THE UNIVERSITY OF CALGARY

Expert Systems for VLSI Computer Aided Design

BY

Mark D. Brinsmead

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

December, 1988

ISBN 0-315-50303-3

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Expert Systems for VLSI Computer Aided Design", submitted by Mark D. Brinsmead in partial fulfillment of the requirements for the degree of Master of Science.

Dr. J. Kendall
Department of Computer Science

Dr. G. Birtwistle
Department of Computer Science

Dr. B. Nowrouzian
Department of Electrical Engineering

Date December 15, 1988

Abstract

Knowledge-based expert systems have found increasing use in a wide variety of applications over the past few years. One area in which they have become particularly popular is design automation, a domain in which most problems are known to have Non-determinisitic Polynomial time complexity. Despite their great popularity, expert systems suffer a number of liabilities in this domain, the greatest being the problem of applicability to design- or synthesis-oriented problems. This liability is often overcome by using an iterative refinement paradigm which allows the design problem to be attacked as a diagnostic problem, for which expert systems are better suited, and for which they are shown to be effective.

This thesis investigates the value and practicality of incorporating stochastic processes in design-oriented expert systems. It is demonstrated that expert systems employing simple iterative refinement are subject to arrest in local minima, and that the use of randomness in the control structure of an expert systems can ameliorate this situation. An experimental expert system using a stochastic procedure based upon Simulated Annealing is developed and used for this demonstration.

Also presented in this thesis is the Expert System Wrapper Environment (`ESWE`), with which the test "experts" were generated. `ESWE` was developed by the author to provide simple, convenient control structures for the synchronisation of sub-tasks within an expert system. It assists program development by automatically adding necessary control patterns to rules written in ART, and provides mechanisms to simplify the manipulation of control facts.

Acknowledgements

The number of groups and individuals due recognition for their contributions to this thesis is large indeed; I shall strive to include all of them, but because of restricted space, must beg forgiveness of those who may be omitted.

I first wish to acknowledge the generous assistance of the Alberta Microelectronic Center, without whose financial support I surely could not have completed this work.

Next, I thank my supervisor John Kendall for having so patiently listened to my questions and dumb ideas, and read so many working drafts of this and other documents. His calm patience and encouragement have proven most valuable.

I am indebted to many faculty members, either for their interesting and worthwhile courses and seminars or for equally interesting informal discussions and conversations. Foremost among these is Graham Birtwistle, who first sparked my interest in VLSI design automation, and encouraged me to enter Graduate Studies.

Thanks are due also to my friends and colleagues at the University of Calgary, and to the excellent members of the support staff, who offered guidance and assistance with equipment problems, and who so cheerfully told me "I told you so" when I suffered catastrophic equipment failure. (My education here has certainly included lessons in more than just Computer Science.) Special thanks are due to Inder Dhingra, Darrin West, Clarence Comsky, Dave Mason, Keith Andrews, Bonnie Bray, and all the members of the VLSI group.

The final word of acknowledgement is unquestionably due my parents, Gilbert and Sheila. Without their love and support, I could have never undertaken this project.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Since the advent of integrated circuit technology in the 1960's, the complexity of integrated circuits has been continuously increasing, from as few as four or five transistors per die initially to as many as one million transistors today. Continuing advances in IC fabrication technology, which promise transistor counts as high as several million for a one centimeter squared die in the imminent future [1] and the possibility of increased die sizes allow for continued growth in the size and complexity of circuits.

As these circuits become increasingly large, the effort required to design, verify, and test them increases combinatorially. The design time for a large integrated circuit [2] may reach hundreds of man years, while exhaustive testing becomes virtually impossible. Consequently, it is often possible for a very complex circuit, such as a 32-bit microprocessor, to be in common use for several years with design flaws still being revealed.

With the goal of reducing the cost of circuit design and improving the reliability of the circuits produced, a great deal of effort has been expended in automating the design process. Over time, design automation efforts have become increasingly ambitious, moving from simple geometrical layout editors, or "polygon pushers", to the automated synthesis of circuit layouts from high-level specifications.

[1] Four-megabit RAMs are already available in sample quantities [Ele88] and sixteen-megabit RAMs are currently under development [Cole 88].

[2] Eg. the design of Motorola's 88000 processor is reported to have taken *only* 20 elapsed months, despite the use of state-of-the-art design tools and a Reduced Instruction Set architecture.

"Silicon compilers", software tools intended to produce mask-level layouts with minimal human interaction, have been in use for some time now. Despite the considerable variety of techniques and approaches, most silicon compilers share a number of common features. Fundamental among these is the subdivision of the general layout problem into "simpler" problems of component placement, network routing, and compaction. Each of these sub-problems, however, is known to be NP-complete [Sastry 82] [Schlag 83] [Sahni 80], and are therefore regarded as too time consuming to solve optimally. Consequently, design automation researchers have been forced to search for suitable heuristics which can be employed to produce placements, routes, or compactions of reasonable[3] quality, without incurring the cost of *exhaustively* searching for an optimum solution. It should be noted, however, that in order for a heuristic to avoid the problems of NP-completeness, "optimality" must be sacrificed, and some notion of "acceptable quality" must be used in its place.

One mechanism which has proven quite valuable in this context is the *knowledge-based expert system.* Knowledge-based expert systems have existed for over a decade and are demonstrably useful for problems which involve some form of diagnosis[4], especially when complete information is not available. It has been thought for some time that expert systems would also prove useful for design-oriented problems such as circuit layout or routing, and considerable experimental evidence has been presented to support this belief [Kim 85, Joobbani 86b, Rosenbloom 85, Subramanya 86, Kollaritsc 85]. Unfortunately, some of these same experiments have demonstrated problems with design-oriented expert systems [Ackland 88]. These problems are now briefly discussed.

---

[3]Typically, "reasonable" is intended to mean "comparable to hand-done work". This, however, is dependent upon the cost of the heuristic; a very expensive (slow) heuristic is expected to produce very high quality results, while we are pleased when a fast simple heuristic works acceptably for very simple cases.

[4]This is the class of problems for which knowledge-based expert systems were originally designed [Shortliffe 76].

1.1.1 The Problem of Local Minima

One of the greatest problems stems primarily from difficulties in expressing design-oriented activities in a way suitable for expert systems. One favored technique [Brewer 86] is that of producing an "initial" design, and iteratively refining it. This, in effect, casts the *design* problem into a *diagnosis* problem, which experts systems are better suited to tackle. Following this approach, the expert system's knowledge is used to recommend localised improvements to the design. These recommendations are usually applied "greedily" [5], trusting the expert system to provide adequate foresight to suggest only moves which will eventually lead to a global optimum.

Heuristics which operate using iterative refinement to minimise (or maximise) some function are members of a class known as "adaptive heuristics". When applied to design in the manner just described, knowledge-based expert systems form a distinct and relatively simple [6] subset of this class, and are typically characterised by their greediness; short-term advances toward the desired goals are readily taken, without regard for the possibility of longer-term advantages of other options. Consequently most knowledge-based expert systems are subject to arrest [7] in local minima in the objective function they seek to optimise, occasionally producing very inferior results.

As an example, consider the task of locating a value for x at which some function $F(x)$ (see Figure 1.1) has the least value. One iterative refinement method for solving this problem begins by selecting an arbitrary value x', and determining the value of $F(x')$. Next, a value x'' is selected, $x'' = x' \pm h$, and the function is evaluated at this point. If the new value, $F(x'')$, is less than the previous, $F(x')$, accept x'' as the current minimum of the function, otherwise x' remains the current minimum. By

[5] I.e. recommendations are accepted whenever they lead to an improvement, and rejected otherwise.

[6] That is, they are simple in terms of adaptation.

[7] I.e., becoming trapped.

repeating this procedure until no further advances can be made, or until no advances have been made in a specified period of time, we slowly converge toward a minimum of the function F.[8]

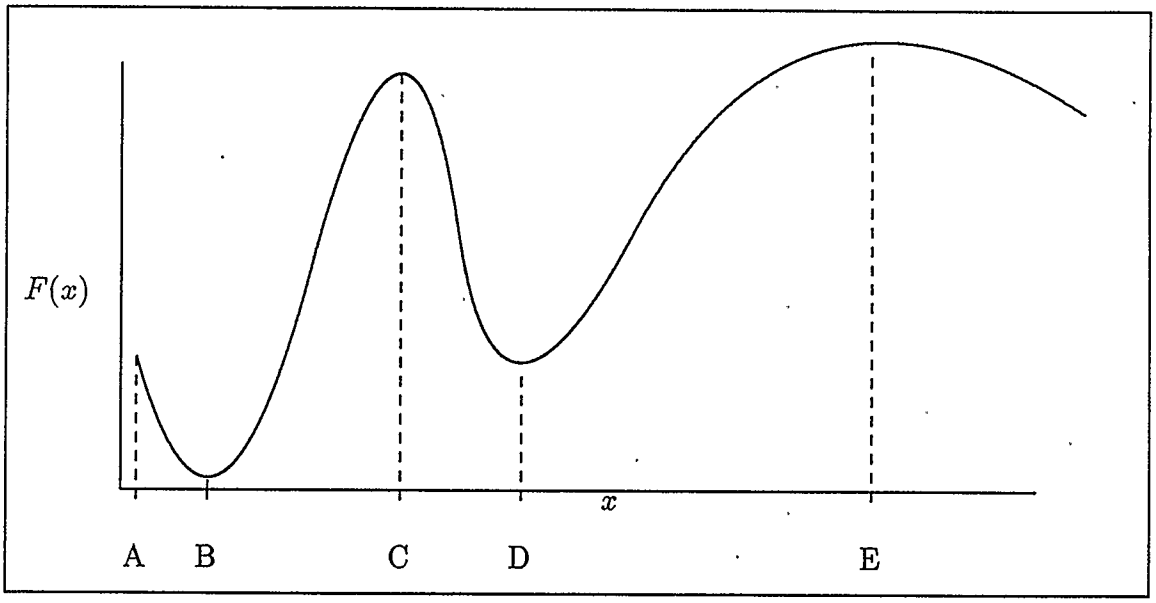

Figure 1.1: Function Optimisation with Local Mimima

This is an example of a "greedy", or opportunistic, iterative refinement procedure, in which short term advances are always capitalised upon without regard for the possible long-term consequences. For example, if the initial x' is chosen such that $A < x' < C$, the final result will (ideally) be B, which is the global minimum of the function F, at least, for the range of F displayed in Figure 1.1. If the initial value of x' is chosen such that $C < x' < E$, then the "optimum" value for x' determined by this method will be D, which is clearly *not* the global minimum of F. In the latter of these two cases, the iterative refinement procedure is said to have arrived (or arrested) at a *local minimum.*

Members of other subsets of adaptive heuristics operate on the premise that there is frequently long-term advantage in occasionally taking short-term losses when try-

[8]It is sometimes possible, if $|h|$ is sufficiently small, and with a small enough time interval, that the technique *may* converge upon a meta-stable value such as E in Figure 1.1.

ing to optimise a function. For certain classes of objective functions, these techniques are able to produce good results much more consistently than are greedy techniques, but require large amounts of time. One technique known as "Simulated Annealing", in particular, is known to converge to a global optimum,[9] but is characterised by very large time requirements.

1.1.2 The Problem of Control and Maintenance

Another major problem displayed by knowledge-based expert systems as applied to design automation is that of sequential control and code maintenance. The normal knowledge-based expert system paradigm calls for complete "modularity" in the rules used, requiring that, wherever possible, rules be described as independent daemons[10] which wait for a suitable circumstance to activate themselves. The exertion of explicit control over the sequence of rule activation is strongly discouraged, as it tends to undermine the effectiveness of the expert system.

Design automation, however, is a domain in which sequential operations are essential. That is, one subtask *must* be completed before the next subtask can commence[11]. Imposing sequential behaviour on groups of rules is relatively simple, requiring only the addition of one or more control-oriented conditions on the activation of each rule. Unfortunately, this quickly becomes unmanageable as the number of rules and subtasks grows, rapidly reaching the point at which modifying the sequence of execution is nearly impossible.

---

[9] Given a sufficiently slow cooling schedule, Simulated Annealing will converge to a solution assymptotically close to the global optimum in finite time with probability 1 [Mitra 85].

[10] A daemon is, in the *software* sense, a piece of code that sits "dormant" waiting for some previously specified circumstance to activate it, performs a function, and then returns to dormancy.

[11] As an example, consider attempting to place the metal traces on a circuit board before chip-locations have been assigned, or trying to change a tire before the lug nuts have been removed. Each task is likely possible, given a sufficiently determined effort, but can be hardly considered productive.

In order to overcome this difficulty, it is necessary to hide some of the details of the sequencing mechanism from the programmer, allowing sequencing conditions to be automatically generated for each rule.

1.2 Thesis Outline

This thesis presents a software environment for the development of rule-based expert systems for very large scale integrated circuit (VLSI) layout, which provides support for the special requirements of expert systems for design-oriented problems. Of particular importance is the inclusion of special structures for controlling the sequence of operations in expert systems and a technique for avoiding problems with experts arriving at local minima. The remainder of the thesis describes a software environment for developing design-oriented expert systems and recommends a paradigm for solving problems with early termination by stochastic means.

Chapter 2 provides a description of the history of integrated circuit design automation, with a particular emphasis placed on the use of heuristics for this purpose. Considerable attention is devoted in this chapter to the background and principles of expert systems and Simulated Annealing.

Chapter 3 describes a software environment, developed by the author, for the description and control of design-oriented expert systems which automatically generates most of the "code" needed to control the sequence of operations in an expert system.

Chapter 4 describes the usefulness of stochastic process in design-oriented expert systems and shows the results of two closely related expert systems specially developed for this project, one of which uses a form of Simulated Annealing as a "meta-heuristic", and the other of which behaves in a more conventional manner.

Finally, Chapter 5 presents conclusions and suggestions for further research in this domain.

Chapter 2

Historical Perspective of Silicon Compilation and Expert Systems

Recent years have seen a growing general awareness of the cost and difficulty associated with designing large scale (LSI) and very large scale (VLSI) integrated circuits. This, together with the considerable competitive advantage of smaller, faster, and more reliable systems resulting from increasing circuit densities, has provided a very strong incentive for the automation of integrated circuit design.

When designing LSI and VLSI circuits by conventional means, a very large portion of the designer's time is spent working at the transistor or gate level. However, due to increasing demands for speed and reliability, which can only be met by increasing chip densities, higher level[1] aspects of design are of considerable importance in VLSI circuit design. As Mead and Conway pointed out in 1979 [Mead 80, pp. v]:

> ... Many LSI chips, such as micro processors, now consist of multiple complex subsystems, and are thus really *integrated systems* rather than integrated circuits."
>
> What we have seen so far is only the beginning. Achievable circuit density now doubles with each passing year or two. Physical principles indicate that transistors can be scaled down to less than 1/100th of their present area [2] and still function as the sort of switching elements with which we can build digital systems. By the late 1980's, it will be possible to fabricate chips containing millions of transistors. The devices and

[1] I.e. register-transfer level and system-level

[2] The lower bound on transistor gate width for digital logic is believed to be approximately 0.25 microns.

> interconnections in such very scale integrated (VLSI) systems will have linear dimensions less than the wavelength of visible light. ...

After ten years, these predictions prove to have been surprisingly accurate. Today, multi-million transistor chips are a reality, but this sort of density has been achieved only with relatively simple and repetitive structures such as random access memories. Today, major issues of microprocessor design include such things as (on-chip) caching schemes, virtual address translation, register files and allocation practices, and floating point support, which had never been considered as part of the domain of microprocessor design only a few years ago. Accordingly, the chip designer of today is called upon for much more than circuit design and layout.

In order to permit designers to concentrate more on these very important high-level issues, as well as to reduce the cost and increase the reliability of VLSI systems, it is useful to automate as much of the low-level "detail" work as possible. The earliest efforts in this regard usually came either in the form of useful or simplifying design abstractions, or in the form of rigorous and exhaustive algorithms intended to optimally solve simple (and tedious) subtasks. The primary focus of much of this effort has been directed toward removing some of the routine tedium from the design process in order to allow highly skilled design experts to focus their energy on the more important issues of the design process.

In recent years, the scope of design automation efforts has expanded, progressing from mechanically assisting the drawing of masks and automatically checking for design rule violations, toward automatically designing or extracting entire layouts. The generally accepted engineering practice for producing circuit layouts is to subdivide the task into the problems of *module placement*, *network routing* (global and detailed), and layout optimisation, or *compaction*. Most frequently, these steps are

regarded as *distinct*, and applied sequentially; they are, however, interdependent, and must be allowed to interact with one another to achieve optimal circuit designs.

Each of these problems has presented considerable difficulty to those trying to automate it; in fact, all three are now known to NP-complete[3] [Sahni 80] [Schlag 83]. NP-completeness does not *necessarily* imply the intractability of these problems; theory allows for the possible existence of an efficient (i.e. polynomial complexity) solution method to NP-complete problems, but no such method is presently known for *any* NP-complete problem.

2.1 Heuristics in Design Automation

As a consequence of the very large computational complexity of most VLSI design problems, design automation researchers have been forced to resort to one of two basic strategies: formulating simplified models under which circuit design may be performed more efficiently; and developing heuristics with which problems may be solved more quickly (but without the assurance of an optimal result[4]). Occasionally, considerable gains have been made with simplified models (such as symbolic layout), but, in general, real progress is only made possible by efficient heuristics for the original problem.

For the purposes of this thesis, a *heuristic* may be regarded as any procedure or technique which is intended to solve a problem by making use of some kind of knowledge or assumptions about that problem (or about likely instances of that problem). In this sense, then, the terms *heuristic* and *heuristic procedure* are interchangeable; an expert system, for example, is referred to as *a heuristic* without regard to how many independent sources of knowledge it may actually encompass.

[3]That is, members of the class of problems having non-deterministic polynomial computational complexity [Aho 74].

[4]Only by failing to assure optimality can the NP complexity problem be avoided.

The aim of a heuristic is to find, with a high probability of success, a "good" [5] solution to a problem in much less time than theory predicts is required to find an optimal solution. For the purpose of this thesis, heuristics may be regarded as falling into two categories: *weak heuristics* which apply knowledge about *problem solving* to a very broad class of problems with little or no regard for the specifics of the problem; and *strong heuristics*, which apply knowledge about a specific problem, or even specific instances of the problem, to that problem only. The class of strong heuristics may be further subdivided into the classes *simple* and *complex*. Simple strong heuristics apply only one or two "pieces" of knowledge [6] to a problem, while complex strong heuristics may contain hundreds or thousands of these "pieces of knowledge".

One example of a weak heuristic is the technique known as "Simulated Annealing" [Kirkpatric 83]. Strong heuristics abound in many domains; examples of strong heuristics for IC design automation are now far too numerous to list here.

The remainder of this chapter describes some of the work done in developing abstractions, strong heuristics, and weak heuristics for LSI and VLSI design automation.

2.2 Design Abstractions

Over time, circuit designers have adopted a number of abstractions with the objective of simplifying their task. Most of these abstractions serve simply to "hide" certain levels of detail, deferring, but not eliminating, the low-level detail. Examples of such abstractions include the use of Register Transfer Language (RTL) descriptions (e.g.

[5] I.e. *near* optimal.

[6] Knowledge, as we normally perceive it, is surely *not* a discrete phenomenon. It is often convenient, however, to attempt to quantize knowledge into convenient parcels, and label these "ideas", "rules-of-thumb", "heuristics", etc.

[Mano 82, pp. 101 – 102]), Schematic Diagrams (at several levels), and Lambda-based Design Rules. These abstractions, while not always directly allowing the automation of circuit design, can often contribute to more efficient design automation by hiding unnecessary detail from computers as well as human designers.

Several abstractions commonly used by designers, have contributed directly or indirectly to the automation of various stages of the layout process. Prominent among these is *symbolic layout.*

The following is a discussion of the form and impact of two such abstractions, namely, symbolic layout and lambda-rules

2.2.1 Symbolic Layout

One of the features of large scale integrated circuit fabrication is the concept of *design rules.* These are rules which define the minimum tolerances permitted on a die for the separation between various types of features, usually stated in microns (10^{-6}m). Violating any of these rules greatly reduces the likelihood of producing a working circuit. These rules depend on such factors as the photo-lithography used in making masks, the accuracy of mask alignment, material used to fabricate the circuit, and deposition temperature and method.

Consequently, design rules are not only complex and numerous, but vary widely from technology to technology, and from process to process. This presents considerable difficulty in circuit design, especially in terms of training new design personnel. The complexity and dynamic nature of these rules often require several years to master. The fluidity of design rules has historically had an adverse impact on the lifetime of circuit libraries and lengthened the design cycle by occasionally requiring large layouts to be modified or redone because of relatively minor process changes.

One of the earliest major abstractions for the simplification of circuit layout is a technique commonly referred to as *symbolic layout.* The specific technique, known

as STICKS layout, is generally credited to Williams [Williams 78]. Prior to the introduction of STICKS, the final stage of IC design, translating circuit schematics to mask geometries, was always done by hand. The primary objective of this stage in the design is to place as many components (transistors, resistors, capacitors, vias and wires) as possible, in as small a space as possible[7]. This is a difficult objective to meet, due to the upper bound on die size imposed by yield considerations, and because of the lower bound on component size imposed by limitations of the fabrication process.

Further, the number and complexity of design rules posed considerable difficulty for designers. They were required to place all circuit features as close together as possible without violating any design rules, however, when a rule violation was detected, or modifications to the layout required the placement of additional components, many thousands of rectangles may have to be shifted to accommodate the changes. Because of the sheer number of rules involved, design rule violations were invariably found in layouts, often subsequent to the first fabrication. The changes required to fix these errors were worse than just tedious, as they often required large (and sometimes *global*) changes, and designers often found themselves introducing new errors while fixing the old ones.

In an effort to relieve designers of the (needless) tedium of manual layout, the STICKS system introduced a high-level notation for the description of circuit topology[8] and a compiler to translate a topological diagram (or STICKS diagram) into compact physical layout. The STICKS diagram further assisted designers by replacing complex mask-level representations of transistors, vias, and other objects with simple, technology-independent symbols. These symbols could later be translated to

[7]Significant advantages of speed and reliability are realised by placing more devices in a single package.

[8]I.e. a description of component locations *relative* to the locations of neighbouring components.

the corresponding mask-level representation of overlapping rectangles of various materials, depending upon the target technology, and the context in which the symbol occurs [9].

The compiler operates by first translating the diagram into a set of *constraints*, a series of inequalities which concisely represent the topology presented in the STICKS diagram. These inequalities, which express relative placement of objects by constraining one object to be placed above or to the left of another. By minimally satisfying all of these equalities, making allowance for the restrictions of the design rules, it is possible to produce an optimal layout of the circuit [10].

At the time the STICKS compiler was constructed, it was considered too difficult to optimally satisfy both horizontal and vertical inequalities simultaneously [11]. Instead, a simpler 1-dimensional compaction [12] technique was devised, which could be used to satisfy constraints first on one axis (horizontal or vertical) and then on the other. Because this approach often produces non-optimal results, there was provision for human interaction with the compiler, so that designers could guide it toward better solutions.

With layout done in this manner, designers no longer had to concern themselves with the tedia of placing components within the limits of the design rules, and modifications were easily made, by changing the STICKS diagram and recompiling it.

---

[9] For example, a considerable variety of vias exist for a given technology, and the specific type required is easily deduced from the types of wires it connects.

[10] This is one definition of *optimal* layout. Other definitions also include minimising parameters such as total wire length, and crosstalk, which cannot be represented as inequalities.

[11] The problem of optimal 2-dimensional compaction has since been shown to be NP-complete [Sastry 82] [Schlag 83]. It is still considered too difficult to be solved optimally.

[12] The time complexity of 1-dimensional compaction has been shown [Ullman 83] to be a polynomial function of the number of objects in the circuit.

2.2.2 Lambda Rules

As described previously in Section 2.2.1, most LSI fabrication processes are characterised by a large set of complex design rules, the application of which poses a major task to designers and design automation tools alike. In 1980, Mead and Conway [Mead 80] helped to alleviate this situation by introducing parameterised design rules, based on a quantity λ defined as one half the narrowest wire permitted by the process. After making a survey of a large number of current NMOS processes, spanning a variety of scales, Mead and Conway produced a relatively small set of generic design rules which were valid across a wide variety of fabrication processes, and which could be reasonably expected to scale up or down. The use of λ-based design rules not only reduced the number of design rules required by a process, but also made designs based on these rules more "portable" and less sensitive to small changes in process technologies.

The use of λ-rules eventually allowed many designers to realise advantages in reduced time for design-rule verification, faster (and often more reliable) development of integrated circuits, and less stringent training requirements for novice designers. In addition, because many designs now became *scalable*, designers were allowed to benefit from advances in photo-lithography and fabrication technologies with little or no modification to their layouts.

2.3 Weak Heuristics

The class of weak heuristics, or weak methods, is often overlooked in a treatment of the subject of heuristics. Frequently, however, some of our most powerful deterministic algorithms result from adaptations of weak methods. Many weak methods, for example, resolution [Robinson 65] are provably complete; that is, it can be proven that if a solution for a problem exists, the weak method *will* find it in finite time.

Unfortunately, while finite time solutions are surely better than infinite time solutions, this is often an empty promise, since, for example, theorists would normally classify even $10^{100}!$ seconds as a finite period of time.

Two classical heuristics for algorithm development are "iterative improvement" and "divide-and-conquer"[Aho 74, pp. 60 – 64]. It is these two, in fact, which give rise to the majority of simple strong heuristics for VLSI design automation.

2.3.1 Divide-And-Conquer

Divide-and-conquer, in the context of computer science, is a general purpose problem solving technique based upon the following principle:

> If a problem is too large to be solved simply, try subdividing it into two or more smaller problems (often smaller instances of the same problem), and then solve these. Afterward, recombine the results for the smaller problems to get the solution to the larger one.

This technique, while known to be very effective, only works well if the sub-problems are not strongly inter-dependent and the solutions to the sub-problems can be re-combined into a solution for the original problem with little enough trouble that the benefits of the sub-division are not lost. The quicksort algorithm, credited to C.A.R. Hoare and described in [Aho 74, pp. 92 – 97] is a classical example of the successful (and recursive) application of divide-and-conquer to the problem of sorting.

Divide-and-conquer has also been applied to VLSI design automation with fair success: the class of "min-cut" algorithms [Breuer 77] are a clear example of the application of divide and conquer to routing. In fact, the place-route-compact paradigm is an application of divide-and-conquer to VLSI layout, but not a successful one. It is commonly accepted that this decomposition of layout into the *separate* steps of

placement, routing and compaction is does *not* lead to optimal design when the results are re-combined.

2.3.2 Iterative Refinement

Iterative refinement is another general purpose problem solving technique, intended for problems involving some form of optimisation: a shortest path; a best partition; or a least cost spanning tree. The technique operates by first arbitrarily selecting a "viable" [13] solution for the problem, and then repeatedly making a *small* perturbation to the current solution (in such a way that the result is also a viable solution), until a "best" [14] solution is found. In order to do this, there must exist some cost function (qualitative or quantitative) which can be used to compare the relative virtues of two viable solutions. Further, the technique also requires that viable solutions be relatively easy to find.

Normally, with iterative refinement, a "greedy" approach is used. After making a perturbation to the current solution, the perturbed solution is accepted (i.e. replaces the original solution) only if the cost function indicates that it is the better of the two. Iteration terminates after no further improvements can be made. This greedy technique is not complete: if an optimum value exists, it will not necessarily be found.

Figure 2.1 describes the general form of the iterative improvement heuristic. It, like the other figures of this chapter, uses a form of "Pseudo Pascal", that is, a language with which we can describe algorithmic semantics without becoming involved in the strict syntax of conventional programming languages.

---

[13] For example, if we are trying to find the shortest path between two vertices in a graph, a viable solution is *any* path between these two vertices. The optimal solution is the best existing viable solution.

[14] I.e., until no more improvements can be made.

```
proc Iterative-Refinement ();
begin
   let State = Generate-Initial-State();
   while not( Completion-Criteria-Met() ) do
      let New-State = Modify( State );
      if Accept( State, New-State )
         then let State = New-State;
   end {While loop};
end {Iterative-Refinement};
```

Figure 2.1: General form of the iterative refinement heuristic.

2.3.2.1 Adaptive Heuristics

Adaptive heuristics [Nahar 86] are a class of general problem solving techniques characterised by the use of iterative refinement of an initial solution, and the dynamic modification of various parameters used to control the generation and acceptance of proposed improvements. The general form of the adaptive heuristic is presented in Figure 2.2.

```
proc Adaptive ();
begin
   let State = Generate-Initial-State();
   while not( Completion-Criteria-Met() ) do
      while not( Inner-Loop-Criteria-Met() ) do
         let New-State = Modify( State );
         if Accept( State, New-State, state-variables )
            then let State = New-State;
      end {Inner while loop};
      Update( Accept, Modify, state-variables );
   end {Outer while loop};
end {Adaptive Heuristic};
```

Figure 2.2: General Form of Adaptive Heuristic

A brief comparison of Figures 2.1 and 2.2 will reveal that the iterative refinement technique is a (degenerate) special case of the class of adaptive heuristics. The class of adaptive heuristics is a very general one, encompassing almost every iterative

improvement strategy currently in use, although allowing a great deal more leeway for adaptation than most actually employ. Interesting sub-classes of adaptive heuristics include Simulated Annealing and many knowledge-based expert systems[15], each of which has found considerable application in the domain of design automation.

2.3.2.2 Simulated Annealing

Simulated Annealing [Kirkpatric 83] is a general purpose stochastic optimisation technique adapted from the Monte Carlo method for determining the minimum energy state of a conglomeration of atoms, originally introduced by Metropolis *et al* in 1953 [Metropolis 53]. Simulated Annealing is a special case of the *adaptive heuristics* described by Nahar, Sahni and Shragowitz [Nahar 86].

The Metropolis technique was proposed as a method of determining the equilibrium state of a relatively small number of particles ($\approx 10^2$ to 10^3) with declining temperature. Kirkpatrick *et al* suggested an analogy between this type of physical process and the process of combinatorial optimisation with iterative refinement. In essence, they likened the cost function used to determine the "quality" of a configuration in the combinatorial optimisation to the energy function in the Metropolis simulation. By adding the concept of *temperature* to the optimisation, using it as a time-dependent parameter controlling the likelihood of increasing the cost or "energy" of the system, they completed the analogy.

A primary requirement for the application of Simulated Annealing to combinatorial optimisation is some sort of objective function which can be used to quantitatively describe the quality of a given solution. This function, often referred to as a *cost function* is assumed, or at least *hoped* to be linearly monotonic in the quality of

[15]Specifically, those which employ a form of iterative refinement

```
proc Simulated-Annealing ();
begin
    let State = Generate-Initial-State();
    let T = Initial-value(temperature);
    let iterations = Initial-value(iterations);
    while not( Allotted-time-exhausted() ) do
        let count = 0;
        while ( count <= iterations ) do
            let New-State = Modify( State );
            if Accept( State, New-State, T ) then
                begin
                    let State = New-State;
                    let count = count + 1;
                end;
        end {Inner while loop};
        Update( T, iterations );
    end {Outer while loop};
end {Simulated Annealing};

proc Accept ( Old-State, New-State, T );
begin
    if ( Cost(New-State) < Cost(Old-State) )
        then return(true);
        else if ( random(0, 1.0] < e^(Cost(Old-State)-Cost(New-State))/T) )
            then return(true);
            else return(false);
end {Acceptance Procedure}

proc Update (T, iterations);
begin
    let T = T * (1 - ε); {typically, 0.01 ≤ ε ≤ 0.2}
    let iterations = f( T, iterations );
end {Update Procedure};
```

Figure 2.3: General Form of the Simulated Annealing Heuristic

the solution. Frequently it is the case that such a function exists when a problem is attacked with any iterative refinement technique[16].

Simulated Annealing has been used in a number of applications to VLSI design automation. Perhaps the most "natural" of these applications is to the problem of two dimensional *layout compaction*, since two dimensional compaction is not entirely unlike the problem for which the Metropolis method was originally developed. Systems which apply Simulated Annealing to compaction include TimberWolf [Sechen 86] and Liu's compaction algorithm [Liu 86].

Simulated Annealing has also been used with considerable success in applications to cell placement [Otten 84] [Gay 85] and global routing [Vecchi 83].The technique is characterised by high quality solutions, guaranteed convergence [Mitra 85], and, unfortunately, very large run-times.

One of the primary advantages of the Simulated Annealing technique is its independence from the problem domain; the technique neither makes nor requires any particular assumptions about the problem to which it is applied, except that the problem *can* be solved by an iterative refinement approach.

2.4 Strong Heuristics For IC Design

As stated in Section 2.1, the complexity of design automation problems has steadily forced researchers to explore heuristics for placement, routing, compaction, etc. Not surprisingly, the majority of of this work has involved the development of *strong* heuristics, those tailored specifically for the problem to which they are applied. Consequently, the number of such heuristic procedures is far too large to allow more than

[16]This is not always the case: all that is really required for iterative refinement is a means of judging the *relative* quality of two possible solutions.

a very small portion to be reviewed here. For a more complete review, readers are referred to [Preas 86].

2.4.1 Simple Strong Heuristics

The classical approach to design automation has typically involved the development of *simple* strong heuristics for use as algorithms for placement and routing. Using the classification of Preas and Karger [Preas 86], these simple strong heuristics may be grouped into two basic classes: constructive heuristics and iterative improvement heuristics.

Heuristics of the first class operate on the principle of incrementally building a circuit one component at a time, fixing the location of each component or wire "permanently" [17]. After all wires and components are placed, no attempt is made to improve the circuit by moving them. The necessity of backtracking to produce good designs leads to a significant cost in terms of execution time, and sometimes leads to infinite regression.

Heuristics of the second class operate on the principle of beginning with a correct, but inefficient, constructive solution, and then try to improve the solution by repeatedly making small localised changes. Several general approaches to deciding which local changes to accept and which to reject are described in Section 2.3.2.1.

The following is a brief survey of several classes of constructive and iterative heuristics for VLSI placement and floorplanning, following the classifications of [Preas 86]. For a more complete survey of this topic, refer also to [Soukup 81] and [Hartoog 86].

[17] In fact, backtracking often occurs. If the placement of one or more components or wires make impossible the addition of a subsequent component or wire, the offending objects are often removed from the circuit to be re-located later.

2.4.1.1 Constructive Heuristics

Of the two major classes of simple strong heuristics, the class of constructive heuristics is the less common. To a large extent, this is because it is more difficult to design an algorithm which can perform well in constructively designing something than it is to design an algorithm that can arbitrarily perturb a design and judge whether the result is an improvement.

Cluster Growth: This is a technique which makes use of connectivity information to sort circuit components into groups that are very strongly connected to one another, and, therefore, closely interrelated logically. A usual method for achieving this is to arbitrarily select one component as a "seed", and then repeatedly select the remaining (unselected) component that is most strongly connected to the cells of the cluster under construction, and add it to the cluster. When there exist no more unselected components whose connectivity is above a given threshold, the cluster is considered complete, and a new one is started.

Ideally, clusters should contain roughly three to eight components: if one grows too large, it may be necessary to break it into sub-clusters by repeating the process above with a higher connectivity threshold. After the clustering process is completed, the circuit can be floorplanned with relative facility, as the size and connectivity of clusters provide considerable clues for reasonable topological arrangements. Because circuits are often specified hierarchically, it is often possible to perform clustering independently of each level of the hierarchy with very good results.

Examples of clustering algorithms are found in [Nixon 84], [Hanan 76], and [Schuler 72].

Partitioning: This is a global placement technique, whose objective is to produce layouts with minimal wiring congestion. This objective is achieved by recursively subdividing the components of a circuit about alternately horizontal and vertical "cut-lines" in such a way that the number of signals crossing the cut-line is minimal. Many algorithms which use this approach suffer from inability to cope with fixed-position components or terminals or over-simplification in the method for determining the number of signals crossing the cut.

Examples of such algorithms are seen in [Breuer 77], [Dunlop 85], and [Corrigan 79].

2.4.1.2 Iterative Improvement Heuristics

As described in Section 2.3, iterative improvement heuristics operate on the principle of beginning with an arbitrary initial design and progressively refining the design by making small localised changes. The difference between weak and strong iterative improvement heuristics lies in the use of domain specific knowledge for generating or accepting proposed improvements. In each case, a comparison procedure, which can judge the relative merits of two (or more) potential solutions, must be provided. This comparison procedure, which may be either qualitative or quantitative, is also often based on domain-specific knowledge. The following is a brief description of several strong iterative improvement heuristics for VLSI floorplanning:

Pairwise Exchange: This is one of the simplest forms of iterative improvement heuristics for VLSI floorplanning. This technique normally performs by selecting a candidate component (cell or transistor) and trying to exchange its position with that of each other component in the circuit. The configuration which is judged best by the comparison procedure is accepted as the result of the iteration. The next iteration proceeds with a different candidate component, until no further improvements are possible.

A generalisation of the pairwise exchange technique is seen in [Goto 79] in which n-tuples (i.e. ordered sets of n items) of components are exchanged in a cyclic fashion; the second taking the place of the first, the third taking the place of the second, and so forth, with the first finally taking the place of the last. Experimental evidence presented in [Goto 79] indicates that cyclic exchange with 4-tuples is most beneficial.

Force Directed Exchange: This is a specialisation of Pairwise Exchange, in which the comparison procedure and cost function employ a heuristic which is analogous to stretching springs between components which are to be connected to one another. The objective of the procedure is to find a configuration in which the sum of all forces exerted by the springs is at a minimum; thus (ideally) producing a good compromise between minimum wire length and minimum area considerations. This is done by considering for interchange only those pairs of components that can be moved in the directions of their net forces.

The primary advantage of this technique is that candidates for exchange can be selected according to the net forces acting on each component, hopefully improving the performance of the algorithm by reducing the number of rejected moves made by the arbitrary pairwise exchange procedure described above. Examples of Force Directed Exchange may be seen in [Sharman 86].

Force Directed Relaxation: This is a technique related to Force Directed Exchange, in which a component is selected as the candidate for relocation, and is moved to the site nearest its equilibrium point (the point at which the net force acting on the component is zero). If another component is displaced from that site (only one component may occupy any given site at one time), it then becomes the candidate component, and the procedure continues until a candidate component is placed in an empty site.

The resulting configuration is then compared to that prior to the sequence of changes, and is accepted only if a net improvement is found. If a net improvement is not found, the entire sequence of moves is rejected.

2.4.2 Complex Strong Heuristics

The distinction between *simple* and *complex* strong heuristics is not especially clear. At what point can it be said that a heuristic procedure employs enough domain specific knowledge to be deemed complex? Perhaps the cut-off could be set at twenty "pieces" of knowledge, or perhaps fifty; such a definition would be clearly pointless, as these "pieces" of knowledge are, at best, very difficult to enumerate.

Therefore, rather than relying on the *quantity* of knowledge used to make this distinction, it is more sensible to pay attention *how the knowledge is used.* The distinction between simple and complex heuristics can then be stated: a simple heuristic employs a fixed and usually *small* quantity of domain specific knowledge; a complex heuristic attempts to provide a framework for describing and applying an arbitrary amount of knowledge, and to extend that knowledge as necessary.

The study and development of techniques for and applications of such complex heuristics has been the subject of much recent Artificial Intelligence (AI) research. It is certainly not the case that all studies under the auspices of AI research fit the definition of complex strong heuristics given above. There is, however, one segment of this research, commonly known as knowledge-based expert systems, which certainly does fit this definition. Prior to discussing the application of knowledge-based expert systems to VLSI design automation (Section 2.6), the history, philosophy, and theory of knowledge-based expert systems will first be elucidated.

2.5 Knowledge-Based Expert Systems

Knowledge-based expert systems are a subset of Artificial Intelligence research which have recently received considerable attention. The principle of the knowledge-based expert system is to cause a computer to mimic the problem solving behaviour of human experts by observing the human expert in action, determining the basic problem-solving steps taken, and encoding these steps in a computer program. This approach to problem solving is normally reserved for very difficult or ill-structured problems which cannot be automated with conventional programming techniques either because there is no known (efficient) algorithm for the problem, or the objectives of the problem are not adequately well-defined to be concisely encoded.

This technique was initially received by some as the solution to all "hard" problems; if a problem could not be solved with conventional techniques, then surely, it was felt, it could be solved by producing an expert system with enough knowledge about the problem domain. Common sense and past experience, however, reveal that this is not the case. Expert systems are only suitable for those problems which human experts are able to solve more quickly or effectively than conventional computer programs. Further, even assuming that the knowledge and behaviour of human experts can be *perfectly* captured and encoded, an expert system can hope, at best, only to match, never to exceed, the peak performance of a human expert or committee of experts. Bleak reality indicates that even this level of achievement is improbable.

Following is a description of the basic structure of knowledge-based expert systems.

2.5.1 Components of a Rule Based Expert System

As initially defined, rule-based expert systems were normally viewed as having two major components: working memory and an "inference engine". Working memory

was used to store both rules and state information, making self-modifying rules a simple matter to implement.

Today, rule-based expert systems are typically composed of three major components: rule memory, working memory, and an "inference engine". The division of rule memory from working memory is largely a conceptual one: it is usually the case that rule memory can still be modified (although often in a constrained manner), but designers may find it convenient to use significantly different underlying representations for rules and data.

These various components once had to be implemented specifically for each individual expert system. Modern expert system environments, often referred to as "expert system shells", now provide the basic components in a ready-made package, and often include several other features which may be useful to expert systems implementors. Although often a great convenience, expert system shells may occasionally be more general than a given application requires, and thus incur an unnecessary overhead. Consequently, expert systems shells, and sometimes expert systems themselves, are often regarded by implementors as *prototyping tools.*

The following sections describe each of these three components individually.

2.5.1.1 Rule Memory

The rule memory of an expert system contains a set of *heuristics* designed to solve a particular class of problems, encoded as production rules. These production rules, derived from Post's "productions" [Post 43], are divided into left-hand side and right-hand side. When viewed as a rules of inference, the objects on one side of the rule are logical antecedents, and the objects on the other side are conclusions to be drawn. When viewed as heuristics, the objects on one side are conditions, and the rules on the other are actions to perform when the conditions are met.

The classical approach to expert systems design makes use of "backward-chaining" [18] (or simply "backward") rules, in which the objects on the left-hand side of the rule are a number of conditions or "goals" that are set as objectives and the objects on the right-hand side of the rule are the conditions under which the goal may be deemed to be met. This form of production rule (which corresponds to the use of Backus-Naur Form (BNF) productions to *recognise* a language) is indeed a very powerful formalism for logical inference, and forms the basis of languages such as PROLOG [Clocksin 84].

Many languages, such as ART [ART87, pp. 14 – 21], also allow a slight variation of the above type of rule, whose left-hand side consists of one or more "conditions", and whose right-hand side consists of actions to perform or conclusions to draw when the conditions on the left-hand side are satisfied. This sort of rule is often called a "forward-chaining" or simply "forward" rule, because inference proceeds from the assumptions and axioms toward the desired goal, much in the way mathematicians formulate proofs. Forward-chaining rules, which are the dual of BNF productions used to generate strings of a language, are often useful for tasks which require the exploration of a problem space, such as design oriented problems.

It is frequently the case that both types of inference, forward- and backward-chaining, can be profitably brought to bear on diverse problems, and they are now frequently intermingled. Each, however, has its caveats: forward rules are simple and flexible programming structures, but subject to problems with control flow; backward rules are elegant, but many programmers find them difficult to write and understand.

[18] The term backward-chaining refers to the mode of inference used: starting from a desired conclusion, and working backward to whatever antecedents and axioms may have been provided.

2.5.1.2 Working Memory

The working memory in an expert system is used to store data which may regarded as a candidate for matching against patterns in rules. Typically, this is information which describes the (instance of) the problem to be solved, and the state of the solution. This part of memory is usually much more dynamic than the rule memory.

The objects stored in the working memory are called "facts". The term "fact" should not be interpreted to imply anything about the *truthfulness* of an assertion, however, the presence of a piece of information in the working memory normally indicates that at least some part of the expert system has cause to believe this fact is true.

2.5.1.3 Inference Engine

The inference engine is the underlying control mechanism for the expert system; it is used to determine which rules are potential candidates for application, and when to apply them. It is also used to control access to the working memory. At the heart of the inference engine is a pattern matching algorithm, used to determine which preconditions for the various rules are currently met. A very typical pattern matching algorithm is the Rete algorithm [Forgy 82], used in both OPS5 [Forgy 81] and ART [ART87].

The inference engine maintains a list of all rules whose patterns match one or more facts in the working memory, called the "agenda". In many languages, such as ART, the elements in this agenda are prioritised. The execution cycle of the inference engine is as follows: first, update the agenda, adding or removing rules as required; next, select a (high priority) rule; finally, execute the actions contained in that rule, and repeat the cycle.

All rules and facts in an expert system are global entities; all rules are assumed to be applicable whenever their conditions are met, and the assertion or retraction of a fact in working memory can potentially affect any or all rules.

2.6 Applications of Expert Systems to VLSI Design Automation

Knowledge-based expert systems techniques have been applied to problems in VLSI design automation in a large variety of contexts, usually with a relatively high degree of success. Much of the work to date has concentrated on some of the "lower" level tasks of circuit layout, such as placement routing, and compaction. Of course, it has already been shown that these "low" level tasks are exceedingly difficult. Expert systems have also been applied to a few much higher level problems, such as automatically producing circuit specifications (eg. in RTL) given a procedural description of the function the circuit is to perform.

The following sections provide a brief review of some of the applications of knowledge-based expert systems to various aspects of VLSI design automation:

2.6.1 STICKS' Knowledge-Based Design Rule Checker

As discussed previously in Section 2.2.1, one of the early advances in VLSI design automation was the development of the STICKS [Williams 78] system, which provided a compiler for translating a topological description of a circuit to patterns suitable for mask generation. While the translation process itself is relatively straight-forward, the sub-task of testing for design rule violations is less simple than it appears.

The complexity of a typical set of design rules and the number of exceptional cases involved made necessary the encapsulation of design rules and their interpretation as a knowledge-based production system.

2.6.2 Expert Systems for Translating Connectivity to Cell Layout

One of the more common applications of knowledge-based expert systems techniques to VLSI design automation is with the problem of translating transistor-connectivity lists to topologies suitable either for direct translation to fabrication masks, or for automated layout by such systems as STICKS [Williams 78] and MULGA [Weste 81]. Aside from actually drawing the masks (which is now done by STICKS-like systems), the description of leaf cell topologies is the lowest level of the VLSI design process. This is, however, in no respect a trivial task, as the two primary components of this task, placing the transistors, and interconnecting (routing) them, are both known to be NP-complete problems [Sahni 80].

The input for this problem consists of a set of ports (the sites at which signals enter and exit the circuit), and a list of interconnections to be made between the transistors, and between transistors and ports. Variations of this problem may require some or all ports to be "pre-placed" (i.e. have pre-assigned positions relative to one another), or may allow the inclusion of a set of constraints upon the relative locations assigned to ports.

The output should consist the description of a complete and compact circuit in which all required transistors have been assigned a position (placed), and which contains a full and error-free description of all the wires needed to provide the required interconnections. Other objectives for the output may include: minimum area; minimum wire length; maximum speed; "optimal" aspect-ratio; conformity with a fixed cell pitch; observance of constraints on wire or transistor placement.

Because of the immense difficulty of producing a complete layout given this amount of information, implementations are normally restricted to laying out "leaf cells"; thus, there is an upper limit of 50 – 100 transistors placed on circuits to be laid out. Naturally, it is not always possible for all (or any two) of these objectives

to be met simultaneously, and the attempt to optimally satisfy even one will prove exceedingly time-consuming. Hence, it is required that a reasonable *compromise* be struck between these conflicting goals.

One application of knowledge-based expert systems to this problem is seen in the TOPOLOGIZER system [Kollaritsc 85]. TOPOLOGIZER is intended to produce design-rule independent symbolic designs (in CMOS) suitable for "compilation" by the MULGA [Weste 81] symbolic layout system. The use of symbolic layout as the output representation has several potential advantages over mask-level layout.

First, at the symbolic layout stage, the only design errors of consequence are illegal wire crossings or overlaps thus greatly reducing the the amount of work to be done with each modification to the circuit. Second, by appropriately handling such parameters as the number and type of routing layers available, the layout system becomes largely technology-independent. This means that a large proportion of the leaf-cells given to the layout system as input can be re-compiled into different fabrication technologies with relatively little difficulty.

TALIB [Kim 85] is a second example of such an application. TALIB produces mask-level layouts of leaf cells in NMOS. It first accepts a hierarchical circuit specification which may include numerous "standard" sub-circuits such as and-gates, or-gates, inverters, or latches. The hierarchical description is then modified as necessary to minimise the number of transistors required. Finally, the hierarchy is flattened; each sub-circuit is replaced by a corresponding set of transistors and interconnections. The circuit is then embedded in a plane using symbolic layout (the use of symbolic layout was found necessary to keep the number of rules at a manageable level), which is subsequently compacted to a single-metal NMOS layout.

A third such system is SPLAT [Sharman 86]. SPLAT is philosophically very similar to TOPOLOGIZER, producing symbolic layout of CMOS leaf cells, however, it is considerably more limited in scope.

2.6.3 Expert Systems for VLSI Routing

Routing, the process of assigning wires or paths to the interconnection networks between components, is a critical portion of describing VLSI circuits at low levels of abstraction. These interconnections often must be made within a number of very restrictive constraints and with a large number of objectives. Constraints on routing include the number of available routing layers (older technologies provided only two, namely metal and polysilicon; modern technologies may now permit four or more), restricted orientation of wires on various layers, placement of terminals, and size and location of wiring areas.

Objectives for routing include 100% completion (i.e. making *all* required connections), minimising vias (connections between wires), minimising wire length, making maximum use of lowest-impedance layers, minimising congestion (i.e. balancing the concentration of wires across the circuit), minimising cross-talk, and minimising area. Frequently, because of the difficulty of finishing partial routes by hand, 100% completion is the only objective fully considered.

Traditionally, the routing problem is sub-divided into three smaller problems: channel assignment, in which the routing area is divided into rectangular segments, called channels; loose routing, in which it is decided which channels each signal will cross; and detailed routing, in which the physical wires for each signal are placed in each channel. Because the job is known to be NP-complete [Sahni 80], and because simple strong heuristic solutions have customarily met with limited success, routing is often considered a suitable task for solution by knowledge-based expert systems.

WEAVER [Joobbani 86a] [Joobbani 86b] is one of the earliest applications of expert systems technology to VLSI routing. WEAVER focuses on detailed routing, the final stage of the routing process. It considers, simultaneously, all important routing metrics, and is shown experimentally to produce more area-efficient routings than routers which consider only a single metric. WEAVER applies a large number (approximately 700) of rules the the problem of routing switchboxes, rectangular areas with fixed signal terminals on all four sides. It is also claimed that it is possible to extend WEAVER to route 'T' and '+' shaped areas in addition to rectangular areas [19].

Another routing expert system which adopts an approach similar to WEAVER's is the B&D router [Keefe 87, Keefe 86]. WEAVER and B&D both contain multiple cooperating "experts", groups of rules which focus on specific subproblems or objectives of the overall system. These experts communicate information to one another using a "blackboard", a publicly shared area of the data base where one expert can post messages for another expert to read. These two expert systems differ primarily in that WEAVER also includes a supervisory "expert" which coordinates the actions of the others, while B&D's experts operate asynchronously. The use of multiple cooperating experts in an expert system is becoming popular.

2.7 Chapter Summary

This chapter has presented an overview of various parts of the VLSI design process, and has described a number of the problems involved therein. Specifically, the majority of tasks involved in VLSI design automation are known to be NP-complete and therefore may require inordinate amounts of processing time to solve optimally. Consequently, researchers trying to automate various stages of the VLSI design process

[19] Areas whose perimeter is not rectangular are often not permitted by routing tools.

are forced to resort to the use of *heuristic procedures*, techniques which encapsulate some form of knowledge or assumption about the task at hand, in an attempt to provide good quality approximations with relatively little computation.

The majority of the work done to date has been concentrated on the development of *weak* heuristics and *simple strong* heuristics. Unfortunately, much of this work has lead to disappointment: the computer programs produced suffered from a tendency to produce either mediocre results in a reasonable amount of time, or relatively good results at the cost of hours or days of computing time [Greene 86]. Often, it appears that weak heuristics are simply *too* general to operate quickly, and simple strong heuristics are too restrictive.

Recent research into *complex strong* heuristics, specifically knowledge-based expert systems, for design automation has been presented as being quite promising, however at least two problems have been identified with this paradigm: First, expert systems do not map nicely onto design-oriented problems; we are usually forced to impose sequential behaviour on portions of the expert system, and this leads to difficulties in design and maintenance. Second, the paradigm most commonly used to map expert systems onto design automation problems is a *greedy* form of iterative refinement; this results in a tendency to arrest[20] at local minima.

The remainder of this thesis addresses these two problems. Chapter 3 describes an extension to the ART expert system tool-kit which automatically generates the portions of code needed to sequence operations in a design-oriented expert system. By producing this portion of the code automatically, programmers are permitted to completely ignore the details of sequencing, allowing them to return to a state where expert systems are viewed as sets of daemons.

[20] I.e., to become "stuck" or terminate prematurely.

Chapter 4 investigates one alternative to greedy iterative refinement: a modified version of Simulated Annealing. By regarding Simulated Annealing as just another member of the same class of weak methods that contains iterative refinement, a new type of expert system which is less prone to arrest at local minima is developed.

Chapter 3

The Expert System Wrapper Environment

Chapters 1 and 2 of this thesis have described a number of important issues pertaining to VLSI design automation. They showed that many interesting and worthwhile problems in design automation are known to be computationally intractable (i.e., NP-complete), and that trying to optimally solve non-trivial instances of these problems may greatly exceed the life expectancy of the individual seeking the solution. Consequently, we are forced to adopt techniques which may frequently produce less than optimal results, but will hopefully produce a close approximation to an optimal result in a reasonable time interval.

To date, the only known means of doing this has been through the use of *heuristics*; the application of some kind of knowledge about a problem in the hope of finding a "short-cut" to a good solution. Evidence for the existence of useful and efficient heuristics is provided daily by people who are able to find acceptably good solutions for a wide variety of problems with relatively little time and effort. Chapter 2 describes several classes of heuristic procedures for design automation, perhaps the most compelling of which is the knowledge-based expert system which seeks to solve problems in with the same knowledge used by people.

Empirical studies [Nahar 86, Hartoog 86] have shown that *strong* heuristics outperform *weak* heuristics; that is, the indications of these studies are that problems are often more effectively solved when using knowledge specific to the problem domain than when using very general purpose knowledge. In fact this should not be surprising, as strong heuristics, unlike the weak, are built upon knowledge *specifically* chosen for the problem they intend to solve.

There is considerable reason to believe that increasing the amount of domain knowledge in a heuristic procedure can greatly augment its problem solving power. The introduction of knowledge based expert systems, which can apply hundreds or thousands of domain specific rules to solve a problem dramatically demonstrated this principle.

Another lesson learned in Chapter 2 is that expert systems, while promising great problem-solving potential in a variety of domains, were originally intended for, and are largely limited to, problems of a diagnostic nature. Implementors have found that expert systems techniques do not always apply well to problems which exhibit implicit structure, or which may require the sequential solution of sub-problems. Many VLSI design automation problems are known to be members of this latter class.

This chapter presents the Expert System Wrapper Environment (ESWE) for describing design-oriented expert systems. Because design problems frequently involve sequential steps, expert systems implementors must enforce sequential behaviour under a paradigm in which it is not intended. Consequently, considerable difficulty may be encountered in designing, implementing, and maintaining such expert systems.

To counter this difficulty, ESWE has been written by the author with the intention of providing expert systems implementors with a high-level abstraction for sequential control of expert system sub-tasks. ESWE is able to automatically generate the control-oriented code fragments needed in each rule of the expert system, according to the high-level specifications provided. This benefits implementors in two ways: first it hides control information encoded in the rules of the expert system, making the encoded *knowledge* more clear; and second, it allows the implementor to modify the sequence of operations (often a necessary step in debugging) *without risk of accidentally altering any of the knowledge encoded in the rules.*

Also included in ESWE is a means of describing fabrication technologies, design rules, and layout primitives; and a constraint-based language for describing the relative placement of ports on the perimeter and in the interior of cells using a syntax similar to that of SHIFT [Liblong 84]. ESWE has been used by the author to implement two simple expert systems for transistor placement, both of which are based upon Sharman's SPLAT [Sharman 86]. These expert systems, which are used to investigate the value of incorporating randomness in a strong heuristic procedure, are the subject of Chapter 4.

ESWE is written in Symbolics Common Lisp [SCL86a, SCL86b, Steele Jr. 84] and runs on Symbolics 3600 computers under Genera 7.1, as an extension to Inference Inc.'s Automated Reasoning Tool (ART) [ART87]. Much of ESWE is written as common-lisp macros in such a way as to provide maximum flexibility in compilation with minimum run-time penalties. ESWE is meant to *extends* the syntax and semantics of ART, complimenting it without *modifying* it. Thus, standard ART code can be executed without modification, even in the presence of the ESWE extensions.

3.1 Objectives of ESWE

Rule-based expert system shells often provide an excellent environment for the rapid development of prototype expert system applications. They generally lack, however, a number of features that would prove useful for design-oriented expert systems. Some of the more noticeably missing features of expert system shells are sequential control mechanisms and facilities for *hierarchical* coding. It is perhaps worth noting that for "conventional" expert systems applications, these features are rarely required. In fact, from the earliest days of expert systems research, implementors were enjoined *not* to employ explicit sequencing in their code, unless it is completely necessary.

Unfortunately, sequential (or at the very least, *synchronous*) behaviour is often required to solve problems of design. When sequencing is required in rule-based systems, it can normally only be achieved by a form of "token-passing". In this approach, a rule – or group of rules – is given "permission to fire" by the assertion of some specific fact in the knowledge base. Upon completion, the currently active group must pass "permission to fire" to another group by asserting the new group's control fact, and retracting its own from the knowledge base.

Thus, "tokens" are effectively passed from one group of rules to the next, causing control to follow the flow of tokens. There are, however, several complications to this apparently simple scheme: each group of rules must expect a *unique* token to enable it, and when its actions are completed may be required to pass on any one of many different tokens depending on which set of rules is to enabled next. A strongly sequenced or synchronised rule-based program (expert system) may degenerate into a completely sequential program, totally devoid of structure.

It is usually the case that a token-passing control paradigm such as this will result in:

- cluttering the logic of a rule by the addition of several control conditions;
- increasing the burden on the programmer by forcing him to keep distinct the meaning of each specific control fact; and
- degrading the performance of the expert system by significantly increasing the number of changes to the knowledge base.[1]

The first two of these issues are addressed directly by ESWE's facility for hierarchical coding of expert systems. The third issue, however, is unavoidable, except

[1]For systems using Rete networks[Forgy 82], such as OPS5 and ART, the performance is measured (and limited) by the number of Working Memory Changes per second (WMC/s) [Gupta 87]. For OPS5 implemented in Franz Lisp on a VAX 11/780, performance is in the neighbourhood of 8 WMC/s [Gupta 87, p. 15]. Other implementations of OPS interpreters are stated to achieve speeds of 40 to 200 WMC/s.

by implementing a new inference engine which specifically recognises *contexts* for pattern matching and rule firing. In fact it may be possible to realise significant performance improvements by restricting pattern matching activities to only those patterns required by rules in "active" contexts. ESWE *does*, however, help to alleviate the degradation of performance by encouraging the use of fewer control facts, and reducing, even eliminating, the use of rules which act strictly for purposes of control.

The primary feature of the ESWE environment is the **defautomaton** macro, which allows the declaration of general finite state automata. Also provided are macros for defining automaton states (which are actually groups of rules) and "experts". In addition, base frameworks for the description of arbitrary CMOS fabrication technologies and tools for describing leaf-cell topologies are provided for the convenience of implementors.

3.1.1 Motivation for Hierarchy in Expert Systems

A frequently used approach to developing expert systems for VLSI design automation tasks is the use of "multiple communicating experts". This paradigm, used by systems such as TOPOLOGIZER [Kollaritsc 85], WEAVER [Joobbani 86b], and B&D [Keefe 87], is used for problems with several complex and reasonably distinct sub-tasks. Each sub-task is assigned to a "domain expert", which contains heuristics applicable to that one task. An additional "expert" is included to serve as an overseer or manager.

Each domain expert, which communicates with the others and with the overseer using a mechanism commonly referred to as a "blackboard", serves to recommend changes to the design state based upon the perspective of its own domain. (For example, domains of expertise in an expert system for routing might include wire length, routability, via minimisation, etc.) The task of the overseer is to accept the recommendations of the domain experts, select one, and allow the domain expert

which made the recommendation to implement it. The overseer thus controls and synchronises the actions of the domain experts, while deciding which recommendations should be acted upon.

The overseer in such an expert system therefore has a number of actions which must be performed sequentially. First, it must request a recommendation from each domain expert. Next, it must evaluate those suggestions, and decide which to act upon. Finally, it must inform the domain experts of its decision, and and prepare to repeat the process.

The previous section described some of the difficulties involved in enforcing strict sequencing on rule-based expert systems. The greatest difficulties derive from the fact that all of the "experts" in a multiple communicating experts system reside in the same rule base, and all messages passed and control structures used to coordinate the various actions must pass through the same *globally accessible* database. Consequently, erroneous behaviour on the part of one "expert" may seriously impact the behaviour of the others. Similarly, modifying the behaviour or sequencing in one expert may have far reaching – and often unforeseen – effects on the rest of the program. ESWE was designed by the author to alleviate these difficulties by providing an environment where the control code for experts is more localised, automatically generated, and can be hidden from the programmer.

The macros provided by ESWE as an extension to the ART system are described in the following sections. These macros are all written in Symbolics Common Lisp, with complete separation of compile-time and run-time activities for minimum overhead. The CMOS design abstractions, described in Section 3.3, are implemented with ART schemata, and thus may not be as easily portable to other expert systems shells as

are the macros. *It is important to recall that* ESWE *is an experimental prototyping environment.*

3.2 ESWE's Interface for Describing Experts

In this section, we review the tools ESWE provides for the description of control structures in design-oriented expert systems. Section 3.2.3 describes ESWE's facilities for describing finite state automata (deterministic or otherwise), while section 3.2.5 describes ESWE's approach to defining experts for sub-domains of VLSI design.

In the following subsections it is important to recall that these sequencing mechanisms were originally designed for use with, and as extensions to, ART [ART87]. For this reason, a brief description of ART will be provided prior to discussing these macros.

3.2.1 A Brief Introduction to ART

ART (Automated Reasoning Tool) is an expert system language or "shell" intended to ease the task of developing knowledge-based programs. The ART system predefines the basic component of an expert system, the inference engine, removing the necessity of implementing this essential for every expert system. ART also provides mechanisms for managing the working memory and rule memory, along with tools for placing knowledge in them.

Rules in ART may be either forward-chaining or backward-chaining. Implicit to all rules in ART is the concept of *patterns*. A pattern is a specification for matching knowledge in the working memory with conditions in the rule memory.

Each rule in ART, whether forward- or backward-chaining, has a number (zero or more) of *conditions* imposed upon it. When all of the conditions in a given rule have been matched to facts in the working memory, the rule is *activated*, and placed in a

queue (called the *agenda*) to await firing. If the conditions on a rule are matched by more than one distinct set of data in the working memory, the rule will be activated several times, once for each distinct set of working memory elements it matched.

Each rule may be supplied with parameter referred to as its *salience*. Saliences may be in the range $-1,000,000$ to $1,000,000$ and the default salience for a rule is 0. *The* salience *of a rule determines the priority of its activated instances when placed on the agenda. That is, rule activations on the agenda are* sorted *according to salience, and no rule may fire if another rule of higher salience is currently on the agenda.* The primary intent of salience declarations is to express the relative importance of rules, not to control the sequence of execution.

The ART database is used to store working memory for an expert system; conceptually, it is completely distinct from the rule memory. The main purpose of the ART inference engine is to match *patterns* stored in the rule memory to *facts* found in the working memory. The act of placing a fact in the ART database is commonly referred to as assertion (i.e., one *asserts* a fact in the database); the act of removing a fact is commonly referred to as retraction (i.e., one *retracts* a fact from the database).

For a more complete description of ART, refer to Appendix A or to the ART Reference Manual [ART87].

3.2.2 Extensions to Semantics for Defrule

The ESWE environment provides several semantic extensions to the definition of rules in ART. Presently, these extensions apply to *forward-chaining* rules only, however the extension to include *backward-chaining* rules is minor[2]. Figure 3.1 provides a simplified BNF description of the syntax of declaring forward-chaining rules with

[2]The extension has not been made as yet because it was not required for the implementation of the STANLEY experts.

```
<defrule}> ::=
   (defrule <name>
      {(declare (salience <integer-expression>))}
      [<{\bf forward-rule-body}> | <{\bf backward-rule-body}>] )

<forward-rule-body> ::= {<logical-conditions>}
                                  <condition>* { => <form>*}

<backward-rule-body> ::= <goal-pattern> <= <condition>*

<condition> ::= (test <safe-form}>) |
                        <pattern>

<safe-form> ::= Any Common-Lisp expression without side-effects,
                and which always returns the same value for a
                given set of arguments.

<integer-expression> ::= Any ART or Common-Lisp expression
                         which evaluates to an integer.

<form> ::= Any valid ART or Common-Lisp expression.
```

Figure 3.1: Simplified BNF for **defrule**. See [ART87] for complete BNF.

ART's **defrule** macro. The two major extensions provided by ESWE are; i) extended *salience* declarations; and ii) a facility for declaring un-named rules.

The first of these extensions allows the use of extended ESWE salience declarations within the body of an otherwise standard ART rule definition. When defining a rule in ART, the only declaration which can be provided is that of the *salience*, or absolute importance, of the rule. Because the ESWE environment seeks to add conceptual hierarchy to the structure of the expert system, it is often more useful to express the *salience* of a rule in relative, rather than absolute terms. The new (extended) syntax of a forward-chaining rule declaration is illustrated in Figure 3.2.

Valid extended salience declarations may include any of the following:

```
<defrule> ::= (defrule <name> {<decl>}
                  [<forward-rule-body> | <backward-rule-body>] )

<decl> ::= (declare {(salience <integer-expression>)}
                    {(default-salience <integer-expression>)}
                    {(relative-salience <integer-expression>)})

<forward-rule-body> ::= {<logical-conditions>}
                        {<condition>}* => {<form>}*
<backward-rule-body> ::= <goal-pattern> <= <condition>*

<condition> ::= (test <safe-form>) | <pattern>

<safe-form> ::= Any Common-Lisp expression without side-effects,
                and which always returns the same value for a
                given set of arguments.

<integer-expression> ::= Any ART or Common-Lisp expression
                         which evaluates to an integer.
<form> ::= Any valid ART or Common-Lisp expression.
```

Figure 3.2: BNF for **defrule** with extended declarations.

default-salience: Declarations of *default-salience* override the default salience values inherited by the rule's enclosing environment. Relative-salience declarations in the rule or enclosing structures will augment the default-salience as normal.

relative-salience: Declarations of *relative-salience* are used to increase the salience of a a rule relative to the (default) salience of the enclosing structure. Thus a rule declared with a *positive* relative salience is more salient than other rules at the same level of enclosure.

salience: Declarations of *salience* state the salience of a rule in *absolute* terms. Such declarations override any default- or relative-salience declaration, as well as any inherited salience value.

The second extension to the syntax of forward-chaining rule definitions amounts to the the automatic generation of a "unique" name for a rule, relieving programmers of the need to provide names when none readily suggest themselves. (This feature is actually employed by the internal code-generating functions of ESWE.) The current implementation of ESWE generates these "unique" rule-names at *compile* time, rather than *load* time, so uniqueness is only guaranteed if the lisp system is not reset between compilation of any two program modules being used. This flaw may be corrected in future versions either by deferring the generation of "unique" rule names to *load* time, or by extending the rules of lexical scope to include symbols such as rule names, and schema names. The latter of these options is discussed in more detail in Section 3.2.3.1.

3.2.3 Defgroup and Defautomaton: Flexible Hierarchical Sequencing

This thesis has emphasised, on numerous occasions, the frequent necessity of imposing sequential behaviour on portions of an expert system. What has not been emphasised is the danger of imposing *too much* sequential behaviour which may cause an expert system to degenerate into a completely sequential program. In providing a sequencing mechanism for rule-based programs, it is necessary to be certain that the mechanism is neither rigid nor forceful: while *portions* of an expert systems may require sequential (or carefully synchronised) execution, it is an essential part of the expert system philosophy that the rules within each of these portions remain as asynchronous as possible.

This section describes ESWE's facility for defining general finite state automata. A finite state automaton (FSA) is a "conceptual" machine, which is described by the following [Brady 77] [Shields 87]:

i) a finite, non-empty set of "states";

ii) a means of receiving 'input' from its surroundings;

iii) a set of possible actions or 'outputs';

iv) a set or function describing which action(s) to perform for a given input when in a given state;

v) a means of deciding what the next state of the automaton is, given the current state.

There exist two main classes of finite automata, deterministic and non deterministic; the distinction is that for deterministic automata, the "next-state" is strictly determined by a function of only the "current" state and the "input", while for a non-deterministic automaton, the function which determines the "next" state may employ other, possibly hidden, variables.

A finite automaton operates by: inspecting the input, and performing the actions prescribed by the current state, which may include producing output; determining which state to act on next; and causing the "current state" to be replaced by the "next" state. This procedure iterates until the "current state" of the automaton is a "terminal state", at which time the automaton halts.

Finite state automata are often used to recognise strings of a language in much the same way as do the productions of a grammar. Finite automata in ESWE no more recognise strings than do the production rules of an expert system implemented on ESWE; in each case we use an operational analog of the formalism, where rather than reading symbols, patterns are matched from a data base and rather than printing symbols, side-effects are generated in the same database.

ESWE's class of finite state automata differ from classic FSAs primarily in the mechanism used to trigger state transitions: normally, a finite state automaton undergoes a state transition whenever an input is received, ESWE's finite state automata may undergo state transitions upon the addition of specific information to the data base but more usually undergo transitions when none of the rules in the currently active group are able to fire.

ESWE's fundamental sequencing mechanism, the **defautomaton** macro (see Figure 3.3), allows programmers to declare "groups" of rules (specified with enclosed **defgroup** declarations) which form the states of the automaton, to designate an initial state for the automaton and to designate one or more final states for the automaton. The finite state automata so declared may be either deterministic or non-deterministic, depending on the functions the programmer specifies to select the next state for each group.

Within the "groups", declared with the **defgroup** macro (see Figure 3.4 on page 56), rules are free to operate completely asynchronously, or may be organised into yet more finite-state automata. While the "nesting" of automata *is* possible, care must be taken not to nest more deeply than necessary, lest operational overhead become prohibitive. It is also necessary for implementors to use this utility within the "spirit" of rule-based programming and to avoid having the expert system degenerate into a – very inefficient – sequential program.

Implicit to these sequencing mechanisms are the concepts of *activated* and *deactivated* code fragments. A deactivated code fragment is a set of one or more rules whose firing has been deliberately inhibited. Each group and automaton makes every rule or group of rules it encloses dependent upon the activation or deactivation of that group or automaton by adding a control condition specific to that group or

```
<defautomaton> ::= (defautomaton (<name>
                       {(declare <aut-decl>*}
                       {<defgroup> | <defrule> |
                              <lisp-or-ART-form>}* )

<aut-decl> ::= (relative-salience <integer>) |
               (default-salience <integer>) |
               (salience <integer>) |
               (conditions (<lhs-pattern>+)) |
               (initial-state <group-name>) |
               (final-state <group-name>) |
               <ART-decl-form>

<name> ::= Any valid (Common-Lisp) symbol.
<lisp-or-ART-form> ::= Any valid Common-Lisp or ART expression.

<group-name> ::=  Any symbol, must also appear as the
                  name in an enclosed <defgroup>

<lhs-pattern> ::=  Any valid ART LHS pattern.

<ART-decl-form> ::=  Any form which may legally appear in an ART
                     (declare ...) expression.
```

Figure 3.3: BNF for the **defautomaton** macro.

automaton to the rule. In plain English, a rule has "permission to fire" if and only if all groups and automata enclosing it are activated.

When an automaton is first defined it is by default disabled or "deactivated", as are all groups contained within it. In this state, rules enclosed in the automaton are prevented from firing by the absence of one or more control facts in the working memory. When an automaton is activated (by invoking the **activate-automaton** function), this fact is asserted in ART's working memory, which allows the firing of rules which are enclosed by the automaton but not by any group. Programmers

may place rules at this level of enclosure to perform initialisation operations for the automation.

In early implementations, ESWE automatically defined two such rules. The first, which had a salience of 0 relative to the rest of the automaton, would cause the group declared as the automaton's initial state to be activated; the second, with a relative salience of -10, would, when fired, cause the automaton to deactivate itself. The first rule was written such that it would fire *once* each time the automáton was activated. Because of its negative relative salience, the second rule should not fire until all *active* rules in the automaton have finished firing. (Programmers should avoid declaring rules with relative salience of less than 0 in order to assure that all automata continue to work as expected.)

More recent versions of the ESWE environment have replaced these rules with more efficient mechanisms. The actions of the first rule, which automatically activated the group corresponding to an automaton's initial state, were absorbed by the (**activate-automaton** ...) function, which is written in Lisp. The actions of the second rule, which automatically deactivated an automaton were eliminated completely, in favour of explicit deactivation of automata using the (**deactivate-automaton** ...) function. Naturally, if the behaviour of the older versions is desired, and considered worth the overhead, users may explicitly add control rules like these themselves.

It is not possible to *pause* the operation of an automaton; once deactivated (via the **deactivate-automaton** function), an automaton may resume operation only from its initial state. This is a consequence of using facts asserted in the data base to control the activation of groups and automata.

Within each group are additional control rules which cause the determination and activation of the automaton's next state, either when the current group is deactivated (by invoking the **deactivate-group** function) or when the group runs out of rules

to fire. These automatically generated control rules are also given a salience of −20 relative to the rest of the group. When the new group to be activated is one of those declared as a "final state", the automaton and all enclosed groups are deactivated.

The following sections present a detailed description of the syntax and semantics of **defgroup** and **defautomaton:**

3.2.3.1 Lexical Scoping in ESWE

The macros provided in the ESWE environment are intended, in part, to provide a form of *lexical scope* within a rule-based expert system. The normal state of affairs in a rule based expert system is to have a large number of rules simultaneously "watching" a common database for a predefined set of patterns. When a rule detects its specific set of patterns in the database it attempts to "fire" itself; that is, to perform the set of actions it has associated with it. (There are several factors which may prevent a given rule from firing, but they are not pertinent to this discussion.)

Frequently it is the case that not all of the rules in an expert system are, or should be, eligible to fire at a given time. Eligibility to fire is controlled and determined simply by adding one or more "patterns" to each rule, and then asserting control facts into the database to trigger a given set of rules. The management of these control "patterns" and facts is usually quite tedious.

The sequencing mechanisms provided by ESWE operate in precisely this manner, by adding control "patterns" to rules, and asserting and retracting control facts to and from the database. The difference is that ESWE provides the required modifications to rules *automatically*, and manages the majority of control data in the database automatically, too.

In developing *hierarchical* tools in the ESWE environment for describing the logical structure of an expert system (or rule-based program), the author adds the concept of lexical scoping to each rule's eligibility to fire. That is, a rule is eligible to fire at a

given time only if all ESWE structures lexically enclosing it are *active* at that moment. Lexical scoping in ESWE is presently restricted to each rule's eligibility to fire and to declarations of salience or conditions. Rule, group, automaton, relation, and schema names are all global entities due to the nature of Common-Lisp symbols, as are all data placed in the database due to the nature and purpose of the database. Lexical scoping *could* be extended to include names also, but only at the cost of *modifying* every such declared name and thus seriously impairing both debugging and run-time operations.

In particular, to extend lexical scoping to identifiers such as rule or group names, it will be necessary to *modify* the symbols. One suitable technique, that commonly used in C++ [Stroustrup 86], an extension to the C [Kernighan 78] programming language, is to append to each identifier a string containing information about the lexical environment. This approach is simple and effective, but unless users are intimately familiar with the transformations performed on symbols, the ability to use run-time monitoring or debugging features of the underlying language are seriously impaired.

3.2.3.2 Hierarchy and Inheritance

Before analysing the detailed syntax and semantics of ESWE's hierarchical sequencing mechanisms, it will be fruitful to digress slightly and discuss the relationship between *hierarchy* and *inheritance.*

The inheritance mechanism is implemented by allowing or requiring each item in the hierarchy (expert, automaton, group, or rule definitions) to include a set of declarations closely resembling those of ART's *defrule* macro. Each of these "items" (except rule declarations because they are the lowest level of the hierarchy) will cause any declarations it is given relating either to salience or "conditions" to be prefixed to the declarations for each other "item" it in turn encloses.

Inheritance is a mechanism which allows expressions at a given level in a hierarchy to obtain properties from expressions at higher (i.e., enclosing) levels in the hierarchy. It may be recalled that in the ART expert system language, as in most others, all rules in the expert system exist at a single level; no rule lexically encloses any other. Consequently, the only way ESWE can express hierarchy is to have expressions at a given level *modify* expressions at lower levels.

One example of the use of inheritance in ESWE is in the propagation of *conditions* from expressions at one level of the hierarchy to the next. Thus, an *automaton* is able to make all lexically contained rules dependent upon its activation status (activated versus deactivated) simply by adding a declaration of the form:

```
(declare (conditions (status <automaton-name> active)))
```

to each of the ART expressions it directly encloses. The conditions declaration is then inherited by each other ART expression enclosed at lower levels. In this way, the firing of any given rule is made conditional upon *all* enclosing groups and automata being active at the same time.

A second use of inheritance is for the propagation of salience declarations. The default salience for any ART expression declared at the *root* of the hierarchy is always zero. (Saliences may be any value within the range of -1 000 000 to 1 000 000.) For an ART expression at a given level in the hierarchy, the salience will be either the value contained in the expression's salience declaration, or, if no such declaration exists, the expression's **default-salience**[3] *plus* the sum of all the expression's **relative-saliences**. (A salience declaration overrides default/relative salience.) Once computed in this manner, the expression's salience value is then propagated to enclosed

[3]If an expression has more than one **default-salience** declaration, the last such declaration takes precedence over all others. Because *inherited* declarations are added *before* other declarations, placing a **default-salience** declaration in a rule overrides any inherited value.

expressions by adding a **default-salience** clause to the beginning of the declarations for each such enclosed expression.

Although ESWE *permits* implementors to override inherited saliences by explicitly declaring either a **default-salience** or a absolute **salience** for an ART expression, they are *strongly discouraged* from doing so. (Although discouraged, this behaviour is not outrightly *forbidden*, as freedom to declare salience may occasionally be required for complete generality.) The internal functions of ESWE generate code which may depend upon some of these inherited characteristics to function properly; overriding default-saliences or declaring a relative-salience of less than -20 may risk upsetting the automatically generated code.

3.2.3.3 Detailed Syntax and Semantics of Defgroup

The **defgroup** macro provided by the ESWE environment is intended to declare groups of mutually asynchronous rules (or automata) which operate within the confines of an automaton. Figure 3.4 illustrates the syntax of **defgroup**.

```
<defgroup> ::=  (defgroup <name>
                    (declare  (next-state <state-spec>)
                              {(default-salience <integer>)}*
                              {(relative-salience <integer>)}*
                              {(salience <integer>)}*
                              {(conditions (<lhs-pattern>+))}*
                              {<ART-decl-forme>}* )
                      {<ruledef> | <lisp-or-ART-form>}* )

<lisp-or-ART-form> ::=  Any valid Common-Lisp or ART expression.
<lhs-pattern> ::=  Any valid ART LHS pattern.

<ART-decl-form> ::=  Any form which may legally appear in an ART
                     (declare ...) expression.
```

Figure 3.4: BNF for the **defgroup** macro.

The most important aspect of the **defgroup** macro is the (declare ...) expression, which is used, among other things, to implement the inheritance mechanism described in the previous section. This is very similar to the declare expression of ART's **defrule** macro, except that the syntax is extended to allow ESWE declarations also.

As implied by the section describing inheritance, the salience and conditions declarations are applicable to *all* ESWE definitions, and they behave for **defgroup** exactly as described in that section. Specific to **defgroup** is the **next-state** declaration. A **defgroup** definition may contain any number (zero or more) of these universal declarations, but must contain *precisely one* **next-state** declaration. The value provided in this declaration may be one of:

- The name of the group which is to be activated after the current group (the group being defined) has completed its actions or been deactivated.
- The name of a function which, when executed, returns as its result the name of the next group to activate. The function may accept no arguments but is free to inspect (or alter) the ART database. Using functions such as this, one may implement conditional branching in an automaton, or even non-deterministic automata. Functions which modify the ART database should be used with caution.
- The common-lisp symbol 'STOP-NOW-PLEASE, which is declared as a final-state for every automaton. This will cause the enclosing automaton to halt and deactivate itself.
- Any symbol contained in the (**final-state** ...) declaration of the enclosing automaton. This will have the same effect as using the symbol 'STOP-NOW-PLEASE.

Declarations in a **defgroup** definition may appear in any order, even though this is not correctly reflected by the BNF of Figure 3.4. It is also "permissible" to include declarations for **initial-state** and **final-state** (which are normally reserved for **defautomaton** definitions) however, the results of doing so are not defined.

The body of a **defgroup** definition may contain any number of (extended) rule definitions or valid lisp or ART expressions. Normally the only lisp or ART declarations that have any true meaning in this context are such things as lisp variable declarations, ART relation declarations, or ESWE expert or automaton definitions[4]. Any other lisp or ART expressions will simply be evaluated (eval'ed) at the time the **defgroup** definition is loaded into the lisp system.

Rule definitions (using either **defrule** or **my-defrule**) enclosed by either a **defgroup** or **defautomaton** definition will be parsed by ESWE and re-written as syntactically correct ART defrule definitions; **defrule** definitions may use the ESWE extended declarations *only* within such a context. A rule whose definition appears within a **defgroup** definition is made eligible to fire exactly when the defined group and all groups or automata enclosing it are activated. This does not mean, however, that the rule *will* fire at this time, or even at all, since it is still bound by ART's regime of salience and pattern matching.

Defgroup definitions should appear only immediately within a **defautomaton** definition. It is not meaningful to place a **defgroup** definition in any other context. The result of doing so is not defined.

This example of the code produced by ESWE demonstrates several points: first, ESWE generates a substantial amount of code for each automaton or group, but all such code is either *pure* ART or Common-Lisp; second, properties are recursively

[4]These *are* valid lisp forms.

inherited through the hierarchy; and third *some* inherited properties can be overridden.

3.2.3.4 Detailed Syntax and Semantics of Defautomaton

Just as the **defgroup** macro is a mechanism for declaring groups of mutually asynchronous rules, the **defautomaton** macro is a mechanism for describing the synchronisation that must be performed *between* these sets of groups. As with the **defgroup** macro, the **defautomaton** macro allows a number of ART-like declarations, *most* of which get passed on to enclosed "items" (automaton, group, or rule definitions) via the ESWE inheritance mechanism. All declarations permitted in the **defgroup** macro, save only the **next-state** declaration, apply identically to the **defautomaton** macro. The result of including a **next-state** declaration in a **defautomaton** definition is not defined.

There are also two ESWE declarations which are specific to the **defautomaton** macro, the **initial-state** declaration, and the **final-state** declaration. The former declaration is used to declare which of the one or more groups enclosed in the automaton is to be first activated (this is done automatically when the automaton is activated). The default **initial-state** is the *first* group immediately enclosed by the automaton. The latter declaration is used to declare additional final states for the automaton.

The body of the automaton definition may include zero or more **defgroup** definitions, zero or more **defrule** definitions, zero or more **defautomaton** definitions, and zero or more ART or Common-Lisp expressions. These expressions and definitions may appear in any order, but *must not* appear before the (declare ...) expression. As is the case with **defgroup**, *any* ART or Common-Lisp expression is legal within a **defautomaton** definition; however, declarative expressions usually make a good deal more sense that executable or side-effecting expressions.

3.2.4 Example of Defautomaton, Defgroup, and My-defrule

Following is a brief example of some of the main features of the ESWE environment. It illustrates the use of groups, and automata, nested automata, and rules for initialisation of automata.

Original Source Code:

```
;-*- Mode: LISP; Syntax: Common-lisp; Package: ART-USER; Base: 10 -*-

(defautomaton (automaton1
   (declare (default-salience 100)
            (conditions ((count ?x) (test (< ?x 10)))))

   (defrule example-of-initial-code
      (declare (relative-salience 20000))
      (count ?y)
      (test (> ?y 5))
      =>
      (format t "This rule will fire only if there is a ~
                 count < 10 and a count > 5~%"))

   (defgroup (group1
      (declare (relative-salience 30)
               (next-state group2))

      (my-defrule
        =>
        (format t "~&Group1 is now active...~%"))))

   (defgroup (group2
      (declare (next-state STOP-NOW-PLEASE))

      (my-defrule
        =>
        (format t "~&Group2 is now active, activating Automaton2~%")
        (activate-automaton 'automaton2)
        (format t "~&Deactivating Automaton1...~%")
        (deactivate-automaton 'automaton1))))

   (defautomaton (automaton2
      (defrule auto-rule-2
         =>
```

```
        (format t "Rule auto-rule-2 fired~%"))))))
```

Passing the above code through the macros defined by ESWE while inhibiting the expansion of macros defined by ART yields the following, considerably more verbose code. Note the use of (eval-when ...) expressions, which force the presence of crucial portaions of the ESWE system when the code is loaded into the LISP (ART) environment.

Code Generated by ESWE:

```
(PROGN
  (EVAL-WHEN (LOAD EVAL)
    (IF (NOT (SCHEMAP 'EXPERT))
        (LOAD '"sym-1:>stanley>stan-compiler-objs.art"))
    (IF (NOT (BOUNDP '*STAN-AUTOMATA-LIST*))
        (LOAD '"Sym-fsa:>stanley>compiler-globals"))
    (IF (ZL:MEMQ 'AUTOMATON1 *STAN-AUTOMATA-LIST*)
        (ERROR "Attempt to redefine automaton ~S" 'AUTOMATON1)
        (PROG NIL
              (ZL:PUSH 'AUTOMATON1 *STAN-AUTOMATA-LIST*)
              (ZL:PUTPROP 'AUTOMATON1 '(INITIAL-STATE GROUP1)
                          'INITIAL-STATE))))

  (DEFSCHEMA AUTOMATON1
             (INSTANCE-OF AUTOMATON)
    (INITIAL-STATE GROUP1)
    (FINAL-STATE STOP-NOW-PLEASE))

  (DEFRULE EXAMPLE-OF-INITIAL-CODE
           (DECLARE (SALIENCE 20100))
    (SCHEMA AUTOMATON1 (STATUS ACTIVE))
    (COUNT ?X)
    (TEST (< ?X 10))
    (COUNT ?Y)
    (TEST (> ?Y 5))
    =>
    (FORMAT T "This rule will fire only if there is a ~
               count < 10 and a count > 5~%" ))

  (PROGN
    (EVAL-WHEN (LOAD EVAL)
      (IF (NOT (SCHEMAP 'EXPERT))
```

```
        (LOAD '"sym-1:>stanley>stan-compiler-objs.art"))
    (IF (NOT (BOUNDP '*STAN-GROUP-LIST*))
        (LOAD '"Sym-fsa:>stanley>compiler-globals"))
    (IF (ZL:MEMQ 'GROUP1 *STAN-GROUP-LIST*)
        (ERROR "attempt to redefine GROUP ~S." 'GROUP1)
        (PUSH 'GROUP1 *STAN-GROUP-LIST*)))

  (PROGN
    (EVAL-WHEN (LOAD EVAL)
      (IF (NOT (SCHEMAP 'EXPERT))
          (LOAD '"sym-1:>stanley>stan-compiler-objs.art")))
    (DEFRULE |my-rule-2|
      (DECLARE (SALIENCE 120))
      (SCHEMA AUTOMATON1 (CURRENT-STATE GROUP1))
      (SCHEMA AUTOMATON1 (STATUS ACTIVE))
      (COUNT ?X)
      (TEST (< ?X 10))
      =>
      (SET-STATE 'AUTOMATON1 'GROUP2)))

  (PROGN
    (EVAL-WHEN (LOAD EVAL)
      (IF (NOT (SCHEMAP 'EXPERT))
          (LOAD '"sym-1:>stanley>stan-compiler-objs.art")))
    (DEFRULE |my-rule-3|
      (DECLARE (SALIENCE 130))
      (SCHEMA AUTOMATON1 (CURRENT-STATE GROUP1))
      (SCHEMA AUTOMATON1 (STATUS ACTIVE))
      (COUNT ?X)
      (TEST (< ?X 10))
      =>
      (FORMAT T "~&Group1 is now active...~%"))))

(PROGN
  (EVAL-WHEN (LOAD EVAL)
    (IF (NOT (SCHEMAP 'EXPERT))
        (LOAD '"sym-1:>stanley>stan-compiler-objs.art"))
    (IF (NOT (BOUNDP '*STAN-GROUP-LIST*))
        (LOAD '"Sym-fsa:>stanley>compiler-globals"))
    (IF (ZL:MEMQ 'GROUP2 *STAN-GROUP-LIST*)
        (ERROR "attempt to redefine GROUP ~S." 'GROUP2)
        (PUSH 'GROUP2 *STAN-GROUP-LIST*)))

  (PROGN
    (EVAL-WHEN (LOAD EVAL)
```

```
      (IF (NOT (SCHEMAP 'EXPERT))
          (LOAD '"sym-1:>stanley>stan-compiler-objs.art")))
    (DEFRULE |my-rule-4|
      (DECLARE (SALIENCE 90))
      (SCHEMA AUTOMATON1 (CURRENT-STATE GROUP2))
      (SCHEMA AUTOMATON1 (STATUS ACTIVE))
      (COUNT ?X)
      (TEST (< ?X 10))
      =>
      (SET-STATE 'AUTOMATON1 'STOP-NOW-PLEASE)))

  (PROGN
    (EVAL-WHEN (LOAD EVAL)
      (IF (NOT (SCHEMAP 'EXPERT))
          (LOAD '"sym-1:>stanley>stan-compiler-objs.art")))
    (DEFRULE |my-rule-5|
      (DECLARE (SALIENCE 100))
      (SCHEMA AUTOMATON1 (CURRENT-STATE GROUP2))
      (SCHEMA AUTOMATON1 (STATUS ACTIVE))
      (COUNT ?X)
      (TEST (< ?X 10))
      =>
      (FORMAT T "~&Group2 is now active, activating Automaton2~%")
      (ACTIVATE-AUTOMATON 'AUTOMATON2)
      (FORMAT T "~&Deactivating Automaton1...~%")
      (DEACTIVATE-AUTOMATON 'AUTOMATON1))))

(PROGN
  (EVAL-WHEN (LOAD EVAL)
    (IF (NOT (SCHEMAP 'EXPERT))
        (LOAD '"sym-1:>stanley>stan-compiler-objs.art"))
    (IF (NOT (BOUNDP '*STAN-AUTOMATA-LIST*))
        (LOAD '"Sym-fsa:>stanley>compiler-globals"))
    (IF (ZL:MEMQ 'AUTOMATON2 *STAN-AUTOMATA-LIST*)
        (ERROR "Attempt to redefine automaton ~S" 'AUTOMATON2)
        (PROG NIL
              (ZL:PUSH 'AUTOMATON2 *STAN-AUTOMATA-LIST*)
              (ZL:PUTPROP 'AUTOMATON2 '(INITIAL-STATE NIL)
                          'INITIAL-STATE))))
  (DEFSCHEMA AUTOMATON2
             (INSTANCE-OF AUTOMATON)
    (INITIAL-STATE NIL)
    (FINAL-STATE STOP-NOW-PLEASE))

  (DEFRULE AUTO-RULE-2
```

```
(DECLARE (SALIENCE 100))
(SCHEMA AUTOMATON2 (STATUS ACTIVE))
(SCHEMA AUTOMATON1 (STATUS ACTIVE))
(COUNT ?X)
(TEST (< ?X 10))
=>
(FORMAT T "Rule auto-rule-2 fired~%"))))
```

Notice in the expanded code, above, that only *pure* ART syntax and expressions are used. All extended declarations have been processed, and replaced by standard ART **salience** declarations. Code has been generated to create the schemata objects required to control the groups and automata and to allow external LISP code to monitor their status.

Next is presented a brief demonstration of executing the code shown above. The input for the demonstration, which simply places two facts in the database and activates the outer automaton, is:

```
(acu:reset)
(art:assert (count 1))
(art:assert (count 7))
(activate-automaton 'automaton1)
(acu:run)
```

The output resulting from the expression `(acu:run)` is:

```
This rule will fire only if there is a count < 10 and a count > 5
This rule will fire only if there is a count < 10 and a count > 5
Group1 is now active...
Group1 is now active...
Group2 is now active, activating Automaton2
Deactivating Automaton1...
No applicable rules.
NIL
```

Note that the some of the rules fired twice. This is because they contained a pair of patterns, one of which looked for a count less than ten, and the other which looked for a pattern between five and ten. The facts `(count 1)` and `(count 7)` matched the former pattern twice, but the latter pattern only once, resulting in $2 \times 1 = 2$

activations of each rule. If the fact (count 1) were removed from the database, each rule would fire only once.

3.2.5 Defexpert: ESWE's Macro for Sequential Experts

In this section, ESWE's macro for describing and sequencing actions in a domain expert is reviewed. In Section 3.2.3 ESWE's macros for describing finite state automata were described. These macros are certainly adequate for the description of a "domain expert", but may tend to be both cumbersome and unnecessarily general. Frequently it is the case that a domain expert requires only a very simple structure, usually, a simple "while" loop with allowances for initialisation and wrap-up is adequate.

```
<defexpert> ::=
     (defexpert (<name>
        {(declare {(relative-salience <integer>)}
                  {(default-salience <integer>)}
                  {(salience <integer>)}
                  {(conditions ({<ART-LHS>}*))}
                  {(initial-state <group-name>)}
                  {(final-state <group-name>)}  )}
        {<lisp-or-ART-form> | <ruledef>}*

        {(initialise {<lisp-or-ART-form> | <ruledef>}*)}
        {(iterate
            {(start-up {<lisp-or-ART-form> | <ruledef>}*)}
            {<lisp-or-ART-form> | <ruledef>}*
            {(clean-up {<lisp-or-ART-form> | <ruledef>}*)})}
        {(wrap-up {<lisp-or-ART-form>}*)
     ))

<lisp-or-ART-form> ::=  Any valid common-lisp or ART form.
```

Figure 3.5: BNF for **defexpert**

The main primitive developed for defining experts is the **defexpert** macro, (see Figure 3.5) which in turn generates **defautomaton, defgroup** and **defrule**

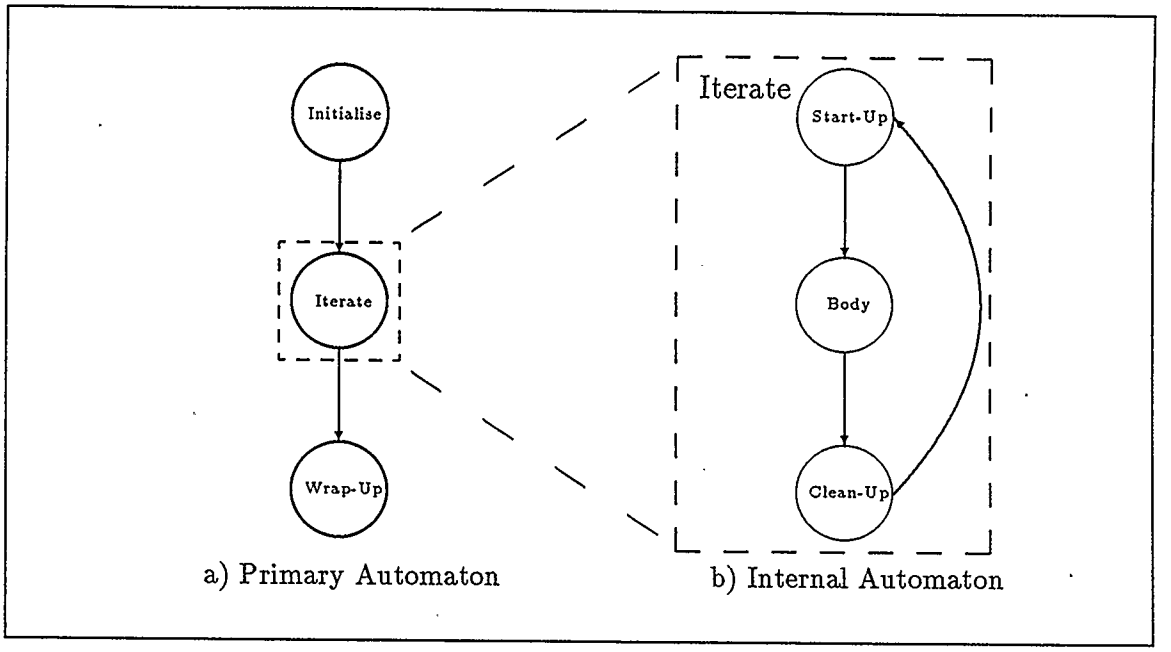

Figure 3.6: State transitions among automata created by **defexpert**.

macros as needed. Defexpert generates two *nested* finite automata (see Figure 3.6 on page 66), which perform the following actions:

1. Initialise the expert.
2. Activate the body of the expert. The body is, itself, an automaton which iteratively performs the following actions:
   (a) Execute some set-up (start-up) code for the current iteration.
   (b) Activate the main body of the loop. The body may, of course, contain *more* finite state automata.
   (c) Execute some clean-up code, in preparation for starting the loop over.
3. Perform some termination actions. These may include, for example, printing a message on a terminal, or clearing the knowledge of unnecessary temporary data.

When inspecting the state diagram of Figure 3.6 it is necessary to recall the nature of state transitions in ESWE FSAs. Most readers will quickly notice that none of the arcs in the state diagram (transitions in the automaton) are labeled. Further, after the hierarchy of the state diagram is flattened, it is immediately apparent that there are *two* possible transitions from the state labeled "Clean-Up", one to the state "Wrap-Up", which terminates the expert, and one to the state "Start-Up", which continues the iteration of the body of the expert. While neither of these transitions is associated with an "input" in any obvious way, the automaton is nonetheless deterministic, and has been deliberately made so. This is because the transition from "Clean-Up" to "Start-Up" is of higher salience (priority) than the transition from "Clean-Up" to "Wrap-Up"; thus the first transition will *always* be made in preference to the second, *except* when the iterative body of the expert is *explicitly* deactivated by a call to the **deactivate-automaton** function. Once such a call is made, the high salience rules of the expert's body are no longer able to fire, and the lower salience rule which causes the state transition from "Clean-Up" to "Wrap-Up" will be able to fire.

3.3 Design Abstraction in ESWE

In addition to control structures for defining expert systems, ESWE provides a user-extensible structure for describing circuit components, and "layout technologies" [5]. The design abstraction facilities of ESWE are implemented using the object-oriented *schema* facility of the ART language [ART87, chapters 9, 12], and are therefore useful both for providing user extensible features and for providing an interface to external code written in Common Lisp.

[5] Layout technologies should not be confused with "fabrication" technologies, for which they serve as an abstraction.

3.3.1 Technology Abstraction

ESWE's technology abstraction, provided by the **deftechnology** macro described in Figure 3.7, allows for the description of CMOS technologies on a layer-by-layer basis. That is, a technology may consist of an arbitrary number of layers, having arbitrary names. Each layer, however, is expected to be declared as one of six predefined types (p-active, n-active, gate, via, interconnect, and virtual) and of one of six types of material (polysilicon, metal, silicide, implant, diffusion, and virtual). Additional information about each layer, such as resistivity, capacitance, and thickness may also be specified, should it be required by the expert system being developed.

"Virtual" is included as a layer type and as a material to allow the specification of layers which may not fall into any of the other categories. It is intended that layers declared as *virtual* be used primarily for passive layers in a technology[6], which are essential to the correct fabrication of devices, but unimportant in considering placement and routing. Naturally, virtual layers may be used for whatever purpose an expert system implementor may see fit; in fact, nothing prevents *all* layers from being declared "virtual", except the loss of what may prove invaluable information about their physical or electrical properties.

A second anticipated use of "virtual" layers is for space-keeping. Design rules are often described in terms of features which are sometimes not easily identifiable. For example, in order to enforce a design rule which requires a separation between vias and gates, it may be helpful to place a rectangle of "material" on a layer called "virtual-gate" over the gate of each transistor. While no such rectangle is required for the fabrication of the device, it is easier to check the boundary conditions between pairs of rectangles on "via" and "virtual-gate" layers than it is to check for conditions between n-tuples of rectangles.

[6]The thinox "overglass" layer of MOSIS Generic CMOS is a good example of such a layer.

```
BNF for DEFTECHNOLOGY:

<deftech> ::=
    (deftechnology <name>
      (LAYERS [<layer-spec>]*)
      (RULES
        [ <rule-spec> ]* )
      {<defobjects>}*
    )

<layer-spec> ::=
     (<layer-name> <layer-type> <layer-material>
                   {<restivity>} {<capacitance>} {<thickness>})

<layer-type> ::= N-ACTIVE | P-ACTIVE | GATE |
                 INTERCONNECT | VIA | VIRTUAL
<layer-material> ::= POLY | METAL | SILICIDE | IMPLANT |
                     DIFFUSION | VIRTUAL

<rule-spec> ::=
    (SEPARATION <layer-name-1> <layer-name-2> <integer>) |
    (SURROUND <layer-name-1> <layer-name-2> <integer>)   |
    (MINIMUM-FEATURE <layer-name> <integer>)             |
    (GATE-OVERLAP <integer>)                             |
    (ACTIVE-OVERLAP <integer>)                           |
```

Figure 3.7: BNF for the Deftechnology Macro

Design rules for a given technology may also be specified using three general types of design rule: Separation rules (for the minimum distance between features on two layers), Surround rules (for the minimum enclosure of features on one layer by features on another), and Minimum-Feature rules (for the minimum features on a given layer).

3.3.2 Layout Abstraction

To complement ESWE's technology abstractions, a facility has also been provided for abstraction of design components and circuit designs. During the design of these abstractions, considerable emphasis was placed on uniformity of representation; this consideration has, in fact, taken precedence over other considerations such a memory use or compilation time.

3.3.2.1 Technology Independent Layout Primitives

There are two technology-independent layout primitives provided by ESWE, "pins" and "wires". All other layout "primitives" must ultimately be described in terms these two. Pins and wires each have several attributes, some optional, others not. Pins serve to represent a location in a grid, possibly associated with a specific layer or a specific electrical network. Wires represent an electrical connection between a pair of pins, possibly on a specific layer, representing a rectangle of specific width. Figures 3.8 and 3.9 contain the ART definitions of these two base primitives.

In addition to slots for electrical net, layer and location, a pin also has slots describing its "parent" and its "constrainedness" (i.e. its state of being constrained). The "parent" slot is used to specify that a given pin (an instance of the schema *sym-pin*) is *part* of another object, eg., a transistor. When a pin is declared as being constrained (i.e. when the value of its constrainedp slot is 'YES'), it is expected not to be moved except as a result of transformations or translations on its parent; the enforcement of this expectation is, however, left to the application.

As with pins, wires include *parent* and *constrainedp* slots. Perhaps surprisingly, they do not contain *width* slots. This is because the sym-wire schemata is not expected to always represent *physical* wires; virtual wires without width or dimension

```
(defschema sym-pin  ; "A symbolic-placement pin"
  (net)
  (layer)
  (loc)
  (parent)           ;; Keep track of object a pin is part of.
  (constrainedp NO)) ;; Is this pin forced to move ONLY along with
                     ;; its parent object? (default: NO)

(defaction translate-pin ( (pin sym-pin) ) (del-pair)
  (prog* ( (location (list*$ (get-schema-value pin 'loc)))
   (del-x (car del-pair))
   (del-y (cadr del-pair))
   (x-loc  (car location))
   (y-loc  (cadr location)) )
    (setq x-loc (or (and (numberp x-loc) x-loc) 0))
    (setq y-loc (or (and (numberp y-loc) y-loc) 0))
    (modify-schema-value
       pin 'loc (seq*$ `(,(+ x-loc del-x)
                         ,(+ y-loc del-y))))))
```

Figure 3.8: Formal ART Specification of Symbolic Pins.

may be needed to indicate electrical connectivity between two objects[7]. It is possible to add a width declaration to wires as required using the (modify-schema-value ...) operation provided by ART.

3.3.2.2 Technology Dependent Layout Primitives

While it is possible to layout circuits with only the two primitives described above, many designers (and expert systems implementors) are apt to find this very low level of abstraction overly tedious. Of course, it is normally necessary that designs eventually be expressed at approximately this level of abstraction for purposes of mask generation.[8] Frequently, it proves useful to think of circuit elements on higher

[7]For example, it is a common occurance in the ELECTRIC design system for "virtual" wires to be used to prevent the design-rule checker from detecting errors in well- and substrate-contacts.

[8]Eg., many designs must be submitted for fabrication using CIF [Mead 80, pp. 115 – 127], which operates at or about this level of abstraction.

```
;   Following is a definition of a symbolic wire. NOTE that
; symbolic wires, by default, have no width; a width may be
; added to a wire  using (modify-schema-value ...)

(defschema av-wire-NET (is-a active-value))

(defschema sym-wire ; "A symbolic placement wire"
  (net)
  (layer)
  (end-1)
  (end-2)
  (parent)
  (constrainedp NO))

(defaction modify-before (av-wire-NET) (schema slot old-val new-val)
  (loop for pin in `(,(get-schema-value schema 'end-1)
     ,(get-schema-value schema 'end-2)) do
    (modify-schema-value pin 'net new-val))
)
(defaction put-before (av-wire-NET) (schema slot new-val)
  (loop for pin in `(,(get-schema-value schema 'end-1)
     ,(get-schema-value schema 'end-2)) do
    (when (schemap pin)
      (modify-schema-value pin 'net new-val)))
)
(add-active-value 'sym-wire 'NET 'av-wire-NET)
```

Figure 3.9: Formal ART Specification of Symbolic Wires.

levels, regarding transistors, contacts, or even logic gates as primitives; the benefits of this are readily gained in simplified design methodologies (it is easier to reason about a "transistor" than a collection of some thirty pins and wires) and simplified design rule checking.

Higher level primitives can be expressed in terms of these base-level primitives, by means of prototypes. (The sym-pin and sym-wire schemata in Figures 3.8 and 3.9, are in effect prototypes for wires and pins; in order to get an object that can be used in a layout one must use an "instance" of one of these prototypes.) ESWE provides

means for generating prototypes for two classes of objects: transistors and contacts (vias). The description of a prototype consists primarily of a list of pins and wires to be created when an instance of the prototype is made. Pin locations are relative to the center of the "prototype", but prototyped objects need not be symmetric. ESWE provides methods for rotating, mirroring, and translating instances of contact and transistor prototypes, although expert system implementors using ESWE may easily redefine these methods as required. Figure 3.10 contains examples of transistor and prototype declarations for the MOSIS Generic CMOS technology.

Appendix B contains a complete sample technology definition based upon MOSIS Generic CMOS.

3.3.3 ESWE Facilities for Cell Description

The final facility provided by ESWE for design abstraction is the **defcell** macro. The least general of technology abstractions, **defcell** is intended to allow designers to easily specify the topology of a standard cell and have that topology entered into the ART database for manipulation by one or more expert systems. The syntax for this macro is illustrated in Figure 3.11.

Defcell allows designers not only to specify the topology of a cell, but also permits the required the description of locations for "ports" at which electrical networks enter or exit the cell. Because **defcell** is intended primarily for use in *symbolic layout* contexts, no provision is made for placing ports at specific locations; ports are placed relative to one another, using constraints on horizontal and vertical position.

For purposes of information or error detection, a number of parameters must be declared for each cell. These are: the electrical networks used (internal and external); the size of the cell (number of rows and columns of transistors); the names of transistors to be used in the cell; and the names, networks, and relative positions

```
(defcontact metal-1-metal-2
  (contacts metal-1 metal-2)
  (pins (m-1 metal-1 (0 0))
        (m-2 metal-2 (0 0))
        (c-1 contact (0 0)))
  (wires (mw-1 metal-1 4 m-1 m-1)
         (mw-2 metal-2 4 m-2 m-2)
         (cw-1 contact 2 c-1 c-1)))

(defschema p-transistor
  (instance-of transistor-prototype)
  (type P-TRANSISTOR)
  (proto-pins (SRC-PIN p-active (0 100))
              (DRN-PIN p-active (0 -100))
              (GATE-PIN-1 poly  (100 0))
              (GATE-PIN-2 poly  (-100 0))
              (INT-PIN-1  p-active (0 0))
              (INT-PIN-2  p-active (0 0))
              (WELL-PIN-1 n-well   (0 100))
              (WELL-PIN-2 n-well   (0 -100)))
  (active-proto-pins SRC-PIN DRN-PIN GATE-PIN-1 GATE-PIN-2)
  (proto-wires
    (SRC-WIRE p-active 5 SRC-PIN INT-PIN-1)
    (DRN-WIRE p-active 5 DRN-PIN INT-PIN-2)
    (GATE-WIRE poly    3 GATE-PIN-1 GATE-PIN-2)
    (WELL-WIRE n-well 19 WELL-PIN-1 WELL-PIN-2))
  (proto-constraints))
```

Figure 3.10: Example of Contact and Transistor Prototypes.

```
<cell-defn> ::=
   (DEFCELL <name>
      (NETWORKS [<netname>]*)
      (SIZE <rows> <columns>)
      (PORTS
         (NORTH {<portspec>}*)
         (SOUTH {<portspec>}*)
         (EAST  {<portspec>}*)
         (WEST  {<portspec>}*)
         (INTERIOR {<portspec>}*)
         (CONST {<port-constraint>}*)
      )
      (TRANSISTORS [<transistor-name>]*)
      (CONSTRAINTS {<constraint>}*)
   )

<portspec> ::=
       (<portname> <network> {<contact-type>})

<port-constraint> ::=
       (<portname-1> = <portname-2>)
     | (<portname-1> >= <portname-2> {+ <integer>})
     | (<portname-1> || <portname-2>)
     | (<portname-1> ^|| <portname-2> {+ <integer>})
```

Figure 3.11: BNF for the Defcell Macro.

of the "ports" used by the cell. The following paragraphs discuss each of these classes separately.

As a measure of error detection, all network names must be declared. A network name may be any valid ART symbol, but because this name will be used as the "name" of a schema, it must not have been used as the name of any other schema.[9] The schemata generated for the declared networks in a cell serve primarily to indicate

[9]This, unfortunately, is also the case for *layer* names, *contact* names, *transistor* names, *cell* names, *port* names, *pin* names, *wire* names, and for names of any other object which may be generated either by *stanley* or by application software.

the existence of these networks, but with user-supplied extensions can be profitably used for such purposes as maintaining lists of strongly connected subnets for routing. For the purposes of the **defcell** macro, the declared network names, along with the predefined networks "PWR" and "GND", comprise the list of all symbols which may legally appear where a network name is required throughout the context of the cell definition.

The size declaration is used to define the number of rows and columns of transistors which may be used in the layout of the cell. This is of particular value to expert systems which produce symbolic layout, but is also pertinent to the design *style* that ESWE targets.

Port declarations are rather complex: they are used to describe the locations of electrical connections between the cell's networks and the outside world. Because ESWE is geared toward expert systems which generate *symbolic layout*, there is little point in describing port locations in physical terms. Further, because the layout of a cell may require the relocation of one or more ports, describing ports in absolute rather than relative terms may prove counterproductive.

When declared, each port must be explicitly placed on a specific *face* of the cell or in the interior of the cell. Exterior, or *face*, ports are automatically constrained either vertically or horizontally to lie on the face (NORTH, SOUTH, EAST or WEST) for which they are declared; further, they are constrained to appear *in the order they are listed* under each face, from bottom to top, or from left to right.[10] By default, ports declared to lie in the interior of the cell remain unconstrained; they should be used with caution, as the port-placement algorithm [Brinsmead 88] supplied by ESWE will place all unconstrained interior ports at the geometrical center of the cell. Pitch-matching between ports may be specified by defining horizontal or vertical

[10]Given these default constraints on *face* ports, it is rarely necessary to provide any addition constraints, except to force pitch-matching.

equivalence relations between pairs of ports on opposite faces. The algorithm used to satisfy the constraints on port placement (to generate the *initial* placement) is described in detail, and verified, in [Brinsmead 88].

The final declaration required by **defcell** is the list of the names of all transistors to be used in the cell. The transistor names, which must obey the same rules as network names, are used solely for type-checking, to help ensure the correctness of subsequent **deftran** definitions which instantiate the transistors and define their networks.

Examples of **defcell** and accompanying **deftran** declarations may be seen in Section 4.3.

3.4 Chapter Summary

Chapter 2 described several difficult problems of VLSI design automation and explained the necessity of heuristics to provide efficient solutions to these problems. That chapter described several classes of heuristic techniques, among them, knowledge-based expert systems.

This chapter presented and described the Expert System Wrapper Environment (ESWE), developed by the author for implementing expert systems for design-oriented problems in general, and for VLSI design problems in particular. By means of the **defgroup**, **defautomaton**, and **defexpert** macros defined by ESWE, programmers are able to easily define sequential or iterative control structures in a rule-based expert system without unduly impairing the asynchronicity of rules which is normally desired in an expert system. These macros provide useful syntactic and semantic extensions to the ART [ART87] expert system shell, allowing implementors to define arbitrary finite state automata to act as control mechanisms within an expert system.

Also presented in this chapter are a number of abstractions for the design of Complimentary Metal Oxide Semiconductor (CMOS) circuits, which have been employed in the implementation an expert system for the symbolic placement of transistors in small VLSI circuit components. This expert system and its variants are the subject of Chapter 4.

Chapter 4

Stochastic Control in Expert Systems

As illustrated in the review in Chapter 2, the expert systems approach to problem solving can be very beneficial for a wide variety of applications, including numerous design-oriented problems. However, when applying rule-based programs to problems of design, developers have found it convenient, or even necessary, to recast the problem as a matter of optimisation rather than constructive synthesis; that is, to start from an arbitrary initial design, and iteratively refine it.

When the problem is a matter of diagnosis, the classical approach to expert systems as described in Chapter 2, has proven most successful; however, this is not necessarily the case for optimisation. The "classical" approach to expert systems design normally tends to produce a "greedy" optimisation, one in which short-term improvements are readily accepted without regard for the possible impact of these decisions on the progress of the optimisation at a later time.

There is a large number of known techniques for overcoming the problem of greediness in optimisation, among them, stochastic processes such as Simulated Annealing. Stochastic techniques are especially appealing for use with expert systems for a number of reasons:

1. The rules used in an expert system are typically uncertain to begin with; it hardly makes sense to rigorously apply a rule (heuristic) that is, a priori, known not to be always effective.

2. When several rules are found to be applicable in a given context, it is not sensible to rigorously select one in favour of another, without a strong and compelling reason.

3. Rigorously deterministic behaviour is reproduceable[1] and invariant. This is often an advantage, especially when testing and debugging. Unfortunately, should a deterministic expert system perform poorly for a given problem instance, the effort of retrying will be fruitless, as the results will always be the same.

4. Stochastic behaviour often helps to broaden the scope of a search, effectively counteracting the single-mindedness of a deterministic approach.

This chapter describes one way in which randomness may be integrated into an expert system to improve overall performance; that is, by adding high-level (or meta-) heuristics based upon Simulated Annealing, which are used to control how a expert system selects and acts on multiple recommendations. This is demonstrated to significantly improve the performance[2] of a simple expert system. Unfortunately, a considerable penalty in computation time is incurred. Other techniques, which involve incorporating randomness into the inference engine of an expert system (so that when several rules are candidates to be fired next the selection is *random* rather than *arbitrary*), are suggested in Chapter 5, for future consideration.

4.1 Expert Systems and Simulated Annealing

When developing an expert system to synthesise designs under an iterative refinement paradigm, it is first necessary to have some means of determining the relative value of two different states in the design space. This is normally done by means of a *cost* function, finding the minimum of which is the object of the optimisation.

[1] Stochastic behaviour controlled by a pseudo-random number generator is also reproduceable, but only when it is *desired* to be.

[2] I.e. the expert system using Simulated Annealing regularly produces superior results, although with a significant penalty in time.

The expert system operates by inspecting the current state of the design, and then making recommendations for changes which might lead to a reduction in the cost function. One of these recommendations is selected and acted upon and the result compared to the state of the design prior to the change. If the cost of the new configuration has improved over the old one, the state resulting from the change is accepted made the new "current" state and the optimisation continues. Otherwise, if the cost has not been reduced, the change is rejected, and another recommendation is selected and acted upon. This procedure continues either until there are no further recommendations for improvement, or until the cost reaches an adequately low value.

This approach has proven to be very successful in solving a number of problems, including channel routing [Keefe 87] [Joobbani 86b] and transistor placement [Kollaritsc 85] [Sharman 86]. It is unfortunately susceptible to a very real problem; that of becoming arrested in a local minimum. It is often the case that during a sequence of iterative improvements the design reaches a state where no single (small) change can reduce the cost function, although the minimum cost has not yet been reached. One well known expert system for symbolic layout, TOPOLOGIZER [Kollaritsc 85], is known to experience this difficulty [Ackland 88], sometimes terminating before a reasonably good layout has been found.

Many techniques for avoiding local extrema are known. Methods which use *backtracking* of some form are able to retreat from local extrema, making a more complete exploration of the search space. Rigorous backtracking techniques such as depth-first and breadth-first searches [Aho 74] are able to guarantee optimal solutions, but only at the expense of (potentially) exploring the *entire* search space. For VLSI design problems, the size of the search space is at least combinatorial in the number of circuit components and interconnections, making exhaustive search techniques impractical.

As we have seen in chapter 2, there exists a technique known as Monte Carlo simulation or Simulated Annealing, which has been demonstrated to be effective in avoiding problems involving local extrema. This immediately suggests the possibility that Monte Carlo techniques may somehow improve the prospects of design completion in expert systems. Further, this also suggests that randomness of other forms may be beneficial to an expert system whose operation is found to be too rigid or shortsighted.

While Simulated Annealing is reported to produce excellent results in many cases, they are all achieved with a large penalty in execution time. A large part of this penalty is incurred during the later stages of the annealing process, when relatively few attempted moves are accepted.

One suggested means of overcoming this problem is presented by Greene and Supowit [Greene 86]. This method, while promising a significant performance improvement over the Metropolis method of Simulated Annealing [Metropolis 53], requires that the result of each possible move be predicted and weighted according to its relative value. Then one of the moves is randomly selected according to the weight it is assigned. While this method assures that no move is ever rejected, it is often necessary to consider a very large number of *potential* moves: for example, when seeking an optimal arrangement of n transistors among n fixed sites, using pairwise interchange, at any given time, there are $n^2 - n$ possible exchanges,[3] each of which must be evaluated before one can be accepted. For even moderate values of n, this can represent a prohibitive overhead.

Another means of overcoming this problem is to use domain-specific heuristics to select moves which *might* be expected to yield favourable results. In order to preserve the underlying principles of Simulated Annealing, these heuristics should be

[3] In cases where more complex interchange schemes are employed, the number of alternatives my grow as large as $n!$.

employed sparingly at very high temperatures [4] (when adverse changes are accepted with high probability), and can be invoked more frequently as the temperature declines. This closely emulates the behaviour of the Simulated Annealing heuristic, which accepts "negative" moves with high probability in the initial stages of the optimisation process, and then becomes progressively greedier as simulation time advances.

4.2 The STANLEY Expert Systems

The notion of employing Simulated Annealing as a meta-heuristic in the control mechanism of an expert system has been tested by the author using a pair of simple expert systems for transistor placement in VLSI leaf cells. These expert systems, henceforth referred to as the STANLEY [5] expert systems, are based upon Sharman's SPLAT [Sharman 86].

At the lowest level, these expert systems are highly similar, both employing a form of Force Directed Exchange (see section 2.4.1.2) as their primary optimisation heuristic, along with a number of rules for selecting probable pairs of candidate transistors for exchange. The difference appears in the control level of these expert systems: the first expert system naively performs a greedy optimisation, always trying the exchange it believes to be most profitable and never accepting an unprofitable exchange; the second has a few additional heuristics intended to occasionally generate arbitrary exchanges, exchanges which it has no particular reason to believe profitable, or for which there may exist reasons to believe that it is *unprofitable.* The latter of these two expert systems, which will henceforth be referred to as the *Optimising Expert*, will occasionally accept "negative" or "unprofitable" exchanges

[4] For a descriptiuon of the use of "temperature" in Simulated Annealing, please refer to Section 2.3.2.2 on page 19.

[5] "STANLEY" is an acronym for "STimulated ANnealing Layout Expert".

according to a well defined annealing schedule, while the former expert system, which will henceforth be referred to as the *Conventional Expert* always rejects negative exchanges without question.

The objective in equipping the *Optimising* Expert with the ability to generate arbitrary moves and to accept unprofitable changes is to provide a simple mechanism by which the expert system can back away from "solutions" which do not provide a global optimum.

In order to perform a good comparison of these two systems, the structure and logic of the Conventional Expert system will first be described, since the Conventional Expert system provides the basis for the Optimising Expert system. Once this has been done, the structure and logic of the Optimising Expert will be presented.

4.2.1 Implementation of The STANLEY Experts

The STANLEY expert systems are tools for the automatic placement of transistors in VLSI leaf cells. Both operate using an iterative refinement heuristic based upon Force Directed Interchange of transistors. In effect, this means that both expert systems employ an underlying algorithm resembling the following:

1. Generate an arbitrary initial placement.
2. Compute the quality of the Current Placement by evaluating `Old-Cost = Cost(Current-Layout)`.
3. Select a pair of transistors and exchange them.
4. Compute the quality of the New Placement by evaluating `New-Cost = Cost(New-Placement)`.
5. If the difference between placements is acceptable, as determined by the function `Accept(`Δ`Cost)`, accept the New Placement. Otherwise, undo the effects of the exchange just attempted.

6. Unless Completion Criteria have been met, repeat from Step 2.

This algorithm is essentially a basic iterative refinement; it is the implementation of the `Cost()` function, `Accept()` function, and Completion Criteria which make this algorithm a Force Directed Exchange procedure. As the `Cost()` function and the selection procedure (which chooses pairs of transistors to exchange) are common to both of the STANLEY expert systems, they will be described first. The peculiarities of each expert will be presented separately.

The STANLEY experts are implemented partly in ART 3.0 and partly in Symbolics Common Lisp. The use of Lisp code was maximised for the sake of performance. The portions of these experts implemented in ART were coded using the macros of the `ESWE` environment, which is the subject of Chapter 3.

4.2.1.1 Cost Function

The cost function employed by the these experts computes a "force analog" based upon the interconnections between pairs of transistors. The term force analog is used because the forces computed between transistors are based upon the analogy of stretching springs between each pair of terminals which are to be electrically connected. The objective of the simulation is to find the configuration in which the sum of attractive forces between transistors is at a minimum, thus producing what is hoped to be a good balance between minimal area and minimum wire length.

For the most part, the force between two electrically connected transistors is a vector quantity whose scalar component is a quadratic function of the distance between the centers of the transistors, that is, $|\vec{F}| \propto r^2$. Because it is intended that wires will eventually be added to the circuit using Manhattan rules (i.e. each wire segment must be parallel to either the horizontal or vertical axis), and to simplify the logic in the rules which reason about forces, the force between a pair of transistors is represented as two separate components, one on the vertical and the other on

the horizontal axis. The vector portions of each component are either positive to indicate that a transistor is being pulled "up" (in the increasing x-direction) or to the "right" (in the increasing y-direction), or negative to indicate the transistor is being pulled "down" or "left".

The *cost* for a given configuration is the sum of all *scalar* components of the *net* force on each transistor. While the cost is being computed, the net force for each transistor is recorded for use by the heuristics which select pairs of candidate transistors to exchange.

Several problem-specific heuristics are "compiled" directly into the cost function. Forces between transistors are weighted according to the layer on which the connection is anticipated to lie. The style of layout to which the expert systems are expected to adhere requires the respective placement of power and ground rails (rectangles of conductive material which distribute the power and ground voltages to the circuit) across the entire top and bottom edges of the circuit; therefore, for connections between a transistor and a power or ground rail, only a vertical component is computed, since wherever the transistor lies, power and ground are always available directly above and below it.

4.2.1.2 Selection Procedure

As with the `Cost()` function, the procedure for selecting pairs of transistors to exchange is essentially the same for both the Conventional and Optimising STANLEY Experts. The primary objective of this procedure is to select either two transistors to exchange positions, or a transistor and an empty space, with the hope of finding a transposition which results in reducing the total force.

Under the Force Directed Exchange paradigm, the layout area is arranged into a fixed (and relatively small) number of rows and columns, which define a grid.

Transistors may be placed only on lattice points in the grid, and no two transistors may occupy the same location.

The design style followed by the STANLEY expert systems places a number of constraints on the freedom of movement of transistors, many of which are reflected in the heuristics of the selection procedure. Primary among these is the requirement that all P-type transistors be placed in the North (upper) half of the circuit and all N-type transistors be placed in the South (lower) half. This restriction is honoured (if possible) by procedure which produces the initial (arbitrary) placement; subsequently, it is enforced by allowing *exchanges* only between transistors of similar type, and *moves* (the "exchange" of a transistor and an empty site) only when the transistor will remain in the appropriate region of the circuit.

The STANLEY experts both maintain a queue of "pending" recommendations for moves; the Conventional Expert treats this as a LIFO (Last In First Out) queue while the Optimising Expert manipulates it rather differently. Initially, the queues are empty; the first time a suggestion for a move is required, a number of reasonable (and in the case of the Optimising Expert, a few unreasonable) suggestions are generated, and placed in the queue, in reverse order of expected success. One suggestion is then removed from the queue, and placed in the ART knowledge base, to be acted upon by the rule-based portions of the expert system.

Subsequently, when a suggestion for a transposition is required, the queue is inspected, and if not empty, a new suggestion is made available for action. To keep the suggestions in the queue up-to-date, when a transistor is successfully moved, or an empty site is successfully occupied, all suggestions involving that transistor or site are removed from the queue. Further, in the case of the Optimising Expert, if the queue has not been re-generated within a specified number of attempted moves

(currently 15), it is emptied, and new suggestions are generated. This approach to move generation is advantageous for several reasons:

- The heuristics for generating suggested moves produce several simultaneous alternatives. Placing all of these alternatives directly into the ART database can be made, with some effort, to result in very similar behaviour (since ART has a natural tendency to operate on most recent data first), but at a considerable penalty to performance, due to the large number of Working Memory Changes required. It is awkward, although not difficult, to cause ART to act on suggestions in LIFO order, however, the main alternative is to allow ART free reign, in which case it will attempt to act on all suggestions simultaneously.

- By having the suggestions queue managed outside ART, it is possible to modify the way in which elements are placed in and removed from the queue with relatively little difficulty. The Optimising Expert, for example, redefines the mechanism for propagating elements from the queue into the ART database.

- If new suggestions were computed for every attempted move, a rejected move will lead to infinite looping, as the new set of suggestions will be identical to the old.

- A successful move has only a localised effect on the forces on transistors. Most suggestions remain equally valid even after several moves have been accepted. To increase the validity of remaining suggestions, suggested moves which involve a transistor sharing a common network with a successfully moved transistor may be removed from the queue.

4.2.1.3 Backtracking

The Optimising Expert is intended to use a very rapid "cooling schedule" [6]. This results in lower computation times, but often causes the Optimising Expert to terminate with a design state which may not be the lowest-cost design state located so far. Since *optimal* results are *not* being sought, the rapid cooling is acceptable, but only if it is possible to return to the lowest-cost design state after the expert terminates. To facilitate such a return, the Optimising Expert is equipped with a simple backtracking facility.

Each time a change is accepted, it is pushed onto a queue. Whenever the expert system enters a new design state whose cost is less than or equal to the lowest cost yet found, the queue of accepted changes is cleared. By undoing each change in the queue when the expert's Simulated Annealing stage completes, it is possible to return to the state of lowest cost.

4.2.2 The Conventional Expert

The Conventional Expert uses a greedy iterative improvement approach common to many design oriented expert systems. Consequently, the `Accept()` function, which determines whether a move under consideration is accepted, is very simple: a move is accepted if and only if the cost for the new configuration is lower than the cost for the old configuration. Attempted moves which present a zero change in cost should not be accepted, and *are not* accepted due to the risk of oscillating forever between two equal-cost states.

The termination criteria for the Conventional Expert are also simple: iteration is complete either when the cost reaches zero (a most unlikely occurrence), or when

[6] A "cooling schedule" is the schedule used to drop the temperature during Simulated Annealing. A common technique is to drop the temperature by a fixed percentage (usually less than fifteen percent) each time a determined number of changes has been accepted.

no moves have been accepted in a fixed number of tries (the threshold is currently forty).

4.2.3 The Optimising Expert

The Optimising Expert is substantially similar to the Conventional Expert described previously, and uses the same basic approach. Force Directed Exchange remains the primary underlying heuristic and the cost function is precisely the same as that used by the Conventional Expert.

The Optimising Expert *extends* the Conventional Expert's iterative refinement technique, by adding stochastic control mechanisms inspired by Simulated Annealing. The incentive for employing Simulated Annealing (or any other stochastic heuristic) in an expert system is to improve its ultimate results by partially relieving the constraints of decisions made early in its execution.

It should be noted that the control mechanism used in the Optimising Expert is *based upon* Simulated Annealing, but has been modified in ways which are not completely compatible with Simulated Annealing. In particular, the `Accept()` function (described in the next section) places a fixed rather than stochastic limit on the magnitude of backward moves accepted at a given temperature. Also, the cooling schedule used in the Optimising Expert is *very* rapid, using only ten iterations per temperature step.[7]

The use of stochastic heuristics in the Optimising Expert has lead to a number of changes in the functions used by the Conventional Expert. The following paragraphs describe these differences:

[7]This is in contrast to more conventional approaches to Simulated Annealing which may use as many as *several thousand* iterations per cooling step.

4.2.3.1 Changes to the Accept() Function

One of the most fundamental changes brought about by the use of the Simulated Annealing heuristic in the Optimising Expert is the introduction of the concept of "temperature". Even though temperature in the usual sense may have little or no meaning in the context of the problem solved by the expert system, its inclusion is essential to Simulated Annealing. The Optimising Expert includes a variable for temperature, $\mathcal{T}$, which is used to represent the 'pseudo-temperature' of the annealing process, that is, as a measure of the "disorder" or permitted randomness in the layout procedure.

Under the Simulated Annealing paradigm, moves which improve (i.e., reduce) the cost of the configuration are always accepted. Moves which do not improve the cost are sometimes accepted, according to a probability function based on an exponential function of pseudo-temperature, $\mathcal{T}$ and the change in cost, ΔC. The Optimising Expert uses temperatures in the range $0 \leq \mathcal{T} < 1$. As time progresses, the temperature is lowered exponentially, making the likelihood of accepting unproductive moves ever diminishing. The `Accept()` function used by the Optimising Expert is shown in Fig. 4.1.

```
(defun Accept (old-cost new-cost temperature)
   (if (or (< new-cost old-cost)
           (and (> temperature (random 1.0))
                (>= (expt temperature 0.7)
                    (/ (- new-cost old-cost) old-cost))))
      (return 'YES)  ;; If acceptable, say YES
      (return 'NO)   ;; else, say NO.
   )
)
```

Figure 4.1: Accept() Function for Optimising Expert.

In the acceptance function, the *random* function is invoked to select a random number in the range [0, ..., 1). This number is compared against the current value

of the pseudo-temperature to determine whether a poor move *might* be tolerated. If this test indicates that an increase in cost may be permitted, a cost ratio, $\frac{\Delta C}{C}$ (the percent change in cost) is then compared to the temperature to control the magnitude of the cost increase. As is normally the case with determining the cooling schedule for Simulated Annealing the parameters of this acceptance function have been determined empirically.

Unlike the usual Simulated Annealing acceptance function, which determines the probability of accepting backward moves as a function related to $e^{-\frac{k\Delta C}{T}}$, the Optimising Expert will not assign *all* moves a non-zero probability of being accepted. This behaviour is acceptable, as the objective in this case is not to generate optimal solutions, but to avoid local minima.

4.2.3.2 Changes to the Suggestions Queue

In many expert systems languages, including ART 3.0, the last fact asserted in the working memory is the most likely to cause a rule to fire. Specifically when a single rule has several instances or activations available to fire (caused by, for example, the presence of several suggested exchanges of transistors), the instance of the rule activated by the *most recent* fact in the working memory will be the first rule fired.

The consequence of this behaviour in test programs was that the *last* recommended change was *always* acted upon, and the time spent generating any other recommendations was often wasted. As described earlier in this section, the problem was overcome, in part, by collecting all the recommendations in a LISP structure (independent of ART's working memory), and then selecting one recommendation and placing it in working memory

The Conventional Expert treated this structure as a queue, producing behaviour very similar to the default behaviour of ART; however, the use of this queue helped

improve performance by reducing traffic through the working memory and by replacing a number of rules with much faster *compiled* Lisp.

The Optimising Expert took somewhat more liberty with the operation of the queue: rather than removing the first element of the queue when a new suggestion was needed, the Optimising Expert randomly selects an element from the queue.

The function used to select elements from the queue may be seen in Fig. 4.2. The probability distribution of this function is dependent on simulation time (pseudo-temperature). Initially, at a very high pseudo-temperature, the function assigns almost equal probability of selection to each element of the queue, but as simulation time advances and the value of $\mathcal{T}$ declines, it is designed to give increasing *preference* to the first elements of the queue. At any time, however, each element in the queue may be selected, with probability greater than zero.

While this technique for selecting suggestions from the queue does not directly contribute to avoiding local extrema in the cost function, it can be of benefit to many programs by preventing one small group of heuristics from dominating the rest through some artifact of the "arbitrary" selection of the next rule to fire.

Upon inspection of the function *choose-element* in figure 4.2, one may notice the fact that the index of the element selected is *explicitly* limited to one less than the length of the queue. In fact, it is quite reasonable to believe the explicit limitation set by the function `max` is unnecessary, as the value of `(random 1.0)` should be a number between 0 (inclusive) and 1.0 (exclusive).

Table 4.1 shows the distributions of elements selected from a queue of length 20 under a variety of temperatures. The figures presented in this table represent the result of ten million trials of the function in figure 4.2, *without the explicit upper bound on the selected index.* Careful inspection will show that the sum of all entries for Trial 2 (center column) in Table 4.1 is only 9,999,999; an out-of-range index was

```
  ;;; Randomly select an element from a queue, giving preference
  ;;; to elements at the head.

(defun choose-element (queue)
  (check-type queue (list))  ;; Make certain QUEUE is a valid list
  (let ((queue-length (length queue))
        (selection (max (1- queue-length)
                        (floor (* queue-length
                                  (expt (random 1.0) (expt temperature 0.7))))
                   )))
     (if (null queue)
         (return NIL)
         (return (nth selection queue))))
)


  ;;; Get an element from the queue bound to QUEUEPTR, and place it in the
  ;;; ART database. Remove the all copies of the element from the queue.

(defun get-selection (QUEUEPTR)
   (check-type QUEUEPTR (symbol))

   (let ((element (choose-element (symbol-value QUEUEPTR))
      (if (null element)                  ;;; If the queue was empty,
          (eval '(assert (out-of-moves)) ;;; Tell ART, otherwise,
          (eval `(assert ,element)))     ;;; place element in ART's database

      (set QUEUEPTR (remove element (symbol-value QUEUEPTR) :test #'equal))
)
```

Figure 4.2: Weighted Random Selection. The function `choose-element` is used to randomly select an element from a queue, giving preference to elements nearest the head. Note the use of `eval` and ` (backquote) macros in `get-selection`; this is necessary because `assert` is a *macro* that cannot be expanded until run time.

selected in one trial in ten million, likely as the result of a floating point rounding error. Placing the explicit limit on the selected element corrects this error, without skewing the probability curve.

4.2.4 Changes to Move Generation

The final change to the Optimising Expert prompted by the inclusion of stochastic control appears in the mechanism for proposing changes to the current layout. In order to help avoid arrest in local minima, the function which generates recommen-

Index	Times Selected			Index	Times Selected		
	Trial 1	Trial 2	Trial 3		Trial 1	Trial 2	Trial 3
0	499 816	68 102	4	10	499 866	542 257	15 391
1	500 999	147 039	0	11	499 662	575 964	35 154
2	499 260	207 512	0	12	500 505	608 624	74 423
3	500 628	260 523	0	13	500 378	641 706	148 085
4	500 402	307 998	5	14	498 902	674 063	280 527
5	500 591	351 559	59	15	499 610	702 733	510 054
6	498 906	393 850	215	16	499 012	733 959	895 147
7	500 291	433 850	709	17	500 670	761 270	1 517 133
8	500 745	469 622	2417	18	499 441	791 971	2 500 417
9	499 935	507 276	6372	19	500 361	820 121	4 013 883

Samples: 10 000 000

Trial 1: tempertaure = 1.0

Trial 2: tempertaure = 0.48202905

Trial 3: tempertaure = 0.03727593

Table 4.1: Distribution of selected elements in a 20 element queue at very high, intermediate, and very low temperatures.

dations for moves has been extended to occasionally suggest *random* changes: either the exchange of a random pair of transistors or moving a random transistor to a randomly selected empty space.

4.3 Runtime Comparison of STANLEY Experts

In order to determine which of the STANLEY experts produces best results, a series of comparative tests was run. Each expert was used to produce transistor placements for three standard cells: a D-type flip-flop (D-TYPE); a 4-to-1 multi-

plexor (MUX4); and an exclusive-or gate (XOR). Before presenting the results of the tests, the specifications of the test circuits will be given.

Because the *Conventional* Expert system is completely deterministic, this test procedure would normally produce ten *identical* placements for each test circuit. In an effort to produce more representative data, the initial placement generated by the Conventional Expert is randomly permuted by exchanging the positions of two pairs of p-transistors and two pairs of n-transistors immediately prior to commencing the iterative refinement phase of the placement procedure.

4.3.1 D-type Circuit

The D-type flip-flop is a fully-complementary CMOS standard cell, with the input on the left and output on the right. The clock signals, PHI and PHIBAR are constrained to enter the cell at the center of the top and bottom edges respectively. This cell originates from the implementation of Joyce's TAMARACK [Joyce 88] microprocessor. Following is the network description for the cell, as given to the STANLEY experts. The topology of this circuit is illustrated schematically in Fig. 4.3.

```
(defcell D-TYPE
  :size (4 4)
  :networks (PWR GND PHI PHIBAR IN OUT alpha beta gamma)
  :ports ((NORTH (pwr.w PWR metal-2)
                 (PHI.n PHI metal-1)
                 (pwr.e PWR metal-2))
          (SOUTH (GND.w GND metal-2)
                 (PHIBAR.s PHIBAR metal-1)
                 (GND.e GND metal-2))
          (EAST  (OUT.e OUT metal-1))
          (WEST  (IN.w IN metal-1))
          (INTERIOR )
          (CONST (PWR.e || NORTH-EAST)
                 (PWR.e =  NORTH-EAST)
                 (PWR.w || NORTH-EAST)
                 (PWR.w =  SOUTH-WEST)
                 (GND.e || SOUTH-WEST)
```

```
                (GND.e =  NORTH-EAST)
                (GND.w || SOUTH-WEST)
                (GND.w =  SOUTH-WEST)
                (PHI.n = PHIBAR.s)
                (IN.w || OUT.e)))
   :TRANSISTORS (mux-p-1 mux-n-1 mux-p-2 mux-n-2 inv-p-1
                 inv-p-2 inv-p-3 inv-n-1 inv-n-2 inv-n-3)
)

;;        NAME      TYPE         GATE   DRAIN SOURCE
;;       ======= ============ ====== ===== ======
(deftran mux-p-1 p-transistor phibar in    alpha)
(deftran mux-n-1 n-transistor phi    in    alpha)
(deftran inv-p-1 p-transistor alpha  PWR   beta)
(deftran inv-n-1 n-transistor alpha  GND   beta)
(deftran inv-p-2 p-transistor beta   PWR   gamma)
(deftran inv-n-2 n-transistor beta   GND   gamma)
(deftran inv-p-3 p-transistor beta   PWR   out)
(deftran inv-n-3 n-transistor beta   GND   out)
(deftran mux-p-2 p-transistor phi    gamma alpha)
(deftran mux-n-2 n-transistor phibar gamma alpha)
```

4.3.2 4-to-1 Multiplexor Circuit

The second cell used to test the STANLEY expert systems is a simple four to one multiplexor, again using fully complementary CMOS conventions. (This cell, however, uses non-restoring logic, and should therefore be used with caution.) Following is the full network description given to the STANLEY experts.This circuit is derived from the color plates of Pucknell and Eshragian's text on VLSI design [Pucknell 88]. The topology of the cell is illustrated in the schematic diagram of Fig. 4.4.

```
(defcell MUX4
  :size (8 4)
  :networks (PWR GND S0 S0bar S1 S1bar I0 I1 I2 I3 OUT alpha beta
             gamma delta epsilon larry moe curly)
  :ports ((NORTH (pwr.w PWR metal-2)
                 (S1.n S1   poly)
                 (s1bar.n S1bar poly)
                 (S0.n S0 poly)
                 (S0bar.n S0bar poly)
```

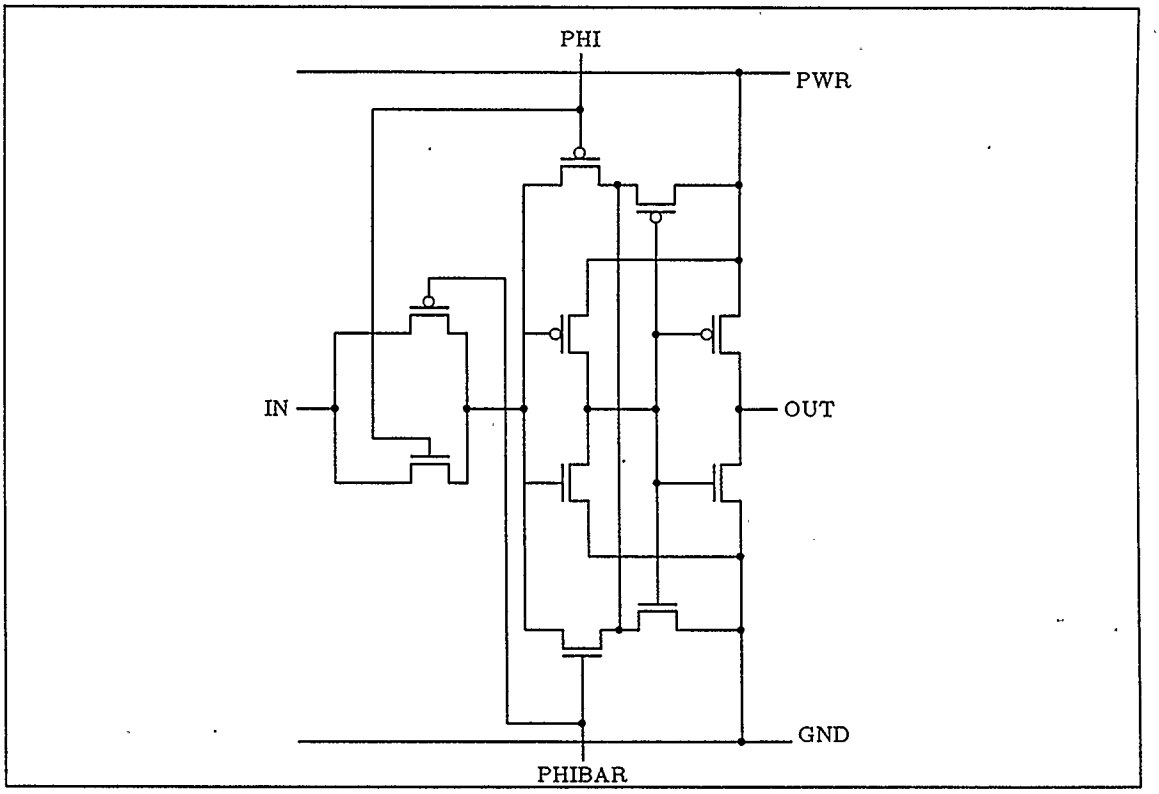

Figure 4.3: Schematic Diagram of D-TYPE Test Circuit

```
        (pwr.e PWR metal-2))
(SOUTH (GND.w GND metal-2)
        (S1.s S1   poly)
        (S1bar.s S1bar poly)
        (S0.s S0 poly)
        (S0bar.s S0bar poly)
        (GND.e GND metal-2))
(EAST   (OUT.e OUT metal-1))
(WEST   (I3.w I3 metal-1)
        (I2.w I2 metal-1)
        (I1.w I1 metal-1)
        (I0.w I0 metal-1))
(INTERIOR )
(CONST (PWR.e || NORTH-EAST)
        (PWR.e =  NORTH-EAST)
        (PWR.w || NORTH-EAST)
        (PWR.w =  SOUTH-WEST)
        (GND.e || SOUTH-WEST)
        (GND.e =  NORTH-EAST)
        (GND.w || SOUTH-WEST)
```

```
                (GND.w =  SOUTH-WEST)
                (S0.n = S0.s)
                (S1.n = S1.s)
                (S0bar.n = S0bar.s)
                (S1bar.n = S1bar.s)))
  :TRANSISTORS (p-1 p-2 p-3 p-4 p-5 p-6 p-7 p-8
                n-1 n-2 n-3 n-4 n-5 n-6 n-7 n-8)
)

;;;             TYPE       GATE    SOURCE   DRAIN
;;;          ============ ====    ======   =====
(deftran p-1 p-transistor S1      I0       alpha)
(deftran p-2 p-transistor S0      alpha    OUT)
(deftran p-3 p-transistor S1      I1       beta)
(deftran p-4 p-transistor S0bar   beta     OUT)
(deftran p-5 p-transistor S1bar   I2       gamma)
(deftran p-6 p-transistor S0      gamma    OUT)
(deftran p-7 p-transistor S1bar   I3       delta)
(deftran p-8 p-transistor S0bar   delta    OUT)

(deftran n-1 n-transistor S1      I3       epsilon)
(deftran n-2 n-transistor S0      epsilon  OUT)
(deftran n-3 n-transistor S1      I2       larry)
(deftran n-4 n-transistor S0bar   larry    OUT)
(deftran n-5 n-transistor S1bar   I1       moe)
(deftran n-6 n-transistor S0      moe      OUT)
(deftran n-7 n-transistor S1bar   I0       curly)
(deftran n-8 n-transistor S0bar   curly    OUT)

(def-bus-net S1)
(def-bus-net S0)
(def-bus-net S1bar)
(def-bus-net S0bar)
```

The MUX4 circuit has one fundamental difference from the other two test circuits. The use of **def-bus-net** declarations in this circuit instruct the placement experts that the networks associated with the select lines (`S0 S1 S0bar` and `S1bar`) are to be regarded as buses. The placement experts react to these declarations by placing additional emphasis on these networks, making it a priority to keep each transistor as close as possible to the select line it uses.

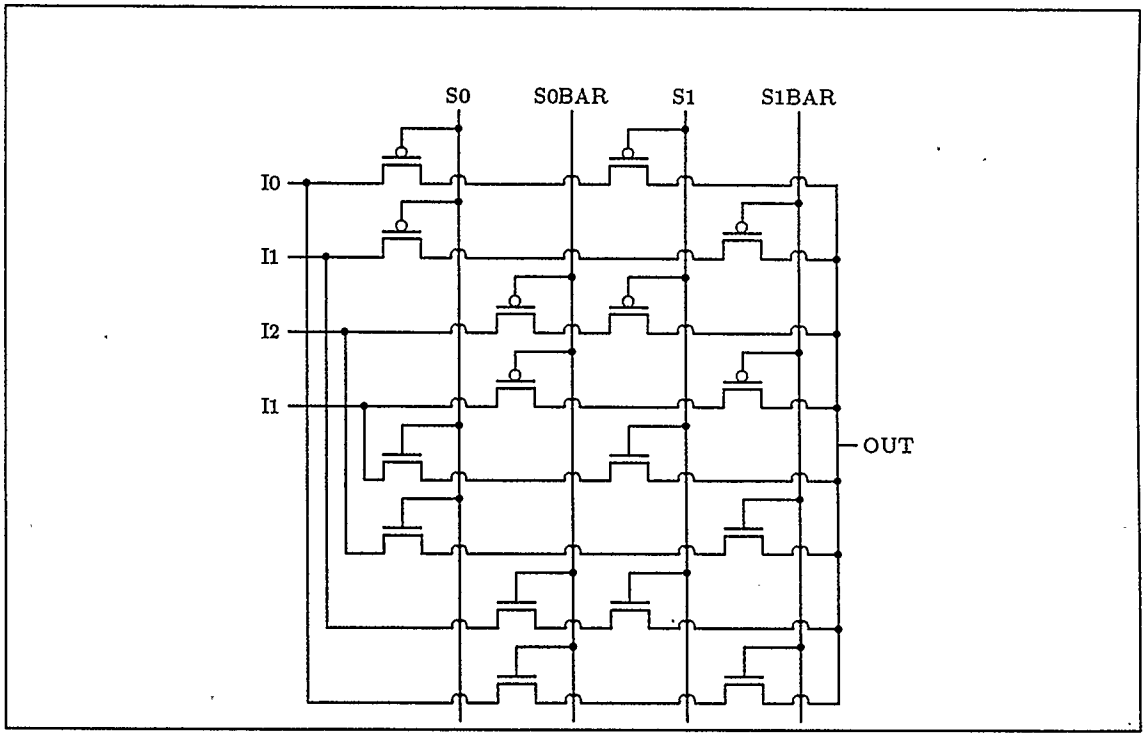

Figure 4.4: Electrical Schematic Diagram of MUX4 Test Circuit

4.3.3 Exclusive-Or Test Circuit

The final test circuit used is a simple two input exclusive or gate. This circuit is derived from the color plates of Pucknell and Eshragian's text on VLSI design [Pucknell 88]. The raw input given to the expert systems is listed below; the topology of the circuit is illustrated in the schematic diagram of Fig 4.5.

```
(defcell XOR
  :size (4 4)
  :networks (PWR GND A B OUT alpha beta gamma delta)
  :ports ((NORTH (pwr.w PWR metal-2)
                 (A.n A poly)
                 (B.n B poly)
                 (OUT.n OUT poly)
                 (pwr.e PWR metal-2))
          (SOUTH (GND.w GND metal-2)
                 (A.s A poly)
                 (B.s B poly)
```

```
                (OUT.s OUT poly)
                (GND.e GND metal-2))
          (EAST )
          (WEST )
          (INTERIOR )
          (CONST (PWR.e || NORTH-EAST)
                 (PWR.e =  NORTH-EAST)
                 (PWR.w || NORTH-EAST)
                 (PWR.w =  SOUTH-WEST)
                 (GND.e || SOUTH-WEST)
                 (GND.e =  NORTH-EAST)
                 (GND.w || SOUTH-WEST)
                 (GND.w =  SOUTH-WEST)
                 (A.n = A.s)
                 (B.n = B.s)
                 (OUT.n = OUT.s)))
  :TRANSISTORS (p-1 p-2 p-3 p-4 p-5 n-1 n-2 n-3 n-4 n-5)
)

;;;       NAME     TYPE         GATE  SOURCE  DRAIN
;;;       ==== ============ ====  ======  =====
(deftran p-1  p-transistor A     PWR     alpha)
(deftran p-2  p-transistor B     PWR     alpha)
(deftran p-3  p-transistor A     PWR     beta)
(deftran p-4  p-transistor B     beta    gamma)
(deftran p-5  p-transistor gamma alpha   OUT)

(deftran n-1  n-transistor A     GND     delta)
(deftran n-2  n-transistor B     delta   OUT)
(deftran n-3  n-transistor A     GND     gamma)
(deftran n-4  n-transistor B     GND     gamma)
(deftran n-5  n-transistor gamma GND     OUT)
```

4.3.4 Testing Procedure

In order to compare the performance of the two expert systems, each one is made to repeat the transistor placement for each of the three test circuits ten times in succession. For each repetition, the initial cost, final cost, percentage difference from the minimal cost[8], and elapsed time for placement[9] are recorded.

[8] "Minimal cost" is the result of the best test run.

[9] This does not include the time required for initial placement, since each expert uses identical initial placement heuristics.

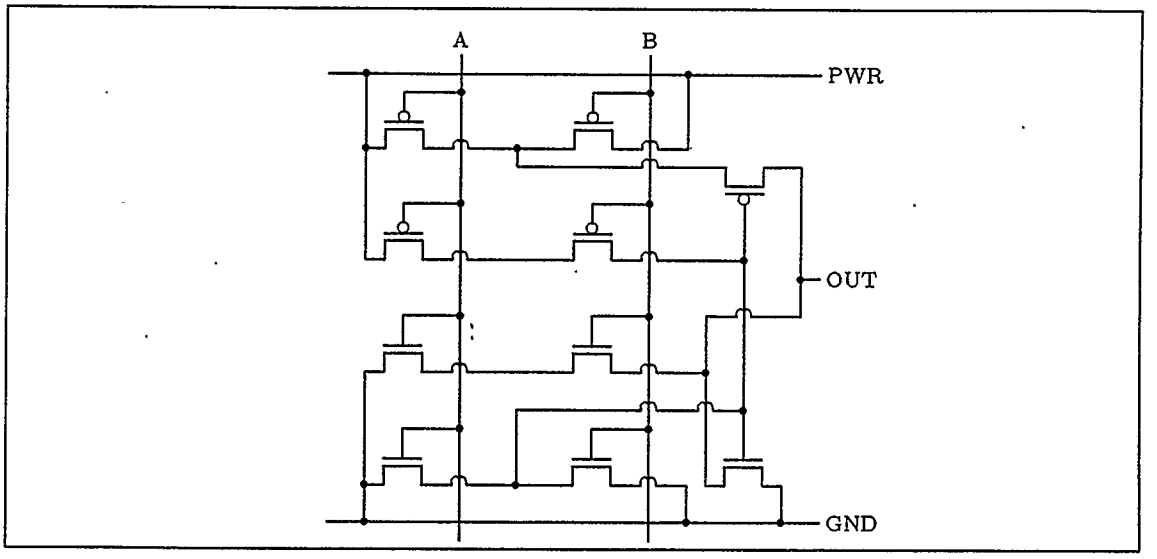

Figure 4.5: Electrical Schematic Diagram of XOR Test Circuit

The data for the Optimising Expert are displayed in Tables 4.2 through 4.4, while the data for the Conventional Expert are displayed in Tables 4.5 through 4.7, all of which are located at the end of this chapter.

4.3.5 Observations

1. Average computation times for the Optimising Expert are much greater than for the Conventional Expert. To a degree, this is a desired result, as the objective of the experiment is to prevent premature termination in expert systems.

2. In many instances, much of the time spent in the Optimising Expert is wasted in continued searches for better solutions, even after a very good solution has already been found. This is because it is very difficult to produce good termination criteria. One termination criterion, which causes the expert to quit after thirty unsuccessful moves, proved to be too strict; occasionally causing the Optimising Expert to terminate while its pseudo-temperature remains relatively high.

3. The test results for the D-TYPE circuit are quite clear. The Optimising Expert uniformly and decisively outperformed the Conventional Expert, to the extent that the even *best* solutions produced by the Conventional Expert were inferior to the *worst* solutions produced by the Optimising Expert.

4. The results produced by the Optimising Expert are remarkably consistent, having a standard deviation (σ_n) of less than one tenth of one percent of the minimum cost. The Conventional Expert was considerably less consistent, with a standard deviation of more than ten percent of the minimum cost; more than one hundred times the standard deviation of the Optimising Expert.

5. As with the D-TYPE circuit, the Optimising Expert considerably outperformed the Conventional Expert. Only occasionally did the Conventional Expert produce results of the quality produced by the Optimising Expert, and in all cases, the former failed to excel the latter. The results for the XOR circuit, while conclusive, are not as striking as for the D-TYPE circuit. Here, the standard deviation of the results produced by the Optimising Expert was nearly four percent of the minimum cost; the standard deviation for the Conventional Expert, however, was more than twelve percent.

   If this small sampling proves to be representative, it leads to the expectation that the Optimising Expert will usually produce good results, while the Conventional will at least occasionally produce *very* bad results.

6. The test data for MUX4 circuit are strongly counter to expectations. As expected, the Optimising Expert required considerably more (a factor of 3.3 times) execution time than did the Conventional Expert, but produced results which had, on average, about 10% *higher* cost than those produced by the Conventional Expert.

   Possible explanations for this behaviour include:

- The Optimising Expert uses a very rapid and *fixed* cooling schedule. The number of moves the Optimising Expert accepts between drops in pseudo-temperature does not depend upon the size of the circuit being compiled. Revising the cooling schedule *will* ameliorate this situation, but at the expense of even greater computation times.

- In addition to a rapid cooling schedule, the Optimising Expert is subject to the same termination criteria as the Conventional Expert; specifically, after forty consecutive moves have been rejected, each expert terminates. This behaviour will lead to premature termination of the Optimising Expert. The purpose of continuing this practice in the Optimising Expert is to *limit* the time spent annealing. Modifying this criterion to take effect only when the probability of accepting backward moves is less than a given threshold (roughly 3% to 5% seems appropriate) will make abrupt premature termination of the Optimising Expert less likely, and effectively assure results which are, at worst, comparable to the Conventional Expert.

- The MUX4 circuit contains a total of sixteen transistors which are to be laid out in a grid of eight rows and four columns. The large amount of empty space considerably decreases the likelihood of the Conventional Expert halting prematurely.

- It is very likely that even if the approach used by the Optimising Expert is intrinsically superior to the approach used by the Conventional Expert, there will still exist some circuits for which the Conventional Expert will excel. Because of its use of "bus-nets", the MUX4 test-circuit is very likely to be one of these.

In an effort to test these hypotheses, the Optimising Expert was revised to use a slower cooling schedule and tried again. The MUX4 circuit was also revised, to allow a placement grid of only four rows and four columns. (In fact, neither expert actually used more than four rows in its final placements for the MUX4 circuit.)

Tests done using the revised MUX4 circuit are summarised in Tables 4.8 and 4.9. In these tests, the Optimising Expert produced results whose final costs were approximately equivalent to those produced by the Conventional Expert.

Test runs of the Revised Optimising Expert, which used a significantly slower cooling schedule, produced results comparable to those produced by the Conventional Expert for the original MUX4 circuit. That one of these tests still produced a relatively high-cost result (although lower than the peak produced by the greedy expert) suggests that the cooling schedule is not solely to blame. Tests of the Revised Optimising Expert on the Revised MUX4 circuit were conducted with relaxed termination criteria, in which the expert was halted only after one hundred and fifty consecutive moves have been rejected. The four test runs for this case all produced *very* good results, but with very large computation time.

It should be noted that even with slower cooling, the lower bound on cost for the MUX4 test circuit was not improved by the Revised Optimising Expert. The consistency of results was, however, greatly improved by slower cooling.

7. The Revised Optimising Expert, despite a slower cooling schedule, still suffers early terminations caused by the termination heuristic described above; removing or modifying this heuristic will lead to better solutions.

8. Both expert systems used in this study are strictly experimental. No effort has been made to optimise the performance of either expert system in any way.

Re-implementing these expert systems using a customised inference engine, or preferably rewriting these expert systems in a language such as LISP or C with a careful view to efficiency, can considerably improve the execution times recorded in this thesis, perhaps by a factor of as much as one hundred times.

9. It is worth noting that both experts were able to produce placements for the MUX4 circuit which had slightly *lower* costs than the relative placement depicted in the schematic of Figure 4.4.

10. Due to limitations on available computation time, a small number of samples have been collected for this study. In order to be statistically valid, more tests must be conducted.

4.3.6 Conclusions

The Optimising Expert presented in this chapter, with suitable modifications to its cooling schedule, has been demonstrated to produce results ranging from comparable to vastly superior to those of the Conventional Expert. It is worth noting that the computation time consumed by the Optimising Expert was, in most cases, much larger than that consumed by the Conventional Expert.

Additional tests were also run comparing the original (unrevised) Optimising Expert to the Conventional Expert, using a revised version of the MUX4 circuit. For the revised circuit, in which transistors must be placed in four rows and four columns (i.e., without empty spaces), the two experts performed almost equally, although, because the Optimising Expert still employed an excessively rapid cooling schedule, the Conventional Expert was able to produce marginally superior results.

Conclusions which may be drawn from these tests are enumerated below:

1. Conventional Expert systems which use greedy heuristics can, and do, arrest at local minima. The consequence of such a premature termination is greatly inferior results for at least some inputs.

2. Expert systems can profitably make use of stochastic heuristics and control structures to avoid arrest at local minima.

3. The appropriate use of the Simulated Annealing heuristic in an expert system is demonstrated to improve best case, worst case, and mean results for at least some data.

4. The Optimising Expert is shown to frequently outperform the Conventional Expert in terms of "cost". Given an adequate cooling schedule, the Optimising Expert will, *at worst*, perform comparably to the Conventional Expert in terms of cost. Therefore, time considerations aside, the Optimising Expert may be safely used to *replace* the Conventional Expert. Further, tests indicate that a very rapid cooling schedule is usually adequate to avoid minima; the Optimising Expert may therefore be used with relatively small penalty in execution time.

5. Difficulties encountered during testing were largely due to problems with cooling schedules and termination criteria; an excessively rapid cooling schedule or overly zealous termination criterion leads to premature termination of the Optimising Expert, resulting in low quality results. Allowing the Optimising Expert to proceed for arbitrarily long periods of time will avoid this problem, but at great expense in terms of computation. Using the Conventional Expert *in conjunction with* the Optimising Expert allows an approximate upper bound on acceptable solution quality to be quickly computed; the Optimising Expert can be forced not to terminate before this upper bound is reached, thus assuring results *no worse* than those provided by the Conventional Expert.

6. The use of the Simulated Annealing heuristic in an expert system is demonstrated to increase the computation time needed. The success to which rapidly cooled Simulated Annealing has been applied implies the possibility that other, more simple, stochastic heuristics may be employed with similar effect and less penalty.

   The scheme outlined above, which first computes a rough upper bound, and then limits time spent after this bound is reached, will help to reduce the computational overhead of Simulated Annealing.

Trial	Elapsed Time (s)	Initial Cost	Minimum Cost	Percent Change	Total Moves
1	3973	6379	1502	5.26	910
2	2763	6379	1775	24.39	588
3	3809	6379	1682	17.87	849
4	3374	6379	1445	1.26	719
5	3116	6379	1512	5.96	651
6	3146	6379	2457	72.18	665
7	2284	6379	1908	33.71	429
8	3698	6379	1602	12.33	773
9	3522	6379	1633	14.44	747
10	3244	6379	1880	31.74	662
Mean	3293	6379	1739.6	21.91	699

Expert: Optimising STANLEY Expert

Circuit: 4-to-1 Multiplexor

Table 4.2: Results for 10 Optimised Trials of MUX4 Circuit.

Trial	Elapsed Time (s)	Initial Cost	Minimum Cost	Percent Change	Total Moves
1	2857	956	561.5	0.00	1115
2	2194	956	562.0	0.09	783
3	2009	956	562.0	0.09	736
4	2003	956	561.5	0.00	721
5	2366	956	561.5	0.00	909
6	1266	956	562.0	0.09	414
7	1232	956	563.2	0.30	413
8	2021	956	562.0	0.09	726
9	2164	956	561.5	0.00	726
10	1650	956	561.5	0.00	553
Mean	1976	956	561.9	0.07	710

Expert: Optimising STANLEY Expert

Circuit: D-TYPE Latch

Table 4.3: Results for 10 Optimised Trials of D-TYPE Circuit.

Trial	Elapsed Time (s)	Initial Cost	Minimum Cost	Percent Change	Total Moves
1	1967	1900	748.9	0.00	763
2	2386	1900	748.9	0.00	957
3	1797	1900	782.1	4.43	692
4	1317	1900	783.1	4.57	467
5	2234	1900	848.4	13.29	849
6	1555	1900	789.6	5.44	574
7	1570	1900	748.9	0.00	585
8	1582	1900	789.6	5.44	582
9	2355	1900	783.1	4.57	877
10	1068	1900	748.9	0.00	345
Mean	1738.6	1900	778.1	3.89	669

Expert: Optimising STANLEY Expert

Circuit: Exclusive-Or Gate

Table 4.4: Results for 10 Optimised Trials of XOR Circuit.

Trial	Elapsed Time (s)	Initial Cost	Minimum Cost	Percent Change	Total Moves
1	831.3	8608	1427	0.00	189
2	983.7	7958	1970	38.05	230
3	952.4	9010	1650	15.63	211
4	1268	7392	1430	0.21	297
5	1247	7334	1427	0.00	287
6	1005	7741	1427	0.00	224
7	840.1	7501	1916	34.27	188
8	879.1	7559	1427	0.00	204
9	1061	6400	1437	0.70	234
10	824.7	7291	1664	16.61	178
Mean	989.2	7679	1578	10.55	224

Expert: Conventional STANLEY Expert

Circuit: 4-to-1 Multiplexor

Table 4.5: Results for 10 Greedy Trials of MUX4 Circuit.

Trial	Elapsed Time (s)	Initial Cost	Minimum Cost	Percent Change	Total Moves
1	206.5	985.3	563.3	0.32	84
2	161.9	1151	758.3	35.05	63
3	195.2	1117	563.3	0.32	73
4	174.7	1104	563.3	0.32	73
5	230.0	1069	631.0	12.38	84
6	171.7	1112	563.3	0.32	64
7	217.5	1057	577.0	2.65	85
8	173.6	1143	563.3	0.32	66
9	143.0	1042	590.5	5.17	58
10	141.1	1035	581.3	3.53	55
Mean	181.52	1082	595.5	6.04	71

Expert: Conventional STANLEY Expert

Circuit: D-TYPE Latch

Table 4.6: Results for 10 Greedy Trials of D-TYPE Circuit.

Trial	Elapsed Time (s)	Initial Cost	Minimum Cost	Percent Change	Total Moves
1	166.5	1900	934.9	24.84	65
2	179.4	2239	941.1	25.66	69
3	172.1	2052	984.4	31.45	71
4	279.4	2232	800.6	6.90	109
5	188.0	2091	833.9	11.35	74
6	172.5	1900	934.9	24.84	67
7	246.6	1820	984.4	31.45	97
8	228.9	1884	794.9	6.142	91
9	228.4	2217	748.9	0.00	83
10	234.1	2307	1039.6	38.82	91
Mean	209.6	2064	899.8	20.14	82

Expert: Conventional STANLEY Expert

Circuit: Exclusive-Or Gate

Table 4.7: Results for 10 Greedy Trials of XOR Circuit.

Trial	Elapsed Time (s)	Initial Cost	Minimum Cost	Percent Change	Total Moves
1	2751	3302	1416	4.27	571
2	4509	3302	1366	0.59	773
3	5434	3302	1358	0.00	1148
4	2372	3302	1392	2.50	490
5	4258	3302	1374	1.18	908
6	3506	3302	1380	1.62	734
7	4129	3302	1366	0.59	854
8	4928	3302	1382	1.77	1057
9	3635	3302	1372	1.03	780
10	5023	3302	1366	0.59	1176
Mean	4055	3302	1377	1.41	849

Expert: Optimising STANLEY Expert

Circuit: Revised 4-to-1 Multiplexor

Table 4.8: Results for 10 Optimised Trials of Revised MUX4 Circuit.

Trial	Elapsed Time (s)	Initial Cost	Minimum Cost	Percent Change	Total Moves
1	359.9	2407	1372	1.03	81
2	407.7	3910	1376	1.33	89
3	467.7	3276	1372	1.03	103
4	454.8	3373	1380	1.62	98
5	409.1	3396	1386	2.06	87
6	436.6	2475	1366	0.59	98
7	472.3	3735	1366	0.59	66
8	468.1	3261	1376	1.33	57
9	296.1	3860	1386	2.06	67
10	520.6	3653	1376	1.33	121
Mean	429.3	3335	1376	1.33	87

Expert: Conventional STANLEY Expert

Circuit: Revised 4-to-1 Multiplexor

Table 4.9: Results for 10 Greedy Trials of Revised MUX4 Circuit.

Trial	Elapsed Time (s)	Initial Cost	Minimum Cost	Percent Change	Total Moves
1	7745	6379	1666	16.74	1714
2	8739	6379	1437	0.70	1936
3	11935	6379	1438	0.77	2677

Expert: Revised Optimising STANLEY Expert

Circuit: 4-to-1 Multiplexor

Table 4.10: Results for 3 Optimised Trials of MUX4 Circuit.

Trial	Elapsed Time (s)	Initial Cost	Minimum Cost	Percent Change	Total Moves
1	17263	3302	1366	0.56	4017
2	21824	3302	1358	0.00	5040
3	17099	3302	1372	1.03	3877
4	33418	3302	1364	0.44	7862

Expert: Revised Optimising STANLEY Expert

Circuit: Revised 4-to-1 Multiplexor

Table 4.11: Results for 4 Optimised Trials of Revised MUX4 Circuit.

Chapter 5

Conclusions And Directions for Future Research

5.1 Conclusions

For a number of years now, the progressive growth of complexity and cost for the design of integrated circuits has provided a continuously increasing incentive to automate as much of the VLSI design process as possible. Knowledge-based expert systems are one technique which holds a strong promise for automating portions of the VLSI design problem which previously have not been regarded as candidates for automation. In the past, implementors have found that expert systems are well suited for problems which involve some form of *diagnosis*. In order to apply expert system techniques to design problems, they often find it helpful to employ iterative refinement in the expert system, effectively recasting a problem of *design* as a problem of diagnosis.

Two specific problems found in current expert system paradigms are the propensity for expert systems built upon a "greedy" iterative refinement heuristic to arrest at local minima, thus producing inferior solutions; and the difficulty of imposing structure and synchronisation on a program written using an inherently asynchronous paradigm.

Due to the overwhelming computational complexity of even the simpler subproblems in VLSI design, such as component placement, we are forced to rely on *heuristics* to find high quality solutions for problems of VLSI design automation in a reasonable period of time.

Defined in this thesis are two primary categories of heuristic: *weak* heuristics make very few assumptions about the problems to which they are applied; and *strong*

heuristics make use of more specific knowledge about the domain in which they are to solve problems. The class of strong heuristics is further subdivided into categories *simple* and *complex*: simple heuristics apply relatively little knowledge to a problem, while complex heuristics apply comparatively large amounts of knowledge.

Instances of weak heuristics described in this thesis include Divide-and-Conquer, iterative refinement, and Simulated Annealing. The latter two of these weak heuristics are both simple members of a more general class of algorithm, referred to by Sahni and Bhatt as "Adaptive Heuristics" [Sahni 80].

Examples of *simple strong* heuristics are cited primarily from the literature describing "conventional" algorithmic approaches to component placement and routing, while examples of *complex strong* heuristics are drawn largely from the literature of artificial intelligence research. For members of both classes of *strong* heuristic, it is noted that one or more *weak* heuristics often form the base upon which domain-specific knowledge has been added.

One empirical study [Sahni 80] compared weak heuristics to strong heuristics. It was concluded that strong heuristics, which make use of domain-specific information, are able to produce results of quality equal or superior to those of weak heuristics (specifically, Simulated Annealing) with *considerably* less computation. A second empirical study [Hartoog 86] provided similar conclusions, but with the additional result that strong heuristics which employ *randomness* can be made to produce superior results to a related deterministic heuristic, albeit at the expense of increased computation time.

The Expert System Wrapper Environment (ESWE) is a software package designed and implemented by the author as an extension to the ART [ART87] expert system shell. The intent of ESWE is to simplify the task of defining sequential or strongly synchronous structures within the inherently asynchronous framework of a rule-based

language. At the heart of ESWE is a group of Common-Lisp [Steele Jr. 84] macros which define structures for describing general deterministic and *non*-deterministic finite state automata (FSAs) for the control of sequencing within an expert system. Also developed by the author, and included in ESWE, are a number of abstractions for various levels of VLSI design, suitable for use in expert systems for placement, routing, or compaction.

ESWE is used by the author to develop a pair of closely related expert systems for the relative placement of transistors in CMOS leaf cell designs. Both expert systems are derived from Sharman's SPLAT [Sharman 86]; one expert uses a common greedy iterative refinement approach, while the other uses a stochastic approach related to (and derived from) Simulated Annealing. The logic and heuristics in each of these expert systems are fundamentally identical; the optimising expert *extends* the behaviour of the non-optimising expert, by introducing several additional stochastic heuristics (used, for example, to generate arbitrary changes to the current placement of transistors), and by adding code and control structures to support the concepts of *pseudo temperature* and an *annealing schedule.* The more structured environment offered by ESWE was found by the author to be quite valuable in the implementation of these expert systems.

Comparisons of these two expert systems show that the Simulated Annealing expert was less prone to arrest at a local minimum than was the conventional expert, despite a *very* rapid cooling schedule. These results are largely consistent with an earlier empirical study performed by Hartoog [Hartoog 86], which indicated that stochastic algorithms are able to achieve an improvement of roughly fifteen percent over rigidly deterministic procedures, although with considerably greater expenditure of CPU time.

Important results gained in the study of stochastic expert systems in this thesis are:

1. Expert systems for VLSI design which use greedy heuristics *can, and do,* become stuck in local minima, at least occasionally producing low quality results.

2. Stochastic heuristics *can* be used effectively to reduce the likelihood of premature arrest.

3. In the stochastic expert system used for this thesis, which employs a form of Simulated Annealing as its primary control mechanism, difficulty was experienced in selecting suitable parameters for the cooling schedule and termination criteria. Conservative parameters are expected to result in uniformly good quality, but at the expense of very large computation times. Less conservative parameters may greatly improve computation times but usually at the expense of sacrificing consistently high quality results.

4. With appropriate cooling and termination parameters, stochastic expert systems may be used to replace deterministic expert systems, since it it is reasonable to expected the stochastic expert to be *at worst* no more prone to premature termination than the deterministic.

5. Difficulties experienced in selecting cooling and termination parameters suggest a profitable collaboration between stochastic and deterministic expert systems. Very fast deterministic heuristics can be used to compute an *upper bound* on the acceptable costs; stochastic experts with otherwise rapid cooling can easily be forced to at least *match* such an upper bound.

5.2 Future Directions

The work done in this thesis suggests several potentially interesting possibilities for future research, both in the development of hierarchical frameworks for design-oriented expert systems, and in the use of stochastic processes in such expert systems.

5.2.1 Hierarchical Expert Systems

As explained in Chapter 3 of this thesis, a number of difficulties arise in extending conventional expert systems shells to allow hierarchical descriptions. A conventional expert system shell has little or no concept of lexical closure. Typically, the highest level construct which includes lexical scope is an individual rule; variables bindings (assignments) made within a rule are efficacious *only within the scope of that one rule.* If variable bindings beyond the scope of a single rule are supported at all, the variables involved are *global* to all rules in the expert system, and the bindings are not permitted to result from pattern matching operations.

Further to this, data placed in the database are completely global. In ART, defining a schema named "foo" in one part of an expert system has the consequence of removing any *previous* definitions made using this name in other parts of the expert system. Consequently, implementors must be exceedingly cautious in selecting names for rules, schemata, and slots, to ensure that no name accidentally duplicated. This is further complicated by the existence of several hundred "base" schemata used internally by the system, whose definitions may also be unwittingly overridden.

Problems involving scoping and duplication of names can be overcome using re-writing macros of the sort found in ESWE, by *modifying* identifiers to reflect the lexical environment in which they are declared. Unfortunately, this has an adverse impact on software development, as implementors who wish to use symbolic debugging aides or who need to interface expert system code with code written in another language (eg.,

LISP) may no longer be certain of the identifiers used internally in their compiled code.

Another interesting aspect of hierarchy in expert systems is the use of the "multiple cooperating expert" paradigm, in which separate "experts" are implemented for each major aspect of a design problem. ESWE provides a prototypical mechanism for defining such experts, but this mechanism is not as full-featured as some implementors might wish. Specifically, because the underlying mechanism used to express hierarchy and control flow in ESWE is a form of token-passing utilising the ART database, *it is* not *possible to reliably suspend an expert and later resume its actions at the same point*[1]. This sort of behaviour can only be provided effectively by designing an inference engine which recognises "contexts", and which allows rules to fire only if they are members of a currently active context.

5.2.2 Stochastic Expert Systems

The major subject of Chapter 4 of this thesis is the use of stochastic processes in expert systems for VLSI design. Several reasons are offered as to why this may be both beneficial and constructive. Expert systems for design-oriented problems are often forced to use an iterative refinement approach to recast a problem of design synthesis into a problem of diagnosis.

Iterative refinement algorithms which do not allow for "hill climbing" or "backtracking" are often subject to arrest at local extrema, finding solutions which are often far from the global optimum. Stochastic "hill climbing" procedures provide a very simple backtracking mechanism, and some, such as Simulated Annealing, may be able to assure solutions assympotically close to the global optimum [Mitra 85]. By providing an expert system with stochastic control mechanisms, as demonstrated in

[1]This can be done, of course, by making use of coroutines. Unfortunately, such structures are not available in most expert system shells.

Chapter 4, it is possible to largely avoid the problem of premature arrest. The particular stochastic mechanism used in the study of Chapter 4 has proven to be quite expensive. Investigations into other, more simple classes of stochastic heuristics are warranted

Another possible technique for incorporating randomness in an expert system stems from the uncertainty of *heuristics*. From the earliest days of expert systems research, techniques for reasoning with uncertain *data* have existed [Buchanan 84]. Design-oriented expert systems, unlike many diagnostic expert systems, often have the luxury of dealing with relatively concrete data; for instance, we know that when a transistor is placed at coordinate (1000, 2000) in a symbolic grid, it is apt to remain there until it is explicitly moved.

Uncertainty still exists in design oriented expert systems, within the very rules and heuristics which are used to manipulate data. It is often the case known that a given rule may be applied effectively in one instance, and yet prove *useless* in another. Further, some rules in an expert system may be known to be more generally applicable than are others. One possibly profitable avenue for future research is the development of a general purpose inference engine which allows the association of a "certainty factor" with each rule in the expert system, and which either fires, or ignores, an activation of a given rule with probability proportional to the certainty factor associated with that rule. Greene and Supowit's Tree Algorithm for the Dynamic Weighted Selection Problem, as presented in [Greene 86] provides one possibly suitable means of randomly selecting a rule activation from among the set of highest-priority activations in the queue.

In summary, stochastic processes are demonstrated to remedy to propensity of some expert systems for design automation to arrest a local minima. The stochastic heuristic used in this thesis, which derives from Simulated Annealing, is demon-

strated to be effective for this purpose, but *may* prove to be unduly expensive. Research into less costly stochastic heuristics for knowledge-based expert systems is therefore warranted.

References

[Ackland 88] B. Ackland. Knowledge based VLSI design. In R. Suaya and G. Birtwistle, editors, *VLSI and Parallel Computation.* Morgan Kauffman, December 1988. – Proceedings of the 1988 Frontiers Conference. To be published.

[Aho 74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[ART87] Inference Inc., 5300 W. Century Blvd., Los Angeles CA. *The ART Reference Manual*, 1987.

[Brady 77] J. M. Brady. *The Theory of Computer Science: A Programming Approach.* Chapman and Hall, New York, science paperbacks edition edition, 1977.

[Breuer 77] M. A. Breuer. A class of Min-Cut placement algorithms. In *Proceedings of the 14th Design Automation Conference*, pages 284 – 290, 1977.

[Brewer 86] F. D. Brewer and D. D. Gajski. An expert-system paradigm for design. In *23rd Design Automation Conference*, pages 62 – 68. DAC, 1986.

[Brinsmead 88] M. D. Brinsmead and John Kendall. An algorithm for the automatic placement of ports in a symbolic-layout VLSI CAD system. Department of Computer Science, Yellow-Series Report 88-312-24, University of Calgary, Calgary, Alberta, July 1988.

[Buchanan 84] B. G. Buchanan and E. H. Shortliffe, editors. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley Publishing Company, Reading, Mass., 1984.

[Clocksin 84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, Heidelberg, Germany, second edition, August 1984.

[Cole 88] B. C. Cole. The next wave: 16-MBit DRAMS from Japan. *Electronics*, 61(4):68 – 69, February 18 1988.

[Corrigan 79] L. I. Corrigan. A placement capability based on partitioning. In *Proceedings of the 16th Design Automation Conference*, pages 406 – 413, 1979.

[Dunlop 85] A. E. Dunlop and B. W. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer-Aided Design*, CAD-4, January 1985.

[Ele88] International newsletter. *Electronics*, 61(14):41, August 1988.

[Forgy 81] C. L. Forgy. OPS5 users' manual. Technical Report CMU-CS-79-135, Carnegie-Mellon University, Pittsburg Pennsylvania, July 1981.

[Forgy 82] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence*, 19:17 – 37, 1982.

[Gay 85] J. G. Gay, R. Richter, and B. J. Berne. Component placement in VLSI circuits using a constant pressure Monte Carlo method. *Integration, The VLSI Journal*, 3(4):271 – 282, December 1985.

[Goto 79] S. Goto. An efficient algorithm for the two-dimensional placement problem in electrical circuit layout. In *Proceedings of the 1979 ISCAS*, pages 850 – 853, 1979.

[Greene 86] J. W. Greene and K. J. Supowit. Simulated annealing without rejected moves. *IEEE Transactions on Computer-Aided Design*, CAD-5(1):221 – 228, January 1986.

[Gupta 87] A. Gupta. *Parallelism in Production Systems.* Morgan Kauffman Publishers, Inc., Los Altos, California, 1987.

[Hanan 76] M. Hanan, P. K. Wolff Sr., and B. J. Agule. Some experimental results on placement techniques. In *Proceedings of the 13th Design Automation Conference*, pages 324 – 342, 1976.

[Hartoog 86] M. R. Hartoog. Analysis of placement procedures for VLSI standard cell layout. In *Proceedings of the 23rd Design Automation Conference*, pages 314 – 319, 1986.

[Joobbani 86a] R. Joobbani. *An Artificial Intelligence Approach to VLSI Routing.* Kluwer Academic Publishers, Hingham, Mass., 1986. Also published as: R., Joobbani, "WEAVER: An Application of Knowledge-Based Expert Systems to Detailed Routing of VLSI Circuits", PhD dissertation, Carneghie-Mellon University, April 1985.

[Joobbani 86b] R. Joobbani and D. Siewiorek. WEAVER: A knowledge-based routing expert. *IEEE Design & Test*, pages 12 – 23, February 1986.

[Joyce 88] J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI*

Specification, Verification and Synthesis, pages 129 – 158. Kluwer Academic Publishers, 1988.

[Keefe 86] M. Keefe and E.J.M Kendall. An expert system for routing in VLSI. In *Proceedings of the 1986 Canadian Conference on Very Large Scale Integration*, pages 337 – 342, Montreal, October 1986. CCVLSI.

[Keefe 87] M. Keefe. An expert system for routing VLSI designs. Master's thesis, University of Calgary, Department of Computer Science, June 1987.

[Kernighan 78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Toronto, 1978.

[Kim 85] J. H. Kim. *Use of Domain Knowledge in Computer Aid of IC Cell Layout Design.* PhD thesis, Carnegie-Mellon University, April 1985.

[Kirkpatric 83] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671 – 680, May 1983.

[Kollaritsc 85] P. W. Kollaritsch and N. Weste. TOPOLOGIZER: An expert system translator of circuit connectivity to symbolic cell layout. *IEEE Journal of Solid State Circuits*, SC-20(3):799 – 804, June 1985.

[Liblong 84] B. Liblong. SHIFT: a structured hierarchical intermediate form for VLSI design tools. Master's thesis, University of Calgary, Department of Computer Science, September 1984.

[Liu 86] E. S. K. Liu. *Two Dimensional IC Layout Compaction.* PhD thesis, University of Calgary, Department of Computer Science, December 1986.

[Mano 82] M. M. Mano. *Computer System Architecture.* Prentice Hall, Engelwoods Cliff, N.J., second edition, 1982.

[Mead 80] C. A. Mead and L. A. Conway. *Introduction to VLSI Design.* Addison-Wesley, Reading Massachusetts, 1980.

[Metropolis 53] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A Teller, and E. Teller. Equation of state calculations for fast computing machines. *Journal of Chemical Physics*, 21(6):1087 – 1092, 1953.

[Minsky 75] M. Minsky. A framework for representing knowledge. In P. H. Winston, editor, *The Psychology of Computer Vision*, chapter 6, pages 211 – 277. McGraw Hill Book Company, 1975.

[Mitra 85] D. Mitra, F. Romeo, and A. Sangiovanni-Vincentelli. Convergence and finite-time behaviour of simulated annealing. Technical Report Memorandum No. UCB/ERL M85/23, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, March 1985.

[Nahar 86] Surendra Nahar, Sartaj Sahni, and Eugene Shragowitz. Simulated annealing and combinatorial optimization. In *23rd Design Automation Conference*, pages 293 – 299, 1986.

[Nixon 84] I. M. Nixon. I.F.: An idiomatic floorplanner. Master's thesis, Edinburgh University, October 1984.

[Otten 84] R. H. J. M. Otten and L. P. P. P. van Ginnekan. Floorplan design using simulated annealing. In *Proceedings of the Interna-*

tional Conference on Computers and Design., pages 96 – 98, Port Chester, 1984. ICCAD.

[Post 43] E. Post. Formal reductions of the general combinatorial problem. *American Journal of Mathematics*, 65:197 – 215, 1943.

[Preas 86] B. T. Preas and P. G. Karger. Automatic placement: A review of current techniques. In *Proceedings of the 23rd Design Automation Conference*, pages 622 – 629, 1986.

[Pucknell 88] D. A. Pucknell and K. Eshraghian. *Basic VLSI Design: Systems and Circuits.* Prentice Hall of Australia Pty Ltd., Sydney Australia, second edition, 1988.

[Robinson 65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23 – 41, 1965.

[Rosenbloom 85] P. S. Rosenbloom, J. E. Laird, J. McDermott, and Alan Newell. R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI7(5):561 – 568, September 1985.

[Sahni 80] S. Sahni and A. Bhatt. Complexity of design automation problems. In *17th Design Automation Conference*, pages 402 – 411, Minneapolis, Minn., June 1980. ACM/IEEE.

[Sastry 82] S. Sastry and A. Parker. The complexity of two-dimensional compaction of VLSI layouts. In *International Conference on Circuits and Computers*, New York, NY, 1982.

[Schlag 83] M. Schlag, Y.Z. Liao, and C.K. Wong. An algorithm for optimal two-dimensional compaction of VLSI layout. *Integration, the VLSI Journal*, 1(2):179 – 209, 1983.

[Schuler 72] D. M. Schuler and E. G. Ulrich. Clustering and linear placement. In *Proceedings of the 9th Design Automation Conference*, pages 50 – 56, 1972.

[SCL86a] Symbolics Inc., 4 New England Tech Center, 555 Virginia Rd., Concord Mass. *Symbolics Common Lisp – Language Concepts*, 1986.

[SCL86b] Symbolics Inc., 4 New England Tech Center, 555 Virginia Rd., Concord Mass. *Symbolics Common Lisp – Language Dictionary*, 1986.

[Sechen 86] C. Sechen and A. Sangiovanni-Vincentelli. TimberWolf3.2: A new standard cell placement and global routing package. In *Proceedings of the 23rd Design Automation Conference*, pages 432 – 439, 1986.

[Sharman 86] D. Sharman. SPLAT: Symbolic cell placement and routing tool. In *Proceedings of the 1986 Canadian Conference on Very Large Scale Integration*, pages 343 – 347, Montreal, October 1986. CCVLSI.

[Shields 87] M. W. Shields. *An Introduction to Automata Theory.* Blackwell Scientific Publications, Palo Alto, 1987.

[Shortliffe 76] E. H. Shortliffe. *Computer-Based Medical Consultations: MYCIN.* American Elsevier, New York, 1976.

[Soukup 81] J. Soukup. Circuit layout. *Proceedings of the IEEE*, 69(10), October 1981.

[Steele Jr. 84] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, Hanover, Mass., 1984.

[Stroustrup 86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Don Mills, Ontario, 1986.

[Subramanya 86] P. A. Subramanyam. Synapse: An expert system for VLSI design. *Computer*, pages 78 – 89, July 1986.

[Ullman 83] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Rockville, Maryland, 1983.

[Vecchi 83] M.P. Vecchi and S. Kirkpatrick. Global wiring by simulated annealing. *IEEE Transactions on Computer-Aided Design*, CAD-2(4):215 – 222, October 1983.

[Weste 81] N. Weste and B. Ackland. A pragmatic approach to topological symbolic IC design. In John P. Gray, editor, *Very Large Scale Integration.* Academic Press, 1981.

[Williams 78] J.D. Williams. STICKS – a graphical compiler for high level LSI design. *AFIPS Conference Proceedings*, 47:289 – 295, June 1978.

Appendix A

The ART Programming Language

ART [ART87] is a language constructed specifically to support the design and implementation of rule-based expert systems. It provides users with a predefined inference engine and database manager, and an established framework for defining rules. Both forward- and backward-chaining rules are supported. In addition, ART provides an object-oriented framework for representing knowledge, which extends the normal knowledge-base operations.

Following is a brief discussion of some of the major issues and features of the ART language. At least surface knowledge of Common-Lisp is assumed; almost any text on Lisp or Common-Lisp should provide adequate background information for this discussion, however, Guy L. Steele Jr.'s text *Common Lisp: The Language* [Steele Jr. 84] is recommended. For complete information on ART, refer to the ART Reference Manual [ART87].

A.1 Viewpoints in Art

ART's knowledge base provides support for "parallel" exploration of multiple models of a given given problem. The mechanism for do this is referred to as "viewpoints". In effect, a viewpoint is a subset of the ART knowledge base which corresponds to some hypothetical or time-related event; by hypothesising a given fact or decision, the ART data base creates a new context, in which conclusions drawn from the hypothesis may be stored. At a later time, a hypothesis my be either accepted or rejected. In each respective case, the "facts" in the viewpoint corresponding to that

hypothesis will either be promoted to the viewpoint in which the hypothesis was first made, or be destroyed entirely.

The ART viewpoint system is very complex, and may operate on many levels. Because the ESWEenvironment make no explicit use of viewpoints (except to make certain that control facts are asserted globally in the root viewpoint) this description will not include a detailed analysis of the viewpoint mechanism. ESWE has been successfully tested with expert systems which make limited used of the viewpoint mechanism, but may fail under extreme circumstances. For a detailed description of viewpoints, refer to [ART87, chap. 7].

A.2 ART's Knowledge Base

At the heart of the ART programming language is the working memory. The ART working memory, or database, is used to coordinate the actions of the rules in the rule memory.

From the user's point of view, ART's knowledge base is comprised of two main forms of entity: propositional facts, and schemata. Sequences, the underlying representation for propositional facts, form the true basis of ART's knowledge base. All the properties and behaviour of schemata may be duplicated using propositional facts (or simply "facts") and rules; schemata provide a more pleasant syntax for describing certain forms of information, and provide procedural features which may prove more efficient than simple ART rules.

A.2.1 Propositional Facts

Facts in ART's database take the form of predicates of First-order logic, using a lisp-like prefix notation. Expressed in Backus-Naur Form (BNF), the syntax of an ART fact is:

```
<fact> ::= (<relation-name> <fact-pattern>*)
<relation-name> ::= <symbol>
<fact-pattern> ::= <symbol> | <number> | ( <fact-pattern>* )
<symbol> ::= any valid Common-Lisp symbol, whose print-name does
             not start with the character '?'.
<number> ::= any valid Common-Lisp numeric constant.
```

In effect, a fact in the ART database has the appearance[1] of a Lisp list, and may contain an arbitrary level of nested sub-lists. The first symbol in the fact is regarded as the name of a relation. Each named relation *must* be declared, and its arity specified. Asserting a fact in the ART data base whose relation-name is not declared, results in a warning message, and a default declaration.

When declaring a relation, it is necessary to fully describe its *arity*, that is, the number of arguments the relation is expected to have. Relations *may* be declared to have arbitrary arity. In addition, relation declarations allow more than simply arity to be specified: it is possible to specify the arity of (required) sublists in the arguments; and a description of the semantics of the relation may be provided to allow conversion between natural language sentences and facts in list-notation.

A.2.2 Schemata

The *schema*[2] system is a useful programming interface provided by ART for the purpose of imposing structure on stored knowledge and defining daemons that can act immediately upon certain types of changes to the knowledge base. Schemata are

[1]Note: this is appearance only. The internal representation is actually a 'sequence'; list notation is used simply for convenience. Functions which directly manipulate the knowledge base must be certain to convert between "lists" and "sequences", or the consistency of the knowledge base will be destroyed.

[2]Singular: schema; plural: schemata.

also useful as an interface between code written in ART and programs written in C or Lisp.

The schema system is, effectively, an object oriented language not at all unlike Minsky's "frames" [Minsky 75]. With the schema representation, a programmer can easily model classes of objects, and describe relationships between classes, subclasses, and instances of objects. Further, programs written in languages can use schemata to locate specific information in the knowledge base with much less difficulty than would be encountered with propositional facts.

A.2.2.1 Example of Schemata

A schema is a named collection of properties or *slots*. The slots contain information which describe or qualify the schema. In the example below, a schema is defined to describe a class of object called "book", and another schema is defined which describes a specific book.

```
(defschema book
   (medium paper)
   (contained-data printed-text)
   (author)
   (title)
   (subject)
   (year)
   (number-of-pages))

(defschema dune
   (instance-of book)
   (author "Frank Herbert")
   (title "Dune")
   (has-sequels Dune-Messiah Children-of-Dune God-Emperor-of-Dune
                Heretics-of-Dune Chapterhouse-Dune)
   (primary-locale Desert-planet-named-Arrakis)
   (subject Science-Fiction))
```

In this example, we describe "book" as a class of objects which contain printed text, and which are printed on paper. (Clearly, other classes of objects, such as

"newspapers", also meet this criteria, but that is not relevant, as most people would agree that a book is not a newspaper and a newspaper is not a book.) In addition to this specific information about what a book *is*, we also encounter a description of the sort of things we might expect to know about *specific* books, such as the author, and year of publication.

Next, we describe Dune as a specific *instance* of the class of objects known as "book", and describe some of the important information about it. There are no entries given for the "year" or "number-of-pages" slots because no copy of the book was handy at the time the example was composed. Absence of this information is left to the expert system application to interpret; possibly its objective will be to deduce the number of pages in the originally published version.

By stating that Dune is an instance of the class book, we know implicitly that Dune contains printed text, and that Dune is made of paper. The ART schema system handles this by way of an inheritance mechanism, which causes instances and subclasses to *automatically* acquire information which has been stated as true about the containing class. Additional information about Dune, is included in the "has-sequels" and "primary-locale" slots, which are not present in the parent schema "book". This is not a problem; we simply have some information about Dune which may not apply to books in general.

It is worth noticing that in this example, all slots have been provided with either zero or one values, except for the "has-sequels" slot in the schema Dune. By default, each slot may contain exactly zero or one values; the "has-sequels" slot in the declaration of Dune will result in an error message from ART, unless the definition is preceded by the declaration:

```
(defschema has-sequels
   (is-a slot)
   (slot-how-many multiple-values))
```

This declaration states that slots named "has-equals" (in *any* schema) may have more than one value.

A.2.2.2 Schemata and Facts

There is a very clear and purposeful duality built into the ART database system, between Schemata and Propositional Facts. A schema is simply a named collection of *slots*; each slot describes a some characteristic of the schema. Within the ART database, slots are simply stored as trinary relations of the form

$$(< \texttt{slot-name} > \ < \texttt{schema-name} > \ < \texttt{slot-value} >).$$

The "author" slot from the Dune example above is represented in the ART knowledge base as:

```
(author dune frank-herbert).
```

Multiple value slots are treated in precisely the same manner, but are represented by one fact for each value. Thus the "has-sequels" slot from the Dune example above is represented as:

```
(has-sequels dune dune-messiah)
(has-sequels dune children-of-dune)
             ⋮
```

Previously, it was implied that facts and schemata are fundamentally the same, and the concept of "duality" described in this section appears to confirm this. Facts and schemata, are fundamentally *similar*, but they are not nearly identical. The schemata system is considerably more than simple "syntactic sugar";[3] it provides several properties not available with simple propositional facts. The inheritance mechanism, for example, is dynamic; adding a slot or changing the default value of

[3]This is often referred to as "syntose" by laconic hard-core hackers.

a slot in a schema will cause *all* descended schemata (those reflexively declared as `instance-of` the modified schema) will be automatically updated.

Schemata also provide a very flexible interface to external code written in languages other than ART. It is possible, for example, to inspect *and modify* schemata in the ART database with code written in LISP or PROLOG, but it is very difficult to inspect facts, and impossible to modify them, except from within ART. For this reason, schemata, and not facts are used to implement most of the features of ESWE.

A.2.3 Specification and Behaviour of Rules in ART

Like OPS5 [Forgy 81] and other expert systems languages, ART makes use of if-then-else type rules. These rules consist of a series of "patterns" to be matched in the working memory, and a number of actions to take when all such patters are matched. These actions often involve modifying the working memory, by adding or deleting facts, but may also include the evaluation of arbitrary Common-Lisp expressions, which may or may not have side-effects on the working memory.

ART's pattern matching algorithm constantly maintains a list of all rules whose "patterns" are matched by one or more sets of facts in the working memory. This list, or agenda, is effectively the set of all candidates for the next rule to be "fired" or executed. Because the firing of a rule can (and usually *does*) modify the working memory, the agenda must be updated before another rule can be selected and fired. When it is time to select the next rule, ART chooses the rule declared to have the highest priority or *salience* (see section A.3); if more than one such rule exists, ART always selects that which was most recently matched. (Thus, the ART agenda effectively becomes a prioritised First In First Out, or FIFO, queue.) This behaviour is not always desirable: when several alternative courses of action are offered, they will *always* be acted upon last to first. This leaves the expert system completely subject to NP-complete problems.

Figure A.1 provides a simplified description of the syntax for declaring rules in ART; for a more complete description, refer to [ART87, pp. appx-a-6 – appx-a-21]. Because ART rules are "global" entities, their names must always be unique from all other rules.

```
<defrule}> ::=
   (defrule <name>
      {(declare (salience <integer-expression>))}
      [<{\bf forward-rule-body}> | <{\bf backward-rule-body}>] )

<forward-rule-body> ::= {<logical-conditions>}
                                    <condition>* { => <form>*}

<backward-rule-body> ::= <goal-pattern> <= <condition>*

<condition> ::= (test <safe-form}>) |
                          <pattern>

<safe-form> ::= Any Common-Lisp expression without side-effects,
                and which always returns the same value for a
                given set of arguments.

<integer-expression> ::= Any ART or Common-Lisp expression
                         which evaluates to an integer.

<form> ::= Any valid ART or Common-Lisp expression.
```

Figure A.1: Simplified BNF for **defrule**. See [ART87] for complete BNF.

A.3 Rules, Salience, and Sequencing in ART

An important aspect of the agenda mechanism described above is the ability to assign priorities to each rule by means of the *salience* declaration[4] (see figure A.1), which

[4]Henceforth, when speaking of the "salience" of a rule, what is meant is "the value appearing in the rule's salience declaration", unless otherwise stated.

takes the form "**(declare (salience <integer>))**". When assigning salience values to rules, the programmer informs ART's inference engine of the relative importance of each rule: rules with very high salience values are inserted at the top of the agenda; rules with very low salience values are placed at the bottom. Thus, a rule with a high salience will fire before (in preference to) to rule with a lower salience.

It is often tempting for programmers to attempt to use the salience declarations [5] of rules as a sequencing mechanism, and, in fact, this can occasionally be done with fair success. Unfortunately, saliences are static; once a rule is defined, its salience cannot be modified, effectively making impossible iteration or branching if salience is used as the sole sequencing mechanism.

The more typical (and officially sanctioned) way to perform sequencing and iteration in a rule-based language is to add special conditions to each rule, and have each rule create a new fact in the working memory, which will allow the "next" rule in the sequence to be selected and fired. This sort of strict control is regarded by some as counter to the spirit of rule-based programming, but, unfortunately, is often necessary.

This method of passing a "token" from one rule to the next in order to control order of execution is certainly very general, but is, unfortunately, very cumbersome to modify. For instance, when the iterative requirements of an expert system require nested loops (fortunately, this is fairly rare), the control facts used for sequencing become so heavily entwined as to be essentially un-maintainable.

[5] This is *not* the intended use of salience declarations.

Appendix B

Sample Technology Declaration

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: ART-USER; -*-

(eval-when (compile eval)
  (if (not (macro-function 'eq-PNAME))
      (load "sym-1:>brinsmead>stanley>compiler-macros"))
  (if (not (macro-function 'DEFCELL))
      (load "sym-1:>brinsmead>stanley>stan-front-end"))
)

(eval-when (load eval)
  (if (not (boundp '*Current-Technology*))
      (load "sym-fsa:>brinsmead>stanley>stanley-globals"))
  (if (not (functionp 'list-of-lists-p))
      (load "sym-1:>brinsmead>stanley>stan-type-preds"))
  (if (not (schemap 'sym-cell))
      (load "sym-fsa:>Brinsmead>stanley>stan-sym-objs"))
  (if (not (functionp '*deftechnology-1*))
      (load "sym-fsa:>brinsmead>stanley>stan-front-end-funs"))
)

(deftechnology MOSIS-GENERIC-CMOS
  (layers
    (p-active p-active diffusion 20 200 1.0)
    (p-well   virtual  virtual   0  0   0)
    (n-active n-active diffusion 20 200 1.0)
    (n-well   virtual  virtual   0  0   0)
    (poly     gate     poly      10 50  1.0)
    (metal-1  interconnect metal 6  20  1.0)
    (metal-2  interconnect metal 6  20  1.0)
    (contact  via          metal 6  20  1.0)
    (gate*     virtual  virtual   0  1.0 0)
  )
  (rules
    (separation p-active p-well 16)
    (well-surround p-active 7)
    (separation n-active n-well 16)
    (well-surround n-active 7)
    (separation poly p-active   1)
    (separation poly n-active   1)
    (separation poly poly       2)
    (separation p-active p-active 2)
    (separation n-active n-active 2)
    (separation metal-1 metal-1   3)
    (separation metal-2 metal-2   4)
    (separation contact gate*      4)
    (separation contact metal-1   3)
```

```
  (separation contact poly      2)
  (minimum-feature  metal-1    3)
  (minimum-feature  metal-2    4)
  (minimum-feature  poly       2)
  (minimum-feature  p-active   3)
  (minimum-feature  n-active   3)
  (minimum-feature  contact    2)
  (minimum-feature  gate*      1)
  (minimum-feature  n-well     1)
  (minimum-feature  p-well     1)
)

(defcontact metal-1-metal-2
  (contacts metal-1 metal-2)
  (pins (m-1 metal-1 (0 0))
        (m-2 metal-2 (0 0))
        (c-1 contact (0 0)))
  (wires (mw-1 metal-1 4 m-1 m-1)
         (mw-2 metal-2 4 m-2 m-2)
         (cw-1 contact 2 c-1 c-1))
)

(defcontact metal-1-poly
  (contacts metal-1 poly)
  (pins (m-1 metal-1 (0 0))
        (p-1 poly    (0 0))
        (c-1 contact (0 0)))
  (wires (mw-1 metal-1 4 m-1 m-1)
         (pw-1 poly    6 p-1 p-1)
         (cw-1 contact 2 c-1 c-1))
)
(defcontact metal-1-p-active
  (contacts metal-1 p-active)
  (pins (m-1 metal-1 (0 0))
        (a-1 p-active (0 0))
        (c-1 contact (0 0)))
  (wires (mw-1 metal-1 4 m-1 m-1)
         (aw-1 p-active 6 a-1 a-1)
         (cw-1 contact 2 c-1 c-1))
)

(defcontact metal-1-n-active
  (contacts metal-1 n-active)
  (pins (m-1 metal-1 (0 0))
        (a-1 n-active (0 0))
        (c-1 contact  (0 0)))
  (wires (mw-1 metal-1 4 m-1 m-1)
         (aw-1 n-active 6 a-1 a-1)
         (cw-1 contact 2 c-1 c-1))
)


(defschema p-transistor
          (instance-of transistor-prototype)
```

```
  (type P-TRANSISTOR)
  (proto-pins
    (SRC-PIN p-active (0 100))
    (DRN-PIN p-active (0 -100))
    (GATE-PIN-1 poly  (100 0))
    (GATE-PIN-2 poly  (-100 0))
    (INT-PIN-1  p-active (0 0))
    (INT-PIN-2  p-active (0 0))
    (WELL-PIN-1 n-well   (0 100))
    (WELL-PIN-2 n-well   (0 -100)))
  (active-proto-pins SRC-PIN DRN-PIN GATE-PIN-1 GATE-PIN-2)
  (proto-wires
    (SRC-WIRE p-active 5 SRC-PIN INT-PIN-1)
    (DRN-WIRE p-active 5 DRN-PIN INT-PIN-2)
    (GATE-WIRE poly    3 GATE-PIN-1 GATE-PIN-2)
    (WELL-WIRE n-well 19 WELL-PIN-1 WELL-PIN-2))
  (proto-constraints
    (lambda (schema)
      (prog (pins-list
             wires-list
             gate-width
             active-width
             src-pin-name
             drn-pin-name
             gate-pin-1-name
             gate-pin-2-name)
            (assert (schemap schema))
            (setq pins-list (get-schema-value schema 'pins))
            )
      )
    )
  )

(defschema n-transistor
           (instance-of transistor-prototype)
  (type N-TRANSISTOR)
  (proto-pins
    (SRC-PIN n-active (0 100))
    (DRN-PIN n-active (0 -100))
    (GATE-PIN-1 poly  (100 0))
    (GATE-PIN-2 poly  (-100 0))
    (INT-PIN-1  n-active (0 0))
    (INT-PIN-2  n-active (0 0))
    (WELL-PIN-1 p-well   (0 100))
    (WELL-PIN-2 p-well   (0 -100)))
  (active-proto-pins SRC-PIN DRN-PIN GATE-PIN-1 GATE-PIN-2)
  (proto-wires
    (SRC-WIRE n-active 5 SRC-PIN INT-PIN-1)
    (DRN-WIRE n-active 5 DRN-PIN INT-PIN-2)
    (GATE-WIRE poly    4 GATE-PIN-1 GATE-PIN-2)
    (WELL-WIRE p-well 19 WELL-PIN-1 WELL-PIN-2))
  (proto-constraints
    (lambda (schema)
      (prog (pins-list
```

```
                wires-list
                gate-width
                active-width
                src-pin-name
                drn-pin-name
                gate-pin-1-name
                gate-pin-2-name)
               (assert (schemap schema))
               (setq pins-list (get-schema-value schema 'pins))
               )
          )
        )
      )
)

(def-state-var main-map ())

;;; Set up the colour map for the layout editing software.
(defcolormap main-map
  (defcolor main-map 0 .77 .69 .00  :mixin nil :transparent nil) ;; N-well
  (defcolor main-map 1 .46 .39 .17  :mixin nil :transparent nil) ;; P-Well
  (defcolor main-map 2 .01 .79 .01  :mixin t :transparent nil)   ;; Active
  (defcolor main-map 3 1.0 .00 .00  :mixin t :transparent t)     ;; Poly
  (defcolor main-map 4 .00 .72 1.0  :mixin t :transparent t)     ;; Metal-1
  (defcolor main-map 5 1.0 .00 1.0  :mixin t :transparent t)     ;; Metal-2
  (defcolor main-map 6 .00 .00 .00  :mixin nil :transparent nil) ;; Text
  (defcolor main-map 7 1.0 1.0 1.0 :mixin nil :transparent nil)  ;; Highlight.
  (defcolor main-map 8 .45 .45 .51 :mixin nil :transparent nil)  ;; Background
)
```