second important issue raised by the first requirement deals with the representation of design alternatives. Again this appears to be a greatly addressed area, but unfortunately the effort tends to concentrate on aspects such as "version serving". As a result the facilities offered by these approaches are not particularly relevant to the needs of the designer regarding alternatives in the early stages of design. For example approaches may offer means of keeping track of the most up to date parts of design in different representations between the members of a design group (Katz et al, 1986 and Gabbe & Subrahmanyam, 1987), but do not aid the designer in exploring new designs. In addition the overall trend with these approaches is to rigidly support the separation of domains theme and so it would be difficult to reconcile them with the representation of designs mentioned earlier.

Actually alternatives can be addressed in a very simple manner by treating different alternatives as separate block diagrams. Additional information can then be added to describe the relationship between diagrams in the same way as between different blocks. This method of treating blocks, block diagrams and alternatives in a homogeneous manner is very attractive as it allows mechanisms developed for one aspect of the representation to be used on the others. This will be seen to be extremely useful when constraints are discussed later. Also having only one scheme greatly simplifies the representation, because then there is no need for transformations between the various design domains. These transformations are generally needed (Walker, 1988), because of the differences (non isomorphism) between the domains in the later stages

of design. This non isomorphism makes the different domain views of the same design topologically different, and thus to maintain interdomain consistency, the transformations are required.

The two aspects, designs and design alternatives mentioned above, go some way towards addressing the second statement of the first requirement, leaving the first. In examining this it is reasonable to assume that actively encouraging designers to pursue design alternatives is primarily a user interface function, however the representation has an important task also, by providing a simple and clear means of portraying alternative designs. In which case, any system that makes the representation of alternatives easy and, or automatic could be viewed as encouraging the user to use them when compared with systems which do not provide any high level support of alternatives. This line of reasoning could be extended to consider an alternative management system which analyses the user machine interaction to determine if the designer is working on a different approach to a problem, and then handles the creation of new alternatives accordingly. Such an approach would go some way towards encouraging the user to explore new design concepts, whereas a "version server" which maintains historic consistency between the different parts of a design would not.

3.2.1.2. The Second Requirement:

The second requirement taken from chapter 2 states:

"Assist with the choice of competing design solutions, for example, enabling evaluations of solutions to be made on the basis of comparative functional simulations."

Two important questions that need to be answered before this requirement can be successively tackled are: What means can be used to distinguish between different designs; and what methods can be used to compare them?

There are many ways of differentiating between designs, limited only by the types of information used. This may be in the functionality of the designs themselves, in that an adder is different from a multiplier, or in other ways, for example this design was produced in 6 weeks and that one in 6 months. The only real limits to this general statement are the relevance of the information to the task and the difficulty in generating that information. In this way any design or designs could be conceivably be

compared on the basis of **any information** which was instrumental to their existence. This is a difficult requirement for any system, and can only be reasonably achieved through reducing the information required to a manageable level.

Each piece of information can be viewed as a constraint, though not totally in the sense of constraint satisfaction, for example where unknown elements in an equation are derived from known values (Tong, 1987), or where designs are synthesised that meet constraints (Chan et al, 1987), but in a more general vein. With this approach any information contained in a design is regarded as a constraint in that it tends to make the design more specific, and therefore constrains it in some way. For instance if there are no requirements other than: "Make something" then the design can be any object, whereas if the design is already an adder then it is not going to factorise easily. Constraints can be classified into many different areas, the following paragraphs outline a few of them relevant to the requirements.

Constraints may be split into explicit constraints, which are specified within a representation, and implicit constraints, which are internalised within the designer. There has been a case to make this second form of information explicit (Mostow, 1985 and Ball, 1990) with regard to aiding the designer. From a practical standpoint this would make the information accessible to all, allowing others (including a design tool) to inspect them for consistency, correctness, or relevancy, and would help prevent them from being forgotten, by reducing the amount of information held in the mind of the designer at any one time (Ball, 1990).

The constraints may be stated requirements, or actual attributes of some stated objects, for example the functionality of a block. This distinction affects the way in which they would constrain a design. Attributes always constrain a design because they describe it some way, and requirements are potentially constraining because they describe some desired aspect, that the attributes should meet. In many cases with partially completed or incorrect designs, this will not be so, and here is where the above usage of constraints differs from many others. For example where constraints are used as blueprints to automatically synthesize possible designs. (Gupta, 1988, and Director et al, 1982).

Another important distinction is that made between domain and non domain constraints. Domain constraints can include such aspects as the design functionality (and behaviour), which can be viewed as some sort of ultimate constraint, and aspects such as device or packaging physics, or power consumption, chip area and speed. Non domain constraints can be anything else which may effect the designs, and can range from when the final design solution is required, to the stability of the design requirements.

Two other forms of constraints can be defined as "musts" and "desirables" and indicate the relative importance of requirements. (mandatory and advisory constraints: Popplestone et al, 1986) An example of a "must" is the mathematical behaviour of a design, perhaps found through analysis or simulation, and cannot vary. On the other hand a "desirable" is just that, an aspect of a design which is desirable, but not absolutely necessary, perhaps being indicative of a better design. An example of both type of constraints would be: The power consumption of the circuit must be less than 5W, but a value less than 2W is desirable.

Two main aspects of all these constraints are apparent. The first is that they reside in the part of the design which they address. In this way, if a constraint affects the top level of a design hierarchy then it is stated at that top level. Secondly the use of constraints is hierarchical in nature, hence many constraints at one level are applicable to lower levels. For example if a design has to be ready by next week, then all parts have to be ready then too. Certain constraints will propagate up the hierarchy (Ball, 1990), consequently a change at a lower level may affect the ones above it in the hierarchy. This type of constraint does however tend to be abstract in nature (Ball, 1990), for example the choice to use a particular technology because it will produce the results wanted (fast enough). In a way this aspect of constraints is a form of constraint propagation, where changes in one place cause a corresponding change elsewhere (Chan & Paulson, 1987). As constraints are changed, heuristics can be used to fill in incomplete information at the various levels in the constraint hierarchy, for example a specified requirement for a design states a maximum delay in a particular design block. The design of that block has been adjusted and its overall delay is not known. A system heuristic is invoked, it derives the critical path delay and hence the overall delay for this block.

The final type of constraint mentioned here is the information produced through simulation. This information is of great importance to designers as it is often the only source of constraint information available, indicating that a design meets or does not meet the stated functional requirements.

There are many different approaches to simulation in the electronic design literature and so it should be relatively easy to find one which satisfies the criteria for this representation. A desirable quality of a simulator at this stage of design is that it is as general purpose as possible, so that the designer may examine a large set of problems, and not be limited to a particular approach, for example logical simulation. The use of mathematical equations to express a design's functionality affords a sufficiently abstract solution to this problem. There are no limitations on how the simulator works, and so the data flow architectures are attractive, due to their simplicity and their non reliance on timing constraints (important in the light that accurate timing is not important in the early stages of design (Ball, 1990)). The use of mathematical equations for the functionality, does allow the same scheme to be used at many levels, from calculus at one, to bit arithmetic at another. For example a designer may use integration at one level in their design, but be concerned with rounding errors due to different floating point representation. This method is conceptually attractive as it avoids the use of multiple representation schemes at different levels, a common tactic due to performance considerations. In any event speed of execution is not an issue in the early stages of design, where designs consist of small numbers of relatively complicated functionality.

It can be seen that very little needs to be done to the basic design and alternative representation to realize the representation of constraints in the early stages of design as most often they are just statements about various aspects of design, and therefore can be treated like any other design information within the representation. Only where constraint propagation occurs, is a means of achieving it required. This use of constraint propagation in design is well represented in the literature (Chan & Paulson, 1987, Mostow, 1989, and Hooton et al, 1988), and can be achieved through the use of specialized heuristics. Simulation is seen as a special case of generating constraint information, its general purpose nature allowing it to test many forms of functionality.

The second issue concerned with meeting the 2nd requirement, involves finding

a method of comparing and contrasting designs in a manner which is fast, simple, consistent, rigorous and clear. There are a number of approaches to solving this type of problem. Expert knowledge can be embedded in the form of heuristics, to provide advice on the right choices as a form of consultant. Another approach uses multi-attribute theories which contain no expert knowledge but require the user to ascribe importance to stated comparison attributes. Calculations involving weighted averages can then indicate the more desirable designs (Humphreys & McFadden, 1980). In the first example the system would reason about the problem itself, whilst the second structures the problem so that the designers can use their own knowledge to solve it. Evans (1988) discusses interactive decision aids and suggests a combined approach which falls between these two extremes, which may de-bias decisions somewhat. Conceptually this type of system might use these theories as the core of a decision system, but help build up the attributes through the use of heuristics which would convert constraints into a form suitable for the decision support system. Also additional functionality would be required in the user interface to correctly structure the problems and present them carefully to the designer.

A combined system of this type would be desirable as the basis for a constraint comparison system as it offers a simple means of addressing the 2nd requirement, and is in close agreement with the overall goal of providing a system which aided the designer in a cooperative manner (Ball, 1990 and Smyth, 1988). The often used consultant based approach would require a very large amount of expert knowledge if it were to address a wide domain, and would exclude the designer from the decision process. In a similar vein a solution using a program such as MAUD (Multi-Attribute Utility Decomposition)(Humphreys & McFadden, 1980), whilst a decision aid, contains no intelligence or domain knowledge and would be of limited value (Evans, 1988). The joint approach would if correctly engineered, combine the best of both methods, extending the decision aid with the ability to derive attributes from design constraints.

3.2.1.3. The Third Requirement:

The third requirement discussed in chapter 2 suggests that a design tool should:

"Superintend the designer's exploratory activity, for example, helping the designer to backtrack if a path proves unpromising (i.e. by providing a record of paths

taken together with the current point of exploration) or suggesting worthwhile paths of investigation (i.e. by suggesting design alternatives)."

It can be seen that this requirement is very wide in overall scope, but the examples given do give an indication of what could be done. It is reasonable to suggest that to achieve this type of assistance a representation or model needs access to a representation of the designer's exploratory activity (Mostow, 1985, and Takala, 1989). The most common way of achieving this is through a history mechanism which logs the commands issued by the designer. Obviously such a recording system can operate at many levels, the lowest may be mouse button clicks, higher ones such as design plans, record refinement heuristics used in automating design (Mostow, 1989). At a higher level still we have abstract design decisions, which outline the reasons why a particular decision was made.

The lowest level history trace can simply be regarded as a record of all design activity including both user-tool interaction and internal tool activity, for instance simulation events. With this type of recording the volume of data created in a typical session can be quite large dependent upon the size and complexity of the tool used. In the lifetime of a project the information recorded would be very large indeed. However the vast majority of this information is superfluous to the needs of the engineer in that the information is too low level in content and would rapidly overwhelm the engineer's working memory: the type of thing that needs to be avoided. Information needs to be presented in a form, which can be assimilated and at a level which would be useful. This is where the concept of design decisions arises. These are high level abstractions of design activity, that in this context portray to the engineering user, the important decisions that led to a particular meaningful event, for example: the creation of a design alternative, or its rejection when compared to others. In keeping with the overall concept of the environment, these design decisions relate different designs to the set of criteria that created them, therefore a design decision to use a particular technology in a design may be because of that technology's superior characteristics, or that the design team was more familiar with it. These are at a different level to those described by Mostow, 1989, in BOGART a tool in which design decisions are menu generated transformation heuristics (design plans) that form a design strategy, and can be "replayed" to partially complete similar designs.

The use of design decisions leads on logically to the development of methods

which can extract these high level decisions from the low level history trace. This type of information could be extracted through a variety of knowledge elicitation techniques including: experimental manipulations and inferred cognitive processes; interviewing and self-report methods; repertory grids; rule induction; and observational methods and protocol analysis (see Evans, 1986 for overview). The exact method used in deriving the heuristics is really not important from the point of view of a representation, however in the light of the psychological work done on the PEDA project by Ball (1990) and biases (Evans, 1988), observational methods are favoured. Work towards automating protocol studies on engineers designing (Burton et. al.) may overcome the inherent slowness of these methods and make them a practical technique for knowledge elicitation.

The number of heuristics in this part of the representation may not be great as a totally automated system, because the aim is to merely superintend the designers exploratory activity, and not replace it. These heuristics could be divided into two broad types: the first are purely automatic, producing decisions directly, for example a design was created because it meets the set of criteria that derived its parents; the second prompts the user for the decision at a particular point. Important issues here are concerned with where do the important design decisions occur, and if the information about the decision cannot be automatically extracted, whether to ask the designer about it then, or later. These are important because the representation should not hinder the designer, which it might if it asked the user the reason for every activity. This type of information would be best obtained through knowledge elicitation techniques, preferably psychological analysis (Evans, 1986).

The remaining issue with regard to this requirement that will be discussed here, deals with the placement of the design decisions within a representation. In many systems the history mechanism is treated separately to the other parts of the representation, but here the overall philosophy is to avoid separating the various aspects, where unnecessary. As a result it is conceptually attractive to place the design decisions in the place in which they are most relevant, for example an alternative which was created for a reason, will contain that reason. The advantages of such a system include: simplicity, in that no complicated schemes are required to relate the history to the rest of the representation and secondly; the placement of the history in the representation allows design decisions to be used as constraints in the constraint comparison system

and in turn the rest of a representation can be advantageously examined by the heuristics when trying to extract decisions.

3.2.1.4. The Fourth Requirement:

The fourth requirement from chapter 2 states:

"Ensure the designer's awareness of design conflicts (e.g. if crucially important constraint requirements have been overlooked when the designer is focusing on a narrow aspect of the overall design solution)."

This requirement like the others is so general, that it can be tackled in many ways. Unfortunately the electronic design literature is less helpful than initial thoughts would suggest due to the automation bias prevalent in the field. Typically conflicts in a design would be resolved through a constraint propagation, or truth maintenance system, preventing "illegal" designs from existing in the first place. It is relatively easy however to adapt these schemes to a constraint checking and reporting role. In addition the asynchronous nature of the problem makes the traditional use of rule based systems in this area quite sensible. A simple modification to such a solution making it compatible with the other requirements involves viewing design conflicts as specific examples of the generalised constraints mentioned earlier. These are attributes which constrain a design in some way. In this approach design conflicts may occur when requirement constraints (requirements) that have been inherited from previous designs (in the alternative hierarchy) cannot be met by a particular design (alternative). This may occur for example when overall speed requirements have not been met. By regarding design conflicts as constraints, any part of the constraint system (including simulations, human and domain constraints) may be used in the generation of design conflicts. As a result, design conflicts can conceptually embrace anything that the constraint system can.

With this knowledge two questions become significant: 1) what constraints are important, and 2) How do we determine that they are being overlooked?

The second question is the easiest to answer in that it requires the existence of some means of checking consistency between constraints. This can almost be met through the use of a classic truth maintenance system but where the conventional approach would propagate the effects of a particular constraint, this use would require

a comparison between them: the first would be the result of the propagated constraint; and the second would be the resident constraint. Again heuristics can be used to perform the propagation and comparison function. This could then be used for two purposes: The first would be to signal the user interface to issue a warning to the designer; The second involves the heuristics producing a constraint as a result, which could also be used by the decision support function as a means of differentiating between designs.

To determine which constraints are important is a little harder, as a method of prioritising conflicts is required. This may be achieved with a similar approach to that for requirement 2, which combines a decision support tool such as MAUD (3.6.6), and user input regarding the priority of certain conflicts. By thresholding the output of the tool, only those constraint violations which are deemed important enough would be signalled to the user, and low priority conflicts would be ignored. This approach could be improved, with the addition of heuristics which would automatically generate the priority of design conflicts. This simple method has the added advantage that it uses the same functionality that is used to satisfy the second requirement.

3.2.1.5. The Fifth Requirement:

The fifth requirement from chapter 2 states:

"Ensure the designer's awareness of inconsistencies in the notation that is being used (e.g. if two different design parameters have been given the same symbolic label)."

This is a very similar request to that in the previous section. And again the lure of viewing mathematical errors and inconsistencies as constraints is attractive, but for slightly different reasons.

From a conceptual point of view, it is very appealing to do this as practically everything is now an aspect of the constraint setup, producing a clean and practical system with the attendant real advantages that the resultant homogeneity entails.

The same system as for the fourth requirement could be ideally used to prioritise these new constraint violations and set the level of warnings produced. In this case however heuristics are certainly required which can detect the various mathematical and logical errors produced. Unfortunately the number required to detect all potential mathematical violations would be quite high. Luckily the fourth requirement only

requires a comparatively simple consistency check and so the realization of such a feature as part of the constraint system would be a straight forward exercise. However in chapter 2, mention was made of much wider ranging mathematical problems and therefore it seems that the wider ranging problem needs to be addressed eventually. Considerable research has been conducted in the literature on symbolic algebra manipulation tools, which would likely reduce the incidence of certain mathematical errors by automating many common manipulations, such as factorization or solving equations. However the internal constraints of such a system if included in a representation would be less visible, and therefore the advantages of a homogeneous system could not be afforded to that part of the representation.

3.2.2. Representational Issues Common to all the Requirements

The aim of this subsection is to outline overall issues which affect the choice of the representation at all levels. These are concerned with: 1) How a representation forms a model of the user; 2) What aspects of design should the representation address; 3) What is the target level of the representation; and 4) Should the representation take into account efficiency and other similar issues.

3.2.2.1. Explicit and Implicit Models of the User.

A representation can express a model of the user in two main ways. In an explicit model, the various stages of activity are distinctly stated and there is a correspondence between important user activity and state changes within the model, for example the user is in a particular state. The second and much less powerful implicit model relies on an indirect method. User activity is still recorded, but there is no categorisation into states. The representation described later in this chapter can be regarded as an explicit model of certain aspects of the early stages of electronic engineering design, but an implicit model of the corresponding user design activity. It is based upon a set of requirements, which were in turn derived from an explicit psychological model (Ball, 1990), but it does not contain that user model in any explicit form. There is an explicit representation for designs, but not the state of the user. Whilst an explicit model of the user is desirable (Ball 1990, Mostow 1985, Ullman, 1988 and

Smyth, 1988) in terms of the knowledge gained about the user design process, it is not essential. Many different models of the user have been suggested (see Williges, 1987 for examples in the human computer interface field), but unfortunately their inner structure depends on the desired end goal. Research on psychologically derived explicit models of users is continuing (Ball, 1990), but until this issue is resolved, or a sufficiently general purpose solution found, it is reasonable to concentrate only on implicit solutions.

3.2.2.2. Providing Functionality

A vitally important consideration when creating a design representation is determining what aspects of design it should address. Obviously there is little point in providing support for something that is not used, for instance in studies involving the early stages of electronic engineering it was found that designers did not use time in any absolute sense except as an overriding constraint (Ball, 1990). The net result would be that there is no point in providing a simulation tool for this stage of design that used time delays (for example conventional logic simulation). However in the same studies designers used constraint criteria to choose between approaches: would the design be fast enough, or would it fit on the integrated circuit die? A system that helped them in this area would be used and therefore the added functionality would not be wasted. The requirements and background stated in chapter 2 create a unique set of goals that the desired representation should address. They cause problems for many common design representations, due to their psychological derivation. A representation may address one area adequately, be totally lacking in another, and offer superfluous functionality in other areas. This is neither ideal or desirable. It is proposed that the representation described later in this chapter is a better solution.

3.2.2.3. The Level of Representation

Another important overall concern is the level at which the representation is targetted. To be useful it needs to aimed at the right level. If the approach is too abstract then it is of little practical use, and if it is too detailed then the solution becomes overly complex. The desired goal therefore is to strike a balance, by describing

the functionality at a level sufficiently low to allow an accurate implementation, but abstract enough to avoid complexity or implementation issues.

3.2.2.4. Efficiency and Implementation Concerns

In a similar vein to the previous comment, it is important that the representation should make no concessions to implementation or efficiency constraints. In this way, the organization of the representation should not mirror a fast simulation model (Barzilai, 1986) or a tripartite behaviour, structure and physical model (Walker, 1988). The view is that these aspects are purely in the implementation domain and should be dealt with there. As a result this allows a basic representation scheme to be abstracted away from factors which are determined by the target language or environment.

3.2.3. The Case for a New Representation

A look at the literature on electronic engineering design will discover a large and active research domain. Over the years there has been a great deal of interest in this field. Simulation has remained the primary area, but expansion has occurred to cover earlier and later parts of the design process. The result is that there is now a plethora of approaches concerned with many aspects of design, including for example, design plans (Mostow, 1989), automated configuration of hardware (Bowen, 1985), and switch-level simulation of integrated circuits (Ashok et. al., 1985).

As research has started moving towards design environments, with the overall goal of incorporating all aspects of the design activity, the ability of tools to aid the designer in other ways has become important. Mostow (1985), Smyth (1988), Ullman 'et. al. (1988) and more recently Ball (1990) have suggested that a better understanding of the user design process (activity) might be advantageous, in pointing out what is required. It was in this vein that the PEDA project was conceived and the basic groundwork for the representation and corresponding implementation laid. The flavour of the PEDA project placed particular constraints on the design of the PEDA tool. Firstly, a cooperative tool was envisaged, and secondly its form and function would be based on requirements derived through psychological studies. It has been suggested in

the preceding pages that it would be difficult to meet those requirements through any conventional representation in the electronic engineering domain, and multiple representational approaches tend to be clumsy and complicated. A new representation can be devised that has none of these deficiencies and has many advantages, in that it can address many perceived issues in a satisfactory manner. Taken together these points made a strong case for deriving a new representation. They can be summarized in overall terms as follows:

1) The Psychological Requirements and basic PEDA approach place important constraints on the representation that make it difficult or clumsy to realize using most electronic engineering design representations, as they often do not tackle the areas which the requirements dictate.

2) Other issues such as the level of detail, functionality, target level of representation, efficiency and implementation concerns of example approaches further improve the case for a new representation.

3) Substantial advantages can be realized with a new representation, including: simplicity, non redundancy, and homogeneity, especially with regard to a constraint comparison system.

3.3. An Idealised Representation for the Early Stages of Engineering Design

An examination of the requirements and the available literature regarding electronic design representations in the preceding section, has shown that there is a strong case for deriving a new representation to satisfy the particular needs of the early stages of electronic engineering design. Methods for realizing the various parts of the representation have also been discussed. In this section those suggestions have been taken and used as the basis of an abstracted and idealised representation, that attempts to address all the points put forward, in this and previous chapters, in a simple and elegant manner. A pictorial view of the representation's main components is given in Figure I.

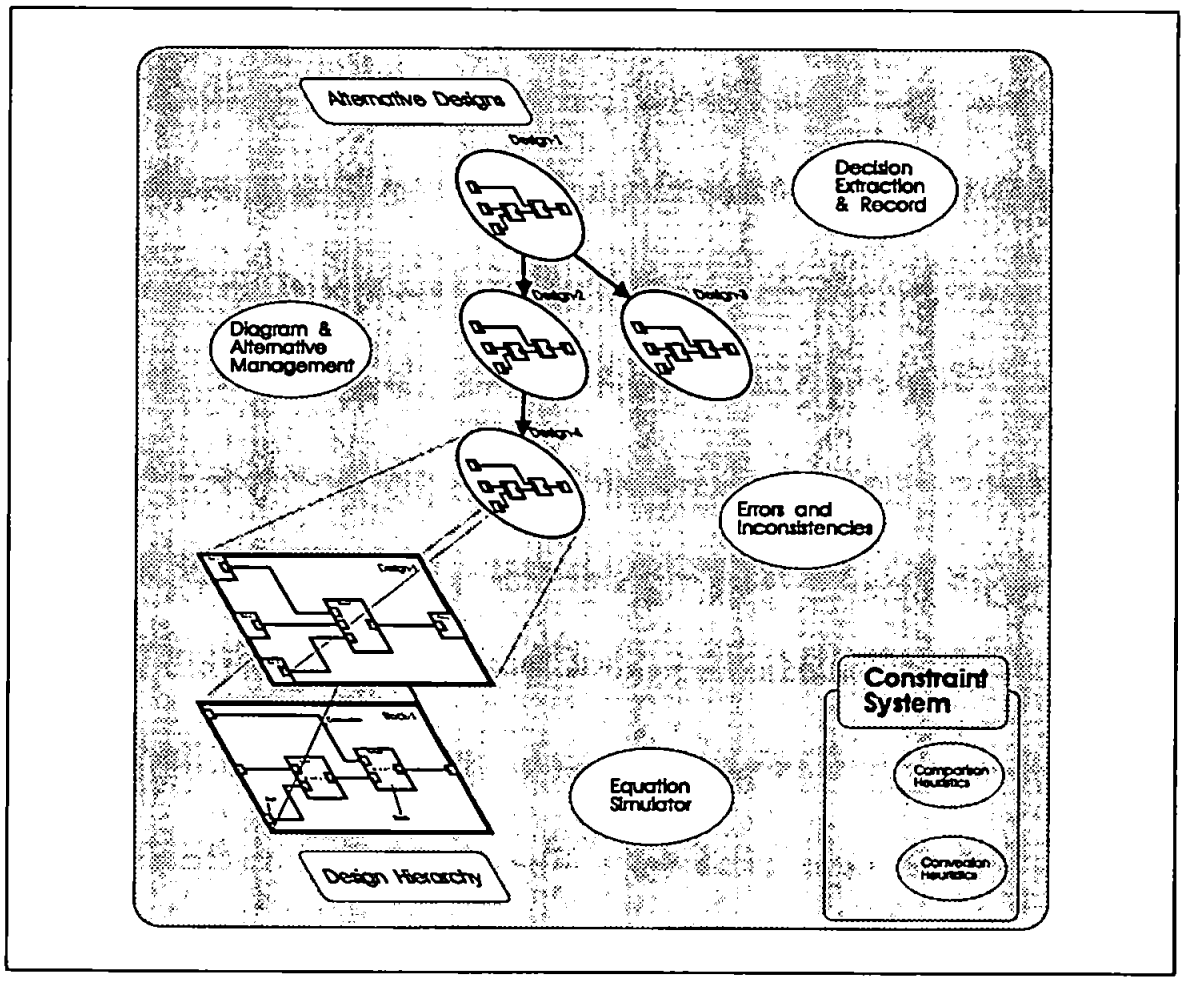

Figure I, a representation for the early stages of design: components

For description purposes, the representation can be easily divided into six basic parts, these are:

1) A merged representation for designs and alternative design hierarchies, based on block diagrams and constraining information.

2) A system for the management of block diagrams and design alternatives.

3) A constraint comparison system for the analysis and selection of alternatives.

4) A block diagram based mathematical equation simulator.

5) A system to extract and record decisions made during design.

6) A system to check for errors and inconsistencies made during design.

These parts are now separately described. It should be noted that the descriptions are in relatively abstract terms. This was done mainly to satisfy the issues mentioned earlier. As a result the representation itself has been kept simple, concise and homogeneous where possible. The complexity inherent in many other approaches is not apparent as it has been moved to the implementation domain.

3.3.1. Block Diagram and Alternative Design Representation

In the abstract representation, designs are held as collections of relevant design information called block diagrams. These diagrams conceptually similar to the familiar electronic engineering block diagram, contain entities called blocks which are arbitrary specified organizations of design data. This organization is achieved along common

hierarchical lines with the most abstract or general information at the top level, and the most decomposed or specific at the bottom. Figure II shows a conceptual view of this aspect of the representation. Note that this is very similar conceptually to many other design representations. The differences arise in the type of information stored and the way it is treated.

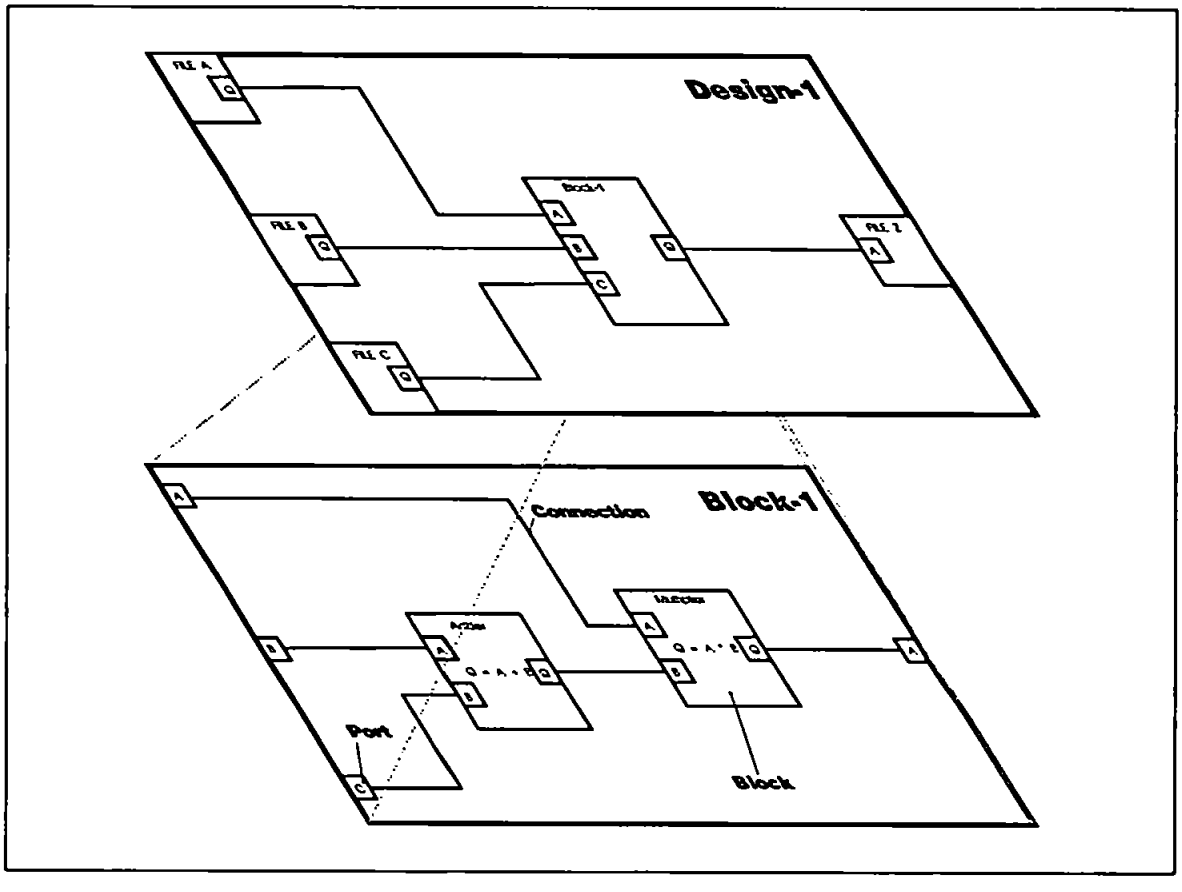

Figure II, block diagrams

Blocks are initially specified by mathematical functionality allowing them to collectively address the functional requirements of the design. However any type of information may reside in a block, not just common design information like physical, functional or behavioural aspects. Anything relevant to the design can be added, for example design time constraints, or designer expertise. This is a basic underlying concept in the representation. The most important point however, is that all information within the representation is regarded as constraint information. This aspect will pay dividends, with regard to constraint comparison, as it will allow the use of all information by the constraint comparison system.

Following on, no major distinctions have been made between the different types of information that can be stored in a block and in the way in which they are represented, and as a result this aspect of the representation can be viewed as being essentially homogeneous. This is an important and essential difference between the representation and many other approaches in the electronic design domain. These tend to be collections of separate representations, where for example the behaviour of a design may be described by a compiled program, whilst its structure is categorized by facts in a database.

Finally, it should be noted that the pictorial conceptual views, show designs in the same style as the PEDA implementation in chapter 4. This is in no way necessary, but was done primarily to help link together the various parts of the representation and that particular implementation .

The representation of alternative designs is addressed, through a simple extension to single designs. This important result is achieved by simply viewing an alternative design hierarchy as a collection of interconnected block diagrams. Each diagram contains additional (constraining) information which not only indicates the evolution of design alternatives, but also the reasons why a particular alternative (or in a loose sense version) was created, for example: "new version of diagram 1 due to diagram changes after simulation." A simple alternative hierarchy is shown in Figure III. In this simple example design-4 is derived from design-2 and both design-2 and design-3 are derived from design-1. When alternative designs are created they may inherit any amount of information from their predecessors, in a similar manner to object oriented inheritance. Thus design-4 may be almost the same as design-2 in the previous example, but may contain some slight differences. The representation of alternatives is kept simple by making no explicit distinction between the various types of related designs, such as alternatives, derivatives or versions. Otherwise conceptual naming problems could occur when a derivative design evolves so much that it becomes to all intents and purposes an alternative design.

It can be seen that the above approach satisfies the requirement for alternative representation in a simple and straightforward manner. Furthermore it satisfies the additional goal of homogeneity with regard to constraints, and by promoting links

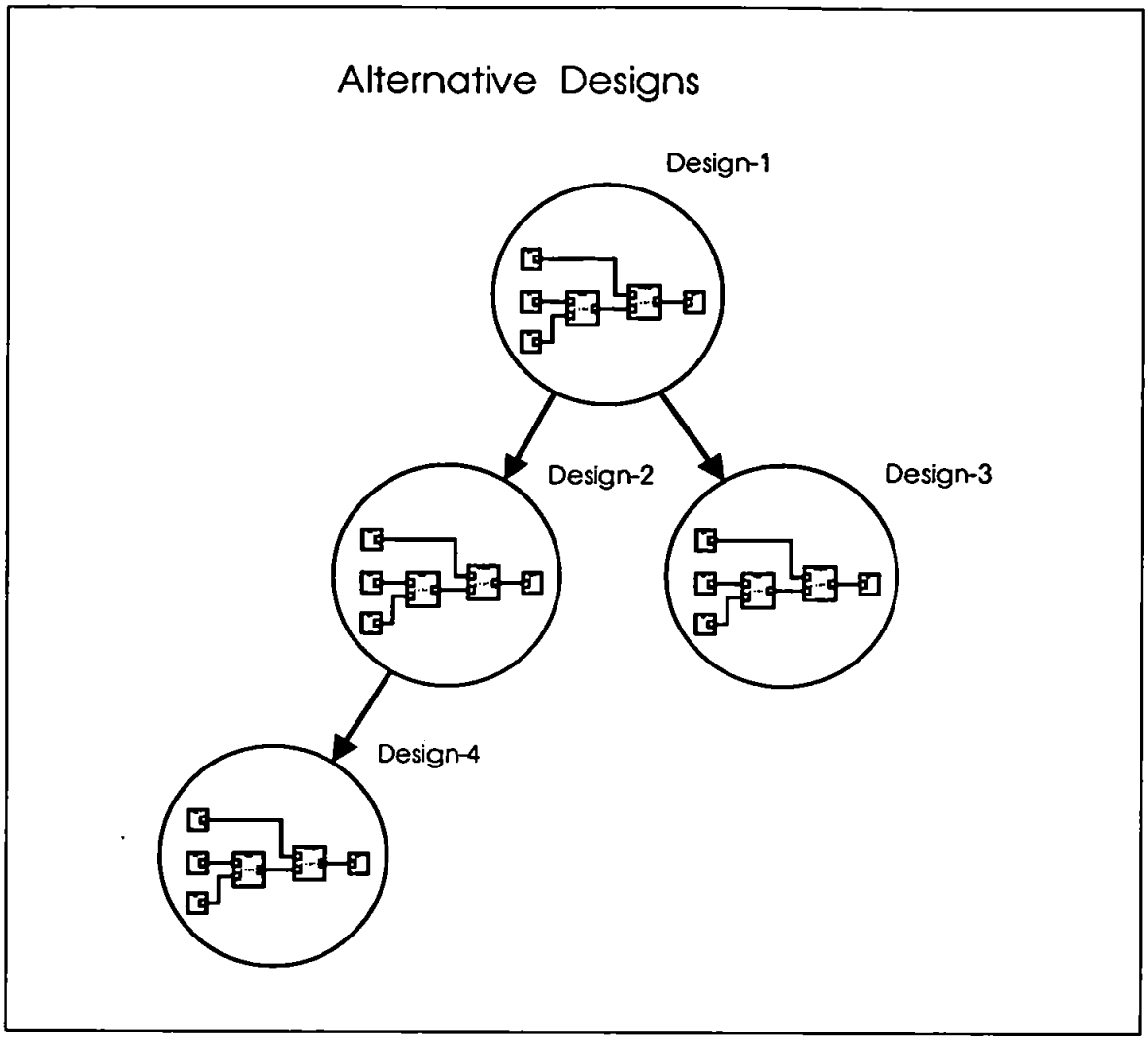

Figure III, alternative designs

between alternatives as constraints, it allows the use of that information in the constraint comparison process.

3.3.2. Block Diagram and Alternative Management

This aspect of the representation is very difficult to describe in the abstract form used in the rest of this section. Many facets of this aspect are firmly in the implementation domain, due to the intimate relationship between them and the aspects which they are managing. However the overall functionality of these components can be discussed. These components are concerned with the management of designs. They control the internal aspects of block diagrams together with the necessary generation, pruning and merging of alternative designs. This is done either automatically via

heuristics as a result of certain design decisions (see 3.3.5), or manually under the control of the user. The automatic management component is the most important concept here as it is an attempt to reduce the cognitive burden on the user. It is necessary due to the potentially very large number of alternatives that can be created during the course of a design.

3.3.3. Constraint Comparison System

The constraint comparison system provides the user with a means of rapidly choosing between a set of alternative designs, and selects those with the most desirable characteristics. Each design is automatically checked against a set of user defined constraints. This is achieved through a mapping operation, in that all required constraints are applied to each alternative design to produce a set of comparison results. A Multi-attribute utility theory approach has been chosen, for this aspect, combined with heuristics to produce the utility values for the various constraints. This approach is ideally suited, as it combines a simple method which can cope with the many constraints, with expandability and an easily understood method. The method can be easily shown through the example of MAUD (Multi-Attribute Utility Decomposition) an automated decision analysis program (Humphreys & McFadden, 1980). In this program a matrix of alternative actions (in this case different designs) is draw up against important attributes (design criteria). Each action can then be evaluated on the basis of all the attributes according to multi-attribute utility theory, where each point in the matrix is given a utility (importance value), and the set of utilities for each action are combined to give an overall utility for each action (design alternative). This final utility is indicative of the overall desirability of a design, and allows the selection between designs on a consistent, systematic and rapid basis.

It has been noted before that one the most important parts of the representation is the fact that almost all information can be used as a constraint. Some constraints are attributes (design criteria), whereas others can be used to produce utilities. There are two main ways by which utilities can be made, the first relies on the user providing them, a method which unfortunately introduces biases due to its subjective basis (Evans, 1988), and the second uses heuristics based upon domain knowledge to provide the values. Both methods are included in the representation, forming a balance between

using the expertise of the user and that of an expert. This approach also allows easy expansion and change, in that heuristics can be added at any time. Figure IV shows a conceptual view of role of the constraint comparison system in the representation.

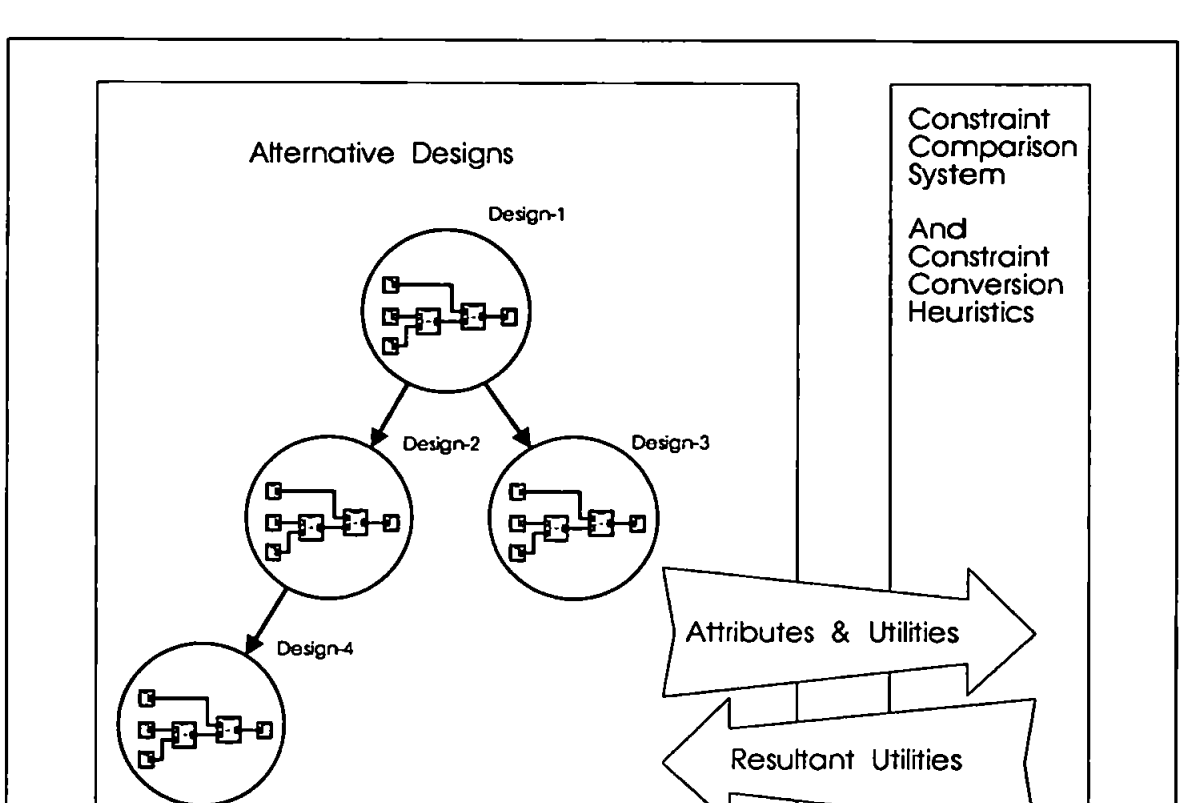

Figure IV, conceptual view of constraint comparison system.

3.3.4. The Equation Based Simulator

In a similar vein to the explanation regarding the management of designs, this section does not describe the basic operation of the simulator as this is an implementation aspect. What is done however is to highlight the important aspects of this part of the representation.

The simulator aspect of the representation is used for two main purposes, the first is its conventional role in providing simulation data to the designer, the second is as part of the constraint system. It is primarily this second role which distinguishes the simulator from others, in that it allows the use of design behaviour as a constraint in

the design comparison process. The simulator is general purpose to suit its requirements at the early stages of design. It can provide both numeric and symbolic equation based simulation of the mathematical behaviour of block diagrams. The use of mathematical equations is important as it allows the simulation of many different types of problems at a high level of abstraction. The simulator is easily extensible because it is limited only by the expressive power of the equations in the blocks.

The representation uses a novel direct data flow approach for simulation in which data flows physically around a block diagram. This important step has been made to avoid the complications of a separate behavioural representation for simulation. Further simplifications are achieved by using Dataflow techniques, avoiding the use of an timing agenda. This can be done because timing information is not important of in the early stages of early electronic design, except as an overall design constraint (Ball, 1990). A conceptual view of the simulation process is shown in Figure V, with numeric data travelling from right to left through a series of blocks and being evaluated as it passes through them according to the equation stored within.

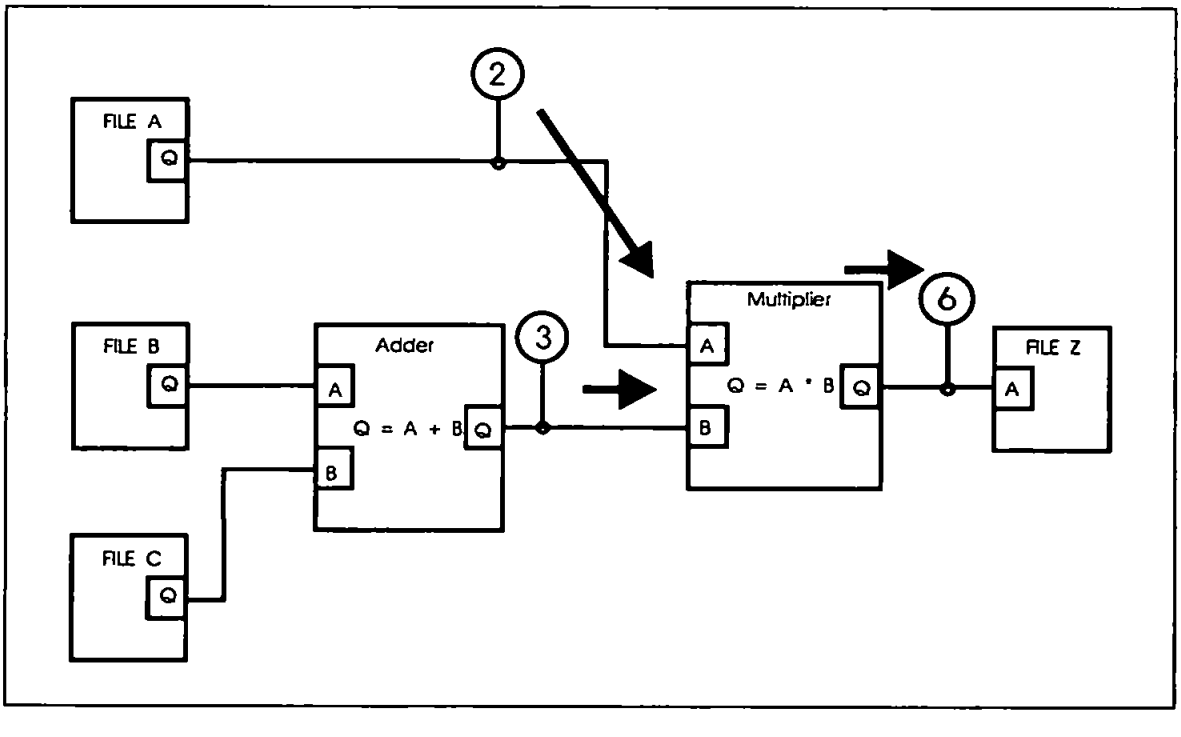

Figure V, conceptual view of simulation process

The usefulness of the simulator in the early stages of the design process is greatly improved by allowing each block in a block diagram to be simulated at a different level from its neighbours, and as a result a design can always be simulated if

it is specified at the top level. This can aid rapid prototyping in that a design need not be fully decomposed to determine its suitability for use. In addition any part of the design can be concentrated upon, and the effects of that change noted without resorting to expanding other parts of the design, again speeding prototyping. This can promote a top down design strategy with designs being verified at each level using simulation. Further improvements, can also be achieved by dividing the prototyping work amongst several designers. They can work in relative isolation, concentrating on their part, whilst specifying the rest of the design at a higher complete level. The isolation is not total however due to the fact that constraints can be applicable between different parts of the same design.

The use of symbolic simulation is an important aspect of the design validation process as it allows designs to be proved equivalent at different levels, through the process of symbolically simulating a design at each level. The results of each simulation can then be algebraically manipulated using a tool such as REDUCE to prove the equations equivalent. A simple example of this process is shown in Figure VI, involving the equation y = Sin(a)*Cos(b).

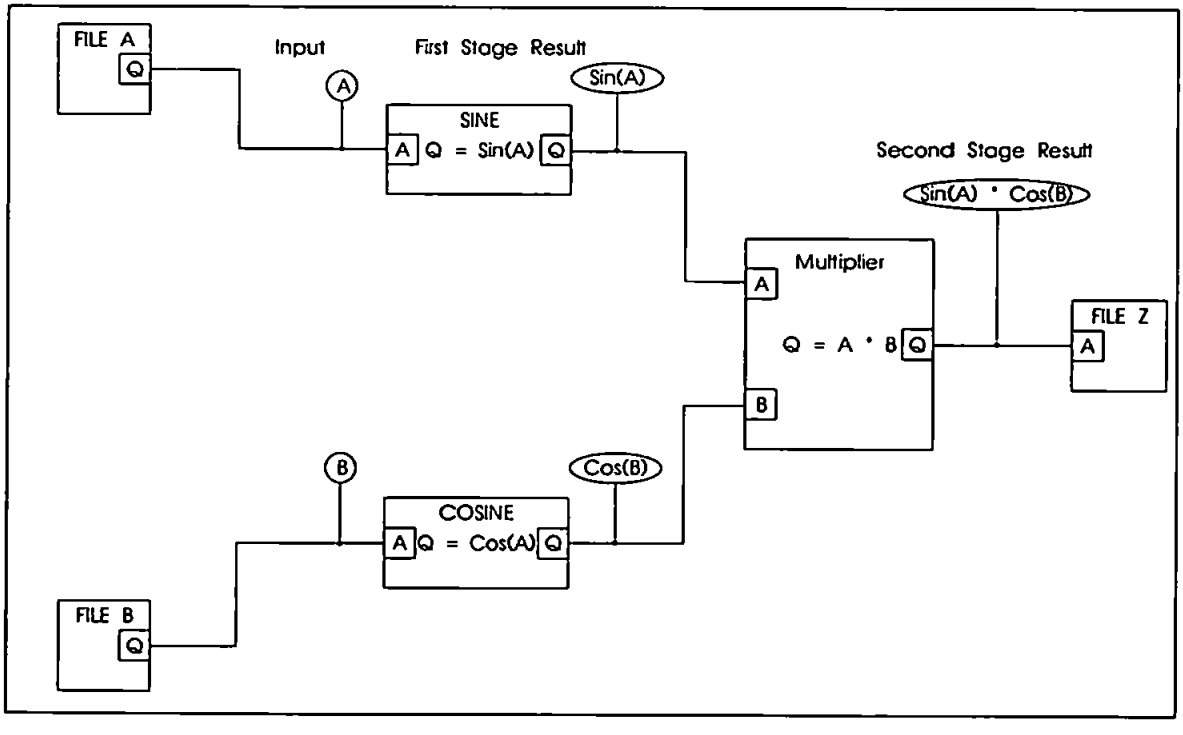

Figure VI, symbolic simulation

3.3.5. Decision Point System

Exploratory behaviour is further aided in the representation by the provision of a record of the decision points at various parts of the design cycle. These points are an abstraction of the user activity and encapsulate a particular decision and the reasons why it was made. Decision point information is an extension to the normal history mechanism and is generated, either automatically in the case where heuristics can determine what is happening, or manually by the user. Decision points are an important part of the representation and can be used in a variety of ways. The primary use of decision points in this representation is to aid designers in backtracking through their designs when a particular path proves unpromising (requirement 4), by providing a record of the important steps taken. They can also be used to help a designer new to a project continue with the design when the original designer is not available for questioning. Also if the decisions encapsulate specific design knowledge then they can be used to complete designs through the process of replaying (Mostow & Fisher, 1987).

Decision points have other uses in other parts of the representation. Most importantly they are constraints so they can become part of the constraint comparison process. Additionally in the design and alternative management system they can be used to trigger automatic creation or merging of alternatives. Alternatives can be created when designs are altered after simulation, when the designer is trying out a different approach, or merged when the designer lumps together a whole series of derived but very similar designs into one.

3.3.6. Error Detection

An important aspect of the representation is concerned with the detection of errors during the course of design activity. It exists in a representation for early electronic engineering design, because the cost of rectifying mistakes in a design rises sharply, as a design progresses from stage to stage. Obviously the earlier errors are found out and corrected, the better. A number of errors are highlighted in the requirements. The first involves inconsistencies in the representation, for example conflicting specifications at different levels in the alternative design hierarchy. The second type involves logical errors either in mathematics or notation. It is relatively

straightforward operation to view both types of errors as cases of constraint mismatching and so integrate them into the overall constraint comparison mechanism. However a mechanism to detect and then alert the user to these errors is still required, and in this representation heuristics are used.

The inconsistencies in requirements are comparatively easy to detect, but the logical errors require some knowledge of the domain, and mathematical errors require a much deeper knowledge concerning the rules of mathematics. However the use of heuristics does allow these features to be addressed incrementally when necessary. A symbolic manipulation tool such as REDUCE or Macsyma could be incorporated, as a replacement to many of the required heuristics. This would break with the overall homogeneity desired of the representation, but would greatly reducing the work required to reduce the incidence of mathematical errors.

3.4. Comparison with Existing Systems or Methods

The aim of this section is to compare and contrast the abstract representation, with other approaches. This is done in two ways: The first compares the representation almost item by item with examples from the literature in the electronic engineering domain, as this allows a straight forward review of the relevant material; In the second combined approaches are examined, and comparisons made.
For the first part the areas of interest are as follows:

(i) Representation of individual designs and design alternatives.
(ii) Constraints and constraint comparison.
(iii) Simulation.
(iv) History and decision record.
(v) Errors and inconsistencies.

3.4.1. Representing Each Design and Design Alternatives

At the centre of all traditional electronic design support systems lies the representation of the design itself. This stems from the early requirements of such systems to represent the design at the circuit diagram stage through schematic capture, and then model the behaviour of the design through simulation. Basically such systems addressed these needs by representing designs as a combination of two aspects: the first structure, in the form of a description or "net list"; and secondly behaviour in the form of preprogrammed functionality.

Over the years these requirements have expanded to cover more aspects of electronic design. The basic separation of structure and behaviour has expanded to cover more areas dependent upon the particular emphasis of the tool, for example structure, behaviour and physical information (Walker, 1988). The rationale behind this particular tactic is to partition the concerns that the designer should consider at any particular

stage of design (Stefic et al, 1982), for example designers should not concern themselves with physical aspects of a VLSI design, whilst examining the behaviour. The representation described earlier in this chapter does not adopt a partitioning scheme (except design hierarchy) for the representation of the different aspects of design. This is due mainly to the target areas of the representation being the cognitive needs of designer, and the early stages of electronic engineering design. In the early stages of design, it has been noted (Ball, 1990) that designers tend to combine certain aspects of a design, with constraints in abstract form breaking the separation of concerns approach (Stefic et. al. , 1981), allowing for example physical factors to influence initial choices of functionality. Secondly a large proportion of the literature produced in this field regarding the representation of designs is concerned with automated design (ie synthesis) and many of the approaches discussed are targetted at that area. (Gupta, 1988, Walker, 1988, Leung et al, 1988, Knapp and Parker, 1985). It can therefore be seen that there is little need to segregate information regrading the various aspects of design into separate domains in a representation which is not targetted at synthesis, and at an area of design where designers tend to merge the various domains anyway.

Hierarchical abstraction, a powerful technique for managing the complexity of design and used by designers in many stages of design including early design (Ball, 1990), is used to great effect in the representations of designs. Unfortunately the tactic of partitioning the design into several areas each with its own hierarchical structure greatly increases the complexity of the overall design representation, when compared to the simple hierarchical structure of designs discussed in this chapter. The representation of designs in this representation is simple and uniform, where a block contains a series of attributes which constrain it, and will itself be embedded in the block diagram structure. The behaviour of blocks is treated like any other attribute, and is specified using a mathematical equation. This is in marked contrast to more common representations or models, in which the behaviour may be deeply embedded within the simulator in the case of any early design tools, or in behavioural descriptions stored separately as behavioural rules, or some behaviour description language. The same representational scheme is used at all levels in the hierarchy, again in contrast to other representations or models where, the design functionality will be specified in different terms, for example at the behavioural, function, logic, gate, circuit and switch levels (Mokkarala et al, 1985). This generality can be achieved, because the use of

mathematical equations allows the modelling of almost any behaviour, and the slowness of a purely equation based approach is not viewed as a problem in the early stages of design, where the number of blocks being modelled is small.

The representation of alternatives or versions within a design support system is one of the key areas mentioned within the psychologically derived requirements (Ball, 1990). Alternatives have been addressed in the literature in a number of ways. Firstly traditional CAD systems have relied on very simple version control systems. In general the user would manually organise the hierarchical structure of alternatives, saving each design and maintaining such information regarding old and new versions of a particular design stage. More recently there has been considerable effort in supporting this type of activity automatically, with examples in this area from Katz and Chang (1986), Chou and Kim (1986), and Gabbe and Subrahmanyam (1986). These approaches tend to address the traditional problems of version control, and include such items as maintaining the most up to date elements of a design in a team environment, maintaining consistency of the various parts of a design in a team environment, and keeping track of the various parts of a particular design going from original, through each refinement.

The use of alternatives in the representation is different however to the classical needs of version control, and can be met with a much simpler system. The requirements indicate that alternatives are used to encapsulate different approaches to a problem, and not maintain consistency between the output of a group of designers. The simple method of individual designs (block diagrams), forming an alternative hierarchy meets these goals. Additional information linking these designs with the reasons why they were made allows the representation to aid backtracking when a design becomes unpromising. The method tightly integrates alternatives into the representation of designs, in contrast to that by Chou and Kim (1986), which is a external system, but similar to Gabbe and Subrahmanyam (1987). Their method however offers a complex system involving original designs followed by decompositions and is more in keeping with more general version control needs. The simple method of representing alternatives in the proposed representation is more in line with the requirements stated in chapter 2.

3.4.2. Constraints and Constraint Comparison

The use of constraints in the representation differs significantly with many other approaches in the electronic engineering design support domain, where it appears that the main use of constraints and constraint based systems has been the automation of certain aspects of the design activity, typically in the synthesis of designs.

In this area constraints are viewed as relations between sets of design parameters (Chan and Paulson, 1987), and as such may be used to critique a design, for example: CRITTER (Kelly, 1984) which performs constraint calculation, propagation and checking of functional and timing behaviour for the tools VEXED (Michell et al, 1985) and REDESIGN (Steinberg and Mitchell, 1984), knowledge based systems that partially automate the decomposition of designs, and changing designs respectively.

These types of constraint (relationships) are commonly handled through knowledge based techniques, involving expert domain knowledge in the form of heuristics, for example OASYS (Harjani et al, 1989) which is a framework for analogue circuit synthesis, or algorithmic approaches, for example OPTIMIZE (Rankin, Siemensma, 1989) a system which uses numerical methods to minimize various cost functions (constraints) in an almost completed design.

The overall impression of these systems is the use of constraints to support the automated machine design process, whereas their use in the representation is to aid the decision making process of the designer, for example Brewer and Gajski, (1986) use constraints and associated heuristics to evaluate designs automatically as part of the design synthesis process.

This approach has a number of disadvantages, the first being that the intelligence of the designer using the tool is being wasted, the second is that the knowledge base would be very large unless the domain of interest was severely constrained. This approach would have severe difficulties in addressing the representation proposed in this chapter due to the fact that all information can be regarded as a constraint, and this is why the combined multi-attribute utility theory and knowledge based approach has been adopted.

3.4.3. Simulation

The modelling of designs through simulation is a very powerful means of verification, and whilst formal proving methods are still in their infancy, it will be used where its cost and time advantages compared to full prototyping are apparent. This has occurred in the electronic engineering domain where many different simulation methods have been proposed, primarily dependent on which area of the design they aim to address, for example Director et al, (1985) outlines switch, circuit and process level simulators in the digital VLSI design domain. In these areas many different constraining factors have influenced these approaches. At the low end or near the device level, where a large number of relatively simple components are being modeled, the emphasis has been towards fast execution speeds, using for example improved sequential (Barzilai et al, 1986) or parallel (Smith, 1986) approaches. This trend towards specialization has been done to achieve realistic simulation times of the more complex designs.

In other areas there has also been a trend towards generalization with the appearance of mixed or multi-level (Tham et al, 1984, and Takasaki et al, 1986) simulators, which combine a number of levels, for example unit delay, multiple delay and timing simulation (Chen et al, 1984). This has been extended at the high end or behaviour level, with the introduction of comparatively general tools based on object oriented (Lathrop and Kirk 1985) and rule based (Singh, 1983) principles. In this case the greater generality of these more flexible methods allows an efficient verification of the more complex designs. Where reasoning about the simulation process itself is important, for example reasoning about temporal aspects, Petri net methods can be useful (see Tadao Murata, 1989 for detailed discussion of applications).

The simulator within the representation can be viewed as having many similarities to simulators in existence in the electronic engineering domain, however it differs in a number of significant areas. The use of mathematical equations to specify behaviour, allows the simulator to work at many levels in design, from a higher level of abstraction than the behavioural simulators to the lowest. This ability allows simulation to be performed at any reasonable level of abstraction in different parts of the design, a fact which makes the simulator a powerful exploration tool as well as an verification tool. This is in contrast to many other simulators in this area which are primarily verification tools.

The simulator is based upon data flow principles, in keeping with the psychological considerations mentioned earlier (Ball, 1990), whereas many simulators in the digital electronic engineering domain use complex event driven schemes, to take into account unneeded timing considerations. The simulator is also tightly integrated with the representation, and in it simulation takes place on the structure, in direct contrast to most other simulation systems, which maintain separate data structures for simulation. This is a moderately important issue as it avoids the conceptual separation of behaviour from structure and allows simulation data to be used in situ as constraint information, further integrating the representation.

3.4.4. History and Decision Record

Electronic design support systems have traditionally supported a means of recording design activity known as the history trace, in which the system logs the selections made by the designer in the order in which they occurred. This simple system can be realized easily and as it is a record of past activity can be used as an aid to exploratory activity. There have been a number of suggestions on improving the usefulness of histories, with for example Mostow, (1985) and Takala, (1989) indicating that the history mechanism may be used as a model of design. Takala suggests that a history mechanism can be improved through expansion into two dimensions, forming a history network for the various design objects. By raising the level of abstraction of histories into design plans Mostow et al, (1989) allow the replaying of these design decision histories on slightly different initial designs, automating the redesign process. These methods are however not applicable to this area of design in that the design decisions required are those which link design alternatives, for example the reasons why a particular choice was made and not the choice itself. If the later definition is used then the extraction of design decisions becomes much easier, for example in BOGART (Mostow et al, 1989) it involves item selections made from menus or graphical displays. To obtain the wider context decisions, a means is required to extract the necessary information. The issue is left comparatively open but heuristics derived from studies of designers may produce the desired level of generality desired.

3.4.5. Errors and Inconsistencies

The detection of user generated errors and inconsistencies is an area that appears to be comparatively unaddressed in electronic engineering, though there are tools such as REDUCE (Hearn, 1984) and Macsyma (Bogen, 1983) which can help the user with algebraic manipulation tasks. (Equation reasoning systems do exist in other domains)

Inconsistencies in designs are generally handled by constraint propagation, or some form of truth maintenance scheme. Though to detect the mathematical errors mentioned in chapter 2 a certain amount of mathematical knowledge is required. This has been done in some areas, for example in mechanical engineering the "design to product" project (Popplestone et al, 1986) which incorporates mathematical knowledge into a large computer aided manufacture system. And although the initial application for this information is in a similar manner to the use of REDUCE and the other tools, it could be used as a basis to detect user generated errors. The most common approaches to knowledge based design in the electronic engineering arena have tended to concentrate on replacing the designer in certain aspects of the design cycle, and whilst this tactic may effectively remove one area where user generated errors can occur, the other areas remain.

3.4.6. Comparison with Combined Approaches

This section discusses the salient parts of a few example approaches to representing electronic engineering designs. The examples have been taken mainly from the electronic design automation arena, due to preponderance of literature on design representation in this area. The comparisons are a little unfair as these approaches to AI design, replicate some aspect of design behaviour, whereas this representation attempts to provide important support for people doing design.

3.4.6.1. "A Conceptual Framework for ASIC Design (Leung, Lisher and Shanblatt, 1988)

This approach covers the making of a conceptual framework for the design of application specific integrated circuits in the digital VLSI arena. It is particularly

interesting in that it incorporates many aspects of that design activity, and possesses many similarities to the representation proposed in this chapter.

The overall emphasis of the representation is centred towards the synthesis of designs, from a decision making perspective, as opposed to the more common transformational model. The framework itself is divided into three areas: the design process; the design hyperspace; and the design repertoire.

The first area or design process comprises the DOEMA (Design Object, Design Engine, System Manager and Expert Assistant) model of the ASIC design activity, and incorporates an implementation of ASIC design methodologies at various levels of abstraction. Design objects are used to describe the target design at a particular level of abstraction, and define the place where the dynamic (process) and static (design information) aspects of ASIC design intersect. This dynamic knowledge is separated and used to form the Design engine which is an embodiment of the mundane and mechanistic aspects of ASIC VLSI design, which involves tasks such as transformation, verification, simulation and test consideration (design for test). The expert assistant is used to make the designer aware of design alternatives, by deciding what the alternatives are at a particular stage and which ones should be considered. This provides a methodology which limits the search space and therefore the number of alternatives examined. The system manager is the final part of the DOEMA model and provides the overall glue to integrate the other aspects.

The second area or design hyperspace consists of those parts of the design which tend to be stable over time, and is divided into a series of frames which are mutually independent (orthogonal), for example the structural, behavioural, and physical domains of the System Architect's Workbench (Walker, 1988). The suggestion is that the particular framing is not unique and depends on the designer's perspective. Two spaces are given as examples: the architectural space; and the algorithm space, which are further subdivided into: functional units, communication and control; and operation, data structure and data dependency respectively. The stated aim of this separation is twofold, the first is to try and make it easier for the designer to recognize design alternatives, the second that the subdivision into frames is possible on real designs.

The final area or design repertoire is a collection of design and analysis techniques used for evaluating design alternatives, these techniques including resource configuration, which finds suitable architectures for a given algorithm ,and algorithm

restructuring which rearranges a given algorithm to better fit a given architecture.

In brief it can be seen that this approach and representation discussed earlier in the chapter, have similar aims, in that they are primarily geared to help the designer pursue alternative approaches in their designs. They differ however in the emphasis of their approach, in that the knowledge embedded in the framework is targetted at the domain and not, as in the case of the representation, at the cognitive needs of the user. In addition the embedded knowledge is used in a closed fashion, in that it helps the user to find alternatives, but uses its own knowledge to find them, whereas the representation helps the designer use his own knowledge. This is an acceptable approach to a narrow domain, where sufficient knowledge can be elicited, but is less useful in wider domains, and where a more cooperative approach is required.

The separation of design information into many frames is another area where the two approaches differ. As said before, this type of approach is less desirable in the early stages of design where the representation is aimed at.

3.4.6.2. "An object based representation for the evolution of VLSI designs" (Gabbe and Subrahmanyam, 1987)

The major aspect of this scheme is the encapsulation of design information from specification to implementation into a framework which represents explicitly the evolving design. It is based upon a transformational model of design which converts specifications to realizations through a set of refinement steps. These designs or versions are organized into a hierarchy of three levels: the architectural level, which determines the way in which the functionality is decomposed into subfunctions; the environmental level, which adds additional constraints such as technological issues; and the realization level, which contains various implementations and the constraints which they satisfy. As a result the architectural level will be composed of hierarchies of decompositions, the environment will contain contexts and realizations will embody refinements.

Design information is also split into three areas: the first deals with the description of designs, from function specification to physical masks; the second contains the mechanisms and the domain knowledge to convert between the various parts in the first area; and the third contains the mechanisms and control knowledge

which decide what to do.

The aim of this model to provide a means for representing iterative design activity. This is done through the use of design modules in the following way: A module realizes some degree of functionality and consists initially of specifications and results; and these results consist of decompositions and a series of refinements in a particular context. For example, a set of specifications may give rise to a number of alternative decompositions. Each decomposition proceeds through a series of refinements to produce a valid design. And each valid design is applicable to a particular context. This representation also includes a history mechanism based on transactions, and a constraint propagation system for truth maintenance.

This approach yields a flexible but complex representation based upon the object oriented paradigm. In a similar manner to the previous approach, it seems that a great deal of effort has been spent on producing many orthogonal spaces. This interesting and common approach whilst perhaps much more important in the later stages of design is less applicable to the early stages. This model like many others appears to be targetted at the problem domain and not at cognitive needs of the designer, and as a result more adequately fits the needs of design synthesis systems. The extensive partitioning scheme also suggests that the model is aimed more towards the implementation of systems, where efficiency and size constraints are more important than the model discussed at the start of this chapter.

3.4.6.3. Walker and Thomas: the System Architect's Workbench.

This is a transformational model used for the synthesis of VLSI designs. It is based on various levels of abstraction and divided into three areas of description comprising behaviour, structure and physical domains. The levels of abstraction are defined as: architecture; algorithmic; functional block; logic; and circuit. For example the behaviour hierarchy covers performance specifications at the top level, going through algorithms, register transfer and boolean equations to electrical characteristics at the bottom.

Transformational heuristics are used to convert from one domain hierarchy to another, and to maintain overall consistency. At each level in each domain a different representation can be used, thus the representation forms the glue to a number a

separate systems. Another reason for the separation of domains is the non isomorphism of designs at this stage of the design activity. For example a behavioural description may not have a one to one correspondence with its equivalent structure.

This approach is basically very different from the representation, as it is geared towards synthesis where the requirements are quite different. As a result alternatives or versions, decision points and a decision support system are not usually explicit parts of synthesis models. In a similar vein the various domains can be merged, if the aim is to address the cognitive needs of designers in the early stages of design.

3.4.6.4. Knapp and Parker Advanced Design AutoMation project (ADAM)

This is a similar synthesis representation to the one previously. In it designs are split into four separate non isomorphic subspaces: dataflow behaviour; structural; physical; and timing and control. Again these spaces are organized in hierarchical manner. In a similar way the other parts of the model are not explicit parts of this representation, but do exist as parts of the larger system, for example the design planner in the ADAM system (Knapp and Parker, 1986). In interesting point here is that time issues in simulation have been separated away from behaviour.

3.5. Summary

The chapter has presented a abstract representation for the early stages of electronic engineering design, which in spite of being simple, tackles the important aspects of that design activity by taking into account the cognitive needs of designers in their work. The representation itself contains an explicit block diagram representation of design alternatives, together with a means to compare and contrast those designs, a dataflow simulator and means to extract and record design decisions, detect errors and inconsistencies made by the designer. The representation is then compared on a piece by piece basis with other approaches, and then in whole against a few representative representation models of electronic design, to show that these systems do not adequately address the cognitive needs of designers in early design, and that it is a valid and useful attempt to do so.

One important aim of the representation was to target it at a level sufficiently concrete that it is a real aid in producing an implementation, but abstract enough to not include many implementation issues. Chapter 4 shows the next step and discusses how an example implementation: PEDA was realized.

References for Chapter 3

Ashok, V., Costello, D., and Sadayappan, P., " Modelling Switch-level Simulation Using Data Flow," Proceedings of the 22nd ACM/IEEE Design Automation Conference, 1985.

Ball, L., "Cognitive Processes in Engineering Design," PhD Thesis, Department of Psychology, Polytechnic South West, Devon, UK, 1990.

Barzilai, Z. & Beece, D. K., "SLS- A fast Switch Level Simulator for Verification and Fault coverage Analysis, Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986

Bogen, R., Golden, J., Genesereth, M., Pavelle, R., Webster, M., Fateman, R., and Doohovskoy, A., Macsyma Reference Manual, The Mathlab Group, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983.

Bowen, J. A., " Automated Configuration Using a Functional Reasoning Approach," Proceedings AISB, 1985.

Brewer, F. D. & Gajski, D. D., "An Expert-System Paradigm for Design, Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986

Chan, W. T. And Paulson Jr, B. C., "Exploratory Design Using Constraints," AI EDAM, pp 59-71, 1987.

Chen, C. F., Lo, C-Y., Nham, H. N., and Subramaniam, P., " The 2nd Generation Motis Mixed-Mode Simulator," Proceedings of the 21st ACM/IEEE Design Automation Conference, 1984.

Chou, H. and Kim, W., "A Unifying Framework for Version Control in a CAD Environment," Proceedings of the Twelfth International Conference on Very large Databases, pp. 336-344, 1986.

Culverhouse, P.F., Ball, L. & Burton, C.J., "A Tool for Tracking Engineering Design in Action," To appear in "Design Studies".

Davis, R. & Shrobe, H ., "Representing Structure and Behaviour of Digital Hardware," Computer, 1983.

Director, S. W., Shen, J. P., Siewiorek, D. P., and Thomas, D. E., "The CMU DA/CAD Project," Research Report No. CMUCAD-81-2, 1982.

Evans, J. St. B. T., "Knowledge Elicitation in the Training and Assessment of High Level Cognitive Skills," Report Prepared for the Army Personnel Research Establishment, 1986.

Evans, J. St. B. T., "Bias in Human Reasoning: Causes and Consequences," Brighton: Erlbaum, 1988.

Gabbe, J. D. and Subrahmanyam, P. A., "An Object-Based Representation for the Evolution of VLSI Designs," Artificial Intelligence in Engineering, vol. 2, no. 4, 1987.

Green, m., "A Methodology for the Specification of Graphical User Interface," ACM Computer Graphics, 1981.

Gupta, Anurag P., "A hierarchical Problem Solving Architecture for Design Synthesis of Single Board Computers," MPhil Thesis Carnegie Mellon University, 22nd February 1988.

Harjani, R., Rutenbar, R. A., and Carley, L. R., "OASYS: A Framework for Analog Circuit Synthesis," IEEE Transaction on Computer Aided Design, vol. 8, no. 12, December 1989.

Hearn, A. C. (ed), REDUCE Users Manual Version 3.2, the Rand Corporation, Santa Monica, 1984.

Hooton, A. R., Aguero, U. and Dasdupta, S, "An Exercise in Plausibility-Driven Design," Computer, 1988.

Humphreys, P. C., & McFadden, W., Experiences with MAUD: Aiding decision structuring versus bootstrapping the decision-maker," Acta Psychologica, vol. 45, pp. 51-69.
1980.

Katz, R. H., Anwarrudin, M., Chang, E., "A Version server for Computer-Aided Design Data," Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986.

Kelly, Van. E., "The CRITTER System," Proceedings of the 21st ACM/IEEE Design Automation Conference, 1984.

Knapp, D. W., and Parker, A. C., "A Design Utility Manager: The ADAM Planning engine," Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986.

Lathrop, R. H. and Kirk, R. S., "AN Extensible Object-Oriented Mixed-Mode Functional Simulation System," Proceedings of the 22nd ACM/IEEE Design Automation Conference, pp. 630-636, 1985.

Leung, S. S., Fisher, P. D., and Shanblatt, M. A., "A Conceptual Framework for ASIC Design," Proceedings of the IEEE, vol. 76, no. 7, pp. 741-755, 1988.

Mentor Graphics, "Vision," Mentor Graphics Newsletter, April 1990.

Mitchell, T., Steinberg, Louis, I., and Shulman, J. S., "A Knowledge-Based Approach to Design," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 7, no. 5, 1985.

Mokkarala, V. R., Fan, A., and Apte, R., "A Unified Approach to Simulation and Timing Verification at the Functional Level," Proceedings of the 22nd ACM/IEEE

Design Automation Conference, 1985.

Mostow, J., "Towards Better Models of the Design Process," The AI magazine, pp.44-57, 1985.

Mostow, J., "Design by Derivational Analogy: Issues in the Automated Replay of design Plans," Rutgers University, Department of Computer Science, AI/Design Project Working Paper No. 80, 1987 .

Mostow, J., Barley, B. and Weinrich, T., "Automated reuse of Design Plans," Rutgers University, Department of Computer Science, AI/Design Project Working Paper No. 146, 1989.

Newell, A & Simon, H. A., "Human Problem Solving," Prentice-Hall, Englewood Cliffs, NJ, 1972.

Popplestone, R., Smithers, T., Corney, J., Koutsou, A., Millington, K., and Sahar, G., "Engineering Design Support Systems," Proceedings of The 1st International Conference on Applications of Artificial Intelligence to Engineering problems, April 1986.

Rankin, P. J. and Siemensma,J. M., "Analogue Circuit Optimization in a Graphical Environment," presented at ICCAD-89.

Sinclair, M. A., Siemieniuch, C. E., and John, P. A., "A User-Centred Approach to Define High-Level Requirements for Next-Generation CAD Systems for Mechanical Engineering," IEEE Transactions on Engineering Management, Vol. 36, No. 4, November 1989.

Singh, N, "MARS: A Multiple Abstraction Rule_Based Simulator," FLAIR Technical report No. 17, Fairchild Laboratory for Artificial Intelligence Research, 40001 Miranda Ave. Palo, Alto, CA 94304, 1983.

Smith, R. J., "Fundamentals of Parallel Logic Simulation," Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986.

Smyth, M., "Articulating the Designer's Mental Codes," LUTCHI Research Centre internal paper (draft) ref: HCC/L/24, 11th May 1988.

Stefic, M., Bobrow, D. G., Bell, A., Brown, H. Conway, L., Tong, C., "The Partitioning of Concerns in Digital System Design," Xerox parc internal paper VLSI-81-3, Dec 1981.

Steinberg, L., and Mitchell, T., "A Knowledge Based Approach to VLSI CAD the Redesign System," Proceedings of the 21st ACM/IEEE Design Automation Conference, 1984.

Tadao Murata, "Petri Nets: Properties, Analysis and Applications," Proceedinds of the IEEE, Vol. 77, No. 4, April 1989.

Takala, T., and Silen, P., "Application of History Mechanism in Architectural Design," Preliminary Proceedings of The Third Eurographics Workshop on Intelligent CAD Systems, April 1989.

Takasaki, S., Sasaki, T., Nomizu, N., Ishikra,H., and Koike, N., "Hall II: A Mixed Hardware Logic Simulation System," Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986.

Tham, K., Willoner, R. and Wimp, D., "Functional Design Verification by Multi-Level Simulation," Proceedings of the 21st ACM/IEEE Design Automation Conference, 1984.

Tong, C., "Toward an engineering science of knowledge based design," Artificial Intelligence in Engineering, vol. 2, no. 3, 1987.

Ullman, D. G., Dietterich, T. G., and Stauffer, L. A., "A Model of the Mechanical design process Based on Empirical Data," AI EDAM, vol. 2, no. 1, pp. 33-52, 1988.

Walker, R. A., "Design Representation and Behavioural Transformation for Algorithmic Level Integrated Circuit design," PhD Thesis: Research Report No. CMUCAD-88-20, SRC-CMU Research centre for Computer-Aided Design, Carnegie Mellon University, April, 1988. (6)

Williges, R. C., "The Use of Models in Human-Computer Interface Design," Ergonomics, vol0. 30, no. 3, pp. 491-502, 1987.

Chapter 4: The PEDA Representation for Early Electronic Engineering Design

4. The PEDA Representation for Early Electronic Engineering Design

4.1. Overall Structure of this Chapter

This chapter discusses a partial implementation of the abstract representation for the early stages of electronic engineering design, discussed in the previous chapter. This implementation has been used as the core of the Plymouth Engineer's Design Assistant (PEDA), a designer support tool for the early stages of electronic engineering design. It was envisaged to promote the rapid generation and selection of alternative designs, according to a set of desired criteria, in a consistent and uniform manner. A brief outline of the various parts of the PEDA system is given first which then leads on to a discussion on the relevant parts of the implementation. With each section there is a brief statement outlining the areas where the implementation interacts with the user interface. The chapter ends with an example showing how the parts are integrated, followed by a summary.

4.2. The Plymouth Engineering Design Assistant : An Overview

The PEDA Environment discussed in Baker et. al. (1989) is a tool designed to address some important needs of engineer designers in the early stages of electronic engineering design. It was developed as part of a joint project in developing a psychologically based engineering design assistant, and was conceived out of a desire to produce a design support environment that better addressed the needs of engineers during the process of designing. This prototype system came to be known as the Plymouth Engineering Design Assistant or PEDA and was originally envisaged as a system that would promote engineer creativity in a natural manner, whilst offering assistance in those areas where human performance is poor. It was intended that PEDA would aid design by offering advice, pointing out inconsistencies and errors in a constructive and helpful manner. This advice would be applicable to a broader range of problems than many expert system approaches, which have tended to concentrate on relatively narrow domains. A psychological study of the way engineers design, would furnish information about the engineering design process, including the strengths and

weaknesses of designers. This knowledge would then be used to build a set of requirements that the PEDA would need to address if it were to be successful in its primary aims.

The system itself is presented to the user as a screen based drawing board that allows manipulation of block diagrams using a mouse. Mathematical blocks are selected from a palette and can be incorporated within a diagram to provide any level of functionality required, from user specified mathematical functions to components such as memory devices. Hierarchical design is supported, and a facility to zoom in a particular level is provided. More optimal design is encouraged by the ability to explore, develop, and compare many alternative designs, using a set of constraint criteria. The generation and deletion of alternatives is monitored and in some cases handled automatically according to a small set of heuristics governing the stage of design. Backtracking of design activity is also supported through a record of design activity.

The PEDA environment has been implemented using the knowledge based system building tool ART, and the COMMON LISP language on a SUN 3/60 workstation. The project was undertaken by three research assistants and their associated supervisors. L. Ball the psychology researcher would investigate the way in which electronic engineers design. The results of his work would be directly useful to G. M. Venner whose investigations would cover the Human Computer Interaction aspects of the project, and to D.G.C. Scothern whose work is the subject of this dissertation.

The PEDA tool can be conceptually divided into two parts, the first is the user interface partially completed by G.M.Venner who tragically died part way through her research, and the second is the implementation of the early design model discussed in chapter 3.

4.2.1. The PEDA User Interface: Overview

The user interface for PEDA has a direct manipulation interface intended to reduce the gulf between user and system during execution and evaluation, and to give the user a feeling of direct engagement (Hutchins et. al., 1985). The system is based upon a drawing board, rather than the more familiar desktop, metaphor. In this case designers create their diagrams using the mouse to select objects from a palette.

Thereafter a pop-up menu associated with a particular created component or object presents the user with the choice of relevant, valid functions that can be performed on it. Other important aspects of the interface are concerned with the communication of various aspects of design knowledge and in the portrayal of the evolution of the design process to the user. A typical view of the user interface to PEDA is shown in Figure VII.

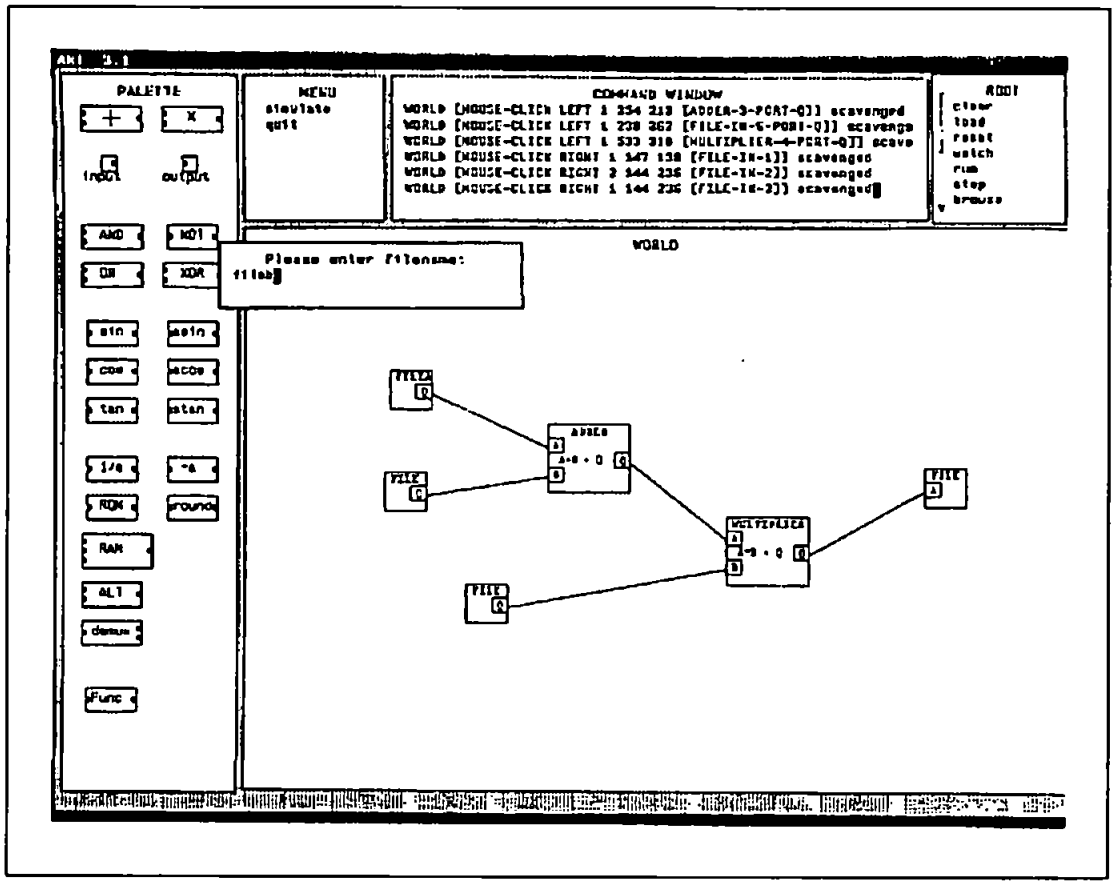

Figure VII, Users view of the PEDA system.

4.2.2. PEDA Internal Design Representation: Overview

The implementation of the early design representation within PEDA provides a framework for representing many important aspects of design activity within the tool.

The core representation consists of:

1) A Base representation for mathematical block diagram hierarchies, with a selection of commonly used blocks in a library.

2) A logical extension to the block diagrams to realize the formation of alternative designs, containing any amount of specifiable information.

3) A small set of heuristics to validate the automatic generation of new alternative and decision information dependent upon user input.

4) A small set of constraints including domain and non domain types, and heuristics to convert them into weightings and provide consistency checks.

5) A Mathematical data flow simulator capable of either numeric or symbolic operation, stateless, interruptible, and not dependent upon the use of an explicit representation of time for its operation.

Figure VIII, PEDA core representation

In the following sections these aspects are explained together with the associated background required. In the text comparisons are made with other schemes. This overlaps slightly with chapter 3, but was thought necessary due to the greater emphasis on implementation issues made here, and in the literature.

4.3. The Representation of Designs Within the PEDA System

4.3.1. The Representation of Individual Designs Within the PEDA System

The representation of a design is one of the most used concepts within the design arena, and many different schemes have been proposed. Within the field of electronic design, hierarchical descriptions involving primarily structure, behaviour and other data are the most common. The abstract representational for early design in

chapter 3 makes very few demands on the exact structure of design representation and as a result a similar scheme has been adopted for the PEDA system. However there are a few notable differences due to the removal of constraining factors such as the separation of concerns (Stefic et al, 1982). The basic elements of an electronic engineer's design are stored as an adaptation of the circuit diagram often used to represent designs on paper or on the screens of Engineering CAD systems. A representation that closely corresponds to a paper design has been adopted, as this leads to a simple, compact form, with little or no conversion required between the visualized and internal artifact.

Unfortunately a more abstract representation than a straight forward logic circuit diagram is really needed and so to produce this more general form, the circuit diagram was redefined as a block diagram and the circuit elements such as gates, transistors and so on, to entities known as functional blocks. At this level the behaviour of the functional blocks can be portrayed by general mathematical functions, and provides considerable freedom in describing function (behaviour). This is in contrast with many commercial electronic engineering CAD systems, which deal with block diagrams at a later stage of design, for instance at the logic diagram level and below, and offer facilities such as fast gate level simulation.

In common with most design representations and the recommendations of the psychological work discussed earlier, the block diagrams are hierarchically organized, allowing the greater complexity of designs at lower levels to be hidden from the engineer unless required. Connectivity between blocks is simply achieved via connections to ports residing within each block. Figure IX shows these basic structural concepts.

The structure was designed so that an Engineer's block diagram could be easily represented. Each block diagram or 'world' consists of a series of interconnected functional blocks (The terms 'block' and 'functional block' are used interchangeably in this dissertation). These blocks can potentially perform almost any mathematical functions, although within PEDA only a representative selection have been defined at present. They are again hierarchical in nature, so that complicated functions can be defined initially at a high level, then later decomposed into a equivalent block diagram, if or when required.

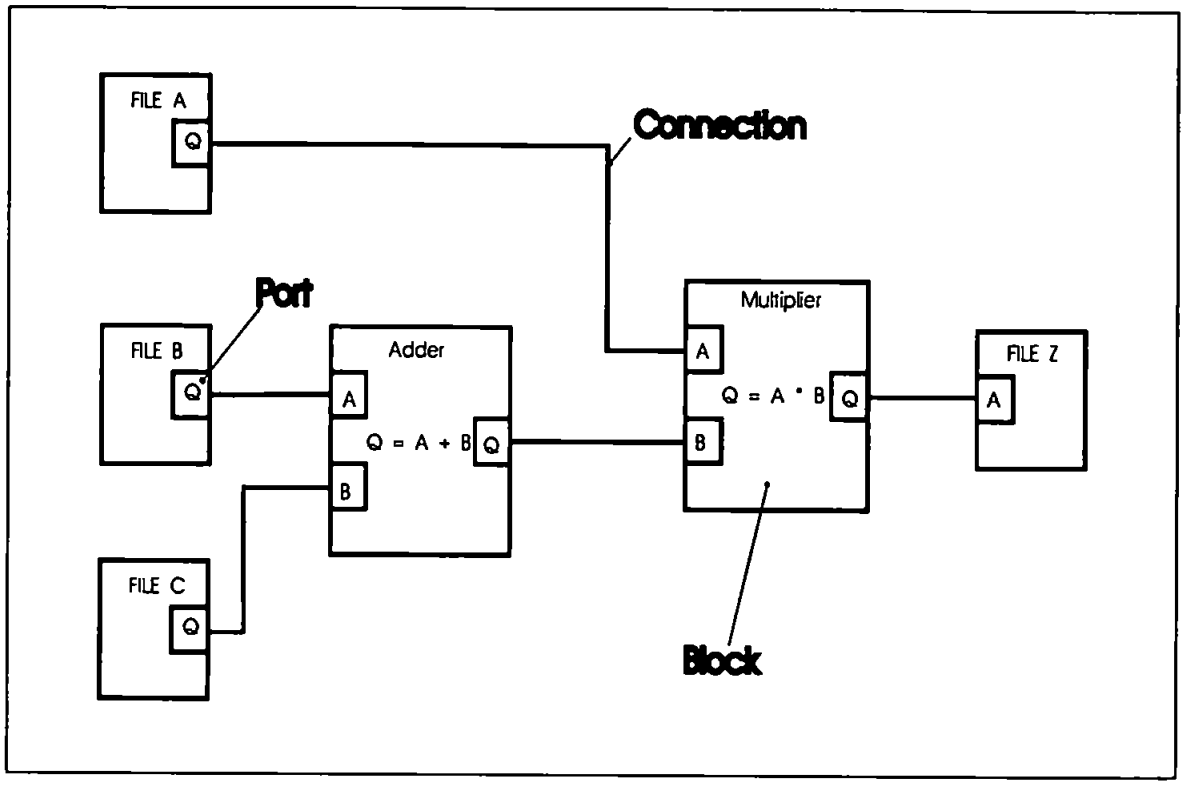

Figure IX, block diagram basic pictorial structure.

4.3.1.1. Functional Blocks

The functional block is as mentioned previously the standard structural unit for representing block diagrams in PEDA. From an implementation perspective, the hierarchical nature of these diagrams can be seen to be important in a number of ways. By allowing blocks to be represented in terms of others, the representation can be made compact and maintainable. In addition it melds well with Object Oriented Techniques (Stefic & Bobrow, 1985), which can greatly reduce the amount of programming effort required. The initial very simple PEDA block hierarchy is shown in Figure X.

Each block can generally perform some form of mathematical function. The places for all-block, alt-block and memory in the hierarchy are associated with the way in which the various blocks behave, and will be discussed in the section devoted to the simulation aspects of PEDA, though in brief: all-blocks require all their inputs valid; alt-blocks need only one; and memories are a special case requiring varying input conditions. The all-blocks provide a small set of useful elements including common mathematical functions. There are three special cases within the all-block category, these are demux, round and the maths function. The demux block produces 'n' outputs

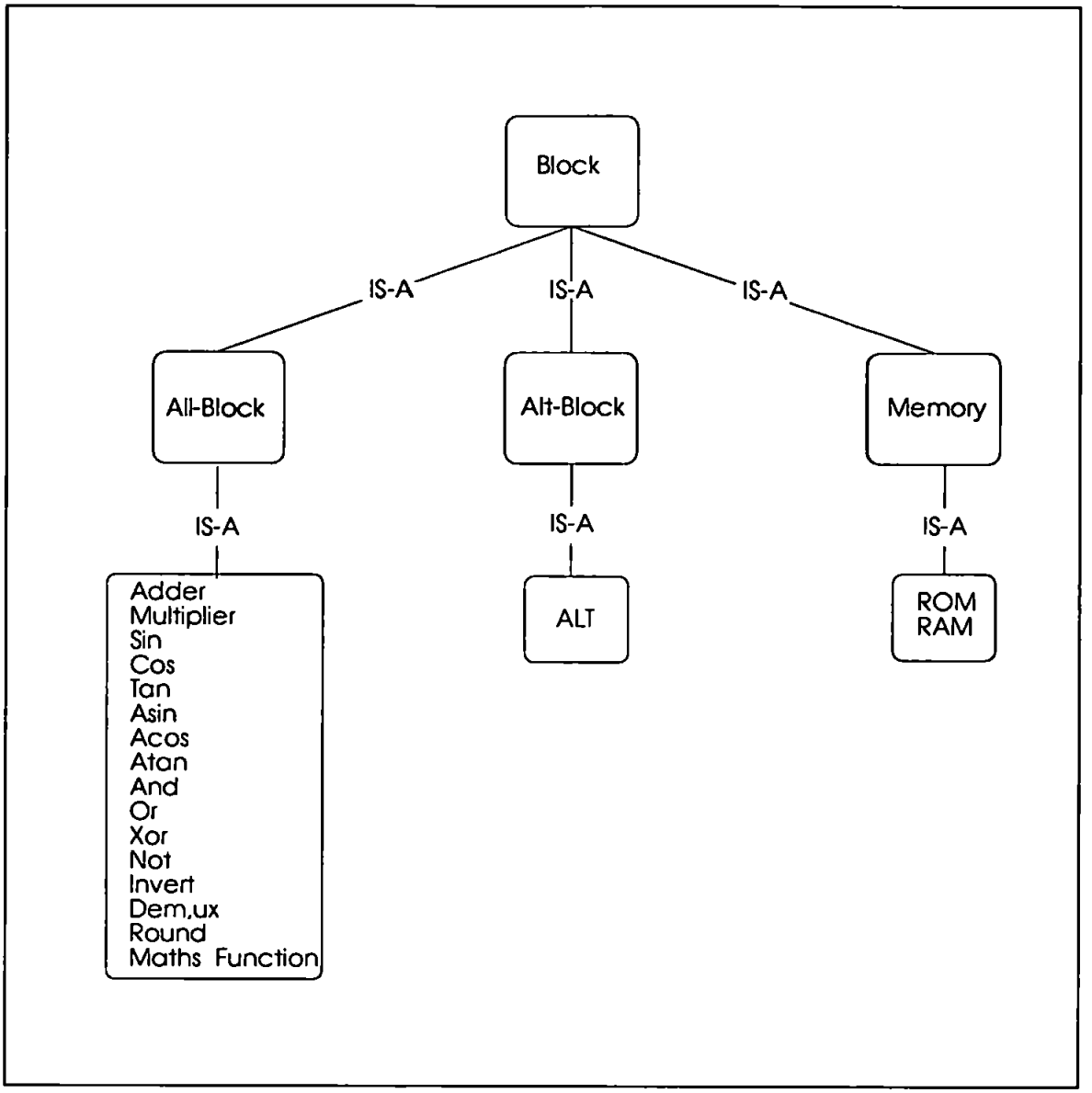

Figure X, PEDA block functionality.

from one input and therefore allows one block output to be connected to the input of many others. The round block performs a variable rounding or truncation operation on an input, allowing experimentation with various data bit widths. The maths function block lets the designer specify an equation, through links to the underlying LISP language, or to symbolic manipulation, and equation solving tools such as REDUCE or Macsyma when operators such as integration are required.

4.3.1.2. PEDA Block Representation

A frame based approach has been adopted in representing functional block diagrams. A simplified view of frames is used in which a frame is viewed as a structure which characterizes an object. This structure contains a number of 'slots', which when filled describe some important aspect of the object. A more complete outline on the use of frames and slots given by Winston (1984). The use of frames for representing a block diagram is convenient and flexible, as it allows any information appropriate to the design to be added as a new 'slot'.

In PEDA each functional block contains a number of slots, which in turn hold various attributes, for example its functional behaviour, structure and other information relating it to other blocks. This is in marked contrast to most models which split the design representation into a number of different domains. In these approaches the Physical, Structural and Behavioural aspects are separated, in accordance with the common consensus on electronic design representation. However as mentioned in chapter 2, it is difficult to separate these aspects in the early stages of electronic engineering design, and so this has not been done in the PEDA environment.

The structure is represented in PEDA as ART Schemata, these are frame like constructs similar in appearance to LISP lists and are discussed in the appendices. An example block shown in Figure XI, will demonstrate the internal structure of blocks.

```
(defschema adder-12
   (instance-of adder)                ; adder-12 is an instance-of adder
   (instance-of block)                ; inherited from adder
   (instance-of all-block)            ; inherited from adder
   (function (if (Nan-Check A B)      ; The Addition Function
      (set-Nan 'Q)
      (setq Q (+ A B)
   )
   (contains                          ; it contains three ports called:
      adder-12-port-A
      adder-12-port-B
      adder-12-port-Q
)
   ; + other slots used by PEDA and ART
)
```

Figure XI, block schema - structure

In this figure the block adder-12 has a slot called instance-of which indicates that adder-12 is an instance of the blocks adder, all-block and block. The instance-of slot is part of the inheritance mechanism and the net result is that adder-12 will inherit set slots and their contents from those other schemata. This is in essence a copying operation, with special rules regarding the how, why and when slots and their values are copied. The mechanism for this will be discussed later. In the example many of these slots have been removed to aid readability. The inheritance mechanism greatly improves productivity and can be seen to be an ideal way in which to generate this type of structure, an instance where there is a good mapping between the language and

problem. The slot 'contains' describes the ports in the block. In this case there are three. Figure XII shows the corresponding structure of these ports.

```
(defschema adder-12-port-A          ; Input port A
  (instance-of input-port)
  (conn-from Or-12-port-Q)          ; Input from another block
  ; + other slots used by PEDA and ART
)

(defschema adder-12-port-B          ; Input port B
  (instance-of input-port)
  (conn-from And-10-port-Q)         ; Input from another block
  ; + other slots used by PEDA
)

(defschema adder-12-port-Q          ; Output port Q of adder-12
  (instance-of output-port)
  (conn-to multiplier-5-port-A)     ; Output to another block
  ; +other slots used by PEDA
)
```

Figure XII, port to port connection - structure

Each port is either an instance-of an input or output port and is connected to its opposite in another block. This is done via the slots conn-from and conn-to. In this example Port A in the adder is connected to Port Q in an Or Block.

The hierarchical nature of block diagram designs can be quite easily accommodated through the creation of further blocks, and linking them to the base block via the slot 'contains'. This is shown in Figure XIII. The block diagram world-1, contains an all-block function-1, which is decomposed into an adder and two multipliers. Data superfluous to this description has again been removed from the block descriptions to aid readability. This data would encompasses the port descriptions and normal inherited information.

```
(defschema world-1
   (instance-of world)
   (contains function-1)
)

(defschema function-1
   (instance-of all-block)
   (contains adder-1)
   (contains multiplier-1)
   (contains multiplier-2)
)

(defschema adder-1
   (instance-of adder)
)

(defschema multiplier-1
   (instance-of multiplier)
)

(defschema multiplier-2
   (instance-of multiplier)
)
```

Figure XIII, a simple block hierarchy

4.3.1.3. Block Templates

When a block is created or copied, the set of slots and values that it has, are dependent upon the type of block it is. When it is created, these values are taken from blocks in a block library, which are used as templates to form the new block structure. When the block is copied, an existing block in a design is used as the template. It can be seen that both operations are very similar. Two mechanisms are used to effect these processes. The first is a straight forward inheritance procedure, using the instance-of relation in ART. This is used to copy slots and values which are identical in the template and created block. The second is a set of rules which are used for slots and values which are different. They are also used to build up the new block structure including any sub blocks or ports in the new block.

When a block is created or copied, the block schemata structure is duplicated. The copying rules use the 'contains' slot value to find successive blocks in the block diagram hierarchy. Each block copied is given a name based upon the type of block it

is. For example adder-1, adder-2 etc. Each copy is then made an instance-of the original block, by giving it a slot 'instance-of' with the value of the original block. The inheritance mechanism takes over and the majority of the remaining slots are created and filled. Finally the 'contains' slot, the interconnectivity and the functionality of the new block structure are then set up to reflect the original. This process is itemised in Figure XIV.

1)	Recursively copy original block using contains slot as a pointer to lower blocks.
2)	Give each copied block a name based upon its type.
3)	Make each new block an instance-of its original, and wait for inheritance to finish.
4)	Make 'contains' slot reflect new structure.
5)	Make interconnectivity reflect new structure.
6)	Make functionality reflect new structure.

Figure XIV, block creation and copying method.

4.3.1.4. Links to the User Interface

Designs are linked quite closely to the user interface. Blocks have a dual identity in that they also have a visual aspect. This is achieved through slots in each block which are used and maintained by the user interface. In addition the initiator for the template copying mechanism is the user interface, so that when the user copies a block icon, the underlying representation is also being copied.

4.3.2. Alternative Designs Within PEDA

4.3.2.1. Alternative Designs

In common with efforts concerning individual designs, the use and representation of alternative designs, or more commonly versions has been extensively discussed in the literature. A common approach mentioned earlier, concentrates on the management of versions in a manner similar to computer software version management. These tools are primarily concerned with versions within the context of projects and the integration of work from different members of design teams. As a result integration of the various partial designs including changes and revisions is handled to maintain consistency.

The management of versions is often treated separately to the designs themselves, and this has advantages and disadvantages. An important advantage is that this type of version system can be used with any design representation tool. This is similar in principle to using a software management tool with any programming language, and as a result a design team can use a preferred system of version control with all their other tools. The main disadvantage with this approach is the lack of integration between the two systems. This is mainly a conceptual problem in the early stages of design, where alternatives and design are intimately entwined. Some recent approaches do integrate designs and aspects of the evolution of designs (versions) through the use of object oriented practices. An interesting use of versions by Lathrop & Kirk (1985), introduces the versional block as part of the design structure, introducing alternatives at any point in the block design diagram.

4.3.3. The PEDA Representation of Alternatives

The requirements for the representation of versions within PEDA are different from many other CAD tools (see chapters 2 & 3). The tool is designed to address the early stages of design, and the aim here is not to provide a method of organizing the most current designs, but to encourage the designer to consider many different design alternatives. Thus the representation is geared more towards alternatives as opposed to versions, in that versions portray the evolution of a design whereas alternatives are viewed as different approaches to the same problem. This is very similar to the concept

of Multiple Worlds (P. Veerkamp et. al., 1989). The structures that address versional issues are therefore not needed, and a simple approach can be adopted, albeit without the flexibility, or complexity of approaches discussed in chapter 3.

The PEDA representation of alternatives relies on information in the form of attributes residing in each block diagram linking them into a alternative hierarchy or tree like structure consisting of derivatives and alternatives in a similar manner to the definition of the version plane by Katz, Anwarudin & Chang (1986). This structure is then superintended by the addition of information again in the form of block attributes explaining the reasons why a particular alternative was made. Figure XV, shows a pictorial representation of an example alternative structure.

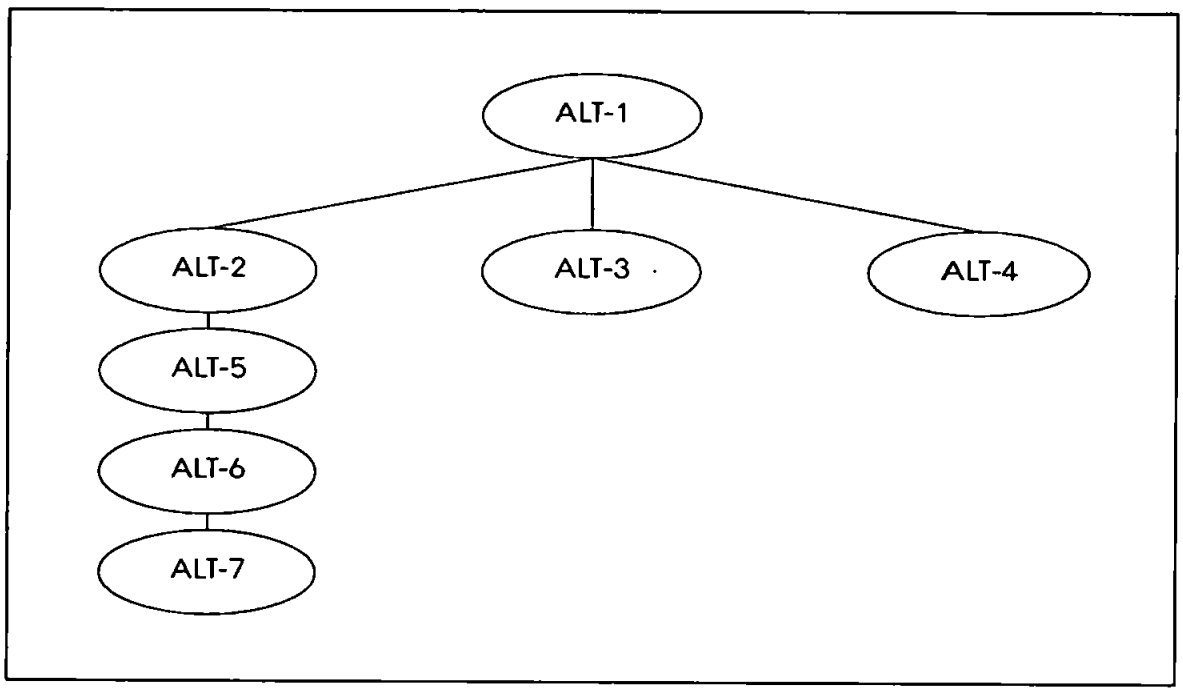

Figure XV, alternative tree.

In this diagram, the alternative designs: alt-2, alt-3 and alt-4 have all been derived from the base design alt-1. Successive changes to alt-2 produces derivatives: alt-5; alt-6 and alt-7. A general rule of thumb for this type of pictorial representation would be, that derivatives are aligned vertically, whilst alternatives are arranged horizontally. However in this example alt-7 could be very different from alt-2, and so could be classified as an alternate design to alt-2. To avoid this form of potential confusion (first mentioned in chapter 3), no explicit distinction is made between alternative and derivative designs in PEDA. Alternatives are made useful, by the

presence of attributes which record the reasons why that alternative was created. A particular alternative will have a whole set of reasons, found by examining it and its preceding alternatives. This is important, because the alternative tree can become quite complex as potential designs are explored, and therefore some means of aiding alternative navigation is necessary.

The previous alternative tree can be used as an example to show the linking of information in the various alternative designs. Figure XVI, shows the textual representation of the seven alternatives alt-1 to alt-7, again with the irrelevant information removed. Each alternative shown contains information which shows which alternative it was derived from and example reasons for their creation.

Finally it should be noted that because alternatives are blocks, there is no need for separate mechanisms to handle their creation and copying.

```
(defschema alt-1
   (instance-of design-alternative)
   (reason "Want a Filter")
)

(defschema alt-2
   (instance-of design-alternative)
   (derived-from alt-1)
   (reason "partial digital, partial analogue")
)

(defschema alt-3
   (instance-of design-alternative)
   (derived-from alt-1)
   (reason "fully digital" )
)

(defschema alt-4
   (instance-of design-alternative)
   (derived-from alt-1)
   (reason "fully analogue")
)

(defschema alt-5
   (instance-of design-alternative)
   (derived-from alt-2)
   (reason "vary input blocks after simulation")
   )

(defschema alt-6
   (instance-of design-alternative)
   (derived-from alt-5)
   (reason "vary output blocks after simulation")
)

(defschema alt-7
   (instance-of design-alternative)
   (derived-from alt-5)
   (reason "tweak filter coefficients after simulation")
)
```

Figure XVI, alternative tree structure

4.4. The Management of Alternatives, and History Tracing.

4.4.1. The Management of Alternatives in PEDA

The management of alternative designs within PEDA can be divided into two parts: The first deals with the conventional aspects of version management for example the saving and restoring of designs; and the second with reducing the cognitive loading on the user.

Very little needs to be said about the parts of PEDA which manage the saving and restoration of designs to disk, as they offer the bare minimum of functionality, providing only saving and restoration of the entire design workspace. Designs are saved in the following simple manner:

1) Form a list of the relevant blocks (schemata).

2) Iterate over the list and Write the text of each schemata in the list to a save file.

The restoration of schemata files is even simpler as ART directly provides the functionality for it.

For the second part, some aspects of alternative management have been automated by the addition of controlling heuristics in order to reduce the effort of creating alternatives during design work. The aim here has been to examine the history trace and the evolving design and determine, where important decisions are being made and then create alternatives together with the reasons for their creation. At present automatic support for this in PEDA is very restricted. Currently the history trace is examined for the occurrence of design changes after a simulation in that alternative. The rationale being that the designer is about to try out a new idea and so a fresh alternative is created, complete with links to the previous alternative and the reason for creating the new alternative being a design alteration after simulation. This simple example is shown pictorially in Figure XVIII, and in outline form in Figure XIX.

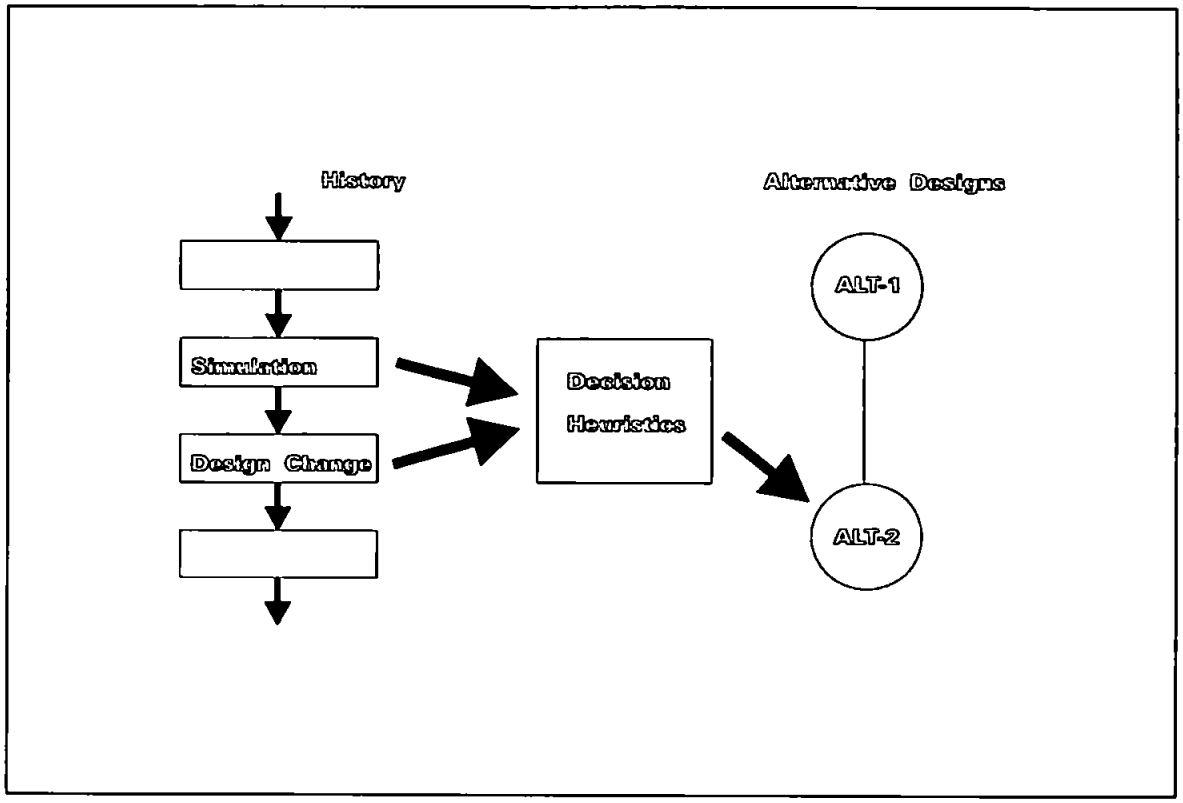

Figure XVIII, automatic creation of alternatives (pictorial)

Rule:

IF current mode is construction
AND last mode was simulating
AND modifying a block diagram

THEN create new alternative from current design

Figure XIX, automated creation of alternative (example rule)

4.4.2. History Tracing.

The history trace is a useful device in electronic design as it offers a partial record of the design steps used to produce a design. Conventional uses of history provide the user with a record of their selections and what the system has done. The information typically in a history trace is of quite a low level, though there are exceptions. For example, Mostow (1989) uses a record of history or "design plan" to

produce alternative designs. The recorded history of design decisions or design selections, made in a previous design can be replayed upon a slightly different starting design to partially complete it. These design decisions are derived from a history of menu commands which are at a comparatively high level.

The Basic history mechanism within PEDA is indeed a straight forward record of all design activity within the system. Whilst this can create vast volumes of data, especially during simulations, it does allow a permanent record of all design work in an albeit low level form. Much of this information is of dubious use to the designer, but it does provide any heuristics present, access to the full history. A sample history trace is given in Figure XX and portrays the commands used to produce a block diagram and subsequent simulation.

```
[create world world-1 1]
[select adder 2]
[create block adder-1 world-1 100 100 3]
[select multiplier 4]
[create block multiplier-1 world-1  200 100 5]
[connect adder-1-portQ multiplier-1-portA 6]
                        .
                        .
                        .
[Start Simulation 100]
```

Figure XX, example history trace.

The history trace is generated in two ways. The first appends a command to all the command rules in the user interface. When the user makes a selection, a rule in the user interface part of PEDA responds. The additional command placed in each user interface command rule places a fact in the ART database, indicating the command, together with relevant arguments and the command number. This number is required so that the order of commands can be determined. The second method monitors the changes to blocks and associated schemata, so that items such as a program trace can be provided. This activity is simple requiring only one rule matching all block schemata, but the generated history trace can be enormous.

4.5. Links to User Interface

It can be seen that both the management of alternatives and history tracing mechanisms are closely involved with the user interface. The history mechanism is actually initiated within the user interface command rules, and the decision rule uses the history trace, in part.

4.6. The PEDA Constraint System

4.6.1. Introduction

The use of constraints in electronic engineering design tools has been touched upon in chapter 3, with systems performing constraint propagation and maintaining consistency. In the abstract representation any information can be viewed as constraining, though it might not be associated with any constraint propagation or truth maintenance system.

This approach has been adopted in PEDA with constraints being any represented information, which might have constraint propagation attributes associated with it. This is due to the fact that the main aim of constraints in the representation (or PEDA for that matter) is to bolster comparisons, and not to maintain consistency, though of course this is sometimes necessary. Constraints generally describe some feature, and for items like the structure, this would entail aspects like the block hierarchy and interconnectivity, or a description of design behaviour. These may or may not have associated constraint propagation heuristics.

Additional constraints in this area which have been included in PEDA are attributes like speed, power, chip area and design time. Some constraint propagation heuristics have been incorporated to propagate these aspects from the lower levels of a design hierarchy to higher ones, deriving for example the critical delay in a block diagram. Certain constraints dictate limits, whereas other dictate desirable features.

These various factors can be explained with the use of an example. Figure XXI shows a partially described top level block, with the unimportant information removed.

```
(defschema design-1
   (instance-of design-alternative)          ; Normal structure
   (contains multiplier-block-1)      ; information
   (contains multiplier-block-2)
   (contains multiplier-block-3)

   (must-have (speed fast))           ; Absolute requirements
   (must-have (chip-area large))
   (must-have (power medium))
   (must-have (design-time medium))

   (desirable (speed very-fast))      ; Desirable requirements
   (desirable (chip-area medium))
   (desirable (power low))
   (desirable (design-time zero))

   (speed fast)                       ; Actual characteristics
   (chip-area large)
   (power medium)
   (design-time low)
)
```

Figure XXI, example use of Constraining Information.

All the information in the design alternative design-1 is a constraint on the design. Starting at the top it can be seen that the design is constrained to be a design-alternative which contains a number of other blocks. These multiplier blocks need not be described here, but by the virtue of their existence they limit the realization of this particular alternative. Other constraints which form a boundary on the design are the must-have's. These invoke the use of heuristics to derive the associated actual values from elsewhere in the design hierarchy. This may be achieved through inheritance (which is a simple form of constraint propagation), for the example chip-area, where the value is propagate from a library component. Heuristics can be used to infer other values, for example estimating the design time required in a large and complex component, or estimating the overall speed of the design. The desirable constraints place additional emphasis on the overall requirements, though at no time are any constraints used to remove a design alternative. They are used to compare designs. The actual characteristics are shown at the bottom of the block, and define the capabilities of the design alternative in actuality.

4.6.2. Constraint System Implementation

4.6.2.1. What Constraints are in PEDA

All information within the PEDA system is regarded as a constraint, however the constraint comparison mechanism deals with only certain types of constraint at present. Constraint requirements are generally specified by the user and are of the form (must-have (<*constraint*> <*value*>)) and (desirable (<*constraint*> <*value*>)). They are usually placed in the top level block diagram description. Constraint values can be specified by the user, inherited from data libraries or derived from existing constraint values in other parts of the design (or other designs), by heuristics. They are generally of the form (<*constraint*> <*value*>). The format of constraint requirements and constraint values is summarized in Figure XXII.

Constraint Requirements:

1) (must-have (<*constraint*> <*value*>)
2) (desirable (<*constraint*> <*value*>)

Constraint Values:

(<*constraint*> <*value*>)

Examples:

Constraint requirements:

(must-have (power low))
(desirable (power very-low)

Constraint value:

(power very-low)

Figure XXII, constraint format

4.6.2.2. How Constraints are used in PEDA

Constraints are primarily used in PEDA to compare alternative designs. This process can naturally be split up into three layers (Figure XXIII):

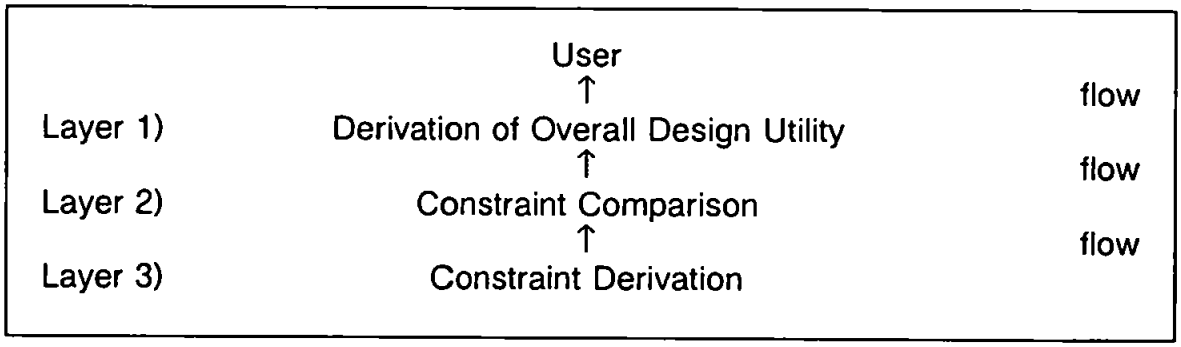

Figure XXIII, constraint system layers

The first layer is concerned with producing an overall value or utility for each design which is used as a direct indicator of the overall suitability of that design with regard to the requirements. This is fed by the results of the second layer which performs the comparison of individual constraints to form a set of desired utilities. The third layer derives the actual constraint values required from available constraint values according to inbuilt algorithms or heuristics. This overall flow of information is also shown in Figure XXIII.

At the bottom layer exists the parts of the constraint system which satisfy requests for constraints which have yet to be derived. Figure XXIV shows the overall method by which such constraint requests are met. As can be seen a rule based approach is used. The figure shows a generic rule, but in actuality a rule for each type of constraint is used.

Rule:

IF can satisfy a constraint request directly:

THEN apply corresponding algorithm or heuristic to available constraints to do so.

Figure XXIV, constraint derivation

The central concept is that any particular unresolved constraint requirement will have a rule trying to satisfy it. A rule will be able to succeed when all it's criteria are met, and as a result may enable other rules to succeed. The process will continue if possible until all unresolved constraints are derived. This effect can be explained with an example. Figure XXV shows the constraint satisfaction rule for the calculation of chip area.

1) Wait until following rule activity has finished and required chip-area is valid.

Simple approach assuming that chip-area calculation are enabled.

Rule: Calculate a block's area.
IF there is a block(i),
AND that block contains a request for block area.
AND that block contains other blocks(ii).
AND all those blocks(ii) have chip areas.

THEN calculate the block's(i) chip-area as being the sum of the blocks'(ii) chip-areas.

Figure XXV, example of constraint derivation: chip area

The calculation of chip area for a block is quite simple being the sum of the chip areas of all blocks contained in that block. Consider a block diagram design 1 containing 2 blocks, block 1 and block 2. Both block 1 and Block 2 contain two blocks, block 3 and block 4, and block 5 and block 6 respectively. Blocks 3 to 5 have chip-areas inherited from a library, the other blocks' chip areas have yet to be determined. The chip area constraint satisfaction rule will try and derive the chip areas for blocks 1, 2 and 3. It cannot do this for the design as the chip areas for blocks 1 and 2 have not been calculated. However it can produce the chip area for block 1, by adding the chip areas for blocks 3 and 4. In a similar manner the chip area for block 2 can be derived. Only after both blocks 1 and 2 have chip areas can the rule derive the chip area for the design. Figure XXVI shows the chain of rule firing and constraint derivation for this example. It should be noted that actual numeric values of chip-area are used in this example. For this type of constraint, numeric values are required and so symbolic terms

```
The design:
        design 1:
                chip area is required.
                contains blocks 1 and 2.
        block 1:
                contains blocks 3 and 4.
        block 2:
                contains blocks 5 and 6.
        block 3:
                chip area 2
        block 4:
                chip area 1
        block 5:
                chip area 3
        block 6:
                chip area 4
Rule firings:

First
        For block 1, chip area is 2 + 1 = 3
Second
        For block 2, chip area is 3 + 4 = 7
Third
        For design 1, chip area is 3 + 7 = 10.

Rule2 fires: chip area required in block 3
Rule1 fires chip area in block 3 is 1 + 3 = 4
Rule1 fires chip area in block 1 is 4 + 2 = 6
```

Figure XXVI, chip area constraint derivation example.

would be converted via a lookup process using user or library derived tables.

The second layer of the constraint system is concerned with constraint comparison, and making sure that the required constraint values are available so that the comparison process can take place. Figure XXVII Shows the overall order of these processes. The constraint requirements are generally supplied by the user or from a library, but the system does cater for the eventuality of deriving constraint requirements if there is a real need, using the same method as for constraint values.

Comparison of requirements and values is done through a semi heuristic technique. Symbolic values are converted into corresponding numerical values using via conversion factors supplied by the user or libraries, where applicable. The difference between these values is then converted to a result in a particular range using heuristics

1) IF The constraint value has not been derived:
THEN derive it: -> Constraint Derivation.

2) IF the constraint requirement has not been derived:
THEN derive it: -> Constraint Derivation.

3) Compare constraint value with constraint requirement, to produce a constraint result.

Figure XXVII, constraint comparison.

or a non linear transfer function. Where a must-have constraint requirement is exceeded the violations can be flagged. An example (shown in Figure XXVIII) using chip area can outline the process of constraint comparison.

design-1:
must have constraint requirement: chip area large.
desirable constraint requirement: chip area medium.
chip area constraint value: 25 units.

lookup values:
chip area large is 100 units.
chip area medium is 10 units.

test for constraint violation:
25 is less than 100 so pass.

Compare constraint:
use heuristic, returns constraint result of 7 when given constraint value of 25 and constraint requirement of 10.

Figure XXVIII, constraint comparison example.

A design has a must have constraint requirement for chip area of large. It also has a desirable chip area of medium. The derived chip area constraint value for the design is 25 units. The derived value is already a numerical value and so need not be modified. A chip area of large is looked up in the technology table for the process under consideration and returns a value of 100 units. Similarly medium gives a value of 10. The value of 25 is less than 100 so a must have constraint violation does not occur. The value of 25 is compared with 10 by a heuristic to give a constraint result or

utility of 7, where 1 is bad and 10 is good. This value is then used by the first layer in the constraint system.

The top layer in the constraint system is concerned with the derivation of the overall design utility for alternative designs. This functionality is outlined in Figure XXIX.

1) Gather all relevant constraint results: -> Constraint Comparison.

2) Derive importance value for each constraint result, from: i) user input; 2) library; 3) other method.

3) If required, convert each importance value into a numeric value using heuristic or algorithmic technique.

4) Multiply each constraint result (numeric) by importance value (numeric), to give overall utility (numeric), for design alternative.

5) If required, convert overall utility into symbolic value (such as low, medium & high) using heuristic or algorithmic technique.

Figure XXIX, derivation of overall design utility

The results from all constraint comparisons are combined using an approach similar to MAUD (discussed in chapter 3). In this method each constraint result is multiplied by an separate importance factor (which is usually linear). The resultant values are than added to give a final utility for each design, after normalization. In addition conversion to and from symbolic and numeric values is provided where necessary.

An example will clarify the operation of the top layer of the constraint system. A set of design alternatives are being examined. These designs design-1 to design-3 are shown in Figure XXX.

The three constraints of importance here are chip-area, power and speed. The constraint results of these constraints for all three designs have already been determined by the lower layers in the constraint system. These results are then multiplied by the corresponding importance values of 0.5, 0.3 and 1 respectively. When added together and normalized (divided by the total number of importance values) the resultant utility

is an indication of the desirability of that design. In this case the design design-1 is the most desirable.

```
(defschema design-1
        (chip-area medium)
        (power high)
        (speed very-fast)
)
(defschema design-2
        (chip-area medium)
        (power low)
        (speed slow)
)
(defschema design-3
        (chip-area high)
        (power medium)
        (speed fast)
)
chip-area lookup factors:
        very-small 10, small 7, medium 5,
        high 3, very-high 1.

Constraint results:

        design-1: chip-area 5, power 3, speed 10.
        design-2: chip-area 5, power 7, speed 3.
        design-3: chip-area 3, power 5, speed 7.

Importance factors:
        chip-area       .5
        power           .3
        speed           1

Final utilities
        design-1: (2.5 + 0.9 + 10)/1.8 = 7.4
        design-2: (2.5 + 2.1 +  3)/1.8 = 4.2
        design-3: (1.5 + 1.5 +  7)/1.8 = 5.6

Selected design: design-1.
```

Figure XXX, overall design utility example

4.6.3. Links to the User Interface

The constraint system is linked to the user interface in a number of places. However unlike some other parts of the representation there is a relatively clear line between the two. The instigation of the design comparison process is the most immediately apparent point. The second is the display of results in the form of overall design utilities. The various lookup tables for importance values and conversions are also points of contact, as is the setting and modification of design constraint requirements and values.

4.7. Simulation of Designs

Simulation has been greatly used by electronic engineers to check the validity of their work. In this broad domain simulation has been greatly used to model both digital and analogue designs, though traditionally commercial CAD packages have concentrated on digital logic simulation of circuit diagrams. More recent commercial efforts have combined analogue and digital simulation techniques, allowing the verification of mixed designs, expanding markedly the usefulness of such tools. In the VLSI arena simulation tools have addressed more stages of the design activity hierarchy, commonly classified as: Behaviour at the top and most abstract; Function; Logic; Gate; Circuit and Switch at the bottom. Performance considerations are considered very important at the later stages, due to the size and complexity of the design at that level, for example: simulating a large (1 million transistor) microprocessor at the switch level requires a very high performance simulator, if reasonable run times are envisaged. High performance is basically required because of the requirements of both simulation speed and accuracy. At the high levels the design will consist of a relatively small number of interconnected elements of corresponding high functionality. As design progresses these will be decomposed into a larger set of elements, until at the bottom individual components are specified. Behavioural simulations can quite accurately model the behaviour of devices, but are comparatively slow. Switch level simulators approximate devices to switches, and are relatively fast. Thus in the initial stages, overall design behaviour of the top level design is best verified using a behavioural simulator, and in the later stages the overall behaviour of the low level

design is adequately verified using a switch level simulator, with the other types in between. A general purpose simulator could be used for all levels, but performance would suffer at the lower levels. This happens with analogue designs which require more accurate device level simulators, such as SPICE, but again the run times can be prohibitive. The net result is that different types of simulator have evolved to offer the best compromise between speed and accuracy at a particular level. Generality is supported by combining approaches into mixed mode simulators, which support a number of levels. This whole area has been well addressed and there are many examples in the design automation literature.

Efforts in the knowledge based arena have produced simulators which tackle the behavioural and functional levels which tend to be based around a high level language, commonly LISP, or an Object Oriented language, though there are a few rule based systems. This can be explained from the fact that the main emphasis of these systems is in applying knowledge based techniques (encoding expert knowledge in heuristics for example) to a particular problem, circuit synthesis for example, and these languages provide an easier platform for this type of work. Also at the higher levels less raw computation is required, as there are less elements to model, though some aspects of behaviour can be troublesome (differential equations and integration for example). This is also applicable in the initial stages of design, where the behaviour is described by mathematical equations, and it is at this stage where the PEDA simulator is designed to operate.

In terms of general usage the relatively common digital circuit simulator is often used to simulate the behaviour of digital logic circuits. However in the context of this project a mathematical function simulator is more general. It simulates the behaviour of connected mathematical functions. Also implicit in the implementation of digital circuit simulators is the notion of time. These simulators model behaviour at specific instants separated by a delay. Approximations to continuous behaviour can be made if the delay is small enough. Another type of simulator follows from data flow principles in which data flows along a network of connections. At the junction of connections are nodes which have a mathematical function associated with them. When all the required data is at a node, it is evaluated and the results of the calculations passed on down the network. This type of arrangement is very flexible and has the advantage that time sequencing is not an integral part of it. It can be regarded as a more general type of

simulator than the digital circuit simulator. The PEDA simulator is of this second type. A simulator was required that possesses a great deal more flexibility than what the simple digital simulator has to offer.

4.7.1. The PEDA Simulator

The simulator within the PEDA environment was conceived as a general purpose system. It was designed like many others to perform a number of roles: the first, offers a means of modelling the behaviour of arbitrarily abstracted block diagrams, placing near the top level of the design activity hierarchy (behaviour...circuit); The second aim is to provide a means of verifying that the behaviour of the block diagram at any level corresponds to the required behaviour in the specifications. The third aim is to produce a rich source of constraint information. These aims are met with the use of multiple paradigms, combining both rule based and object oriented programming techniques, to produce a simple yet powerful simulator engine. The underlying richness of the implementation languages are used to full effect, allowing the specification of design behaviour as mathematical equations, stored within the blocks themselves. Abstract simulation is performed with no explicit use of events or delays, in accordance with the psychological requirements, and abstract representation. However if these devices are required, they can be accommodated easily, by changing the functional behaviour of blocks, through the addition of an extra port on each block which would receive synchronisation data.

4.7.2. PEDA Simulator Operation

The PEDA simulator provides the basic underlying mechanism by which the behaviour of block diagrams is modelled. It controls the flow of data between blocks and the subsequent equation evaluation within them. There is a close relationship between data, behaviour and the structure in that the data is physically moved around the structure during simulation.

The basic operation of the simulator is as follows:

1) Data packets are created at the outputs of input file blocks, using the data stored within a file.
2) Data packets move along the connections between blocks, from the output of one block to the input of the next.
3) The presence of all the required data packets at the inputs to the block will cause it to evaluate them according to the equation specified within.
4) The resultant evaluation will create new packets at the output(s) of the block and the packets at the inputs are destroyed.
5) Output file blocks store their input data in a file.

The previous steps are not completed in any set sequence, making the simulator essentially asynchronous. It is in fact data driven, and operations are performed on a demand basis, though regulating mechanisms have been added to improve the interactive performance with respect to the user, spreading the evaluation of data across all viable blocks in time. If this were not done evaluation may be performed in a batch processing manner, with the processing of all the data through one block at a time. This is a problem due to the recency action of the ART inference engine, in which the most recent patterns to match have a greater priority than new ones (see appendices).

4.7.3. PEDA Simulator Implementation

The simulator combines the use of rules and Object Oriented LISP written in ART. Its operation is centred around the movement and manipulation of data in the form of data packets around the block diagram. This approach is in keeping with the general philosophy of the tool, in that the operation is closely matched to the conceptual views of simulating the operation of a block diagram, and simplicity. The explanation has been divided into a number of areas: The first discusses the makeup of data packets, how they are moved around the block diagram and how they are maintained. The second deals with the data driven operation of the blocks in response to valid data at their inputs, and the third deals with the evaluation mechanism of the data within the blocks themselves.

4.7.3.1. Packets

PEDA uses a flexible arrangement for the representation of Data Packets. They may be simple structures embodying just data and location, or they might contain varying additional information dependant on need. The inheritance mechanism is used both for new packets and the associated functionality required. As a result new types of packet can be easily added. Various numeric data types are supported at variable levels of accuracy or uncertainty. An example packet is shown in Figure XXXII.

```
(schema P-1
  (instance-of packet)          ; A data Packet.
  (node multiplier-1-portA)     ; It is associated with this port.
  (data-type fixnum)            ; A 32 bit integer.
  (overflow null)               ; Set to true is overflow has occurred.
  (data 7)                      ; Has a value of 7.0
)
```

Figure XXXII, packet structure

This particular packet P-1 contains the integer value 7. The value 7 is accurate because an overflow did not occur when it was created. The packet is at present situated in the input port, port A on multiplier-1.

4.7.3.2. Packet Movement

Packets are moved by altering two types of slot. The first is the node slot in a packet, the second is the packet-link slot in a port. The node slot is used to associate a packet with a particular port, and the packet-link slot is used to maintain an ordered queue of all the packets residing at a particular port. To Move a packet the node slot is altered, and the corresponding packet-link slots in the source and destination ports changed to reflect this. A typical packet-link slot in a port is shown in Figure XXXIII.). Packet queues are maintained so that ordering of data is preserved when a number of packets are at a port, for even though exact timing information is not required in the simulator, the order of data must be maintained for the results to be meaningful. The decision to move a particular packet is made by rules, candidate packets are handled

on a random basis, and as a result packet movement is essentially asynchronous. An outline of the simple rule which moves packets is given in Figure XXXIV

```
(defschema input-file-1-portQ
  (instance-of port)                ; This is a port.
  (conn-to multiplier-1-portA)      ; It is connected to another port.
  (packet-link (P-1 P-2 P-3 P-4))   ; The queue of packets.
)
```

Figure XXXIII, packet queue

```
Rule:  Faster packet move
       IF there is a port
       AND that port has data packets
       AND that port is connected to another port

       THEN move the data packets to the other port *

*Note moving packet just involves modifying the packet link slot.
```

Figure XXXIV, outline of packet move rule

4.7.3.3. Packet Maintenance

Packets are maintained, through the use of inheritable OOP code. This handles all aspects of packet maintenance, including creation, deletion, and slot manipulation.

4.7.3.4. Data Driven Operation of Blocks

The operation of most blocks is managed by a small number of rules. They observe the input ports of blocks and invoke the data evaluation mechanism when the input criteria are satisfied. This generally occurs when data packets reside at all the inputs of a block. The exceptions to this simple approach are blocks with non standard input requirements, such as memory devices, files and the ALT block. In these cases

slightly modified rules have been used. Figure XXXV shows an outline of the relevant rule for the all block.

Rule: function all block
IF there is an all block
AND there is a data packet associated with all its input ports

THEN invoke the data evaluation functionality for that block.

Figure XXXV, all block evaluation invoking rule.

4.7.3.5. Data Evaluation

Data evaluation is achieved by applying the functionality described within a block to the data at the input ports. Object Oriented Programming techniques have been used to achieve this with basic types of blocks, which include All-blocks, Alt blocks and the memory devices. Figure XXXVI shows how this is done for the all block.

1) Examine the function-conn-in slots to determine the input variables and the ports which are associated with them.

2) Do the same for the function-conn-out slots and output ports.

3) Get the relevant input data and bind it to the corresponding variables.

4) Evaluate (execute) the functionality of the block defined in the function slot.

5) Take results and convert to data packets and place in output ports.

6) Remove input packets.

Figure XXXVI, all block evaluation

4.7.4. Links to User Interface

The simulator is relatively unconnected to the user interface. The only direct control starts or stops simulation, all other effects are achieved by manipulating the blocks themselves or observing the results. For example if floating point results are required then the data type slot of the relevant blocks is changed to floating point. Similarly an observation of the packets gives a simulation trace.

4.7.5. PEDA Simulation Example

This basic operation of the PEDA simulator is best outlined with an example. The block diagram for a very simple design is shown in Figure XXXVII.

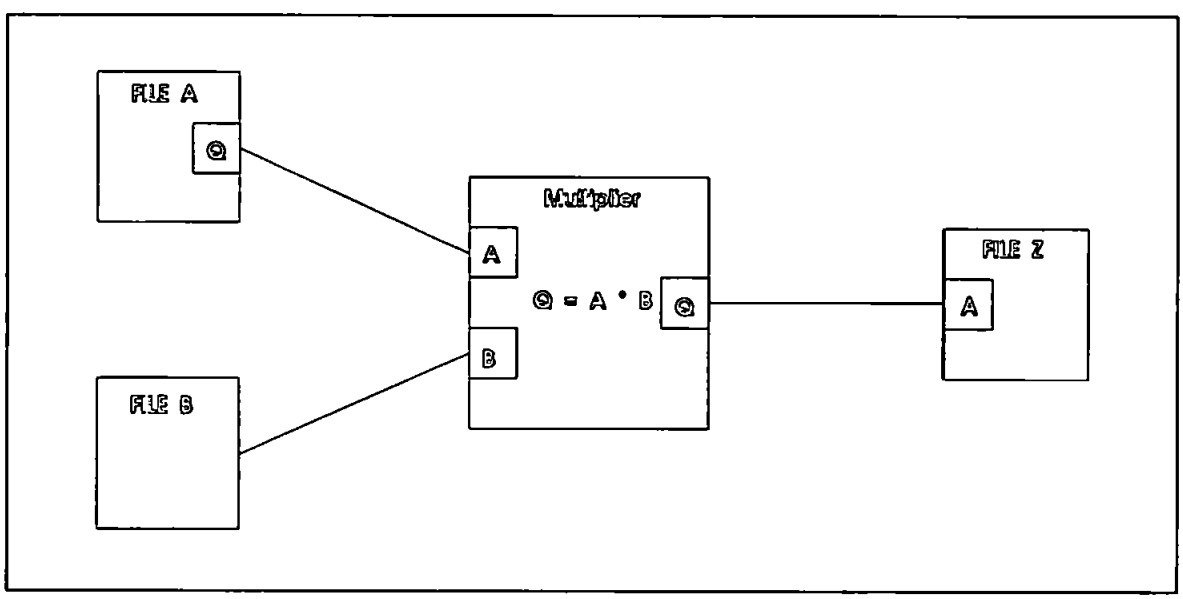

Figure XXXVII, example

The design consists of a 2 input multiplier, fed from two input files: fileA & fileB, and is in turn connected to one output file: fileZ. Initially data is read in from the input files and is placed on their outputs as data packets (Figure XXXVIII). These data packets are moved along the connections linking the input file blocks to the multiplier (Figure XXXIX). When the all the inputs to the multiplier are ready, the data is evaluated, the old data is destroyed and the result made available at the output as a new packet (Figure XL). This in turn moves along the connection to the output file block, where the data is stored in a file, and the now redundant data packet is destroyed

(Figure XLI). The above set of sequences are performed until the input data is exhausted.

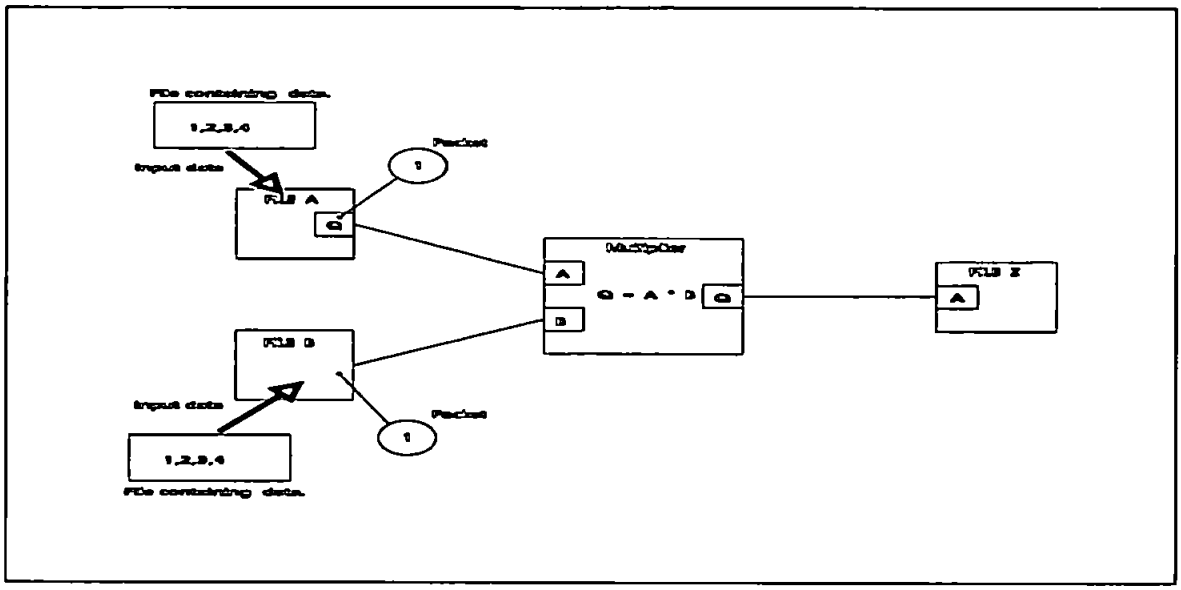

Figure XXXVIII, input data from files

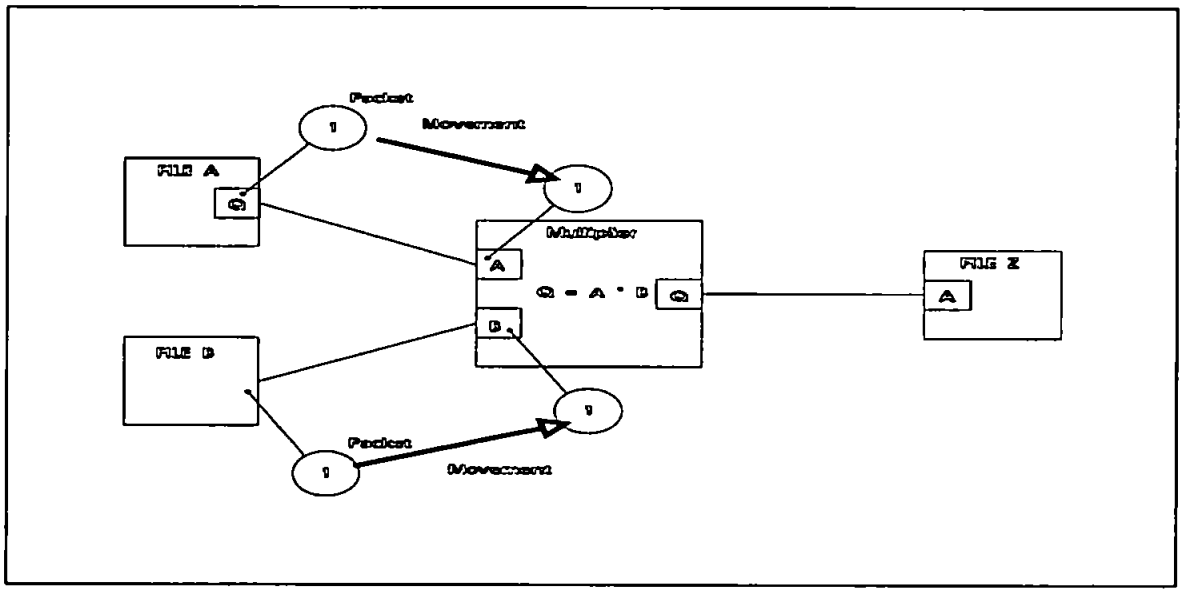

Figure XXXIX, packet movement between blocks

The overall approach of the simulator is centred upon the fact that the rules and OOP code provide the means by which the behaviour is simulated, and are not themselves descriptions of the behaviour. This confers some advantages in that it allows the implementation of the simulator to be changed without affecting the simulation behaviour, and secondly it allows the modelling of arbitrarily difficult behaviour. In this

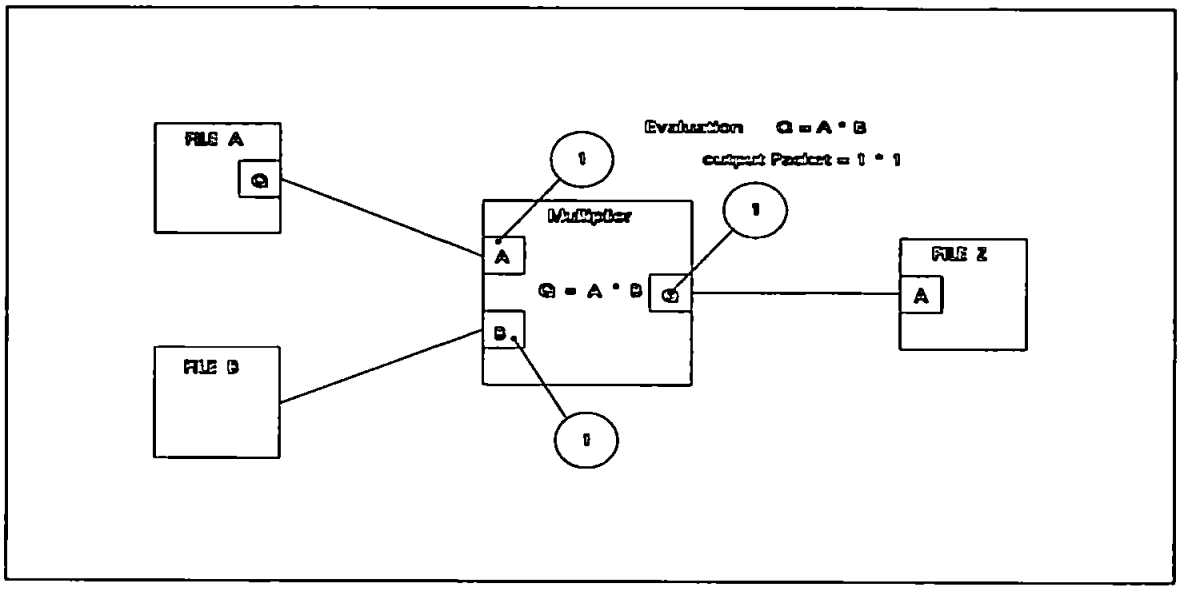

Figure XL, evaluation of data

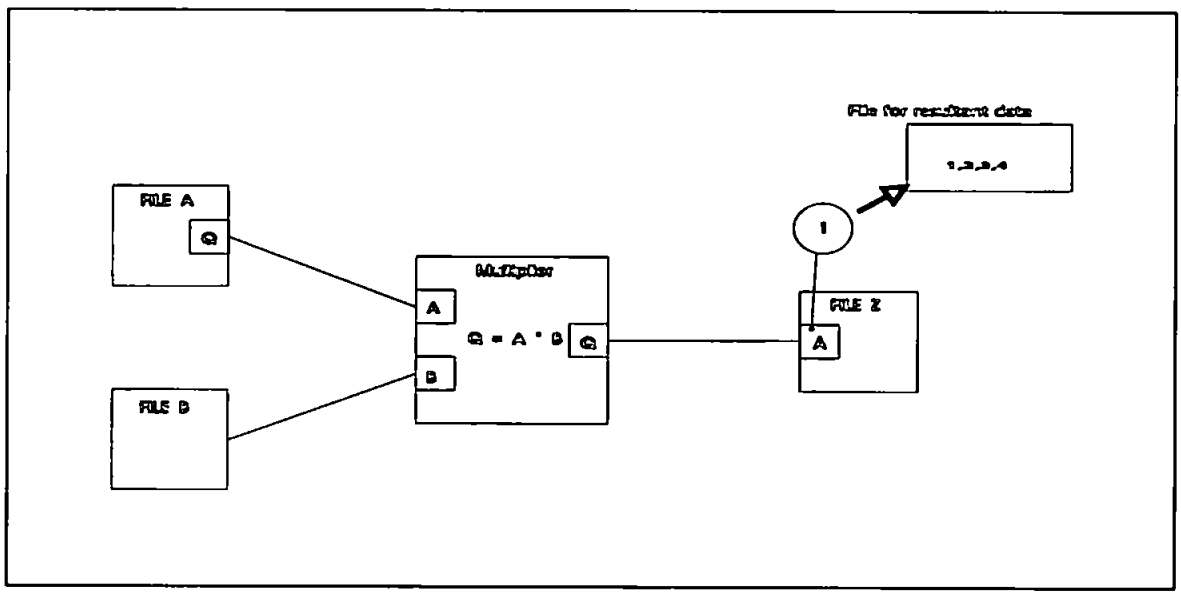

Figure XLI, output data to files

implementation the behaviour is described in LISP, but within these confines it can be any mathematical equation as long as the description language can manage it. A common attribute of many simulators is that they limit the modelling of behaviour to a small set of functions, usually logical. This is true to some extent in this case, but the limits are only specific to the implementation language, and not the method in general.

4.7.6. Feedback and the Alternative(Alt) Block.

In a event driven data flow simulator a number of problems can arise when feedback is used. In these cases at least one of a block's inputs is dependent on its output. An example of feedback is shown in Figure XLII. In this port B on the multiplier is dependent on its output from port Q. The majority of blocks require that all their inputs are available before evaluation will occur. As a result a block with feedback will never evaluate. This problem is solved in the PEDA simulator with a special type of block called the Alt block. This block does not obey the general rule of evaluation and will pass any input to its output. When this block is used in the feedback path in conjunction with a setup data from an input file block, the feedback problem can be avoided. A secondary advantage of this approach is that it makes explicit the initial conditions of all blocks in the block diagram. The placement of the Alt block can also be seen in Figure XLII.

File A contains the data to be used during the simulation, whilst File i contains the initialisation data for port B of the multiplier. Without this data simulation would not proceed as the input requirements for the multiplier would not be satisfied.

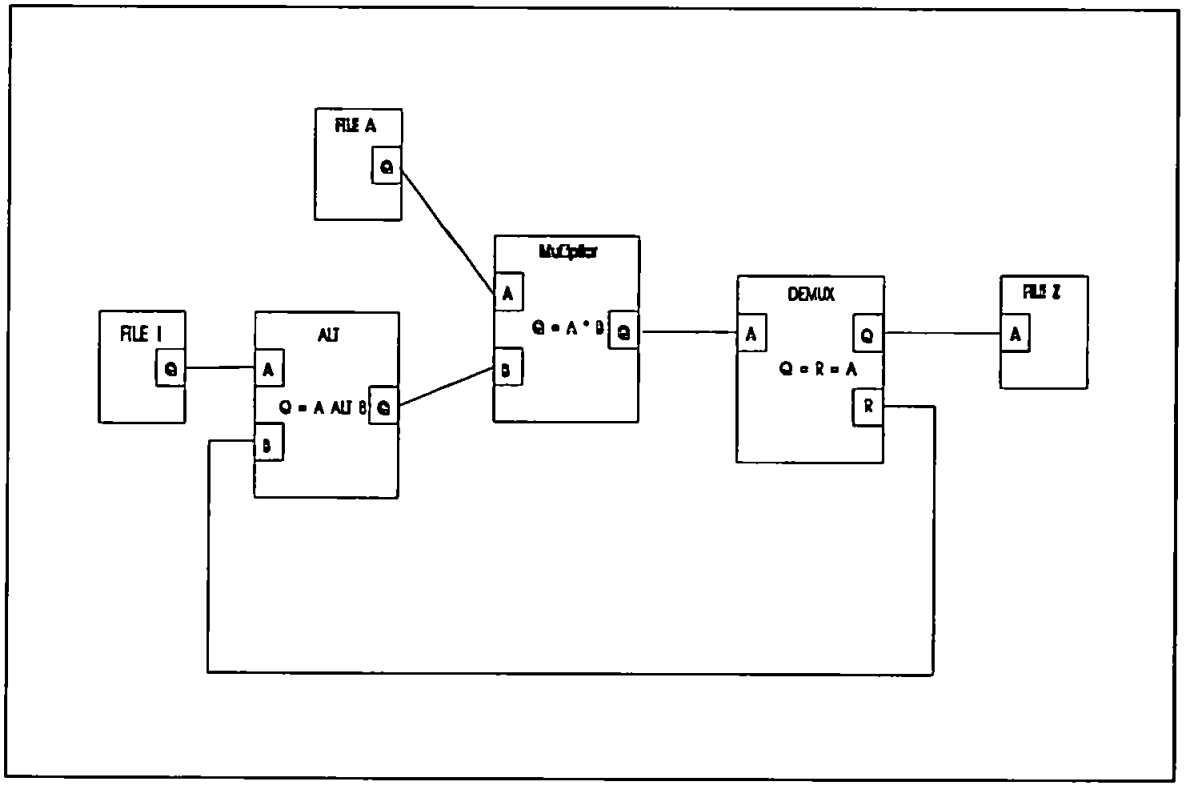

Figure XLII, use of the alternative block.

4.8. Integration in the PEDA Representation: An Example

The following example shows how the basic parts of the PEDA implementation are used and work together. The example given is that of a engineering producing a set of different designs and then comparing them.

The designer interacts with the PEDA system using the user interface. To start a new design the user selects a block from the library palette. When this block is placed upon the drawing board, a skeletal block is created, containing mainly graphical data. The template copying mechanism is invoked and the block is filled out to contain all the required data. The first block in this example is an adder. The block structure of the adder created is shown in Figure XLIII.

It should be noted that the graphical information maintained by the user interface has not been included in this example. The adder has two input and one output port, a number of constraint values for chip area, power, speed and design time, the functionality of an adder (function slot), and links between the functionality and the ports (function-conn slots).

Two more blocks are created for this design, both adders, adder-2 and adder-3. They are very similar to adder-1 and so are not shown. For this design the outputs of adder-1 and adder-2 are connected to the inputs of adder-3. To do this the conn-from and conn-to slots of the respective adders' ports are modified to point to the corresponding ports. This can be seen in Figure XLIV.

Four input files and one output file are added to the design, and linked to the unused input and output ports.

The user interface can then be used to modify the behaviour of the blocks specifying the base number type that each block works with. This data is stored in the blocks and will be used by the simulator. When the simulator is invoked the user interface is used to watch and display the results of the simulation. The dynamic operation of the simulator is explained in section 4.7 with an example in 4.7.5.

At any time the history tracing mechanism can be enabled, producing a record of events in the ART fact database. This is discussed in section 4.4 an example partial history trace is shown in Figure XX.

If after simulation the designer decides to modify the design, then the decision test rule will fire and signal the creation of a new design alternative. The template rules

```
(defschema adder-1
   (instance-of all-block)
   (instance-of adder)
   (chip-area medium)
   (power low)
   (speed 100)
   (design-time v-low)
   (function (Setq Q (+ A B)))
   (function-conn-in (function A in adder-1-port-A))
   (function-conn-in (function B in adder-1-port-B))
   (function-conn-out (function Q to adder-1-port-B))
   (contains adder-1-port-A)
   (contains adder-1-port-B)
   (contains adder-1-port-Q)
)

(defschema adder-1-port-A          (defschema adder-1-port-B
   (instance-of input-port)           (instance-of input-port)
   (contained-in adder-1)             (contained-in adder-1)
   (direction in)                     (direction in)
   (conn-from)                        (conn-from)
)                                  )

(defschema adder-1-port-Q
   (instance-of output-port)
   (contained-in adder-1)
   (direction out)
   (conn-to)
)
```

Figure XLIII, example adder block structure

will build up the new alternative from the original. The designer can also instigate a new alternative from the user interface. The decision test rule is discussed in section 4.4.1.

With a number of separate designs, the user may wish to compared them. The user interface is used to select the particular constraints needed for comparison and a library of lookup tables for the conversions between symbolic and numeric values. At this stage the user may decide to input a new set of importance values or use an in house derived library. The comparison process can then be started. All required constraints that need to be found are derived using the built in heuristics. The constraint values are then compared with the requirements to give the set of results or utilities, and these are combined to form an overall utility using the importance values as weighting

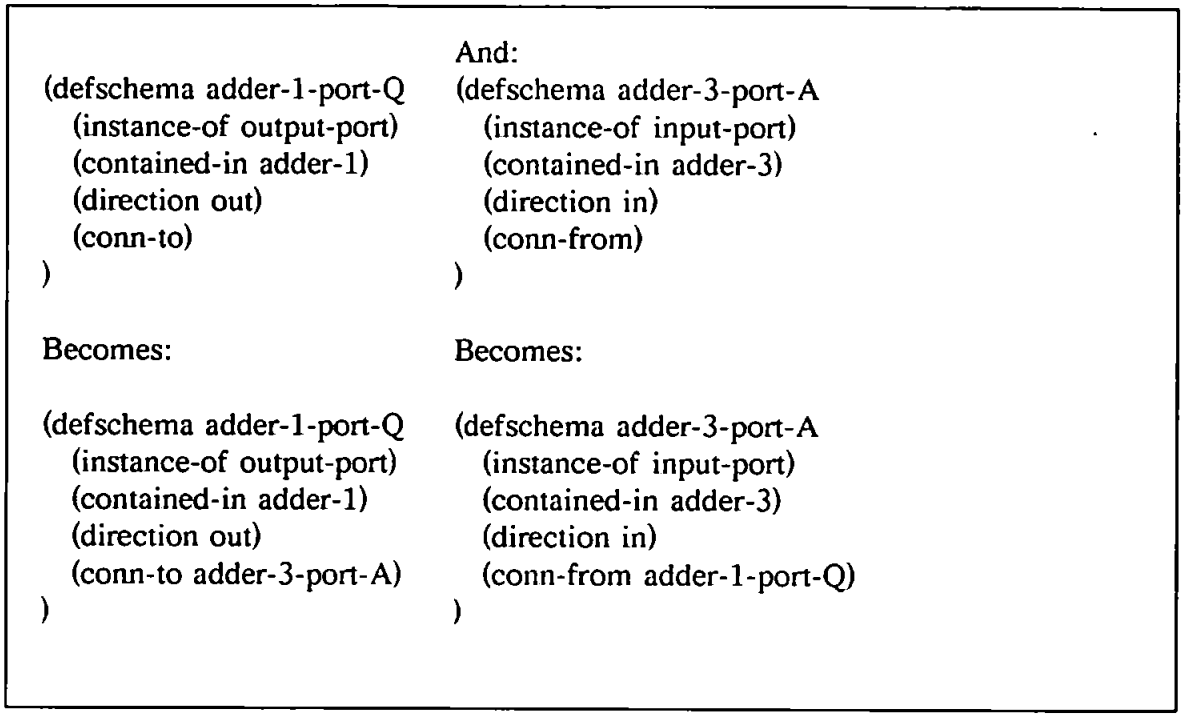

Figure XLIV, modification of conn-to and conn-to slots

factors. The final utilities for each alternative are displayed and the highest one highlighted by the user interface. This process is discussed in section 4.6.2.2

The constraint system can be used in a variety of ways. For example, a potential alternative design can be rapidly evaluated to test its suitability. Alternatively the criteria may be changed to determine the effects on the designs. The net result is that the tool facilitates easy application of 'what if' strategies.

4.9. Summary

The aim of this chapter has been to show how the relatively abstract representation of early electronic engineering design discussed in chapter three has been used to form a design tool called the Plymouth Engineer's Design Assistant (PEDA), targeted at some of the users cognitive needs. The description has been split into the same broad categories as chapter 3, and the key parts of the implementation are described in moderate detail. At the same time the implementation is compared with other approaches which have substantial implementational aspects. The emphasis has been on portraying the simplicity of the implementation, which is due in part to the simplicity of the abstract representation on which it is based, and a disregard for many implementation constraints. These include for example, the size and speed of the final

tool, which can only be justifiably done in a research context. Certain parts of the implementation have been discussed in more detail, for example the data flow simulator to show this simplicity, and also to show how a design and its behaviour take place at the same level, where the data physically moves around the design itself. This and other issues raised by the work will be discussed in the next chapter.

References for Chapter 4

Baker, K. D., Ball, L. J., Culverhouse, P. F., Dennis, I., Evans, J. st. B. T., Jagodzinski, A. P., Pearce, P. D., Scothern, D. G. C., and Venner, G. M., "A Psychologically Based Intelligent design Aid," in (to appear) Intelligent CAD Systems 3: Practical Experience and Evaluation, ed P. Veerkamp (Ed.), Berlin: Springer-Verlag.

Hutchings , E. L., Hollan, J. D. & Norman, D. A.,"Direct Manipulation Interfaces," Human Computer Interaction, Vol. 1, pp. 311-338, 1985

Katz, R. H., Anwarrudin, M., Chang, E., "A Version server for Computer-Aided Design Data," Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986.

Lathrop, R. H. and Kirk, R. S., "AN Extensible Object-Oriented Mixed-Mode Functional Simulation System," Proceedings of the 22nd ACM/IEEE Design Automation Conference, pp. 630-636, 1985.

Mostow, J., Barley, B. and Weinrich, T., "Automated reuse of Design Plans," Rutgers University, Department of Computer Science, AI/Design Project Working Paper No. 146, 1989.

Stefic, M., Bobrow, D. G., Bell, A., Brown, H. Conway, L., Tong, C., "The Partitioning of Concerns in Digital System Design," Xerox parc internal paper VLSI-81-3, Dec 1981.

Stefic, M, & Bobrow, D. G., "Object-Oriented Programming: Themes and Variations," The AI Magazine, 1985.

Veerkamp, P., Kwiers, R. P., and Hagen, Paul ten, "Design Process Represntation in IDDL," Preliminary Proceedings of the Third Eurographics Workshop on Intelligent CAD Systems, April 1989.

Winston, P. H., "Artificial Intelligence 2nd Ed," Addison-Wesley, 1984.

Chapter 5: Contributions, Final discussion and Further Work

5. Contributions, Final Discussion And Further Work

5.1. Overall Structure of This Chapter

The aim of this chapter is to bring together the statements of the preceding four chapters into a summary of the ideas and research embodied in the work to date. The chapter begins with a description of what contributions the research has achieved to date, both in terms of the idealised abstract representation of early electronic engineering design, and the partial realization of this representation in PEDA, a cooperative tool designed to aid the engineer in this area. This is then followed by a general discussion of some of the important issues concerned with electronic design representational that have been raised during the course the research and have been outlined in the main text. A discussion of the problems encountered during the research leads on to an outline of topics in which further work may be carried out. The chapter and dissertation is rounded off with a few concluding remarks.

5.2. Contributions

The work described within this thesis has been concerned primarily with two main points. The first deals with the abstract definition of a idealised representation for early electronic engineering design; The second is concerned with an implementation of key parts of that representation as the core of a cooperative electronic engineering design tool known as the Plymouth Engineer's Design Assistant (PEDA).

The representation differs significantly from many others in that its basis has been shifted away from an analysis of the end problem towards the cognitive needs of the designers themselves. To achieve this aim the requirements for the representation have based primarily on the results of psychological research in this field (Ball, 1990). This psychological emphasis has produced a representation which caters mainly for the generation and selection of design alternatives. It is based upon the merging of all the information regarding an early design into two loose hierarchies: The first deals with the decomposition of equation based block diagrams: And the second orders these diagrams into collections of alternative designs. All information held within the representation is conceptually viewed as constraining a design or set of designs and can

be used as criteria to choose between them. The implementation realizes the two hierarchy representation, addresses some important aspects of the constraint system and the navigation of the alternative hierarchy. Also included are a rule based, multilevel, data flow equation based simulator. The overall features of both the representation and to a lesser extent the implementation are itemized below:

1) A merged representation for block diagram and alternative design hierarchies (versions).

2) A System for the management of block diagrams and design alternatives.

3) A constraint comparison system for the analysis and selection of alternatives.

4) A block diagram based mathematical equation simulator

5) A system to extract and record decisions made during design.

6) A system to check for errors and inconsistencies made during design.

5.3. Final Discussion

This section attempts to cover some important issues or points that were raised during the research. Some of these issues have already been covered in the previous chapters, but are included here with the others not only for convenience, but to put them all in a common perspective.

5.3.1. The Overall Approach to Applying Knowledge Based Techniques to Design Tools

The use of AI or knowledge based techniques in design support tools still has much to offer the designer. Traditionally tools based upon these techniques have

supported the designer from two quite different perspectives (Smithers et al, 1989): The first creates involves creating an automatic design system, that replaces some aspect of the design activity using domain specific knowledge elicited from experts; The second adopts a cooperative approach and attempts to provide some form of intelligent design support system which will aid the designer. It can be seen that the approach used has many very important implications for tool designers and end users alike.

The first and more common approach has been and will continue to be for some time an area of extensive research, especially in the digital VLSI domain. However there are a number of problems to this strategy, the most apparent resulting from the increasing sophistication of design tools in this area. The need for powerful systems, has spurred the production of more capable and wider ranging tools which have slowly moved towards the early stages of design. As this occurs they tend to require increasingly greater amounts of embedded design knowledge to address all these areas, and it can be seen that a general purpose system based upon these principles, would need a vast amount of expert knowledge that would given present day technology, require a long time to elicit. This great repository of knowledge would also create additional problems, with regard to the verification and maintenance of the information stored within.

The second class of systems however offer an arguably superior approach in that they realise assistance by providing support for those areas in which the performance of engineers has traditionally been poor or tedious, and not providing assistance in those areas in which they are good at. This targeting of support will hopefully produce useful systems which are achievable using current technology, however it seems that the main problems are replaced with others which are associated with the means of deriving what types of support are required.

In areas where it has been shown that certain traditional knowledge elicitation techniques fail, other techniques are required (Evans, 1986). The psychological studies upon which the requirements for early electronic engineering design in this thesis are based are slow and laborious (Ball, 1990). As a result fast and accurate methods will be necessary if the aim of producing systems which meet the overall needs of designers is to be met.

5.3.2. The Target Domain of Knowledge Based Systems.

To find the areas where assistance should be provided in a design support system it appears that traditionally, introspective and intuitive methods have been used. This has produced a large number of contemporary systems which are primarily oriented at the design problem, in both classes of system described in the previous subsection. As a result systems have been produced to address designers' needs through satisfying the domain needs, with the main point being that if a system removes the designer from a particular mundane or repetitive aspect of design, then assistance is being afforded.

In reply to such techniques it would be reasonable to counter that in depth studies of the way in which designers design are urgently required to find out what are designers' needs after all. It does indeed look worrying that we are not sure that we are addressing the real needs of engineers or not, especially in the light that it is indeed very difficult and time consuming to elicit that knowledge (Evans, 1986 & Ball, 1990). This has been to some degree and approaches that look into the human aspects of design have been made. Unfortunately these have tended to tackle only the Human Computer Interaction (HCI) aspects of systems, and not the underlying design process model or representation. Suggestion have recently been made for truly cooperative systems which are domain independent (Smyth, 1988). Other studies (Ball, 1990 & Ullman et al, 1988) make similar recommendations, and add weight due to the techniques used to remove biases from introspective accounting that is often used. It is therefore reasonable to suggest that such techniques could be advantageously used to produce the requirements for truly cooperative systems.

5.3.3. The Complexity Inherent in Design Systems

An examination of the electronic design arena in both the research and commercial field will show that in general the tools are become more and more complex and that this trend shows no real signs of stopping. This is the normal result of tool designers adding more functionality to their products.

As systems' complexity increases they become harder to manage and maintain. This can be generally tackled in two ways: The first uses tools and methodologies to manage the complexity; The second attempts to reduce the complexity in the first place.

The psychological requirements for a system that addresses the early stages of electronic engineering design has been shown to produce a simple abstract representation and consequently relatively uncomplicated end system. This representation does however attempt to some important aspects of that activity, which are not covered by other tools, in a consistent, homogeneous and uncomplicated manner.

It can therefore be seen that there are real benefits to be won in overall system complexity and usefulness to the use of psychological methods in deriving the requirements for cooperative systems in the early stages of design at least.
However there may be real benefits to using similar techniques in other parts of the design activity.

5.3.4. The Target Level of Representations

Another point made apparent by the research and previously mentioned was concerned with the level at which a representation is targetted. This is a simple issue, but has important ramifications for all later aspects of design, in that if the representation is placed at too high a level, or described in too abstract terms, then it can become vague and then loose its usefulness, by allowing too much freedom in a subsequent implementation. Whereas, if it is pitched at too low a level, then it may lose clarity amidst the clutter of implementation issues.

It appears that many representations in electronic engineering design seem to be targetted at too low a level and as a result their description is full of implementation details, regarding the particular language details for example. If these details are removed, not only does the representation become simpler and clearer, but also independent of constraints such as language and related paradigm issues.

5.3.5. How Target Languages Shape Representations

Another interesting observation has been the apparent effect that the target language has upon the structure of many representations. In these it is often very difficult to separate the representation from the language and the representation is often described in language terms. For example an object oriented representation will mirror closely the object oriented features of the chosen implementation language, or a representation separating structure and behaviour will based on a language which promotes this. Whilst it may be argued that representations may be developed independent of the target language this appears not to be so in the electronic engineering domain, because the representations are not described in language independent terms.

In a similar vein, representations are greatly influenced by the paradigm preferences of the designers, thus instead of getting a representation which mirrors the internal aspects of the design activity we get for example: object oriented representations; rule based representations; blackboard representations; and database representations. To some degree, these trends cannot be avoided as languages are often used as conceptual aids, but the implementation should be derived from the abstract representation and not the other way around.

5.3.6. The Use of the Separation of Concerns in Representations

The subdivision of the representation of designs into a number of non interacting domains of description is a common theme in electronic engineering design and can be viewed as an extension of "Divide and Conquer" complexity reduction techniques used by engineers in many disciplines. However it appears that care should be shown in deciding how to partition the information, and in the early electronic engineering design activity it would be difficult, due to the amount of interaction between the commonly accepted domains, for example structure, behaviour and physical. As design tools improve the different types of information that are stored will increase and in return so will the potential number of domains. To help reduce the chaos that may result, further study will be needed to determine the areas and the degree of partitioning that best model the different parts of the design activity.

5.3.7. The Similarity Between the Representation of Software and Engineering Designs

An interesting observation found during the course of the work, was the apparent similarity between electronic engineering and software design representations especially at the functional level, for example logical block diagrams in commercial electronic design tools, and ACP diagrams in MASCOT. This statement of the obvious, does however have the implication that research in electronic engineering design may have much to gain from work in the software gain and vice versa. In addition it does indicate that it may be possible to produce tools that address both domains if the areas of commonality are found.

5.4. Problems Encountered

This section outlines some of areas in which problems were encountered during the research. Not all the problems are discussed here as a few are apparent from the previous section on issues.

5.4.1. The Ambiguity of Terminology (Design Process)

This problem arose out the need for communication between the members of the interdisciplinary group of researchers and their supervisors on the PEDA project. Two facts emerged from the resultant dialogue: The first was that phrases and terminology were not consistent between members of the different professions and as a result a great deal of time was spent in achieving a reasonably dialogue; The second problem arouse out of definitions in common use in the engineering design assistant arena. These were phrases like "Design Process" and "Constraint" which tended to have a number of definitions in the literature, and made the process of analysis quite difficult.

5.4.2. Limitations of Target Languages

This is a problem common to most attempts at implementation, in that there is a semantic gap between the language and the intended artifact, in this case the representation.

It is reasonable to state that this will always be a problem, but the combination of effective language and environment can reduce this greatly. Even so, with a language such as the chosen ART, there are a number of problems. There may be a paradigmatic gap, between the representation and the implementation, for instance the abstracted representation may be hierarchical and the language may use a blackboard. This can be solved with multi-paradigm languages such as ART, but the results can be clumsy and inelegant. The language may be low level, and so too much time is spent worrying about details. The features of the language or environment may be extensive but poorly integrated, for example in ART different object oriented mechanisms are used for both schemata and icons. Some features may be extremely powerful and others infuriatingly limited, as again was the case with ART's reasoning mechanisms and user interface respectively. The sophisticated environments generally have long learning curves, but once learnt are very productive tools. And finally the resultant systems tend to be resource hungry, and run slowly when compared to more conventional tools.

5.5. Further Work

It can be seen from the text of chapter 4 that the pilot PEDA system incorporates only part of the functionality required by the psychological requirements and abstract representation. There is therefore a great deal of scope for further work on the tool in many areas, not only on the implementation but on refinements to the abstract representation itself, areas such as classic version control and communication with other tools such as REDUCE or Macsymma. The following sections precis the individual areas targetted for further work.

5.5.1. Representation of Designs and Design Alternatives

The representation of designs as alternatives are one of the most completely implemented parts of the representation, though it must be said that the approach chosen is not particularly efficient from the point of view of the memory required by designs or the speed of creating or subsequent manipulation of design alternatives. These factors can be addressed in a number of ways, and are indeed a priority target for additional research due to the slowness of the particular implementation of the PEDA tool. However the advances made in these areas in LISP the underlying language for the type of tools used to create PEDA may make such effort wasted. In fact this was one reason why efficiency considerations were not deemed important in the production of PEDA in the first place.

The second area that can be examined is fundamental to the representation as it deals with the number of constraints used. At present the implementation deals with constraints in a totally free format. The psychological requirements suggest the types of information required as a minimum, and it has been the aim of the work to keep this type of information as small as possible. However in real systems the amount of information available can become quite large and so it is an important area for further work to determine what specific domain information (ie constraints) are the best to use, from the point of view of the constraint comparison system.

5.5.2. Management of Designs and Design Alternatives

The functionality within the PEDA implementation that deals with the management of designs and their alternatives is in fact quite minimal, providing the barest of necessary features. In a more comprehensive system this would need to be addressed, through an examination of the abundant literature in the area of software version management or involving studies (perhaps psychologically based) to gather a better understanding of the actual requirements in this particular area.

5.5.3. The Constraint Comparison System

Again, the constraint comparison system has only been sufficiently implemented to test out the ideas embodied in the abstract representation and would need to be extended greatly if it were to meet the aims of the psychological requirements fully. The number of constraint heuristics could be considerably extended, but the number and type of constraints need to be known as mentioned before. In addition the multi-attribute constraint comparison engine requires further investigation before completing.

5.5.4. The Simulator

The simulator like the representation of designs and alternatives is one of the most completed parts of the implementation, though again there are many aspects that warrant further review.

The simulator like many other parts of the implementation is quite inefficient in terms of execution speed and other normal constraints, and although this is adequate for very small test designs, it needs to be tackled if the PEDA tool is to be effectively used on larger work. Simulation is an area which has been extensively examined over the years and improvements to this system should be realizable within the context of its conception, although it is a comparatively high level simulator and much work has concentrated upon improving the speed of low level simulators.

It may also be advantageous to investigate interfacing a symbolic manipulation tool such as REDUCE or Macsymma to the simulator, greatly improving its capabilities, and at the same time providing facilities such as equation solving. However these tools tend to be large, offer many features that may not be required and tend to be closed systems providing limited access to a knowledge based system.

5.5.5. The Decision Point System

In a similar vein to the previous topics a great deal of work can be conducted on the decision point system. This could utilize considerable weight of literature available on decision analysis and studies intended to elicit generalized and specific heuristics on detecting and extracting the important design decisions and the reasons

behind them. These ideas could be joined into a framework that then provided a semi-automated means of annotating designs with the additional information required to tell the designs why a particular design decision was made.

5.5.6. The Detection and Correction of Errors

This is an area that has not been covered at all in the PEDA implementation, and as such there are plenty of opportunities for further work here. There are many possible avenues to explore, concerned with the detection of mathematical constraint violations, and the adoption of REDUCE or Macsymma as a base symbolic manipulation tool together with heuristics to detect other constraint violations, in a similar way to truth based systems, may be a reasonable way forward.
Again the problems of inaccessibility may force other schemes to be investigated.

5.5.7. Implementations in Other Languages or Environments

One of the fundamental aspects of the abstract representation is that it is implementation independent and therefore attempts to avoid specific details about languages or particular representational schemes. These issues are dealt with by a particular implementation. The aim here has been to simplify the abstract representation by removing these considerations, and make the abstract representation as generic as possible. The specific implementations can therefore take advantage of the features of a particular language or environment, or improvements in algorithms and heuristics, whilst still addressing the needs of the abstract representation. A possible avenue for further work is produce implementations in different environments or languages, taking into account the knowledge gained from other approaches, but without having to resort to a particular language paradigm, which may occur if the representation was derived from that language.

5.6. Concluding Remarks

The overall aim of this thesis has been to produce a knowledge based design support tool for the early stages of electronic engineering design. This has been done by first presenting a new idealised abstract representation for those stages whose purpose is to direct the production of cooperative design tools to the areas of that activity that have been shown in the literature to be important. By being based upon this psychological input the representation is targetted at those needs specifically, instead of through side effect and as a result contains far less redundant functionality in this area than many other design systems targetted at later stages of design activity.

In addition it is shown that by addressing the cognitive needs of designers, any explicit or implicit knowledge held by the representation and targetted at those needs will be useful in wider range of problems than systems targetted at a particular end problem, because the cognitive problems experienced by the designer are common across many problems.

References for Chapter 5

Ball, L., "Cognitive Processes in Engineering Design," PhD Thesis, Department of Psychology, Polytechnic South West, Devon, UK, 1990.

Smithers, T., Conkie, A., Doheny, J., Logan, B., and Millington, K., "Design as Intelligent Behaviour: An AI in Design Research Programme," Fourth International Conference on Applications of Artificial Intelligence in Engineering, July 1989.

Evans, J. St. B. T., "Knowledge Elicitation in the Training and Assessment of High Level Cognitive Skills," Report Prepared for the Army Personnel Research Establishment, 1986.

Smyth, M., "Articulating the Designer's Mental Codes," LUTCHI Research Centre internal paper (draft) ref: HCC/L/24, 11th May 1988.

Ullman, D. G., Dietterich, T. G., and Stauffer, L. A., "A Model of the Mechanical design process Based on Empirical Data," AI EDAM, vol. 2, no. 1, pp. 33-52, 1988.

Appendix A

A. PEDA in Use

The following figures give a view of the PEDA tool in use. They show the user interface primarily developed by G.M. Venner, discussed briefly in the main text. The interface is basically a screen based driven drawing board, using a mouse for the direct manipulation of block diagrams. Figure XXVI shows the form of the interface, where the designer is using the tool to produce a trial FIR filter. The main work area is in the centre, where block diagrams can be built up from the palette of common blocks on the left. Operations are performed either using the fixed menu which indicates overall options available, Pop up menus or direct manipulation of the various visible blocks in the palette or main window. All menus are context based, indicating the current options available for any selected object. In this case a Pop up menu is visible and one of the multiplier blocks is about to be moved.

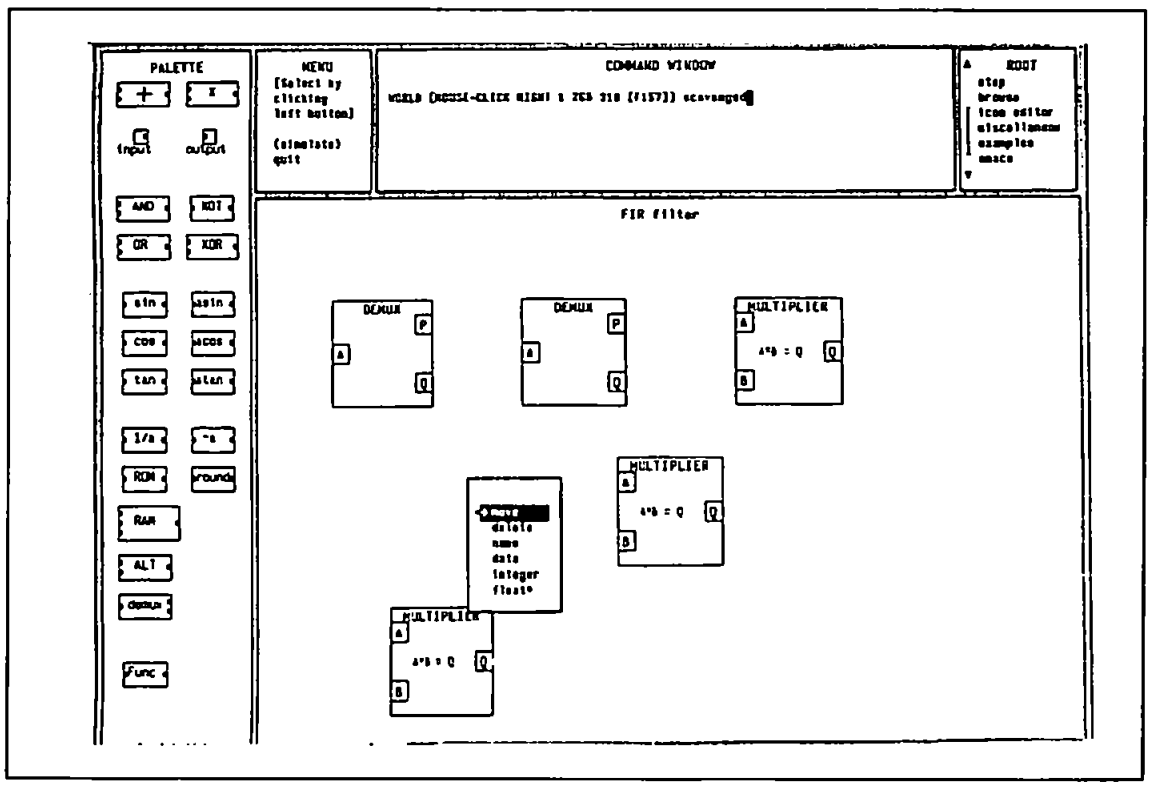

Figure XXVI, block diagram construction.

The second figure shows the completed FIR example. It has a main input, an output and three coefficient inputs C1, C2 & C3. In addition there are three multipliers and two adders. The Demux blocks perform no other function than converting one input into two outputs. Connectivity is indicated by lines between the ports on the respective blocks.

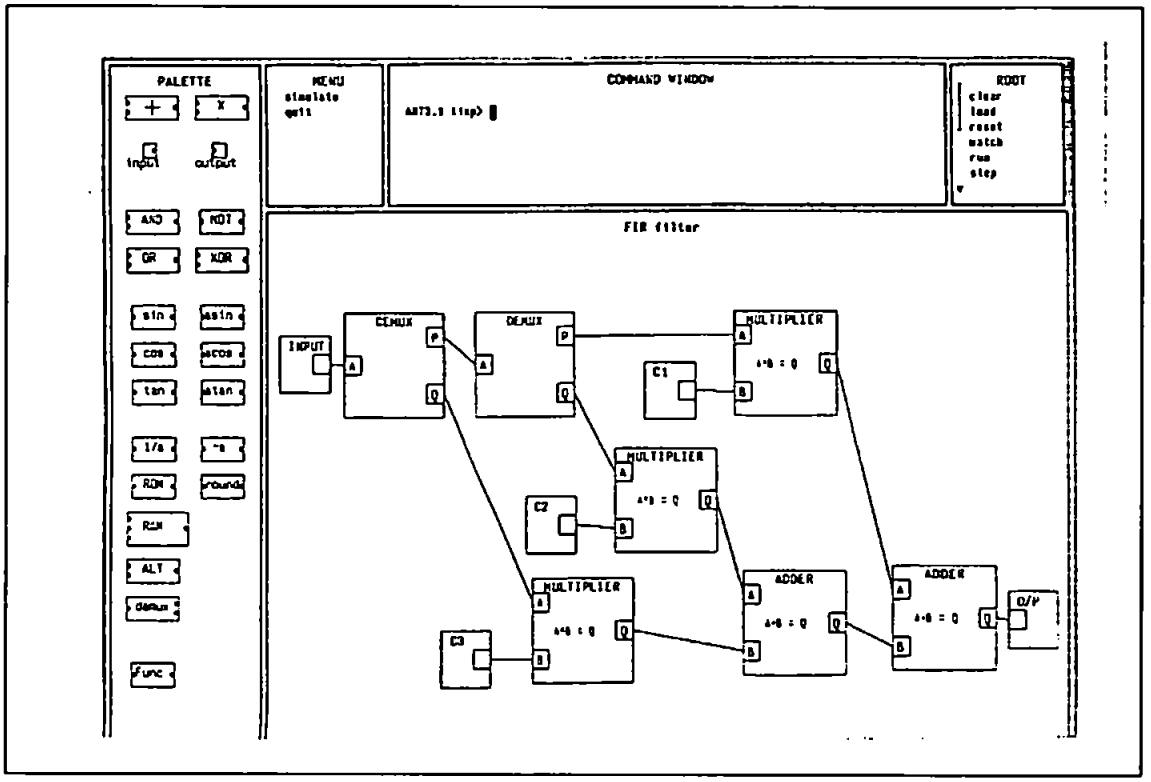

Figure XXVII, completed block

The Results of a simulation run are indicated in Figure XXVIII. Ports can be probed, to examine the data passing through them. When this happens the port is darkened and a probe window appears. As simulation proceeds the data through each probed port can be seen in the corresponding probe window. The interesting point to be noted is that changes can be done on the fly, with results being produced incrementally. Finally it should be noted that this is indeed an incorrect design, and will produce the wrong results. For correct operation a delay is required between each demux and multiplier block.

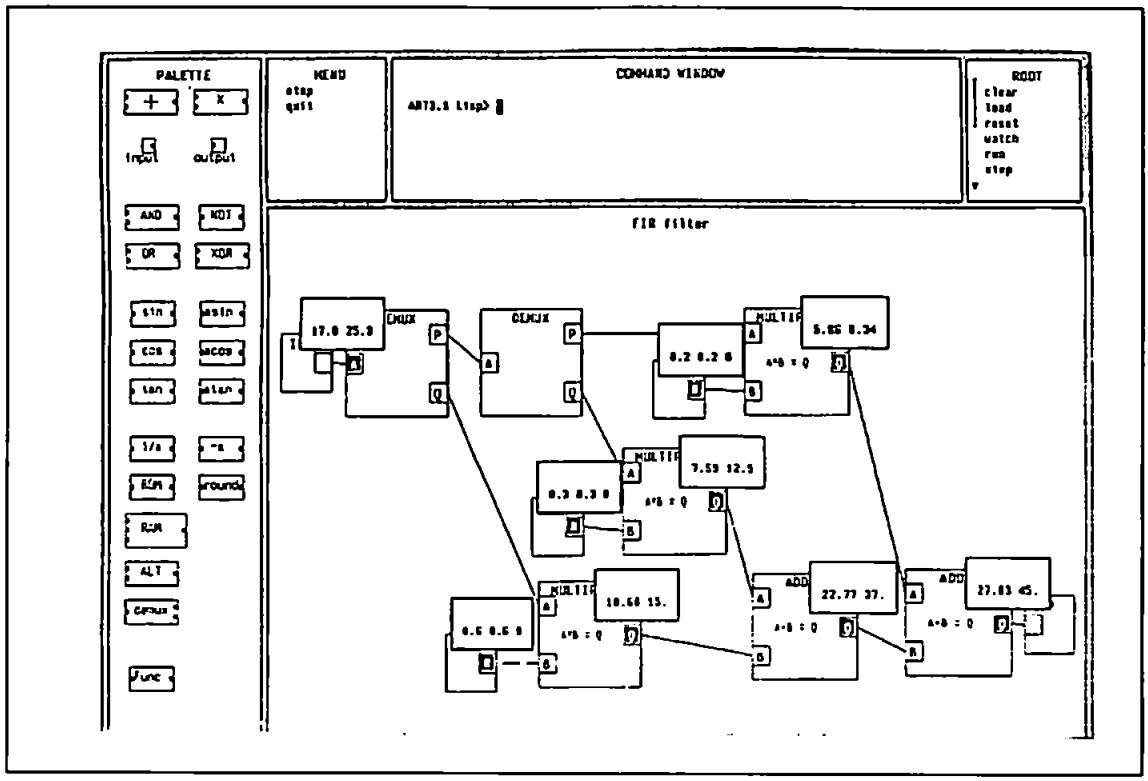

Figure XXVIII, block diagram simulation.

Appendix B:

B. The ART Expert System Development Tool

This appendix outlines and describes the capabilities of the Automated Reasoning Tool (ART), a large and complex hybrid expert system development tool, which was used extensively to investigate and develop the PEDA environment discussed in chapter 4. Although important in the implementation of the PEDA tool, it was not thought that an explanation of ART was appropriate to the main text. However for completeness and an overview of the associated terminology it is included here in the appendices.

Introduction

ART is one of group of large, hybrid general purpose expert system development tools, which combine a number of paradigmatic approaches, to facilitate the production of knowledge based systems. The ART system documentation lists its primary features as:

1) A language for knowledge representation and programming.
2) An Inference Engine.
3) A complete programming environment.

The ART Language

The ART language can be subdivided into a number of areas. These are:

1) Facts
2) Schemata
3) Rules
4) Viewpoints

Facts

A fact is a means of representing knowledge or information in ART. Each fact is a separate item of information and is stored in an area known as the Fact database. All facts are unique and are numbered. To the programmer Facts appear in two forms 1): as text separated by spaces and wrapped in round brackets, for example:

```
(friends Hillary John Lee)
```

or 2) within square brackets and preceded by a unique fact number, for example:

```
f-1211[friends Hillary John Lee]
```

The first form is used within the ART programming langauge, which is an extension to COMMON LISP, and has a similar syntax, the second is the printed form of facts residing in the fact database. Facts are relatively free format, variable length entities, similar in conception to LISP lists.

Schemata

ART Schemata are used to organize knowledge about items which are related to one another. To the programmer schemata consist of a number of 'slots' which contain 'values'. A example schema is shown below.

```
(defschema bug-eyed-alien                ; Name
   (has-legs yes)                        ; Attribute slot
   (has-suckers no)                      ; Attribute slot
   (number-of legs 2)                    ; Attribute slot
   (type alien)                          ; Relation slot
)
```

The schema has an overall name, and several named slots. Each named slot can contain attributes or the name of another schema. In the latter case, this forms a relational link between the two schemata and allows the schema system in ART to automatically maintain the logical consistency of information represented as schemata. For example, if the relational link is an inheritance relation, then its presence in a schema allows the inheritable slots and their values, in the related schema, to be inherited, in the schema with the link. If the relation slot is then subsequently altered then the inherited slots will subsequently change to reflect this. For example:

1) An Empty schema:

```
(defschema alien-I-saw
                                    ; No slots yet
)
```

2) Report bug-eyed alien;

```
(defschema alien-I-saw
  (instance-of bug-eyed-alien)      ; inheriting relation slot
  (has-legs yes)                    ; Attributes slot inherited
  (has-suckers no)                          ; from bug-eyed alien.
  (number-of legs 2)                ; "                 "
  (type alien)                      ; Relation slot causing
)                                   ; inheritance.
```

3) Realize mistake:

```
(defschema alien-I-saw
        (instance-of false-sighting)   ; new inheriting relation slot
        (type none)                    ; this attribute now inherited
)
```

This can also be shown pictorially in Figure XXXIV.

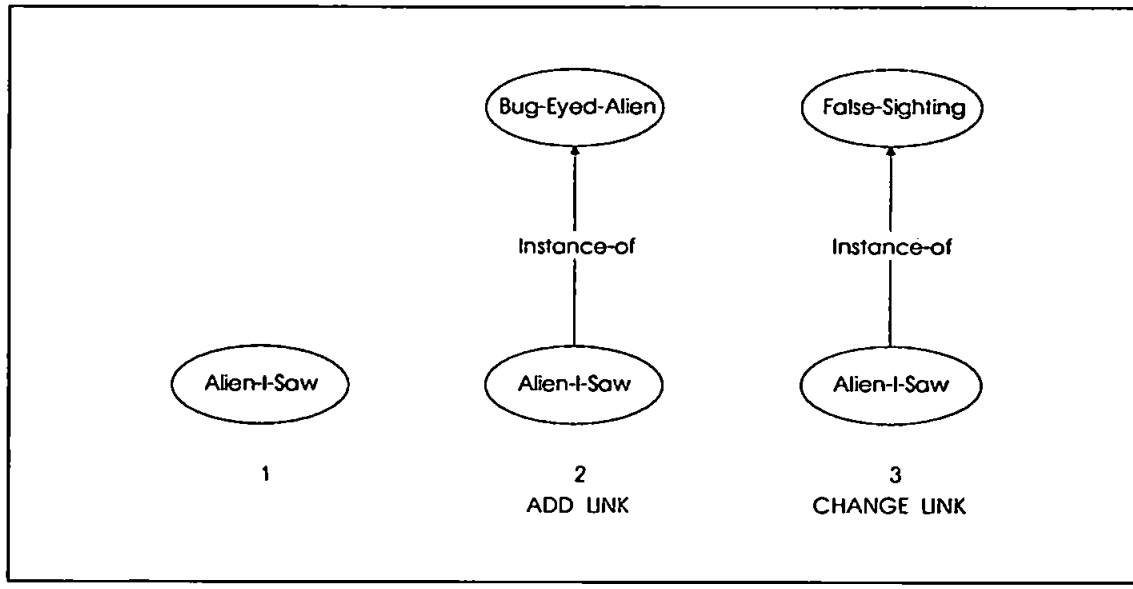

Figure XXXIV

The inheritance mechanism is very flexible and can be tailored extensively to requirements.

There is an intimate relationship between schemata and facts within ART. The information within a schema exists not only as a schema definition, but as a series of fact triplets of the form (<slot> <schema> <value>), for example:

The schema:

```
(defschema todays-catch
   (no-of-cod 10)
   (no-of-conger 4)
   (no-of-haddock 4)
)
```

exists also as the facts:

```
(no-of-cod todays-catch 10)
(no-of-conger todays-catch 4)
(no-of-haddock todays-catch 4)
```

When a schema is modified, there are corresponding changes to the fact database and vice versa. This is done to present a uniform interface to the rule pattern matching system, allowing rules to work with both conventional and schema derived facts.

Rules

ART rules are a means of defining the procedural knowledge that is available to an application. Two major types of rules are provided: 1) Forward chaining rules respond to facts by taking action; and 2) backward chaining rules respond to goals by trying to satisfy them.

In ART a rule has a left hand side (LHS) and a right hand side (RHS). The LHS contains a set of patterns which the rule tries to match to all the facts available in the fact database. The RHS contains procedural code which is executed if the match was successful. This is the same mechanism as condition-action pairs in forward chaining production systems. An example of a simple forward chaining rule in ART is as follows:

```
(defrule find-all-persons-with-blue-eyes
  (instance-of ?name person)              ; look for somebody
  (colour-of-eyes ?name blue)             ; this person has blue eyes
=>
  (printout t ?name " has blue eyes")     ; print result if true
)
```

This rule contains two patterns, the first looks for persons in the fact database, the second looks for eyes that are blue. The '?name' is a variable that has been used to link the patterns together. The rule can only fire if there are facts in existence in which there is a person, and that person has blue eyes.

Backward chaining is achieved through an extension to the forward system, with special goal fact patterns causing rules to produce facts which satisfy the patterns in a conventional forward chaining rules. An example of this is shown next.

```
(defrule calculate-volume-of-cylinder
  (goal (volume ?name ?volume))           ; volume required
  (instance-of ?name cylinder)            ; of a cylinder,
  (radius ?name ?radius)                  ; Both radius
  (length ?name ?length)                  ; and length specified.
=>
  (setf ?volume (* ?radius ?radius ?length 3.14159)); Now calculate volume
  (assert (volume ?name ?volume))         ; and Assert in database.
)
```

The goal pattern in this rule would be linked to a corresponding pattern in another forward chaining rule. If that pattern needs to be matched, then this goal directed rule will be activated.

Viewpoints

Viewpoints are a powerful means of segregating data into separate models of a situation that an application is considering. This is not done by creating multiple databases and copying data, as this would be very time consuming for large models, but through different viewpoints on one database. This is achieved through tag information on each fact, which lists which viewpoint it appears and the one it is removed from. This mechanism which extends to multiple dimensions, enables the production of rapid hypothetical reasoning systems, and is a key feature of ART.

Actions

Actions are an Object Oriented (OOP) addition to ART that permits the association of inheritable procedural code to schemata slots. This code can be invoked from procedural code, or automatically when schemata data is accessed, modified or removed. Multiple inheritance is also allowed.

Inference Engine

ART contains an inference engine that uses declarative and procedural knowledge captured in the ART language to derive conclusions about an application. Making use of known facts, it attempts to match patterns in rules and then apply their consequences, often generating more facts, in an interactive manner, until a set of goals are met. This operation takes place in cycles, which are divided into three steps. These are:

1) Match
2) Select
3) Fire.

Match

Facts are matched against patterns in the LHS of rules. Every time a rule's patterns are satisfied the rule is said to be 'activated', and its name is placed on the rule agenda together with the associated facts which satisfied the match. The rule agenda is a list of pending rules waiting to be executed.

Select

When all matching has occurred, one pending rule is selected from the agenda. This is either random, or based upon an importance value or 'salience' number, specified in the rule definition.

Fire

The selected rule is then executed or 'fired' which involves executing the procedural code in its RHS.

The above cycle then repeats, reforming the rule agenda each time until there are no new matches, or the inference engine is halted. This procedure makes ART's rule based operation effectively data driven, and if order is required it has to added externally. Also, as the rule agenda is reformed each cycle is created new each cycle, it is possible for some rules which were originally activated, to never fire, if a change to the fact database prevents the match for a particular pattern match. On the other hand it is quite possible for a rule to fire many times if there a many combinations of facts which match its patterns. This happens often, due to the flexibility of the pattern matching mechanism and the use of variables and wildcards in the patterns. Endless matching is prevented through a scheme called refractoriness in which a particular fact combination is only matched once to a particular rule pattern.

Programming Environment

ART includes an interactive development environment called the ART Studio, which includes the two editors vi and EMACS, and offers tracing and debugging aids to the developer. The Studio exists in two forms, for driving Window and Character based interfaces. The window based interface, provides extra facilities, for the production of graphical interfaces, using icons, graphical images, and mice based menu systems.

Appendix C

C. PEDA Core Implementation Program Code

```
;;; -*- Mode:ART; Package:art-user; Base:10. -*-
;;; Basic PEDA Schema Definitions: Blocks.
;;;
;;; Links to iconic representation data removed

;;; Switch for joint PEDA or just representation

(deffacts version
  (dinos version)
; (joint-version)
)

;;; Basic PEDA schema slot definitions

(defschema our-instance-of
  (instance-of inh-relation)
  (inverse our-has-instances)
  (slot-what share-value)
  (new-relations
    (instance-of (?domain)(?range))
  )
)

(defschema copy-of
  (instance-of relation)
  (new-relations (is-a (?domain)(?range)))
)

;; inheritance of block structure handled by copy-of methods

(defschema contains
  (instance-of relation)
  (inverse contained-in)
)

(defschema conn-to      ;; builds up conn-to links in a copy
  (instance-of relation)
  (inverse conn-from)
)

(defschema function-conn-in
  (instance-of slot)
  (slot-what nothing)
  (slot-how-many multiple-values)
)

(defschema function-conn-out
  (instance-of slot)
  (slot-what nothing)
  (slot-how-many multiple-values)
)

(defschema function
  (instance-of slot)
  (slot-what share-value)
)
```

```
(defschema stream
  (instance-of slot)
  (slot-what nothing) ;;don't inherit
)

(defschema functions
  (instance-of relation)
)

(defschema has-fired
  (functions)
)

(defschema has-not-fired
  (functions)
)

(defschema function-fired
  (instance-of relation)
  (slot-what share-value)
  (slot-how-many single-value)
  (new-relations
    (functions (?range)(?domain)))
)

(defschema template
  (instance-of relation)
  (new-relations (is-a (?domain)(?range)))
)


(defschema template
  (instance-of inh-relation)
)

;;; Simulator data type specs

(defschema data-type
  (instance-of relation)
  (slot-how-many single-value)
  (slot-what share-value)
  (size)
  (printing-representation (princ))
)

(defschema number
  (is-a data-type)
)

(defschema rational
  (is-a number)
)

(defschema ratio
  (is-a rational)
)
```

```
(defschema integer
  (is-a rational)
)

(defschema fixnum
  (is-a integer)
  (size 32)
)

(defschema bignum
  (is-a integer)
  (size not-known)
)

(defschema float
  (is-a number)
)

(defschema short-float
  (is-a float)
  (size (23 8))
)

(defschema single-float
  (is-a float)
  (size not-known)
)

(defschema double-float
  (is-a float)
  (size not-known)
)

(defschema long-float
  (is-a float)
  (size not-known)
)

(defschema complex
  (is-a number)
)

;;; Start Defining Blocks Now

(defschema world          ;; Where block diagrams live
  (instance-of instantiated-window-icon)
  (window world)
  (display-parameters)
  (menu world)
  (no-of-options 5)
)

(defschema base-world
  (is-a world)
)
```

```
;;; Central Definition of block

(defschema  block
  (function-conn-in)
  (function-conn-out)
  (function)
  (copy-of)
  (template)
  (result-data-type fixnum)
  (function-firings 0)
  (function-fired has-not-fired)
  (result-data-type fixnum);;; default data type
    ; + iconic representation data
)


(defschema alt-block ;;; blocks that do not require all
  (is-a block)       ;;;inputs
)

(defschema all-block ;;; blocks that do  require all inputs
  (is-a block)
)

(defschema port
  (is-a block)
  (conn-to)
  (conn-from)
  (packet-link)
)

(defschema input-port
  (is-a port)
  (direction in)
; + gill's iconic representation data

)

(defschema output-port
  (is-a port)
  (direction out)
; + gill's iconic representation data
)

(defschema file
  (is-a block)
  (direction)
  (stream)
  (data-type)
  (data-count 1)
)

(defschema memory
  (is-a functional-block)
  (size)
  (data)
  (function ("function performed by rule "))
)
```

```
(defschema ROM
  (is-a memory)
)

(defschema RAM
  (is-a memory)
)

(defschema round
  (is-a all-block)
  (bits)
  (function ("function performed by action"))
  (result-data-type)
)

(defschema packet
  (node free-packets)
  (data-type null)
  (overflow null)
  (data null)
  (has-instances)
)

(defschema data-file
  (data-type integer)
  (data)
)

(defschema free-packets ;;; the place where all packets
  (is-a port)           ;;;eventually go
  (packet-link ())
)

(defschema simulator-defaults
  (mode construction)
  (last-mode construction)
  (no-of-words-input-at-a-time 1) ;;;from files.
  (no-of-functions-at-a-time 1)   ;;;max five evals per rule
)                                 ;;fire

(defglobal ?*function-eval* = 0)
```

```
;;;This is a library of common functions.
;;; We have:
; adder
; multiplier
; alt
; function
; recip
; And
; Or
; Xor
; not
; invert
; sin
; cos
; tan
; asin
; acos
; atan
; Reduce
; demux
; Round
; input-file
; output-file


;;; Templates have been used to reduce repetitive effort
;;; Added test constraints for a low power moderate speed
;;; library.
;;; Iconic (display) information has been removed.

(defschema adder
  (is-a all-block)
  (template 2mux1)
  (chip-area medium)
  (power low)
  (speed 100)
  (design-time v-low)
  (function (if (Nan-check A B)
                (set-Nan 'Q)
                (setq Q (+ A B))
            )
    )
)

(defschema multiplier
  (is-a all-block)
  (chip-area medium)
  (power medium)
  (speed 200)
  (design-time v-low)
  (template 2mux1)
  (function (if (Nan-check A B)
                (set-Nan 'Q)
                (setq Q (* A B))
            )
  )
)
```

```
(defschema Bool-and
  (is-a all-block)
  (chip-area low)
  (power low)
  (speed 50)
  (design-time v-low)
  (template 2mux1)
  (function (if (Nan-check A B)
              (set-Nan 'Q)
              (setq Q (and A B))
            )
  )
)

(defschema alt
  (template 2mux1)
  (is-a alt-block)
  (function (if (equalp A 'NAN) (setq Q B)(setq Q A)))
)

(defschema Bool-Or
  (is-a all-block)
  (chip-area low)
  (power low)
  (speed 50)
  (design-time v-low)
  (template 2mux1)
  (function (if (Nan-check A B)
              (set-Nan 'Q)
              (setq Q (ior A B))
            )
  )
)

(defschema Bool-Xor
  (is-a all-block)
  (chip-area low)
    (power low)
    (speed 90)
    (design-time v-low)
    (template 2mux1)
    (function (if (Nan-check A B)
                (set-Nan 'Q)
                (setq Q (xor A B))
              )
  )
)

(defschema demux
  (is-a all-block)
  (template 1mux2)
  (function (if (Nan-check A)
              (set-Nan 'P 'Q)
              (setq P A Q A)
            )
  )
)
```

```
(defschema recip
    (is-a all-block)
    (chip-area high)
    (power medium)
    (speed 1000)
    (design-time v-low)
    (template 1mux1)
    (function
    (cond
      ((Nan-check A)(set-Nan 'Q))
      ((equalp A 0)
       (progn (printout t t "Error: trying to divide a -
              number by 0 is a bad move")
              (break)))
      (T (setq Q (/ 1 A)))
    )
  )
)

(defschema Bool-not
  (is-a all-block)
  (chip-area v-low)
  (power v-low)
  (speed 10)
  (design-time v-low)
  (template 1mux1)
  (function (if (Nan-check A)
              (set-Nan 'Q)
              (setq Q (not A))
          )
  )
)

(defschema invert
  (is-a all-block)
  (chip-area medium)
  (power low)
  (speed 150)
  (design-time v-low)
  (template 1mux1)
  (function (if (Nan-check A)
          (set-Nan 'Q)
              (setq Q (- A))
          )
    )
)
```

```
(defschema sin
  (is-a all-block)
    (chip-area high)
    (power medium)
    (speed 2000)
    (design-time v-low)
    (template 1mux1)
    (function (if (Nan-check A)
                  (set-Nan 'Q)
                  (setq Q (sin A))
              )
    )
)

(defschema cos
  (is-a all-block)
  (chip-area high)
  (power medium)
  (speed 2000)
    (design-time v-low)
    (template 1mux1)
    (function (if (Nan-check A)
                  (set-Nan 'Q)
                  (setq Q (cos A))
              )
    )
)

(defschema tan
  (is-a all-block)
  (chip-area high)
  (power medium)
  (speed 2000)
  (design-time v-low)
  (template 1mux1)
  (function (if (Nan-check A)
                (set-Nan 'Q)
                (setq Q (tan A))
            )
    )
)

(defschema asin
  (is-a all-block)
    (template 1mux1)
    (chip-area high)
    (power medium)
    (speed 2000)
    (design-time v-low)
    (function (if (Nan-check A)
                  (set-Nan 'Q)
                  (setq Q (asin A))
              )
    )
)
```

```
(defschema acos
    (is-a all-block)
    (template 1mux1)
    (chip-area high)
    (power medium)
    (speed 2000)
    (design-time v-low)
    (function (if (Nan-check A)
              (set-Nan 'Q)
              (setq Q (acos A))
          )
    )
)

(defschema atan
  (is-a all-block)
    (template 1mux1)
    (chip-area high)
    (power medium)
    (speed 2000)
    (design-time v-low)
    (function (if (Nan-check A)
              (set-Nan 'Q)
              (setq Q (atan A))
          )
    )
)

(defschema Reduce
 ; not implemented
)

(defschema input-file
  (is-a file)
  (template 0mux1)
  (direction in)
)

(defschema output-file
  (is-a file)
  (template 1mux0)
  (direction out)
)

(defschema round1
  (is-a round)
  (template 1mux1)
    (chip-area medium)
    (power low)
    (speed 200)
    (design-time v-low)
)
```

```
;;; Now template definitions

(DEFSCHEMA 2MUX1
  (is-a block)
  (CONTAINS 2mux1-port-A 2mux1-port-b 2mux1-port-q)
  (function-conn-in (function A in 2mux1-port-A))
  (function-conn-in (function B in 2mux1-port-B))
  (function-conn-out (function Q to 2mux1-port-Q))
)

(DEFSCHEMA 2mux1-port-a
  (is-a input-port)
)

(DEFSCHEMA 2mux1-port-b
  (is-a input-port)
)

(DEFSCHEMA 2mux1-port-q
  (is-a output-port)
)

(DEFSCHEMA 1MUX1
  (IS-A BLOCK)
  (CONTAINS 1mux1-port-A 1mux1-port-q)
  (function-conn-in  (function A in 1mux1-port-A))
  (function-conn-out (function Q to 1mux1-port-Q))
)

(DEFSCHEMA 1mux1-port-a
  (is-a input-port)
)

(DEFSCHEMA 1mux1-port-q
  (is-a output-port)
)

(DEFSCHEMA 1MUX2
  (IS-A BLOCK)
  (CONTAINS 1mux2-port-A 1mux2-port-q 1mux2-port-r)
  (function-conn-in  (function A in 1mux2-port-A))
  (function-conn-in  (function Q to 1mux2-port-Q))
  (function-conn-out (function R to 2mux1-port-R))
)

(DEFSCHEMA 1mux2-port-a
  (is-a input-port)
)

(DEFSCHEMA 1mux2-port-q
  (is-a output-port)
)

(DEFSCHEMA 1mux2-port-r
  (is-a output-port)
)
```

```
(DEFSCHEMA 1MUX0
  (IS-A BLOCK)
  (CONTAINS 1mux0-port-portA)
  (function-conn-in  (function A in 1mux0-port-A))
)

(DEFSCHEMA 1mux0-port-a
  (is-a input-port)
)

(DEFSCHEMA 0MUX1
  (IS-A BLOCK)
  (CONTAINS 0mux1-port-q)
  (function-conn-out (function Q to 0mux1-port-Q))
)

(DEFSCHEMA 0mux1-port-q
  (is-a output-port)
)
```

```
;General source-block template copying rules

(defglobal  ?*template-salience*
            = (- *maximum-salience* 100))

;;; Most of template copying handled by normal inheritance
;;; mechanisms, but
;;; some slots are special:

(defrule template-slot-contains
  (declare (salience (- ?*template-salience* 20)))
  (template ?object ?original-object)
  (contains ?original-object ?original-contents)
  (not (contains ?object
         =(our-get-icon-name ?original-object
           ?original-contents ?object)))
=>
  (bind ?object-contents
    (our-get-icon-name ?original-object
      ?original-contents ?object))
  (assert
   (contains ?object ?object-contents)
   (template ?object-contents ?original-contents)
  )
)

(defrule template-slot-input
  (declare (salience (- ?*template-salience* 20)))
  (template ?object ?original-object)
  (input ?original-object ?)
  (not (input ?object (?object)))
=>
  (assert (input ?object (?object)))
)
```

```
;;; The next rule bulids up the correct function-conn slots
;;; in each new block. This process is a lot easier
;;; to do procedurally. Most of the hassle is in
;;; correct pattern matching

(defrule template-slot-function-conn
  (declare (salience (- ?*template-salience* 20)))
  (template ?new-port ?original-port)
  (contains ?original-block ?original-port)
  (template ?new-block ?original-block)
  (contains ?new-block ?new-port)
  (split
    ((function-conn-in ?original-block
        (function ?var in ?original-port))
    (not (function-conn-in ?new-block
        (function ?var in ?new-port)))
    =>
      (assert
        (function-conn-in ?new-block
          (function ?var in ?new-port)))
      )
    ((function-conn-out ?original-block
          (function ?var to ?original-port))
     (not (function-conn-out ?new-block
          (function ?var to ?new-port)))
      =>
      (assert (function-conn-out ?new-block
              (function ?var to ?new-port)))
    )
   )
=>
)
```

```
;;; General source-block copying rules
;;; Very similar to template fill rules
;;; But uses copy of relation to signify that a copy should
;;; be made.
;;; Only the slots which don't inherit normally are given
;;; special treatment.
;;; The rest take pot luck with the is-a inheritance
;;; mechanism.

;(in-package 'au)
defglobal ?*copy-salience* = (- *maximum-salience* 100))

;;; Copy-of-fill not used as copy-of relation asserts is-a

;;; relation automatically

#|
(defrule copy-of-fill
  (declare (salience ?*copy-salience*))
  (copy-of ?object ?original-object)
  (instance-of ?original-object ?parent)
  (not (instance-of ?object ?parent)) ;; not asserted
=>
  (assert (instance-of ?object ?parent))
)
|#

(defrule copy-slot-contains
  (declare (salience (- ?*copy-salience* 20)))
  (copy-of ?object ?original-object)
  (contains ?original-object ?original-contents)
  (not (contains ?object =(our-get-icon-name
       ?original-object ?original-contents ?object)))
=>
  (bind ?object-contents
    (our-get-icon-name ?original-object
      ?original-contents ?object))
  (assert
    (contains ?object ?object-contents)
    (copy-of ?object-contents ?original-contents)
  )
)

(defrule copy-slot-input
  (declare (salience (- ?*copy-salience* 20)))
  (copy-of ?object ?original-object)
  (input ?original-object ?)
  (not (input ?object (?object)))
=>
  (assert (input ?object (?object)))
)
```

```
(defrule copy-slot-function-conn
  (declare (salience (- ?*copy-salience* 20)))
  (copy-of ?new-port ?original-port)
  (contains ?original-block ?original-port)
  (copy-of ?new-block ?original-block)
  (contains ?new-block ?new-port)
  (split
    ((function-conn-in ?original-block
      (function ?var in ?original-port))
     (not (function-conn-in ?new-block
          (function ?var in ?new-port)))
     =>
     (assert (function-conn-in ?new-block
             (function ?var in ?new-port)))
    )
    ((function-conn-out ?original-block
      (function ?var to ?original-port))
     (not (function-conn-out ?new-block
          (function ?var to ?new-port)))
     =>
       (assert (function-conn-out ?new-block
               (function ?var to ?new-port)))
    )
  )
=>
)
```

```
(defrule copy-slot-conn-to
  (declare (salience (- ?*copy-salience* 30)))
  (schema ?original-source-port
    (conn-to ?original-dest-port)
  )
  (schema ?copy-source-port
    (not (conn-from ?)) ;; make sure not a dest port
    (not (conn-to ?))   ;; has no conn-to value
    (copy-of ?original-source-port)
  )
  (schema ?copy-dest-port
    (copy-of ?original-dest-port)
  )
  (schema ?original-source-block
    (contains ?original-source-port)
  )
  (schema ?original-dest-block
    (contains ?original-dest-port)
  )
  (schema ?copy-source-block
    (contains ?copy-source-port)
    (copy-of ?original-source-block)
  )
  (schema ?copy-dest-block
    (contains ?copy-dest-port)
    (copy-of ?original-dest-block)
  )
=>
  (assert (schema ?copy-source-port
            (conn-to ?copy-dest-port)
          )
  )
)
```

```
;;; -*- Mode:ART; Package:art-user; Base:10. -*-

;;; Rules For The Behaviour of different types of block

(defrule function-all-block
  (declare (salience (+ ?*function-eval* 2)))
  (schema simulator-defaults
      (mode simulating)
  )
  (schema ?fb
      (instance-of all-block)
;     (active yes) ;;;for multi mode
      (function-fired has-not-fired)  ;;; only do once,
      (function  ?)       ;;; others a chance this cycle
      (function-conn-in (function ? in ?))
  )
  (forall (function-conn-in ?fb
          (function ? in ?port)) ;;; each and every input
      (instance-of ?port port)  ;;; must be ready
      (packet-link ?port ?)     ;;; multiple firing
  )
=>
  (invoke 'do-function-eval ?fb)
  (modify (schema ?fb (function-fired has-fired)))
)

(defrule function-ALT-block
  (declare (salience (+ ?*function-eval* 2)))
  (schema simulator-defaults
      (mode simulating)
  )
  (schema ?fb
      (instance-of alt-block)
;     (active yes) ;;;for multi mode
      (function-fired has-not-fired) ;;;only do once, give
      (function-conn-in (function ? in ?)) ;;; others a
      (function  ?)                ;;; chance this cycle
  )
  (exists
    (function-conn-in ?fb
      (function ? in ?port))       ;;; any input can be ready
    (instance-of ?port port)
    (packet-link ?port ?)          ;;; multiple firing
  )
=>
    (invoke 'do-function-eval ?fb)
)
```

```
(defrule function-ROM
  (declare (salience (+ ?*function-eval* 2)))
  (schema simulator-defaults
      (mode simulating)
  )
  (schema ?fb
    (instance-of ROM)
;     (active yes) ;;;for multi mode
      (function-fired has-not-fired) ;;; only do once, give
      (function  ?)        ;;; others a chance this cycle
      (function-conn-in (function address in ?port-address))
      (function-conn-out
        (function data-out to ?port-data-out))
      (data ?data) ;;; data of form (1 2 3 4 5 6 7....) etc.
  )
  (schema ?port-address
      (instance-of port)
      (packet-link (?packet $?))
  )
  (schema ?port-data-out
    (instance-of port)
  )
=>
  (bind ?address (get-data ?packet))
  (destroy-packet ?packet)
  (if ((length$ ?data) <= ?address)
    then (printout t t "error address " ?address
                  " exceeds size of ROM: " ?fb)
    else (create-packet-with-data 'packet
      ?port-data-out (nth ?address (list$ ?data)))
  )
  (modify (schema ?fb (function-fired has-fired)))
)

(defrule function-RAM-read
  (declare (salience (+ ?*function-eval* 2)))
  (schema simulator-defaults
      (mode simulating)
  )
  (schema ?fb
      (instance-of RAM)
;     (active yes) ;;;for multi mode
      (function-fired has-not-fired) ;;;only do once, give
      ;;; others a chance this cycle
      (function-conn-in (function address in ?port-address))
      (function-conn-in (function R-W in ?port-R-W))
      (function-conn-out
        (function data-out to ?port-data-out))
      (data ?data);; data of form (1 2 3 4 5 6 7....) etc.
  )
  (schema ?port-address
      (instance-of port)
      (packet-link  (?address-packet $?))
  )
  (schema ?port-R-W
      (instance-of port) ;;; always an address
      (packet-link  (?R-W-packet $?))
  )
```

```
   (schema ?port-data-out
       (instance-of port)
   )
   (schema ?R-W-packet  ;;; not using the get-data method
     (instance-of packet)
       (data R)
   )
=>
   (bind ?address (get-data ?address-packet))
   (destroy-packet ?address-packet)
   (destroy-packet ?R-W-packet)
   (if ((length$ ?data) <= ?address)
     then (printout t t "error address "
            ?address " exceeds size of RAM: " ?fb)
       else (create-packet-with-data 'packet
             ?port-data-out (nth ?address (list$ ?data)))
   )
   (modify (schema ?fb (function-fired has-fired)))
)

(defrule function-RAM-write
   (declare (salience (+ ?*function-eval* 2)))
   (schema simulator-defaults
       (mode simulating)
   )
   (schema ?fb
       (instance-of RAM)
;      (active yes) ;;;for multi mode
       (function-fired has-not-fired) ;;;only do once, give
       ;;; others a chance this cycle
       (function-conn-in (function address in ?port-address))
       (function-conn-in (function R-W in ?port-R-W))
       (function-conn-in (function data-in in ?port-data-in))
       (data ?data) ;; data of form (1 2 3 4 5 6 7....) etc.
   )
   (schema ?port-address
       (instance-of port)
       (packet-link  (?address-packet $?))
   )
   (schema ?port-R-W
       (instance-of port)
       (packet-link  (?R-W-packet $?))
   )
   (schema ?port-data-in
       (instance-of port)
       (packet-link (?data-packet $?))
   )
   (schema ?R-W-packet ;;; not using the get-data method
       (instance-of packet)
       (data W)
   )
=>
   (bind ?address (get-data ?address-packet))
   (bind ?input-data (get-data ?data-packet))
   (bind ?data-lst (list$ ?data))
   (if ((length$ ?data) <= ?address)
     then (printout t t "error address "
            ?address " exceeds size of RAM: " ?fb)
```

```
    else (progn
           (setf (nth ?address ?data-lst) ?input-data)
               (modify-schema-value ?fb 'data
                 (seq$ ?data-lst))
           )
  )
  (destroy-packet ?address-packet)
  (destroy-packet ?data-packet)
  (destroy-packet ?R-W-packet)
  (modify (schema ?fb (function-fired has-fired)))
)

(defrule file-input-block ;;; behaves  just like any other
  (declare (salience (+ ?*function-eval* 2)))  ;;; block
  (schema simulator-defaults
    (mode simulating)
  )
  (schema ?file-block
    (instance-of file)
    (function-fired  has-not-fired)
    (direction in)
    (stream ?file-name)
    (contains-ports ?node)
    (data-count ?data-count)
  )
  (schema ?file-name
;   (data-type ?data-type)
      (data ?data-sequence)
  )
  (test (>= (length$ ?data-sequence) ?data-count))
=>
  (printout t t " " ?data-count " " ?data-sequence)
  (bind ?new-data (nth$ ?data-sequence ?data-count))
  (bind ?packet (create-packet 'packet))
  (put-data-type ?packet (type-of ?new-data))
  (insert-data ?packet ?new-data)
  (place-packet ?packet ?node)
  (modify-schema-value ?file-block
    'data-count (+ ?data-count 1))
  (modify-schema-value ?file-block
    'function-fired 'has-fired)
)

(defrule file-output-block ;;; behaves just like any other
  (declare (salience (+ ?*function-eval* 2))) ;;; block
  (schema simulator-defaults
    (mode simulating))
  (schema ?file-block
    (instance-of file)
    (function-fired has-not-fired)
    (direction out)
      (stream ?file-name)
      (contains-ports ?node))
  (schema ?node
    (packet-link (?packet $?))
  )
=>
  (bind ?existing-data (get-schema-value ?file-name 'data))
```

```
  (bind ?new-data (get-data ?packet))
  (destroy-packet ?packet)
  (modify-schema-value ?file-name 'data
    (seq$ (nconc (list$ ?existing-data) (list ?new-data))))
  (modify-schema-value ?file-block
    'function-fired 'has-fired))

;;; Now the rule that moves the packets along connections
;;; between blocks

(defrule faster-packet-move
  (declare(salience (+ ?*function-eval* 2)))
  (schema simulator-defaults
    (mode simulating))
  (schema ?port
    (instance-of port)
    (conn-to ?dest)
    (packet-link ?source-list))
=>
  (for packet-name in$ ?source-list
    do
      (move-packet packet-name ?port ?dest)
  )
)

;;; This rule waits until all functions that can fire, have

(defrule last-function-done
  (declare (salience (- ?*function-eval* 10)))
  (schema simulator-defaults
      (mode simulating)
    )
    (schema has-fired
      (functions ?))
=>
  (bind ?x (get-schema-value 'has-fired 'functions))
  (for fb in ?x do
      (modify-schema-value fb 'function-fired
          'has-not-fired)
  )
)

;;; Initization rules:

(defrule simulator-initialise
  (schema simulator-defaults
      (mode construction)
  )
=>
  (init-packets)
  (clear-function-firings-slots)
  (resize-data-slot-in-memory-schema)
)
```

```
;;; changed to behave like blocks 15/5/89

(setq *data-file-dir* "~dino/ART/GROUP/")

;;;Once for all files

(defrule load-all-input-files
  (schema ?file-block
    (instance-of file)
    (direction in)
    (stream ?file-name)
  )
=>
  (let  ((the-file (merge-pathnames *data-file-dir*
                    (format nil "~a.art" ?file-name))))
    (if (probe-file the-file) then (art-load the-file))
  )
)

(defrule save-files
  (declare (salience (- ?*function-eval* 12)))
  (schema simulator-defaults
    (mode simulating))
  (schema ?file-name
    (instance-of data-file))
=>
  (let  ((the-file (merge-pathnames *data-file-dir*
                    (format nil "~a.art" ?file-name))))
    (with-open-file
      #L(output-stream the-file  :direction
        :output :if-exists :supersede)
      (printout output-stream (list-schema ?file-name))
    )
  )
)

;;;Each run through reset data count

(defrule initialise-file-blocks
  (declare (salience (+ ?*function-eval* 3)))
  (schema simulator-defaults
    (mode simulating)
  )
  (schema ?file-block
    (instance-of file)
    (direction in)
    (stream ?file-name)
    (contains-ports ?node)
  )
  (schema ?file-name
    (data ?data-sequence)
  )
=>
  (modify
    (schema ?file-block
      (data-count 1)
      (function-fired has-not-fired)
```

)
)
)

```
;;; -*- Mode:ART; Package:art-user; Base:10. -*-
;;;
;;; The action do-function-eval provides the means by which
;;; a block evaluates its data
;;;
;;; The action Map-vars-to-ports forms a list of equation
;;; variables and their corresponding packets, and calls
;;; do-eval.
;;;
;;; The purpose of do-eval varies between the types of
;;; block, but for all-blocks it:
;;;
;;; 1) binds the input variables to the input packets'
;;;  values.
;;; 2) evaluates the equation in the block's function slot.
;;; 3) rounds the results to the required value
;;; 4) creates output packets at the right ports, containing
;;; the results.
;;; 5) Destroys the input packets.

;; start with do-function eval


(defaction do-function-eval (block)()
  (map-vars-to-ports  block)
)

; Now debugging before method

(defaction (map-vars-to-ports before) (block)()
  (printout t t "map-vars-to-ports called on " block)
)
```

```
;; Now get a list of variables and their packets

(defaction map-vars-to-ports (block)()
  (let  ((f-c-i-list
            (get-schema-value block 'function-conn-in))
              ;;; list of sequences of slot function-conn-in
         (f-c-o-list
            (get-schema-value block 'function-conn-out))
              ;;; same for function-conn-out
         (input-vars nil)
         (output-vars nil)
         (packets nil)
        )
    (setq input-vars  ; form list of input variables
      (for f-c-i in f-c-i-list
       collect (list (nth$ f-c-i 2)(nth$ f-c-i 4))
      )
     )
      (setq output-vars  ; form list of output variables
        (for f-c-o in f-c-o-list
          collect (list (nth$ f-c-o 2)(nth$ f-c-o 4))
        )
      )
      (setq packets ; produce a list of input packet lists
                    ; in correct order
        (for var-pair in input-vars
          collect
          (if (slot-null (nth 1 var-pair) 'packet-link)
            ;if slot is nil then
              NIL
            ; else
              (car
                (list$ (get-schema-value (nth 1 var-pair)
                           'packet-link)))
           )
        )
      )
      (do-eval block input-vars output-vars  packets)
  )
)

(defaction ; debugging before action (method)
  (do-eval before)(block)(input-vars output-vars packets)
  (printout t t "do-eval called on " block
    " with " input-vars output-vars packets)
)
```

```
(defaction ; main action
  do-eval (all-block)(input-vars output-vars packets)
  (let  ((packet-name nil) (result-data-type
            (get-schema-value all-block 'result-data-type))
        )
    ;;; progv creates new dynamic variables
    ;;; and restores old ones when it finishes
    (progv (mapcar #'car input-vars)
      (mapcar #'get-data packets) ; variables now bound
      (mapcar #'destroy-packet packets) ; remove packets
                                        ; from inputs
        ;;; Now evaluate the function slot's contents
        (eval (invoke 'perform-function all-block))
        ;;; Now create o/p schemata and populate
        (for var-pair in output-vars
        do
          (setq packet-name (create-packet 'packet))
          (put-data-type packet-name result-data-type)
          (insert-data packet-name  (eval (nth 0 var-pair)))
                                ;;; result of evaluation
          (place-packet packet-name (nth 1 var-pair))
                  ;; move packet now to port, eg portQ
        )
      );;; end progv
  ) ;;end let
)

(defaction do-eval
  (alt-block)(input-vars output-vars packets)
  (let ((packet-name nil)
          (result-data-type (get-schema-value alt-block
                              'result-data-type))
       )
    ;;;progv creates new dynamic variables and restores old
    ;;;ones when it finishes
      (progv (mapcar #'car input-vars)
        (mapcar
          #'(lambda (x)(if x (get-data x) 'NAN)) packets)
              ; variables now bound
      (mapcar
          #'(lambda (x)(if x (destroy-packet x))) packets)
              ; remove packets from inputs
      ;;; Now evaluate the function slot's contents
      (eval (invoke 'perform-function alt-block))
      ;;; Now create o/p schemata and populate
      (for var-pair in output-vars
        do
          (setq packet-name (create-packet 'packet))
          (put-data-type packet-name result-data-type)
          (insert-data packet-name  (eval (nth 0 var-pair)))
            ;;; result of evaluation
          (place-packet packet-name (nth 1 var-pair))
            ;; move packet now to port, eg portQ
      )
    );;; end progv
  ) ;;end let
)
```

```
(defaction perform-function (block)()
  (list*$ (get-schema-value block 'function))
)

(defaction ;; clean up operrations
  (do-eval after) (block)(input-vars output-vars packets)
  ;;; update the function-firings slot
  (modify-schema-value block
    'function-firings
    (+ (get-schema-value block 'function-firings) 1))
)

(defaction
  do-eval (round)(input-vars output-vars packets)
  (let* ((input-port (second (car input-vars)))
         (output-port (second (car output-vars)))
         (input-packet(car packets))
         (bits (get-schema-value round 'bits))
         (data (get-data input-packet ))
         (data-type (get-data-type input-packet))
         (result-data-type
          (get-schema-value round 'result-data-type))
         (overflow-list (get-overflow input-packet))
         (output-packet nil)
       )
    (destroy-packet input-packet)
    (multiple-value-bind (result overflow)
        (our-round result-data-type (list$ bits) data)
        (setq output-packet (create-packet 'packet))
        (put-data-type output-packet result-data-type)
        (insert-data output-packet result)
        (if overflow then
          (put-overflow output-packet round))
        (place-packet output-packet output-port)
    )
  )
)

(defaction (our-round before)(number)(bits data) ; debugging
  (printout t t "our-round called on " number )
)

(defaction ; round a fixnum to a number of bits
  our-round (fixnum)(bits data)
  (let* ((sign (signum data))
         (number (abs data))
         (result ;;; performs round
          (* (mask-field
              (byte (- bits  1) 0) number)  sign))
       )
;   (printout t t "our-round 'fixnum' called on " data )
;   (printout t t "result is " result)
    (if (> (integer-length data) bits)
        (values result 'overflow)
        (values result nil)
    )
  )
)
```

```
;;; this action is a little complicated
;;; first it braks up the data into three integers ->
;;; mantissa, exponent and sign.
;;; the mantissa is a fraction ie the bit order is reversed
;;; so masking is done from the left
;;; the exponent is an integer so it is done the same way as
;;; a fixnum


(defaction ; round short float using bit mask
  our-round (short-float)(bits data)
  (let* ((bits-m (first bits))
         (bits-e (second bits))
         (result nil)
         (e-sign nil)
         (mantissa nil)
         (exponent nil)
         (m-sign nil)
         (m-p-mantissa nil)
         (m-p-exponent nil)
         (m-p-m-sign nil)
        )
;    (printout t t "our-round 'short-float' called on " data
)
     (multiple-value-setq
       (m-p-mantissa m-p-exponent m-p-m-sign)
       (integer-decode-float most-positive-short-float)
     )
       (setq max-digits (integer-length m-p-mantissa))
       ;;; no of digits in the largest short-float "about 23"
       (multiple-value-setq (mantissa exponent m-sign)
         (integer-decode-float data)
       ;;; break up the data into its components
       )
       (setq e-sign (signum exponent))
       (setq m-r
         (* (mask-field
              (byte max-digits (- max-digits bits-m))
               mantissa)  m-sign))
         ;; should be ok , needs looking into.
       (setq e-r (* (mask-field (byte (- bits-e  1) 0)
                    (abs exponent))  e-sign));; this is ok
     ;; now check for overflow and return result
       (setq result (scale-float m-r e-r))
;      (printout t t "result is " result)
       (if (not (equalp result data))
         (values result 'overflow)
         (values result nil)
       )
  )
)
```

```
;;; -*- Mode:ART; Package:art-user; Base:10. -*-
;;; Packet operations now supported :
;;; (don't address packets directly now!!!)
;;;
;; (coerce-to <type> <data>)
;; (insert-data <packet> <data>
;; (put-data-type <packet> <data>
;; (put-overflow <packet> <data>
;; (get-data <packet>)
;; (get-data-type <packet>)
;; (get-overflow <packet>)
;; (create-packet 'packet)
;; (destroy-packet <packet>)
;; (move-packet <packet> <from> <to>)
;; (create-packet-with-data 'packet<data>)
;; (place-packet <packet> <port>)
;;; non active value version !!!!!

(defaction ; change type catch NANS
  (coerce-to whopper) (number)(data)
    (if (equalp data 'Nan)(values 'Nan)
      (whopper-continue number data)
    )
)

(defaction coerce-to (number)(data)
  (convert (type-of data) number data)
)

(defaction convert (number number)(data)
  (coerce data number)
)

(defaction convert (float integer)(data)
  (values (coerce (round data) integer))
)

(defaction convert (integer float)(data)
  (values (coerce (round data) integer))
)

(defaction convert (number ratio)(data)
  (values (coerce (rationalize data) ratio))
)

(defaction convert (number complex)(data)
  (values (coerce (complex data) complex))
)

(defaction convert (complex number)(data)
  (format t "Warning converting data type ~
  from ~A to ~A, you will lose information ~
  on: ~A ~%"  complex number data)
  (values (coerce (realpart data) number))
)
```

```
(defaction get-data (packet)()
|#  (printout t t "get-data "
      (get-schema-value packet 'data) " from packet "
      packet)#|
  (values (get-schema-value packet 'data))
)

(defaction get-data-type (packet)()
|#  (printout t t "get-data-type "
    (get-schema-value packet 'data-type)
    " from packet " packet)#|
  (values (get-schema-value packet 'data-type))
)

(defaction get-overflow (packet)()
|#  (printout t t "get-overflow " (get-schema-value packet
      'overflow) " from packet " packet)#|
  (values (get-schema-value packet 'overflow))
)

(defaction insert-data (packet)(data)
  (let  ((data-type (get-data-type packet))
         (new-data nil)
        )
;   (printout t t data)
;   (printout t t (equalp data 'null))
    (cond
      ((equalp data 'null)
          (modify-schema-value packet 'data 'null))
      ((schemap data-type)
          (progn
          ;;(printout t t "doing insert data with " data)
            (setq new-data
              (coerce-to (get-data-type packet) data))
          |#(printout t t "insert-data " data " in packet "
                packet)#|
            (modify-schema-value packet 'data new-data)
           )
      )
      (T
       (progn
        (printout t t
         "Error !! trying to insert data of unknown type: "
          data-type)
        (break)
       )
      )
     )
    )
; (printout t t "done insert-data")
)
```

```
(defaction put-data-type (packet)(data-type)
|#(printout t t "put-data-type "
  data-type " in packet " packet)#|
  (cond
    ((equalp data-type 'null)
      (modify-schema-value packet 'data-type 'null))
    ((schemap data-type)
      (modify-schema-value packet 'data-type data-type))
    (T (progn
        (printout t t
      "Error !! trying to put-data-type of unknown type: "
          data-type)
        (break)
      )
     )
    )
)

(defaction put-overflow (packet)(data)
; (printout t t "put-overflow " data " in packet " packet)
  (modify-schema-value packet 'overflow data)
)

(defaction add-to-packet-link (packet port)()
; (format t "adding to packet-link ~A ~A~%" packet port)
  (modify-schema-value port 'packet-link
    (create$
      (append
        (list$ (get-schema-value port 'packet-link))
              (list packet)) t)
  )
)

(defaction delete-from-packet-link (port)()
  (let* ((old-value
           (list$ (get-schema-value port 'packet-link)))
           (new-value (cdr old-value))
         )
      (if new-value
        (modify-schema-value port 'packet-link
          (create$ new-value t))
        (retract-schema-value port 'packet-link)
      )
  )
)

(defaction  make-packet (packet)()
  (let* ((packet-name (gentemp "P-")))
    (put-schema-value 'packet 'has-instances  packet-name)
;     (printout t t "make-packet "  packet)
      (values packet-name)
    )
)
```

```
(defaction use-existing-packet (packet)()
  (let* ((packet-list
          (list$ (get-schema-value
                  'free-packets 'packet-link))))
    |#(printout t t "use-existing-packet " (car
       packet-list))#|
;; explicit change to packet-link slot now done.-
;; no active value
      (invoke 'delete-from-packet-link 'free-packets)
      (values (car packet-list))
  )
)

(defaction create-packet (packet)()
; (printout t t "create-packet" )
  (let ((packet-list
          (list$ (get-schema-value 'free-packets
                  'packet-link))))
      (if packet-list
        (setq packet-name ;; use existing packet
          (invoke 'use-existing-packet packet))
          ;; otherwise make one
          (setq packet-name (invoke 'make-packet packet))
    )
    ;; explicit change to packet-link slot now done.-
    ;; no active value
      (values packet-name)
  )
)

(defaction destroy-packet (packet)()
; (printout t t "destroy-packet " packet)
  (let* ((port (get-schema-value packet 'node)))
      ;; explicit change to packet-link slot now done.- no
      ;; active value
      (delete-from-packet-link port)
      (put-data-type packet 'null) ;;;data slot now null
      (put-overflow packet  'null) ;;;data slot now null
      (insert-data packet   'null) ;;;data slot now null
    (modify-schema-value packet 'node 'free-packets)
    ;; explicit change to packet-link slot now done.-
    ;; no active value
    (add-to-packet-link packet 'free-packets)
  )
)

(defaction move-packet (packet (source port)(dest port))()
  (invoke 'delete-from-packet-link source)
    (modify-schema-value packet 'node dest)
    (invoke 'add-to-packet-link packet dest)
)

(defaction place-packet (packet (dest port))()
  (modify-schema-value packet 'node dest)
  (invoke 'add-to-packet-link packet dest)
)
```

```
(defaction ;;; quick shorcut
    create-packet-with-data (packet)(data)
  (insert-data (create-packet packet) data)
)
```

```
;;; Simple decision heuristic to test ideas

(defrule decision1
  (declare (salience *maximum-salience*))
  ;; to we have a instantiation of a component
  (or
   (utterance ?time ?window
    (mouse-click ?button & left ?times & 1 ? ? ))
   (utterance ?time ?window
    (menu-mouse-click ?x-w ?y-w (?block) ?button & right
       ?times & 1 ? ? (?area & #L| delete|)))
  )
  ;; ignore the old utterances
  (not (and (utterance ?high-time & -?time ? ?)
            (test (> ?high-time ?time))))

  (schema current
    (mode construction)     ;; we are now constricting
    (last-mode simulating) ;; and we were simulating
  )
  (schema ?window-icon ;; bind variables for new world
    (window ?window)
    (display-parameters (?x ?y ?w ?h))
  )
=>
  ;; say we are creating new world
  (printout t t "creating new world as changes to "
     ?window-icon " after a simulation")

 (bind ?new-world (get-icon-name ?window-icon))
 ;; Create newwindow for new world:
 ;; User interface stuff
 (create-window ?new-world  'graphics
    (incf  ?x  10) ?y
    (+ ?x ?w) (+ ?y ?h)
    (string ?new-world)
 )
 (#Lai::create-window-icon ?new-world)
 ;; create new world now
 ;; slots will be filled in by inheritance
 ;; and slot fill rules
 (assert (schema ?new-world
     (our-instance-of world)
     (child-world-of ?window) ;; link to parent world
     (reason "changes after a simulation") ;; noddy reason
     (window ?new-world)
   )
 )
 ;; create rest of the block diagram
 (for icon in
      (list*$ (get-schema-value ?window-icon
           'our-contains-icons))
      bind new-icon
   do
   (setq new-icon (get-icon-name icon))
   (our-create-icon icon new-icon) ;; make a copy
   (our-add-icon-to-window new-icon ?new-world)
```

```
    (parse
      `(assert (schema ,?new-world
                 (our-contains-icons ,new-icon)
               )
      )
    )
   )
   ; (printout t t "done")
   (refresh-window ?new-world)
   (modify (schema current
       (last-mode construction)
    )
   )
  )
```

```
;;; Define test constraints

(defschema constraints
  (kinds chip-area design-time power)
)

(defschema chip-area
  (values (v-large large medium low v-low))
  (v-large 10000)
  (large   1000)
  (medium 100)
  (low 10)
  (v-low 1)
)

(defschema design-time
  (values (v-long long medium short v-short))
  (v-long 10000)
  (long   1000)
  (medium 100)
  (short 10)
  (v-short 1)
)

(defschema power
  (values (v-high high medium low v-low))
  (v-high 10000)
  (high   1000)
  (medium 100)
  (low 10)
  (v-low 1)
)

(defschema speed
  (values (v-fast fast medium slow v-slow))
  (v-fast 10)
  (fast  100)
  (medium 1000)
  (slow 10000)
  (v-slow 100000)
)
;; use iterative method this time instead of rule based
approach

(defun calc-constraint-value (schema constraint)
  (let ((constraint-list
          (remove nil
            (calc-constraint-list schema constraint))))
    (if (null constraint-list) nil
     (values-list (mapcar #'+ constraint-list))
    )
  )
)
(defun calc-constraint-list (schema constraint)
  (nconc (list (get-schema-value constraint
           (get-schema-value schema constraint)))
   (for block in
```

```
        (list*$ (get-schema-value schema 'contains-blocks))
      join
        (calc-constraint-list block constraint)
   )
  )
)
;;; Interface rules for calculating constraint values

(defrule calc-constraint-time
?f <-  (utterance ?time ?window (menu-mouse-click ?x-w ?y-w
         (()) ?button & right ?times & 1 ? ? (|time|)))
  (schema ?window-icon
    (window ?window)
  )
  (schema design-time
    (values ?values)
  )
=>
  (printout "the range of times is "
    (for value in (list*$ ?values)
      collect (list value
        (get-schema-value 'design-time value))))
  ;; returns number, requires wrap around
  ;; to convert back to values
  (calc-constraint-value ?window-icon 'design-time)
)

(defrule calc-constraint-chip-area
?f <-  (utterance ?time ?window (menu-mouse-click ?x-w ?y-w
(()) ?button & right ?times & 1 ? ? (|area|)))
  (schema ?window-icon
    (window ?window)
  )
=>
  (calc-constraint-value ?window-icon 'chip-area)
)

(defrule calc-constraint-power
?f <-  (utterance ?time ?window (menu-mouse-click ?x-w ?y-w
(()) ?button & right ?times & 1 ? ? (|power|)))
  (schema ?window-icon
    (window ?window)
  )
=>
  (calc-constraint-value ?window-icon 'power)
)

(defrule calc-constraint-speed
?f <-  (utterance ?time ?window (menu-mouse-click ?x-w ?y-w
(()) ?button & right ?times & 1 ? ? (|speed|)))
  (schema ?window-icon
    (window ?window)
  )
=>
 ;; link into calc path seq stuff not completed
)
```

```
;;; -*- Mode:ART; Package:art-user; Base:10. -*-
;;;
;;; Using viewpoints.
;;; these set of rule extract the separate paths from an
;;; interconnected set of blocks so that the delay
;;; associated with a path can be calculated.
;;; Feedback loops and feedforward paths are identified
;;; Not fully integrated with PEDA.

(defrule start-path-seq  ;;; start path generation
  (schema current
    (find paths)
  )
  (schema ?output
    (instance-of port)
    (conn-to ?next)
    (our-contained-in ?block)
  )
  (not (schema ?input &~?output
         (instance-of port)
   (conn-from ?)
         (our-contained-in ?block)
       );;; find an entry point
  )
=>
  (sprout
   (assert (path-seq ?output ?next));;;; and start there
  )
)

(defrule continue-path-seq-1
  (schema current
    (find paths)
  )
  (schema ?input
    (instance-of port)
    (conn-from ?)
    (our-contained-in ?block)
  )
  (schema ?output &~?input
    (instance-of port)
    (conn-to ?next)
    (our-contained-in ?block)
  )
  (path-seq  $?body ?input)
;;; forget about circle here
=>
  (sprout
   (assert (path-seq $?body ?input ?output ?next))
  )
)
```

```
(defrule prevent-to-much-circle  ;;; stop loops fom being
repeated indefinitely
  (declare (salience *constraint-salience*))
  (schema current
    (find paths)
  )
  (path-seq  $? ?from $?body ?from $? ?from $?)
=>
  (poison "loop travelled more than once")
)


(defrule clean-up
  (declare (salience -5))
  (schema current
    (find paths)
  )
  (viewpoint ?vp
?x<- (path-seq $?body)
  )
  (viewpoint ?vp2
     (path-seq $?body ? $?)
  )
=>
  (retract ?x)
)

(defrule clean-up-2
  (declare (salience -5))
  (schema current
    (find paths)
  )
?x<-(path-seq  $? ?body $? ?body)
=>
  (retract ?x)
)


(defrule clean-up-3
  (declare (salience -6))
  (schema current
    (find paths)
  )
  (path-seq  $?)
=>
  (believe ?root "collaspsing path-seq to single-level")
)

(defrule print-path-seq
  (declare (salience -10))
  (schema current
    (print paths)
  )
  (viewpoint ?vp
       ?x<-  (path-seq $?body)
         (not (path-seq $?body ?))
  )
=>
```

```
(printout t t
  (create$ (list 'path-seq #L:splice (list$ ?body))))
)


(modify-schema-value 'current 'find 'paths)
```

```
;;;; Auxiliary Functions

(defun member-relation-p (schema relation value)
  (let ((slot-value (get-schema-value schema relation)))
    (cond ((null slot-value) NIL)
          ((symbolp slot-value)
            (if (equalp slot-value value) slot-value))
          ((listp slot-value)
            (member value (list*$ slot-value)))
    )
  )
)

(defun init-packets ()
  (for packet in
    (list*$ (get-schema-value 'packet 'has-instances))
    do
      (if packet
        (destroy-packet packet)
      )
  )
)

(defun  clear-function-firings-slots ()
  (for the-block in
    (list*$ (get-schema-value 'block 'has-instances))
    do
      (modify-schema-value the-block 'function-firings 0)
  )
)

(defun resize-data-slot-in-memory-schema ()
  (for memory in
    (list*$ (get-schema-value 'memory 'has-instances))
    bind size data new-data
    do
    (if memory
      (if (setq size (get-schema-value memory 'size))
        (if (slot-null memory 'data)
          (setq new-data
            (make-list size :initial-element 'NAN))
          (progn
            (setq data
              (list$ (get-schema-value memory 'data)))
            (setq new-data
              (replace
               (make-list size :initial-element 'NAN) data))
          )
        )
      )
      (modify-schema-value memory 'data new-data)
    )
  )
)
```

```
(defun Nan-check (&rest vars) ;; returns t if any vars are
;; (N)ot (A) (N)umber
    (not (for var in vars
           always
           (not (equal var 'Nan))
          )
    )
  )

(defun set-Nan (&rest vars)
    (for var in vars
      do
      (setf (symbol-value var) 'Nan)
    )
  )


; function our-get-icon-name, returns new-icon name
; given mux mux-text-1 adder, returns: adder-text-1

(defun our-get-icon-name
       (original-parent original-icon new-parent)
  (prog1
    (intern
     (concatenate
      'string (string new-parent)
      (subseq (string original-icon)
        (mismatch (string original-parent)
          (string original-icon) :test #'char-equal)
      )
     )
    )
  )
)

(defun insert-schema-value (schema slot value)
  (if (not (slotp schema slot))(slotc schema slot))
  (if (eql (slot-get-type schema slot 'slot-how-many)
           'single-value)
    (modify-schema-value schema slot value)
    (put-schema-value schema slot value)
  )
)
```