REPRESENTING ENGINEERING DESIGN KNOWLEDGE IN AN OBJECT-ORIENTED CONSTRAINT LOGIC PROGRAMMING LANGUAGE

by

Dong Li

B.Eng. Tsinghua University, Beijing, China, 1990

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

in the School
of
Engineering Science

© Dong Li 1993
SIMON FRASER UNIVERSITY
March 1993

All rights reserved. This work may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

APPROVAL

Name: Dong Li

Degree: Master of Applied Science

Title of thesis: **Representing Engineering Design Knowledge In An Object-Oriented Constraint Logic Programming Language**

Examining Committee: Dr. John C. Dill, Chairman

____

Dr. John D. Jones, Senior Supervisor

____

Dr. William S. Havens, Supervisor

____

Dr. William A. Gruver, Examiner

Date Approved: March 30, 1993

ABSTRACT

We intend to develop a two-part knowledge base to support the design of a range of artifacts. A prototype knowledge base has been built which can support the design of Stirling engine heat exchangers. The knowledge base is implemented with the Echidna constraint reasoning system, which incorporates constraint logic programming, truth maintenance and dependency-directed intelligent backtracking, all in an object-oriented framework. In developing this knowledge base, we face a trade-off between generality and efficiency. To solve this problem, we introduced a new form of knowledge representation which incorporates scaling information, a form not generally available in other knowledge-based systems.

In this thesis, we show how the knowledge base is created and how some of the knowledge underlying the designer's sense of scale can be formally represented in a knowledge base. A design example is also included.

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted be me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

"Representing Engineering Design Knowledge In An Object-Oriented Constraint Logic Programming Language"

Author:

(signature)

Dong LI
(name)

March 31, 1993
(date)

To My Family

ACKNOWLEDGMENTS

First and foremost, I would like to thank my senior supervisor, Dr. John Dewey Jones, who has been a constant source of encouragement and inspiration to me. His lectures first quickened my interest in this research. He has given me numerous suggestions and comments during the course of the research and in the proofreading of this thesis. I am very grateful for these things, and much more.

I wish to express my deep appreciation to other members of my thesis committee — Dr. William A. Gruver, Dr. William S. Havens and Dr. John C. Dill, for their time and effort in reading my thesis.

The assistance provided by the staff of Expert Systems Lab — Sue Sidebottom, Miron Cuperman and Sumo Kindersley, is also appreciated. The School's Graduate Secretary, Mrs. Brigitte Rabold, with her optimism and personality, made the daily jaunt to the General Office interesting.

Finally, but by no means last, I owe my family more than I have a right to owe anyone, and certainly more than words can say.

Contents

ABSTRACT iii

ACKNOWLEDGMENTS v

LIST OF FIGURES ix

LIST OF TABLES x

ABBREVIATIONS xi

1 Introduction 1

1.1 Design and Knowledge 1

1.2 Expert Systems 4

1.3 Motivation 6

2 Echidna 8

2.1 Object-Oriented Programming 9

2.2 Constraint Logic Programming 10

2.2.1 Backtracking 11

2.2.2 Constraint Propagation 13

2.2.3 A Combination of Backtracking and Constraint Propagation . 13

2.3 Potential Problems With Echidna's Constraint Propagation Mechanism 14

3 A Knowledge Base For Stirling Engine Design 18

3.1 A Brief Introduction To Stirling Engines 18

3.1.1 What Is A Stirling engine? 19

3.1.2 How Does It Work? 20

3.2 Structure of The Knowledge Base 23

3.3 How Is The Knowledge Base Used ? 27

4 A Sense of Scale 28

4.1 Why Do We Need Scaling ? 28

4.2 Creating Scaled KB 30

4.3 Application of Monotonicity Analysis 33

4.3.1 The Problem 33

4.3.2 Basics About Monotonicity Analysis 34

4.3.3 Application 38

5 Testing The Knowledge Base 43

5.1 Testing For Correctness 43

5.2 Improvement of Efficiency 48

6 A Detailed Design Example 51

6.1 Design Requirements 51

6.2 Echidna's Decision 53

7 Summary And Future Work **67**

7.1 Progress So Far 67

7.2 Future Work 68

A Sample Files **71**

B Echidna's Output of The Example in Chapter 6 **95**

References **108**

List of Figures

3.1 Alpha Configuration 21

3.2 Beta Configuration 22

3.3 Gamma Configuration 22

3.4 Classification Hierarchy 26

3.5 Component Hierarchy 26

4.1 Monotonicity Analysis Process 37

5.1 Cylindrical and annular capsule 46

List of Tables

4.1 Monotonicity Table 39

4.2 Monotonicity Table 1 40

4.3 Monotonicity Table 2 41

6.1 Design Parameters 52

6.2 Heater 60

6.3 The parameters of a single tube in the heater 61

6.4 Cooler 62

6.5 The parameters of a single tube in the cooler 63

6.6 Regenerator 64

6.7 Capsule 65

6.8 Matrix 66

ABBREVIATIONS

CLP	Constraint Logic Programming
CPS	Cycle Per Second
CSP	Constraint Satisfaction Problem
HP	Horse Power
IED	InnerExternalDiameter
OID	OuterInternalDiameter
OOPS	Object-Oriented Programming System

Chapter 1

Introduction

1.1 Design and Knowledge

Design is the most essential and fundamental engineering activity. It results from an expression of need for a component, product or system and entails several steps in producing a solution to fill that need. These steps typically include a precise definition of design requirements; collection and analysis of pertinent information; synthesis of analyzed information into various configurations; and evaluation of the merits of alternative possible solutions; and the selection, maybe after several iterations, of a solution that will best satisfy all the constraints.

Design can be generally categorized as either *creative* or *routine*. In the former class, design is open-ended : design goals are ill-specified, and there is no storehouse of effective decompositions, nor any design plans for subproblems. This kind of design is very difficult and is rarely done; it leads to a major invention or completely new

products. One example would be the Manhattan Project[1]. In the latter class, effective problem decompositions are known, detailed design plans for the component problems are known, and the criteria for success are clear and quantitative. For example, configuring a computer system from available components. Most designs fall into *routine* design category.

In this thesis, I am not going to write about how we can come up with new inventions. Instead, I will restrict the discussion to routine engineering design. In particular, I will illustrate the application of a new approach to represent design knowledge and, as an example, its application in the development of a knowledge-based system which can support the design of the Stirling engine heat exchangers (background knowledge of Stirling engine will be given in chapter 3). The complexity of this kind of design tasks is due not only to the variety of combinations of requirements, but also to the numerous components and subcomponents, each of which must be specified to satisfy the initial requirements, their immediate consequences, the consequences of other design decisions, as well as the constraints of various kinds that a component of this kind will have.

Engineering design is a knowledge-based activity. In order to design a particular set of artifacts, a computer-aided design system should possess knowledge about that set. To give some ideas of the kind of knowledge we are dealing with, let's draw an example from the knowledge base of Stirling engine heat exchanger (further details

[1]The Manhattan Project was conducted in the United States from 1940-1945; it was initiated in consequence of a letter from Einstein and Leo Szilard to President Roosevelt in which they suggested it might be possible to make an atomic bomb (and that whoever was first to make such a bomb would probably win the war.) The project involved many of the leading physicists of the time – Hans Bethe, Lawrence Oppenheimer, Richard Feynman, Freeman Dyson, etc. No-one had made anything of the sort before, and until the first test in the New Mexico desert, no-one knew whether it was even possible.

of the knowledge base will be shown in chapter 4). Suppose the set of artifacts we are trying to design is the the set of heat exchangers, then we need to know their type (fluid-fluid or regenerative), their physical size (diameter, length, thickness of the wall), the kind of materials they are made of (aluminum, iron or steel), etc. The knowledge is *hierarchical*, for example, tubular heat exchangers, plain heat exchangers, and finned heat exchangers are all subclasses of fluid-fluid heat exchangers: they share some common properties, yet they keep their own ones. The example I gave may look simple, while the real world design is much much more complicated and the ever-increasing level of sophistication in design knowledge presents a great challenge for the designer.

Many research and development efforts are underway to make knowledge base technology suitable for engineering applications, [Ashley 92], [Papalambros 87], [Reichgelt 91], [Rinderle 91], and [Ward 87], to name several examples . One difficult task is to decide in what form the knowledge should be represented. A good versus bad representation can make an enormous difference to the success or failure of a knowledge-based system. It seems that there is an inevitable trade-off between the generality and the efficiency of knowledge representation. Brown and Chandrasekaran's AIR-CYL system is a notable example of a highly efficient and extremely specific knowledge base [Brown 89]. Knowledge bases written in this way cannot be re-used to support design of a different class of artifacts. Since it is expensive to develop a knowledge base, there are very few applications for which the cost of developing an original knowledge base could be justfied. At the other extreme, we have the example of *How Things Work* project at Stanford University [Cutkosky 90]. This knowledge base basically consists of the laws of physics and mechanics. Of course, knowledge bases written in terms of laws of physics and mechanics would be eminently reusable. But it is a daunting task

to design, say, a pair of scissors, from these laws.

We have to balance generality and efficiency in order to find an appropriate level of knowledge representation. Then we have to face the question — *What is the appropriate level to represent design knowledge* ? It is safe to say that the appropriate level of knowledge representation depends on the specific design subject, or in other words, the adequacy of the representation depends on the task we are trying to perform. However, we can also say that to be appropriate, the representation should be suitable for each stage of the design process; design requirements should always be expressible in terms of the representation; the same representation must be useful for all knowledge in the subject area and the representation should come naturally to humans. We mentioned that knowledge representation is hierarchical, the knowledge representation should be able to express the design specifications at the highest level; also, the explication of any statement should be independent of any other explication and statements can always be explicated at the next level down. The lowest level of the language corresponds to detailed engineering drawing which a CAM system could translate into physical operations.

Knowledge can be represented in different forms: expert systems, databases, algorithms etc. *Expert Systems* is the traditional way of design knowledge representation. We will give a brief review of expert systems technology next.

1.2 Expert Systems

Expert systems are computer programs to reason about some specific domains using specialized databases known as knowledge bases [Hayes-Roth 83]. There are

two major components in an expert system : the knowledge base *(knowledge of the experts)* and the inference engine. When given an input in the form of a query, the inference engine consults the knowledge base to produce an answer. Expert systems differ from conventional programs in that the former distinguish the knowledge base from the inference engine while the latter simply combine the knowledge with the control algorithm.

Expert systems can be classified into two categories: *Rule-based* and *Model-based* expert systems. The first generation of expert systems were rule-based systems, representing knowledge only in the form of production rules:

```
IF     <antecedent(s)>
THEN   <consequent(s)>
```

One notable example of rule-based expert system is R1 (a knowledge-based VAX configurating system) [Sell 1985], which was developed by John McDermott and his colleagues at the Carnegie-Mellon University at the request of Digital Equipment Corporation (DEC). R1 can translate customer's orders into complete and coherent configurations. R1 went into operation in January 1980. DEC calculated that by 1984 they would have required 80 more staff without R1, and they were convienced that R1 did the job much better than people could do it. Other examples of rule-based expert systems include: MYCIN (blood diagnosis) [Buchanan 84], MACSYMA (symbolic mathematics) [Martin 71], PRIDE (paper handling system) [Mittal 86], and PROSPECTOR (mineral exploration) [Duda 79].

Though these systems can achieve *expert* level of performance, their domains are highly restricted and they depend too much on the production-rule representation.

When the number of rules become very large, the rule-based systems are very difficult to organize, maintain and update. Moreover, they have no knowledge of the components of the domain, the way those components are connected or the way those components behave and interact. Therefore rule-based expert systems are subject to many limitations, they are recommended and successful only when the given problem domain is quite small, well-understood and when there is a general agreement among domain experts [Luger 89].

By comparison, model-based expert systems have many advantages. Instead of using a collection of condition/action pairs obtained from a domain expert, model-based systems use a model to represent a behavioural theory about a certain class of artifact [Davis 84]. The model has a structure resulting naturally from the structure of the artifact being modelled, and can take advantage of similarities between different components and assemblies. Changes to the artifact can be naturally included as changes to the model, therefore the maintenance of model-based expert systems is easier. Mostly important, the model can be used for more than one task. For example, in the Stirling engine heat exchanger knowledge base described in the later chapters, the knowledge base can support the creation of multiple instances of components, say, a heater, a regenerator, and a cooler, all from the same knowledge base. The model can be used in design as well as in analysis.

1.3 Motivation

The purpose of this research is to develop a model-based expert system for computer-aided design based on an object-oriented constraint logic programming language,

Echidna. Our research group is working on two different design domains: architectural design and mechanical engineering design. My involvement is in the area of mechanical engineering design. The work described in this thesis is the development of a two-part knowledge base that will ultimately support design of a range of artifacts. We made a distinction between *facts* and *strategies*; we also incorporate *scaling* information in our knowledge base. These features are not found in other current knowledge-based systems. I have been working on a prototype of such a knowledge base that supports Stirling engine heat exchanger design. We believe that Echidna allows us to take a new design approach which can permit our system to take over greater initiative and wider areas of expertise in the future.

The rest of this thesis is organized as follows. In chapter 2, some background knowledge of the developing tool, Echidna, is given. Chapter 3 gives a brief introduction to the design subject, the Stirling engine. Chapter 4 addresses our new approach to represent design knowledge and the application of Monotonicity Analysis in the development of the knowledge base. The method to test the knowledge base and a detailed design example is discussed in chapter 5 and chapter 6, respectively. Chapter 7 summarizes the results and outlines some work for future research. Some sample files of the knowledge base and Echidna's output files of the design example are included in the appendices.

Chapter 2

Echidna

Echidna is a new type of constraint logic programming (CLP) language for model-based expert systems applications. It is under ongoing development by William S. Havens and his colleagues of the Expert Systems Laboratory at Simon Fraser University. The language improves upon the limitations of existing expert system languages by combining aspects of schema-based knowledge representations, constraint logic programming, and intelligent backtracking. This chapter provides a limited review of Echidna, more information regarding Echidna can be found in [Havens 90] and [Sidebottom 92]. Two research areas related to our work — object-oriented programming and constraint logic programming are also discussed. The vast quantity of publications covering these topics prohibits a complete review, but this review is representative of current work and provides a firm foundation for further study.

2.1 Object-Oriented Programming

Object-oriented programming is a paradigm in which a software system is decomposed into subsystems based on objects [Zeigler 90]. Computation is done by objects exchanging messages among themselves. The paradigm enhances software maintainability, extensibility and reusability.

In OOPS (Object-Oriented Programming Systems), an object is a conglomerate of data structures and associated operations, usually representing a real world counterpart. Objects are usually not defined individually, instead, a class definition provides a template for generating any number of instances, each one an identical copy of a basic prototype. Objects are usually given generic descriptions so that classes may be generated at will. Classes of objects form a hierarchy in which they are arranged according to their degree of generality.

Echidna is object-oriented. It allows us to represent knowledge in a modular form, and we can test, add or remove the modules independently. Echidna also provides *inheritance*[1] so it allows us to move from general to specific, that is, a schema may be defined as a subclass of a more general parent schema and inherits its properties and methods. In this way, it reduces the need for new code. Representations of knowledge shown to be effective in one area can therefore be generalized to other applications.

In Echidna, each object has two types of information associated with it. First, there are attributes and their values. Second, there are small programs, called *methods*, which can be used to perform certain calculations. As mentioned earlier, objects communicate with each other by sending *messages*. Syntactically, a message is the

[1] Echidna supports only single inheritance; a class has exactly one superclass.

name of a method followed by arguments, which is sent to a schema instance via an operator; a method is a predicate in the logic programming sense and is defined within a particular schema (except *Global* methods which are defined outside of any schema definition in the knowledge base). The message-passing paradigm enforces modularity, data hiding and well-defined interactions between objects.

Here is an example from the Stirling engine heat exchanger knowledge base:

$$Metal : density(MetalDensity).$$

In this example, Metal is an object; density(MetalDensity) is a message in the object-oriented programming sense. The object named Metal receives the message *density*/1[2]. Interpreting the message causes a clause from the method *density*/1 in the object Metal to be chosen and called, which in turn, finds the appropriate value of density and unifies it with the variable named MetalDensity.

2.2 Constraint Logic Programming

We are dealing with constraint-satisfaction problems (CSP). They can be stated as follows: Suppose there is a given set of variables, a finite and discrete domain for each variable, and a set of constraints which are persistent data links between variables. The flow of information between variables is through the constraints. Each constraint is defined over the set or some subset of the variables and limits the combinations of

[2]A logic term is often referred to by its *functor* which is the function and arity of a term f/n, here *density* is the function and arity is 1, therefore this message can be referred to as *density*/1.

values those variables can take. Design is constraint oriented, constraints are continually being added, deleted and modified throughout the design process [Mittal 90]. If we can find one assignment of values to the variables which can satisfy all the constraints, the assignment is the solution or one of the solutions [Hentenryck 89]. We have two rather different schemes for solving CSP: backtracking and constraint propagation [Kumar 92]. Next I will discuss these two schemes.

2.2.1 Backtracking

The backtracking scheme is an improvement on the generate-and-test paradigm, which simply generates and tests each possible combination of the variables until the first combination that satisfies all the constraints is found. The number of combinations considered by this method is the size of the Cartesian product of all the variable domains. The backtracking paradigm instantiates variables sequentially. As soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial instantiation violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has alternatives available. Whenever a partial instantiation violates a constraint, backtracking can eliminate a subspace from the Cartesian product of the variable domains.

Although the standard backtracking (depth-first search chronological backtracking) does prune significant portions of the search tree, it still takes exponential time in the number of variables in the worst case. Besides, it often suffers from *thrashing*, that is, the search in different parts of the space keeps failing for the same reasons.

To correct the inefficiency of chronological backtracking, Echidna uses dependency-directed backtracking [Stallman 77]. When a failure occurs, chronological backtracking simply undoes the most recent choice, regardless of whether that choice is the cause of the failure. Obviously, this is not good and often causes redundant work. In contrast, dependency-directed backtracking looks for the most recent choice 'involved' in the failure (the cause). Once the cause is found, it is tried for an alternative value.

Let's look at a simplified example. Suppose there is a query:

$$? - m1(A, B), m2(C, D, E), m3(A).$$

Initially m1(A) succeeds in binding A to 10 and B to 20 and it takes a long time to bind C, D, E to 30, 40, 50 respectively. If later, m3(A) fails for some reason, standard chronological backtracking will undo both m2(C,D,E) and m1(A,B). However, m2(C,D,E) has nothing to do with the failure in m3(A) since they share no common arguments, the real culprit is m1(A,B) for it assigned 10 to variable A in the first place. As it requires a large amount of work to succeed in binding C, D, and E to their present values, it is a waste to do it all over again. Intelligent backtracking can detect that m2(C,D,E) is independent of the failure m3(A), therefore keeps the binding results of m2(C,D,E) and backtracks directly to m1(A,B) to find a new value to A[3]. Then m3(A) will be tested with the new value of A from m1(A,B). Several iterations may be needed before all the variables can be ground.

[3]Though B is independent of the failure m3(A), it has to be undone since m1(A,B) involves both A and B.

2.2.2 Constraint Propagation

During constraint propagation, constraints are applied to restrict variable domains to only those values that can participate in the solution to the problem, thus constraint propagation significantly reduces the search space. By performing constraint propagation, the original CSP is essentially transformed into a simpler CSP. In the process of constraint propagation, certain failures are identified, and the search space is effectively reduced so that these failures are not encountered at all in the search space of the transformed CSP. Constraint propagation is implemented using an arc consistency algorithm [Mackworth 77] adapted to k-ary relations over discrete domains. The constraint propagation scheme has the disadvantage of being expensive; experiments by many researchers with a variety of problems indicate that it is better to apply constraint propagation only in a limited form [Haralick 80; Dechter 89; Prosser 91].

2.2.3 A Combination of Backtracking and Constraint Propagation

A good way to solve CSP is to embed a constraint-propagation algorithm inside a backtracking algorithm. However, to avoid the thrashing problem of the standard backtracking, a truth maintenance system [Doyle 79; McDermott 91] is needed to support intelligent backtracking. The way it works is as follows: A variable is assigned some value, and a justification for this value is noted. Then similarly, a default value is assigned to some other variable and is justified. At this time, the system checks whether the current assignments violate any constraint. If they do, then a new node

is created that essentially denotes that the pair of values for the two variables in question is not allowed. This node is also used to justify another value assignment to one of the variables. This process continues until a consistent assignment is found for all the variables.

Echidna has a built-in mechanism to control dependency-directed backtracking and constraint propagation [Sidebottom 91]. It is quite powerful in solving CSPs, but it still has some problems which we will discuss next.

2.3 Potential Problems With Echidna's Constraint Propagation Mechanism

Echidna has two built-in tools to perform constraint propagation, namely, *indomain* and *split*. *Indomain* will select a unique value for a single variable, while *split* takes a list of variables, halves the domain of each, selects one half of each domain, then enforces constraints; if no failures occur, *split* repeats the process until the variable domains have been halved a predetermined number of times. *Split* might be thought of as allowing breadth-first search. Ideally, the *split* command would divide the domains of its variables into intervals and a set of constraints would then be applied which would eliminate all of those intervals in which the constraints failed to hold.

In the process of testing our Stirling engine knowledge base, we discovered a problem with the constraint propagation mechanism of Echidna and it further led us to the idea of adding *scaling* information to the knowledge. We think this is a new and

useful form of knowledge representation. I will first describe the problem we found, then in chapter 4, discuss our new form of knowledge representation.

In Echidna, each variable has an initial domain, that is, it has an initial range of possible values. We can impose constraints on these variables and take advantage of the built-in constraint propagation mechanism; the range mentioned earlier will be restricted progressively until the variable has a single value. This is always true in theory, however, we found that how successful a constraint is in restricting the domain of a variable depends on the size of the domains of any other variables that may appear in the constraint. Consider the following example.

```
schema try
      {
              [1,    100000] A.
              [1,    100000]  B.
              [1,    10] C.

          try :-
              A =:= 2 ,
              B =:= 3 ,
              C =:= A + B ,
              print(C).
      }
```

We would expect Echidna to come up with an answer C =:= 5. Instead, we will get C =:= [1.984375, 10]. So variable C is not ground even after constraint

propagation.

There is a way to solve this problem: we could set the 'precision[4]' from its default value precision(6) to a very high value, say, precision(24). This can give us C =:=5. In this very simple example, increasing precision may not be a bad idea, however, in most cases, increasing precision is quite costly and slows the computation process significantly. So generally this is not a good solution.

We found that if we reduce the initial ranges of A and B and set them to the same order of that of C, then we can obtain the expected result.

```
schema try
       {
              [1,    10] A.
              [1,    10]  B.
              [1,    10] C.

          try :-
              A =:= 2 ,
              B =:= 3 ,
              C =:= A + B ,
              print(C).
       }
```

[4]The *Precision* is a global variable internal to Echidna. It is an integer representing the degree to which a real interval variable is refined. The global precision can be increased from its initial value of 6 to a maximum value of 48, and the precision of individual variables can also be set to a multiple of the global precision.

This gives us $C =:= [4.9375, 5.148438]$. Thus C is refined to very close to 5.

We have encountered several other similar problems and we arrived at a conclusion that we should avoid using large domains for variables. However in real world design, many design variables do have large domains, for instance, the power output of a Stirling engine can be as small as 1 watt and as large as 1 megawatt, a range of six orders of magnitude. This may be a problem when we use Echidna to develop the knowledge base to support Stirling engine design. We have to find a way to avoid the problem. The solution we found is based on 'scaling'. This aspect will be addressed in chapter 4.

Chapter 3

A Knowledge Base For Stirling Engine Design

3.1 A Brief Introduction To Stirling Engines

This chapter gives an introduction to Stirling engines. For more details on the subject, the reader can refer to [Walker 80], [Ross 77], and [Jones 82].

3.1.1 What Is A Stirling engine?

The history of the Stirling engine can be traced to 1819 when a Scottish inventor, Robert Stirling, built the original device. During the last century, Stirling engines competed with the steam engine, until they were both replaced by the internal combustion engine around the beginning of this century.

A Stirling engine, like the gasoline, diesel, and jet engines with which we are more familiar, is a heat engine; that is, an engine that derives its power from heat. However, unlike those other engines, a Stirling engine obtains its heat from outside, rather than inside, the working cylinders. The Stirling engine is quite omnivorous with respect to fuel; literally, any source of heat, as long as its temperature is high enough, will power a Stirling engine. This last statement is true of any other externally heated engines, like the steam engines, but Stirling engines hold the promise of developing the most power for a given supply of heat (or fuel) of the practical alternatives presently known; besides, it can also use stored heat. In terms of pollutants, Stirling engines are among the cleanest heat engines available. Stirling engines can have the same high efficiency and part-load performance of diesel engines, can match the high specific output of gasoline engines and yet have the favorable low-speed torque characteristics of steam engines. These factors all account for the present interest in the Stirling engines. Stirling engines can be used as prime movers, heat pumps and pressure generators, and they are beginning to attract attention for their use in artificial heart, household refrigeration and solar-power applications A market niche is also emerging for the use of Stirling machines in the cooling of computer chips [O'Connor 92].

3.1.2 How Does It Work?

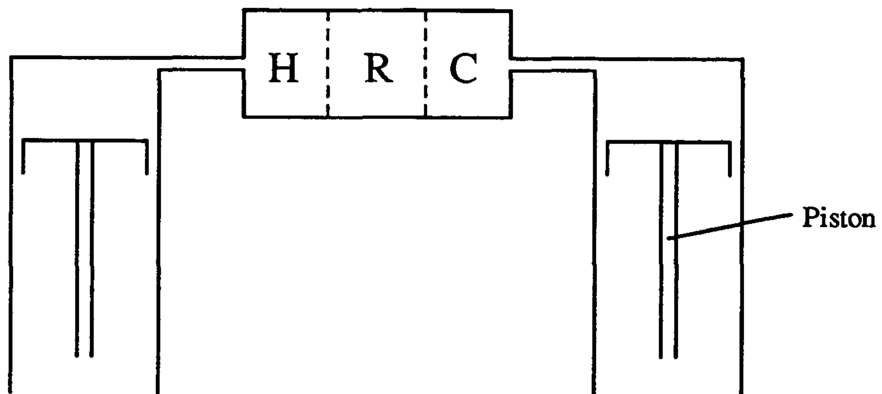
A Stirling engine operates on a *closed* regenerative thermodynamic cycle[1], with cyclic compression and expansion of the working fluid at different temperature levels. The Stirling engine operates with a fixed mass of working fluid rather than periodically taking in working fluid and exhausting it to the atmosphere after heating and expansion. The flow of working fluid is controlled by *volume changes*, and there is a net conversion of heat to work or vice versa.

The basic operation involves heating and cooling the working gas. The working gas is stored in the cylinder. In Stirling engines, as in other heat engines, the heater is an essential element. Heat is absorbed from the hot end and rejected at the cold end, and work is done equal to the difference between those two quantities of heat. The Stirling engine also contains two other heat exchangers, namely the regenerator and the cooler, which have no analogues in the internal combustion engine. A regenerator is a heat exchanger in which heat is transferred from the working fluid to a solid matrix and back again over the cycle. It consists of a capsule containing the matrix material, which may be stacked wire screens, a metal foam, or a packed bed of spheres. The regenerator removes the residual heat in the working fluid after its expansion and stores the heat for re-absorbtion later in the cycle. When the fluid is compressed, heat generated in this process will be removed from the cooler. A cooler is similar to a heater but its function is cooling rather than heating.

There are many configurations of the Stirling engine. Three common configurations are shown below. For detailed descriptions, refer to [Walker 80] and [Jones 82].

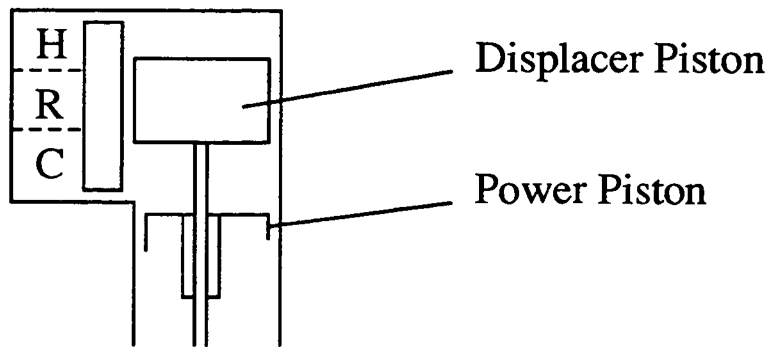
[1]The Stirling engine doesn't operate on Stirling cycle, rather, it approximates to the Otto cycle.

In our present knowledge base, we consider the design of the heater, cooler and regenerator. We will extend our knowledge base to include the design of other components such as the displacer piston and power piston.



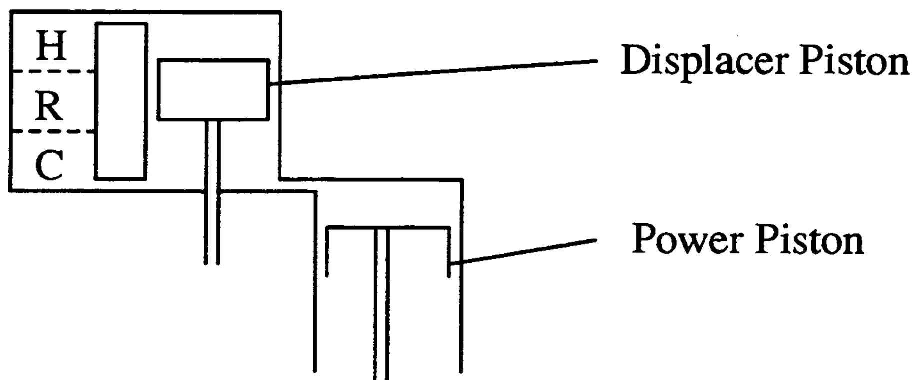
H: heater
R: regenerator
C: cooler

Figure 3.1: Alpha Configuration



(2) Beta

Figure 3.2: Beta Configuration



(3) Gamma

Figure 3.3: Gamma Configuration

3.2 Structure of The Knowledge Base

We developed a two-part knowledge base to support the design of Stirling engine heat exchangers. The knowledge base represents two kinds of knowledge: *facts* concerning the properties of potential design components; and *strategies* for combining these components into an artifact that can meet given requirements. The former kinds correspond to descriptions of the behaviour of the components in terms of the laws of physics; the latter kinds correspond to methods for selecting and assembling these components. Facts are usually expressed as constraints, while strategies are expressed by the ordering of clauses.

Here are some examples of *facts*:

$Area =:= 0.25 * pi * Diameter * Diameter$,

Density of hydrogen is 0.088 kg/m^3.

Facts are, of course, always true, but they provide no guidance by themselves.

In contrast to rules which are *always* true, design strategies are heuristics which are *usually* valid. They are guiding principles which the designers use to make intelligent decisions based on previous experience. Examples of *strategies* :

```
If the heatflow is more than 1000W, the tubular
heat exchanger is probably the best to choose;
If the heatflow is between 10W and 1000W, then
finned heat exchanger may be most suitable;
If the heatflow is less than 10W, plain fluid-
fluid heat exchanger should be used.
```

```
Determine the size of the engine according to
the power requirement.
Use the heuristic "stroke==bore" to determine
the cylinder and piston dimensions.
```

The distinction we made is not found in traditional CAD systems: knowledge of the set of artifacts versus knowledge of the particular artifact being designed to meet the current design requirements. The former knowledge is unchanging, the latter is built up gradually over one or more design sessions. Why do we need to make the distinction? Firstly, it opens the possibility of using different strategies on the same set of facts; Secondly, it allows facts about components to be used for different design tasks and to be checked independently.

Ideally, *facts* and *strategies* would be entirely independent, but in some cases, this division would be quite fuzzy when examined closely. For example, in a Stirling engine, the heater is usually connected to the expansion space of the engine. If we write this into the *facts* knowledge base, we have precluded using the heat exchanger model for designing other artifacts containing heat exchangers. If we are to make the division, we cannot make it on ideological grounds, rather, we must make it on pragmatic grounds. The deciding factor is, as we mentioned earlier, the tradeoff between generality and efficiency; the closer to uninterpreted physical laws we make the *facts*, the more general the applicability of the *factual* knowledge base, but the more difficult the task of synthesising a solution to any given problem. The rule we followed for deciding how to make the division is *start specific, work towards the general*; this way we are sure that the knowledge base will be good for at least one

thing.

The knowledge bases we developed in this research are of two types: the first is called '*agent ∗ ∗.kb*' are concerned with strategies for design, such as agentHX.kb and agentMat.kb; the second is concerned with facts about possible components of a design, such as reg.kb and material.kb. Sample files are included in Appendix A. The separation between agents and components is intended to increase the re-useability of the knowledge recorded in the knowledge bases.

My work mainly focuses on the development of the *factual* knowledge base which consists of schemata representing potential components of a Stirling engine. We are using an object-oriented approach by which the engine is represented in terms of two orthogonal hierarchies, a classification hierarchy and a component hierarchy. In our example of the Stirling engine knowledge base, there is a description of a generic heat exchanger; a *fluid-fluid* heat exchanger can be defined as a subclass of the generic heat exchanger, and it inherits the properties and methods of that class (It may also have particular properties and methods of its own). Consequently a *tubular* heat exchanger can then be defined as a subclass of the *fluid-fluid* heat exchanger, and inherits the properties and methods of that class. This hierarchy is illustrated in Figure 3.4.

The component hierarchy describes the 'part of' relationship between objects, for example, matrix and capsule are both components of the regenerator (Figure 3.5). Certain methods may make explicit use of this hierarchy; for example, the total mass of an assembly can be found by recursively finding and summing the masses of all its subassemblies.

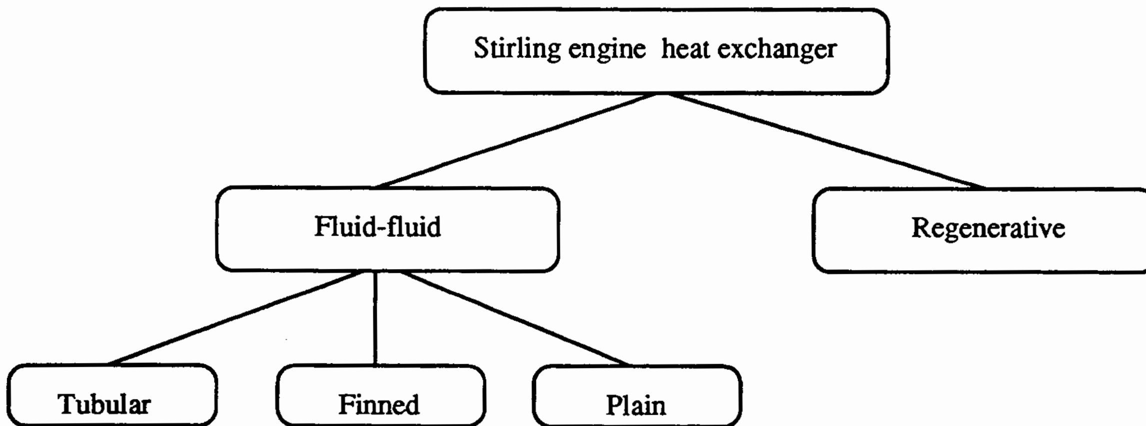


Figure 3.4: Classification Hierarchy

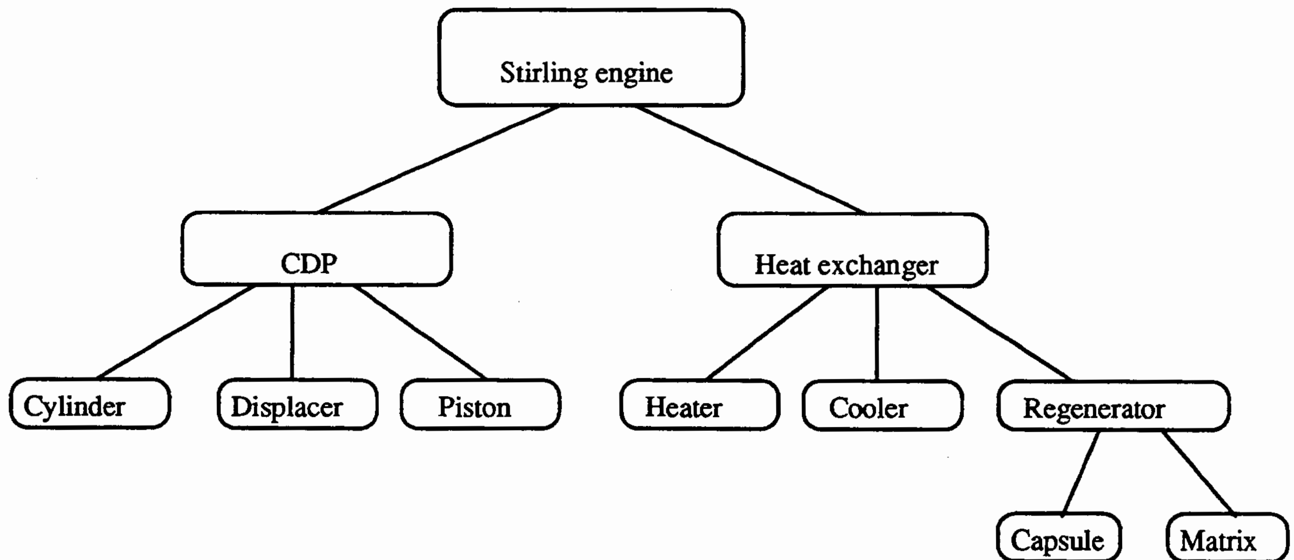


Figure 3.5: Component Hierarchy

There are a number of parameters associated with each object, for example, InnerDiameter, OuterDiameter and Length are all parameters of a tube. Parameters can be either numeric or symbolic. Besides those numeric parameters mentioned above, a tube also has a parameter 'material' which may be any one of a library of material types represented in the knowledge base. An object is fully specified when all its parameters have been assigned values and all of its components, if any, have been fully specified. The values which parameters may take are restricted by constraints defined between objects, reflecting the relevant physical relationships. The component hierarchy is of more importance, as it is the key to extending the knowledge base into other areas of knowledge.

3.3 How Is The Knowledge Base Used ?

Before use, the knowledge base represents the generic engine, or the set of all possible engines: none of its parameters has been assigned a value and no components have been selected. When the designer specifies a particular set of design requirements, the set of engines meeting these requirements shrinks as the constraints propagate, restricting the possible values of the parameters. The parameter values of each component are initially unknown, but will be gradually determined as the implications of the design requirements and the agent's heuristic decisions are propagated through the network of constraints. It may happen that the initial choice of components does not lead to any solution; in this case, Echidna uses dependency-directed backtracking to identify and change the choices responsible for the failure. In chapter 6 we will present a detailed example to describe the operation of the knowledge base.

Chapter 4

A Sense of Scale

4.1 Why Do We Need Scaling ?

While testing and modifying the knowledge bases, we found that many variables in the constraints have very large initial domains. This made it hard for Echidna to ground some of the variables after constraint propagation, and, sometimes, there would be no solution.

The problem could be ameliorated if we reduced the initial domain size of the variables. But one would immediately argue that by doing this we also have reduced the generality of the knowledge bases and we should not trade off efficiency against generality in this way.

To solve this problem, we introduce the concept of scaling.

An experienced designer has a good sense of scale, and uses it routinely. For example, if we ask an experienced automobile engineer — *Can you make an internal combustion engine that can deliver 500 HP and fit inside a soup can ?* He or she would say no, and we would expect him or her to say this immediately. Yet it would be very difficult to point to any physical principle that makes this goal impossible. There might be no shorter way of *proving* it impossible than by examining a variety of attempts to reach the goal, and seeing that each one fails — and each might fail for different reasons, one because of limitations in the strength of materials, one because of the limited reaction rates of any possible fuel/air mixtures. As we lower the power requirement, we eventually come to an engine that could be built, but it is impossible to identify a sharp threshold between impossibility and possibility.

For each schema in the knowledge base, we distinguish a number of different possible scales, each corresponding to certain domains of the schema's variables. Before creating an instance of the schema at run time, the design agent looks at the required performance and decides which of the variable sizes is *most likely* to meet that level of performance, then restricts the variable domains appropriately. The use of scaling increases the efficiency of the design process in two ways: firstly, at the level of the design heuristics: failures will occur as soon as the knowledge base begins to consider an unsuitable design rather than occurring after a long series of choices and constraint propagation; secondly, at the level of constraint propagation, where as we mentioned earlier, constraint propagation is more powerful if the domain of the variables are reduced.

Once the idea of scaling has been introduced, we recognize that a lot of engineering design knowledge can be represented in this way. As the scale of a design changes, we

will cross the threshold at which different design methods become appropriate. For example, if the design is to store small quantities of electrical energy, then a capacitor might be a suitable device, but if we want to store enough energy to start a car, we would look at battery storage. So we will make the choice of design methods depend on scale, as well as the initial variable domains. More discussions on the idea of scaling can be found in [Jones 93].

4.2 Creating Scaled KB

In order to incorporate *scaling* information in the knowledge base, we can, for instance, replace the component *tubular heat exchanger* with a family of components: *very small tubular heat exchanger*, *small tubular heat exchanger*, *medium tubular heat exchanger*, *large tubular heat exchanger*, and *very large tubular heat exchanger*. The members of a family basically share the same set of constraints, though sometimes some minor adjustments are needed as the scale changes – for example, the loss in engine efficiency due to heat transfer to the cylinder walls must be included when designing a very small engine, but is negligible when designing a very large engine.

We need to provide a label to each member of a family showing the upper and lower bounds on the values of some of the schema variables. Then, when the design agent reads the labels, it can avoid choosing components for which the desired performance was out of range. This obviously increases the efficiency at the level of design heuristics, which we mentioned earlier.

If we know there is a unique order in which variables will be ground, we can select bounds on those that will be ground first, and deduce bounds on the other variables

from these. We introduce two guidelines which help us to decide the order in which these variables will be ground.

The first one is the distinction between *design parameters* and *functional requirements.* The functional requirements are those properties of the artifact whose values are supplied to the designer as targets to be achieved, while design parameters are those properties whose values the designer may set to meet those targets [Suh 90]. For example, we want to design a tubular heat exchanger which can handle 2000 watts of heat flow. We can set *heat flow* as the functional requirement. In order to meet this requirement, the designer can choose the value of the diameter of the heater, the internal and external diameter of the tube and the length of the tube. These are design parameters. After we make the distinction, we would want to set bounds on the design parameters first, and deduce bounds on the functional requirements. However, this distinction cannot be applied rigorously since we intend to make the knowledge base reusable for a variety of design problems: some problems may involve maximizing the power output, others may involve maximizing efficiency, etc.

When the distinction mentioned above is not clear, we may make use of a related distinction from object-oriented programming,*private* versus *public* variables of an object. The idea is that some properties of the component are relevant to the design of the rest of the artifact, while others are only important when we are designing that particular component. For example, in our Stirling engine heat exchanger knowledge base, we can set 'PressureDrop' as a *public* variable because it is needed to calculate the efficiency of the engine as a whole; by contrast we can set 'Density' of material for the regenerator matrix as a *private* variable since it doesn't directly affect anything outside the regenerator. We choose to set bounds on the private variables first, and

deduce bounds on the public variables.

It must be admitted that these two guidelines will in general be inadequate to determine a unique order in which to set bounds on the schema's variables. However this is not a fatal problem, for it only affects the efficiency of the knowledge base, not its correctness or completeness.

Scaling appears to be quite straightforward: choose reasonable upper and lower bounds on some variables in a given schema, and deduce upper and lower bounds on the remaining variables. However, we need to be very careful in doing this so as to make the knowledge efficient. Consider the following example:

```
[1,  10]   X.
[1,  25]   Y.
Y =:=  X * 2.
```

Suppose this is part of a knowledge base for design. In the course of design, we may use 'indomain' to choose a random value of X from its range. Whatever value we choose, the knowledge base will be consistent. But if instead we begin by choosing a value of Y, there is the possibility that we will choose a value between 20 and 25, in which case there will be a failure. Although Echidna can recover from the failure by backtracking, we have wasted time by making a choice which we could have avoided. So it is important that the ranges chosen for the variables be consistent with each other, given the constraints in the knowledge base. The principle we need to follow is – For any value chosen from the initial domain of a variable, there must be at least one value in the initial domains of each of the other schema variables such that the schema constraints can be satisfied [Mackworth 77]. This principle tells us to restrict

the initial domains of variables, to eliminate inconsistent values. At the same time, we do not want to restrict the domains more strictly than this principle requires, or we will eliminate potentially valid solutions, making the knowledge base incomplete.

In a complex schema, a given variable may appear in several constraints, and it may be difficult to tell which constraint actually determines the upper and lower bounds on its potential values. A systematic way is needed. That's our next topic – *Monotonicity Analysis.*

4.3 Application of Monotonicity Analysis

4.3.1 The Problem

In a scaled knowledge base, every variable is assigned some domain. We discovered an interesting problem and we were able to solve it using the *Monotonicity Analysis* technique.

In reg.kb (regenerator), there are five variables: R [Reynolds number], h [convective heat transfer coefficient], Nu [Nusselt number], D_H [hydraulic diameter of the passage in the matrix], and G [mass flux]. They are related by three constraints:

$$Nu = CR^n$$

$$h = NuK/D_H$$

$$R = GD_H/\mu$$ [1]

[1] C, K and μ are constants.

We have discovered that for the given set of constraints and the given initial bounds on R, h, Nu and D_H (none of which we want to change), G should be set within a certain range and we want to find an explicit expression for this range. The expression should be in terms of the limits for the other variables involved: $R_{High}, R_{Low}, h_{High}, h_{Low}$, etc. The expression should ensure that, for any value in the range allowed for G, there will be values in the initial ranges of R, h, Nu, and D_H consistent with this value.

We can do this by using the axioms of interval arithmetic, however, the reader may easily verify that simply eliminating variables from this system of equations will give different bounds on G, depending on which variables are selected for elimination. This is not a particularly complicated problem. It would be much more confusing if there are many variables and constraints involved. Clearly, a systematic approach is required. Fortunately, such an approach exists, it is called *Monotonicity Analysis.*

4.3.2 Basics About Monotonicity Analysis

Monotonicity Analysis is an interactive partial optimization technique to reduce the dimensionality of the problem and detect flaws in the problem formulation. At this point, it is necessary to give the reader some background knowledge of *Monotonicity Analysis* by introducing some simple but important concepts and principles. For further details, please refer to [Papalambros 88].

Basic Concepts

- A discrete or continuous function f(x) is *strictly globally monotonically increasing* over a domain Ω if and only if $f(x_1)<f(x_2)\ \forall\ x_1, x_2 \in \Omega : x_2>x_1$. The monotonicity of a function with respect to (w.r.t.) a variable is designated by a + or - superscript for the argument in the list of the function. For example, $f(x_1^+, x_2^-)$ implies that the function is monotonically increasing w.r.t. x_1 and decreasing w.r.t. x_2.

- A function having a minimum (maximum) will be called *well bounded* from below (above).

- A constraint is said to be *unconditionally active* or *critical* if elimination of it will lead to an unbounded or degenerate solution. For an inequality constraint, this means that it must be satisfied with strict equality. If a constraint is critical for more than one variable, we say it has *multiple criticality.* An *active* equality constraint can be changed to an *active* inequality constraint, the direction of which defines the *directionality* of the equality.

- A set of constraints is said to be *conditionally active* if elimination of all the set will lead to an unbounded or degenerate solution.

- Any variable that occurs in an active constraint is described as *relevant*; Any variable occurring in all constraints of a conditionally active set must be *relevant.*

Two Monotonicity Principles

The First Monotonicity Principle (MP1)

In a well-constrained objective function every increasing (decreasing) variable is bounded below (above) by at least one active constraint.

The Second Monotonocity Principle (MP2)

Every monotonic nonobjective variable[2] in a well-bounded problem is either

- irrelevant and can be deleted from the problem together with all constraints in which it occurs, or
- relevant and bounded by two active constraints, one from above and one from below.

Monotonicity Analysis Process

We show in the next page a flow chart which describes the process of *Monotonicity Analysis*. First we need to put the problem statements into a standard qualitative form and build a model for the particular problem. Then we check out the monotonicities of the variables and put this information into a table called *Monotonicity Table*. Next we applied monotonicity principles (MP1 and MP2) to check the problem model. If we find that the model is *well bounded*, we can proceed and apply classical optimization methods, such as Gradient descent, Newton's method, Simplex algorithm, etc. If at this stage, we have already found that the problem is not *well bounded*, we should go back to check the problem statements because probably there is something wrong with

[2]If there is a monotonic variable which occurs in the constraint(s) but not in the objective, it is called a nonobjective variable.

them, or the problem doesn't have a solution. By using *Monotonicity Analysis*, we can reduce the dimensionality of the problem and avoid wasting effort on a problem which doesn't have a solution.

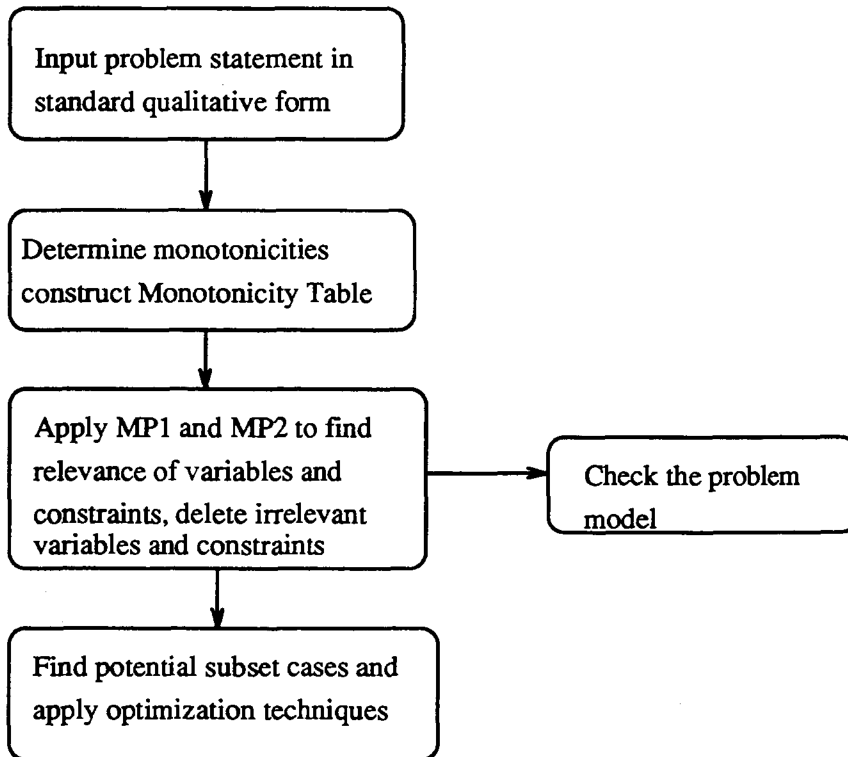


Figure 4.1: Monotonicity Analysis Process

4.3.3 Application

Let's go back to the problem of finding explicit limits of G. To find G $[G_{Min}, G_{Max}]$, separate the problem into two parts. First, find G_{Min}; second, find G_{Max}.

To find G_{Min}, write the problem in a standard form:

Objective:	minimize G	
Subject to:	(1) $Nu = CR^n$	
	(2) $h = NuK/D_H$	
	(3) $R = GD_H/\mu$	
	(4) $h - h_{High} \leq 0$	;h_{High} is the upper bound on h
	(5) $-h + h_{Low} \leq 0$	;h_{Low} is the lower bound on h
	(6) $R - R_{High} \leq 0$	;R_{High} is the upper bound on R
	(7) $-R + R_{Low} \leq 0$	;R_{Low} is the lower bound on R
	(8) $Nu - Nu_{High} \leq 0$	;$Nu_{High}$is the upper bound on Nu
	(9) $-Nu + Nu_{Low} \leq 0$	;$Nu_{Low}$is the lower bound on Nu
	(10) $D_H - D_{HHigh} \leq 0$	;$D_{HHigh}$is the upper bound on D_H
	(11) $-D_H + D_{HLow} \leq 0$	;D_{HLow} is the lower bound on D_H

We put the information into a form called the *Monotonicity Table.*

Objective: minimize G

Monotonic variable	G	Nu	h	D_H	R
Relevance	*			*	*
Direction +	+				
Direction -	...				
(1) $Nu = CR^n$		?			?
(2) $h = NuK/D_H$		?	?	?	
(3) $R = GD_H/\mu$	$*_-$			$*_-$	$*_+$
(4) $h - h_{High} \leq 0$			+		
(5) $-h + h_{Low} \leq 0$			-		
(6) $R - R_{High} \leq 0$					+
(7) $-R + R_{Low} \leq 0$					-
(8) $Nu - Nu_{High} \leq 0$		+			
(9) $-Nu + Nu_{Low} \leq 0$		-			
(10) $D_H - D_{HHigh} \leq 0$				+	
(11) $-D_H + D_{HLow} \leq 0$				-	

Table 4.1: Monotonicity Table

This table shows the result of the first cycle of *monotonicity analysis.* G is called the *objective variable* since it appears in the objective we seek to minimize, so we can add an asterisk to cells that have G in them. This shows that G is a *relevant* variable. A " + " is added to the Direction + cell because G increases in the objective to be minimized. We can also find that only constraint (3) bounds G, so constraint (3) must be critical and bounds G from " $-$ " direction. This also shows that R and D_H are also relevant although they do not appear in the objective function, so we can put asterisks under D_H and R. The multiple criticality of constraint (3) indicates that it can be used to eliminate a variable. Since the objective and the constraint (3) both have variable G, it can be eliminated. Two other equality constraints have a common

variable Nu, so we can substitute constraint(1) into constraint(2). In the meantime, the G column and the Nu column can be deleted, together with constraint (8) and (9). Now the *Monotonicity Table* is simplified and looks like this:

Objective: minimize $\mu R/D_H$

Monotonic variable	h	D_H	R
Relevance		*	*
Direction +			+
Direction -		-	
(2)′ $h = KCR^n/D_H$	?	*	
(4) $h - h_{High} \leq 0$	+		
(5) $-h + h_{Low} \leq 0$	-		
(6) $R - R_{High} \leq 0$			+
(7) $-R + R_{Low} \leq 0$			-
(10) $D_H - D_{HHigh} \leq 0$		+	
(11) $-D_H + D_{HLow} \leq 0$		-	

Table 4.2: Monotonicity Table 1

The only equality constraint (2)′ and the new objective have common variables R and D_H, this indicates that we can further eliminate one variable. In this case D_H should be eliminated together with constraint (10) and (11)[3]. At this point the monotonicity table looks much simpler:

[3]We can eliminate R and get another lower bound on G which we can prove to be greater than the one we get by eliminating D_H.

Objective: minimize $\mu h R^{1-n}/CK$

Monotonic variable	h	R
Relevance	*	*
Direction +	+	+
Direction -		
(4) $h - h_{High} \leq 0$	+	
(5) $-h + h_{Low} \leq 0$	-	
(6) $R - R_{High} \leq 0$		+
(7) $-R + R_{Low} \leq 0$		-

Table 4.3: Monotonicity Table 2

To find G_{Min}, from the *Monotonicity Table*, we can see that constraint (5) and (7) bound h and R from below respectively, therefore,

$$G_{Min} = \mu h_{Low} R_{Low}^{1-n}/CK$$

It is a similar task to find G_{Max}. Following the same step, we get

$$G_{Max} = \mu h_{High} R_{High}^{1-n}/CK$$

So the bounds on G are

$$\mu h_{Low} R_{Low}^{1-n}/CK \leq G \leq \mu h_{High} R_{High}^{1-n}/CK$$

which can be expressed as

$$G \in [\mu h_{Low} R_{Low}^{1-n}/CK, \quad \mu h_{High} R_{High}^{1-n}/CK]$$

Thus, we can put explicit bounds on the variable G, given the constraints it appears in and the limits on the other variables in the constraints.

In this case, the technique of *Monotonicity Analysis* can lead us directly to the solution. It can be applied to more complicated problems where there are many more variables and reduce the complexity of the problem to some extent.

Chapter 5

Testing The Knowledge Base

5.1 Testing For Correctness

The combination of scaled initial domains and constraint propagation gives us a powerful tool for representing engineering design knowledge. However, it is highly probable that a random combination of constraints and initial domains gives no solution, therefore great care is needed to ensure that the domains and constraints are consistent.

For each schema in the knowledge base, we write a 'testing schema' which can invoke the schema being tested, and can set the values of its variables via accessors[1].

The first we need to do in testing a schema is to test that the set of constraints *does* have a solution. To do this, we can work out a trial case by hand calculation and then

[1]An accessor is a kind of Echidna method, it provides access, for either retrieving or setting the instance variables.

feed these values to the testing schema. Next, we face the task of selecting consistent sets of values for the initial domains. The approach we used is the 'bottom up' method : one chooses a certain subset of schema variables as fundamental (functional or private variables as we mentioned in chapter 4); their values specify the *structure* of the device being designed and would go onto engineering drawings. We choose their domains, then from these we calculate the domains of other variables, which describe the *performance* of the device.

It should be pointed out that the above approach may not always yield *practical* results. For example, the feasible domain for the tube diameter of a small tubular heat exchanger is [0.001, 0.01] metre, and the tube length is in the range [0.01, 1] metre, but it is ridiculous to design an heat exchanger with 0.001 metre diameter tubes 1 metre long, as the pressure drop would be unacceptably high. So some modifications to the 'bottom up' methods are necessary. We can work out the possible domains of the variables and we may further restrict their domains to eliminate designs that would perform poorly.

Obviously failures may occur in the course of testing the knowledge base. There are different kinds of failures. Sometimes Echidna will have to choose one of several alternatives. If it rejects an alternative, it will show 'unify failure'. Let's look at an example. In material.kb, we have a method 'SetProps(Species)'.

```
setProps(Species):-
    setViscosity(Species,Viscosity),
    .....
setViscosity(hydrogen,0.00000896).
setViscosity(helium,0.000023).
```

```
setViscosity(nitrogen,0.00000178).
```

Suppose we have chosen 'nitrogen' to be the Species. When Echidna evaluates 'SetProps(Species)', we will see:

```
Tracing Failure:
unify clause failure:
setProps(nitrogen) :- setViscosity(Species, Viscosity)
        at ./material.kb:39 ...
   on clause 1 due to argument 1
echidna 23> c
Tracing Failure:
unify clause failure:
setProps(nitrogen) :- setViscosity(Species, Viscosity)
        at ./material.kb:39 ...
   on clause 2 due to argument 1
```

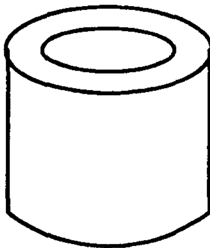
The first 'unify' failure is due to ' on clause 1 due to argument 1', since clause 1 is setViscosity(hydrogen,0.00000896) and argument 1 here is hydrogen, and it fails to unify with nitrogen; similarly, argument 1 in clause 2 is helium and it cannot unify with nitrogen either. The unification will succeed when clause 3 is called because it has nitrogen as argument 1.

So 'unify failure' is not necessarily anything to worry about, we can just tell Echidna to continue (by hitting 'c') and it will go on to choose other alternatives. However, if all alternatives have failed, Echidna will say 'meta failure[2]'. These are

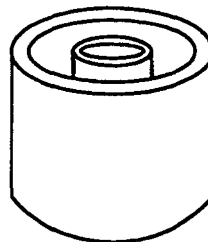
[2] Also called 'deep failure'.

serious failures since they indicate that probably one or some of the variables have wrong initial domains. One of my tasks as a knowledge engineer is to detect and fix these wrong initial domains.

Sometimes an apparently reasonable knowledge base can contain contradictions. Here is an interesting example. The capsule is an important component of a regenerator. There are two kinds of capsules, namely, cylindrical capsule and annular capsule. The cylindrical capsule looks just like a tin can (Figure 5.1(a)); the annular capsule looks like a tin can with another can inside it, concentric with the first (Figure 5.1(b)).



(a) Cylindrical Capsule



(b) Annular Capsule

Figure 5.1: Cylindrical and annular capsule

The cylindrical capsule is essentially a can that will be filled with some finely-divided material (the *matrix*) to absorb and re-emit heat. For the annular capsule, it is the space between the two cans that gets filled with the material.

In capsule.kb, the internal and external diameters of the cylindrical capsule are both scaled to have the same initial domains. This seems reasonable; the wall thickness separating them will only be a few millimeters. If I set the internal diameter to the upper limit of its domain, the external diameter is constrained to have the same value, since it must be greater than or equal to the internal diameter, but cannot exceed the upper limit of the domain. As a result, the wall thickness drops to zero, and this produces a propagate failure. The solution to this problem is to first set the domains of the internal diameter, then to set bounds on the thickness of the wall, and lastly to calculate the external diameter using

$$ExternalDiameter =:= InternalDiameter + 2 * WallThickness.$$

Thus we can avoid the above problem.

When testing a *big* annular capsule, I encountered another problem. Since the void volume is the difference between the volume enclosed by the outer can and the volume occupied by the inner can, it can be very small even when the dimensions of the cans are very large. Specifically, its size depends on the difference between the OuterInternalDiameter (OID) and the InnerExternalDiameter (IED). The void volume calculated this way may drop below the minimum acceptable limit. It is unreasonable to have a tiny volume in a capsule we've declared to be big.

We thought about several ways to deal with this problem. First, we could set the limits for the two diameters so that the difference between them would always be big enough. This is not a good solution, because then the upper bound of the IED would have to be smaller than the lower bound of the OID, which is too strong a constraint.

A second alternative would be to leave the bounds as they are, but add an explicit constraint that $OID > IED + \epsilon$, where ϵ is some suitable number big enough to

prevent the failure. If we did this, failure may occur in some cases where the above constraint is not satisfied. But this is acceptable because the failures indicate that we are asking the knowledge to do something unreasonable, for example, we ask for a *big* heat exchanger with a tiny volume. The knowledge base should fail when the requirements are unrealistic.

A third alternative would be to do nothing. Then failure would occur and Echidna would backtrack and either change the value of OID or IED; or, if neither of these worked, to the decision that the capsule should be big.

The second alternative is the best. It makes it clearer to the users what should fail and what shouldn't.

5.2 Improvement of Efficiency

Echidna, like Prolog, is a *declarative* language. As opposed to *procedural* languages like FORTRAN and C, it is not always easy to tell what order the statements will execute in. Echidna has an added level of complexity because of the persistent constraints. Part of the theory of declarative language says that we shouldn't worry about the order of execution; as long as the statements in the knowledge base define a problem which has a solution, the knowledge base will eventually find it. But if we are concerned with efficiency, we do have to consider the order of execution. As a rule of thumb, we should put the statement which is most likely to be chosen in the first place, followed by the statement which is the second most likely to be chosen, and so on and so forth. For example, in the testing program 'testHX.kb', we have

```
order chooseFF.
chooseFF(HeatFlow, functionType Function):-
   chooseFFAccordingToHeatFlow(HeatFlow, Function).
chooseFF(HeatFlow, Function):-
   chooseFFOtherwise(HeatFlow, Function).
```

Here we put 'chooseFFAccordingToHeatFlow(HeatFlow, Function).' before ' chooseFFOtherwise(HeatFlow, Function).' since the former is more likely to be chosen.

There are other principles we may follow to determine the order of the statements. For instance, when we use 'tubeChoose' method to choose the number of tubes, we follow the simple principle of 'the fewer, the better', therefore we can order the clauses as:

```
order tubeChoose.
tubeChoose(50).
tubeChoose(60).
tubeChoose(70).
......
```

In the course of testing the knowledge base, sometimes I found that Echidna took a long time to come up with a solution. Of course we have a scaled knowledge base. If we are at the threshhold, Echidna may need to have several iterations of backtracking until it finds the appropriate scale, but if we know that we are not at the threshhold, we expect Echidna to make the decision quickly. In these cases, if Echidna still spends a long time, we should investigate the problem and check whether we can put the statements in a better order to improve efficiency. Let's look at an example from

the Stirling engine heat exchanger knowledge base. The design objective is a heater and we want to send a message to it. First we tried:

```
test1:-
      Heater:diameter(Diameter).
chooseFF(HeatFlow, heating).
```

If we put the statements like this, the initialization method in test1 sends a message to the heater, setting the value of its diameter, then it calls a second method, 'chooseFF' to create the heater. The message about the diameter doesn't reach the heater until all of the method 'chooseFF' has executed. This takes quite a long time. Alternatively, we tried:

```
test2:-
      Heater:diameter(Diameter),
      chooseFF(HeatFlow, heating).
```

If we rearrange the statements, the initialization method in test2 sends a message to the heater, then creates the heater itself. This time the message about the diameter reaches the heater as soon as it is created.

We would expect that the message to the heater would be delivered as soon as the heater came into existence, we should avoid the delay in the first example and follow the way in the second example. We reported this to the Echidna system supporter; maybe in future versions, they can improve on this and make the system more efficient.

Chapter 6

A Detailed Design Example

The present knowledge base can successfully create three kinds of heat exchangers, namely heater, cooler and regenerator. We will go through a detailed design example to trace Echidna's calculations and explain how choices were made.

6.1 Design Requirements

We have designed part of an engine and we expect Echidna to design the heat exchangers. The requirements are:

1. In the heater, the mean temperature difference between cylinder and working gas: $DeltaT < 150K$;

2. In the cooler, the mean temperature difference between cylinder and working gas: $DeltaT < 100K$;

3. Must supply thermal power that can keep the temperature difference between the entering and leaving fluid at 100K.

4. The pressure drop should not be too big.

The first three requirements are written in 'textHX.kb', the last requirement is not formally stated in the knowledge base, but the pressure drop is calculated and the knowledge engineer will reject the design with pressure drop that is too big. The design parameters are shown in Table 6.1. These values are assigned in 'testHX.kb' file as well.

Heat exchanger diameter	0.5 *m*
Piston Stroke	0.5 *m*
Piston Clearance	0.001 *m*
Cylinder wall thickness	0.01 *m*
Atmospheric pressure	100000 *Pascal*
Mean cycle pressure (Pm)	100000 *Pascal*
Cycle per second (CPS)	1
Source temperature	900 *K*
Sink temperature	300 *K*
Working gas	nitrogen

Table 6.1: Design Parameters

The above parameters are supplied to Echidna when we load 'testHX.kb' file to the Echidna interface. We expect Echidna to calculate the amount of heat flow, the mass flow rate, the pressure drop, etc. Also we expect Echidna to determine the type of heat exchanger and the detailed dimensions of its subcomponents.

6.2 Echidna's Decision

In this section, we will go through some of the decision-making steps to show how Echidna gets the final design.

The first method Echidna called is estimate(Ve, Size). We can determine the size of the heat exchanger by the value of its working volume Ve, which is calculated as:

$$Ve =:= 3.1415 * 0.25 * Diameter * *2 * Stroke$$

From Table 6.1, we can find the values of Diameter and Stroke, therefore we get $Ve = 0.09817187m^3$.

There is a relationship between the physical size of a heat exchanger and the amount of thermal power it can handle. This relationship is quite complicated since there are many factors involved, such as the choice of geometry, the working gas, the pressure and so on. Still, we know that a heat exchanger the size of a coffee cup can't exchange a mega watt of heat, and we know it would be wasteful to use a heat exchanger the size of a house to exchange a few watts. In the course of developing our knowledge base, we simply follow the engineer's rule of thumb:

```
estimateSize(Ve, verySmall):-
        Ve   > 1.0e-09,
        Ve   < 0.000001.
estimateSize(Ve, small):-
        Ve   > 5.0e-07,
        Ve   < 0.0001.
```

```
estimateSize(Ve, medium):-
      Ve   > 5.0e-05,
      Ve   < 0.001.
estimateSize(Ve, big):-
      Ve   > 0.0005,
      Ve   < 0.1.
estimateSize(Ve, veryBig):-
      Ve   > 0.05,
      Ve   < 10.
```

When we run the knowledge base, we get

```
Tracing Failure:
propagate failure:
estimateSize(0.09817187, verySmall) :- ... Ve < 1e-06
           at ./testHX.kb:279 ...
echidna 26> c
Tracing Failure:
propagate failure:
estimateSize(0.09817187, small) :- ... Ve < 0.0001
           at ./testHX.kb:283 ...
echidna 27> c
Tracing Failure:
propagate failure:
estimateSize(0.09817187, medium) :- ... Ve < 0.001
           at ./testHX.kb:287 ...
```

We can see that Echidna first tried to unify Ve with the first argument in

```
estimateSize(Ve, verySmall):-
Ve   > 1.0e-09,
Ve   < 0.000001.
```

and it failed. It should fail since Ve does not fit into this interval. So Echidna went back to check the next method and it failed for the same reason. It finally succeeded when it called:

```
estimateSize(Ve, big):-
Ve   > 0.0005,
Ve   < 0.1.
```

In this case 'Size' is big. We can then restrict our search to *big* heat exchanger domains.

The second method Echidna called is estimatePower(HeatFlow, Power). We have a constraint: $HeatFlow =:= HotMassFlowRate * SpecificHeat * 100$ From the inputs and previous calculations we can get the value of HotMassFlowRate and SpecificHeat:

Gas:specificHeat ([500, 5000] SpecificHeat),

Gas:density ([0.05, 5.0] GasDensity),

$HotDensity =:= GasDensity * Pm * ReferenceTemperature$

$/(AtmosphericPressure * SourceTemperature)$,

$HotVolumetricFlow =:= Ve * CPS$,

$$HotMassFlowRate =:= HotVolumetricFlow * HotDensity,$$

Therefore, we get Heatflow = [3394.774, 3394.797] and this unifies with

```
estimatePower(HeatFlow, highPower):-
  HeatFlow > 100.0,
  HeatFlow < 100000.0.
```

Hence, we get power range is highPower and we can restrict the design search space accordingly.

Next, we issue our first design goal: designHeater. We have

```
designHeater:-
[0.2, 200]   DeltaT < 150,
chooseFF(HeatFlow, heating).
```

Echidna tries to unify 'chooseFF' method with one from the following:

```
chooseFF(HeatFlow, functionType Function):-
     chooseFFAccordingToHeatFlow(HeatFlow, Function).
chooseFF(HeatFlow, Function):-
     chooseFFOtherwise(HeatFlow, Function).
```

It will try chooseFFAccordingToHeatFlow first (Function is instantiated as heating). If it fails, it will backtrack to this point later and try chooseFFOtherwise[1]. Now we

[1] Usually the choice of heat exchanger is based on the heat flow, we put chooseFFOtherwise here in case chooseFFAccordingToHeatFlow doesn't have any solution. This is for the completeness of the knowledge base.

have:

```
chooseFFAccordingToHeatFlow(HeatFlow, Function):-
     HeatFlow > 1000,
     createHX(tubularFF, Function).
chooseFFAccordingToHeatFlow(HeatFlow,Function):-
     HeatFlow < 1000,
     HeatFlow > 10,
     createHX(finnedFF, Function).
chooseFFAccordingToHeatFlow(HeatFlow, Function):-
    HeatFlow < 10,
    createHX(plainFF, Function).
```

We have Heatflow = [3394.774, 3394.797], obviously createHX(tubularFF, Heating) is called. Since the heat flow is large, Echidna chooses a tubular heat exchanger, which has a large surface area. We further restrict the search space to tubular heat exchanger.

```
createHX(tubularFF, heating):-
  tubeChoose([50,150] NumberOfHotTubes),
  interval HotTubeFlow =:= HotVolumetricFlow/NumberOfHotTubes,
  TubeDesigner isa tubeDesigner(Pm,Pressure,Size,SourceTemperature,
                                HotTubeFlow,tube HotTube),
  MaterialChooser:choosePVMaterial(Diameter, WallThickness, Pm,
                             SourceTemperature,solids HotMetal),
  Heater isa tubular(heating,Gas,HeatFlow,HotMetal,HotTube,
```

```
                        HotMassFlowRate,NumberOfHotTubes,Pm,Power,
                    Pressure,Size,SourceTemperature,HotVolumetricFlow),
    HotTube:choose.
```

The first choice made here is the number of tubes. Echidna just searched the domain of 'NumberOfHotTubes', and selected the first number 50. Next it calculated HotTubeFlow from $HotTubeFlow =:= HotVolumetricFlow/NumberOfHotTubes.$ Then it called the TubeDesigner to create the tube and the MaterialChooser to choose the tube material[2]. Next Echidna tried to create the tubular heater. The schema tubular is in 'heat.kb'. In heat.kb we have

```
schema tubular:fluidFluid
{
tubular(Function,Gas,HeatFlow,Metal,Tube,MassFlowRate,NumberOfTubes,
        Pm, Power,Pressure,Size,T,VolumetricFlow):-
    setSize,
    fluidFluid,
    Geometry = tubularFF,
    Tube:internalDiameter(ReynoldsDiameter),
    Tube:length(Length),
    setCorrelations(R,A,C,M,N).
 }
```

We can see that tubular is a subclass of 'fluidFluid', which in turn is a subclass of

[2]In our knowledge base, MaterialChooser is a separate method, it is not part of the TubeDesigner. The purpose is to increase the reusability of the method.

'heatExchange'. So when Echidna tried to create a tubular heater, it would have called all the relevant methods in class 'fluidFluid' and 'heatExchange'.

The reader might notice the message HotTube:choose. We use it to call the *choose* method in tube.kb. Before the 'choose' method is called, some variables, such as InternalDiameter and Wallthickness of the tube, still have large domains. For this reason, we don't know whether we really have a consistent design. After the method is called, Echidna uses 'indomain' to pin down these variables and check them.

Echidna takes about 30 seconds to come up with the design of a heater. Following similar steps, Echidna can create a cooler and a regenerator. We can print out the results and check whether they are reasonable. If the designer is not satisfied with them, he or she can go back and change some design parameters. In our example, the results looks quite feasible. The Echidna output file is included in Appendix B. Here we summarize the results in tables.

Heater Type	tubular
Surface Area available for heat transfer	$0.0021 m^2$
Cross Area	$6.79e-05 m^2$
Mean Temperature Difference between gas and wall	127.5 K
Friction Factor	0.0035
Heat Transfer Coefficient	12934 W/m^2K
Heat Flux	1.65e+05 W/m^2K
Mass Flux	500 Kg/m^2S
Heater Length	0.01 m
Mass	0.25 Kg
Mass Flow Rate	0.034 Kg/s
Pressure Drop	27 *Pascal*
Nusselt Number	651
Reynolds Number	367497
Reynolds Diameter	0.0013 m
Volume	3.87e-07 m^3
Number of Tubes	50
Coefficient in f-Re correlation	0.045
Coefficient in Nu-Re Correlation	0.022
Exponent in f-Re correlation	-0.2
Exponent in Nu-Re Correlation	0.8
Power Range	high
Pressure Range	low
Size Range	big
Temperature Range	hot
Metal Species	Aluminum
Metal Density	$2700 kg/m^3$
Metal Conductivity	110 W/mK
Metal Specificheat	330 J/kgK

Table 6.2: Heater

External Diameter	0.015 m
External Area	0.00048 m^2
External Cross Area	0.00018 m^2
Internal Diameter	0.0013 m
Internal Area	4.11e-05 m^2
Internal Cross Area	1.34e-06 m^2
Tube Length	0.01 m
Tube Wall Thickness	0.007 m
Void Volume	7.75e-09 m^3
External Volume	1.86e-06 m^3
Mass	0.005 Kg
Metal Density	2700 Kg/m^3
Pressure Range	low
Size Range	big
Metal Species	Aluminum
Metal Density	2700kg/m^3
Metal Conductivity	110 W/mK
Metal Specificheat	330 J/kgK

Table 6.3: The parameters of a single tube in the heater

Cooler Type	tubular
Surface Area available for heat transfer	$0.0045m^2$
Cross Area	$0.00033m^2$
Mean Temperature Difference between gas and wall	$99.7K$
Friction Factor	0.0033
Heat Transfer Coefficient	7442 W/m^2K
Heat Flux	742561 W/m^2K
Mass Flux	305 Kg/m^2S
Cooler Length	0.01 m
Mass	0.25 Kg
Mass Flow Rate	0.102 Kg/S
Pressure Drop	6.55 $Pascal$
Nusselt Number	833
Reynolds Number	500000
Reynolds Diameter	0.0029 m
Volume	1.56e-06 m^3
Number of Tubes	50
Coefficient in f-Re correlation	0.045
Coefficient in Nu-Re Correlation	0.023
Exponent in f-Re correlation	-0.2
Exponent in Nu-Re Correlation	0.8
Power Range	high
Pressure Range	low
Size Range	big
Temperature Range	room temperature
Metal Species	Aluminum
Metal Density	$2700kg/m^3$
Metal Conductivity	110 W/mK
Metal Specificheat	330 J/kgK

Table 6.4: Cooler

External Diameter	0.0156 m
External Area	0.00048 m^2
External Cross Area	0.00019 m^2
Internal Diameter	0.0029 m
Internal Area	9.14e-05 m^2
Internal Cross Area	6.65e-06 m^2
Tube Length	0.01 m
Tube Wall Thickness	0.0063 m
Void Volume	5.45e-08 m^3
External Volume	1.91e-06 m^3
Mass	0.005 Kg
Metal Density	2700 Kg/m^3
Pressure Range	low
Size Range	big
Metal Species	Aluminum
Metal Density	2700kg/m^3
Metal Conductivity	110 W/mK
Metal Specificheat	330 J/kgK

Table 6.5: The parameters of a single tube in the cooler

Regenerator Type	regenerative
Surface Area available for heat transfer	$7.067m^2$
Cross Area	$0.177m^2$
Mean Temperature Difference between gas and wall	5.6
Friction Factor	0.065
Heat Transfer Coefficient	85.3 W/m^2K
Heat Flux	479 W/m^2K
Mass Flux	0.19 Kg/m^2S
Regenerator Length	0.1 m
Mass	9.61 Kg
Mass Flow Rate	0.034 Kg/S
Pressure Drop	0.23 $Pascal$
Nusselt Number	32.8
Reynolds Number	1079
Reynolds Diameter	0.01 m
Volume	0.0176 m^3
Number of Regenerators	1
Coefficient in f-Re correlation	20
Coefficient in Nu-Re Correlation	1
Exponent in f-Re correlation	-0.8
Exponent in Nu-Re Correlation	0.50
Power Range	high
Pressure Range	low
Size Range	big
Temperature Range	room temperature
Metal Species	Aluminum
Metal Density	$2700kg/m^3$
Metal Conductivity	110 W/mK
Metal Specificheat	330 J/kgK

Table 6.6: Regenerator

Capsule Geometry	cylindrical
External Diameter	0.51 m
External Area	0.163 m^2
External Cross Area	0.212 m^2
Internal Diameter	0.5 m
Internal Area	0.157 m^2
Internal Cross Area	0.196 m^2
Length	0.1 m
Wall Thickness	0.01m
Void Volume	0.019 m^3
External Volume	0.0212 m^3
Mass	4.32 Kg
Metal Species	Aluminum
Metal Density	2700kg/m^3
Metal Conductivity	110 W/mK
Metal Specificheat	330 J/kgK

Table 6.7: Capsule

Matrix Type	meshMatrix
Porosity	0.9
Density	270 Kg/m^3
Specific Area	360 m^2/kgK
Reynolds Diameter	0.01 m
Reynolds Number	1079
Fibre Diameter	2e-05 m
Wire Diameter	0.0011 m
Sphere Radius	2e-05 m
Metal Species	Aluminum
Metal Density	2700kg/m^3
Metal Conductivity	110 W/mK
Metal Specificheat	330 J/kgK

Table 6.8: Matrix

Chapter 7

Summary And Future Work

7.1 Progress So Far

The development of the knowledge base is laborious and takes up a great deal of time. The progress is relatively slow, partly because I am not very familiar with the technology of Stirling engine heat exchangers; partly because Echidna, the development tool, is still in an experimental stage. Based on the work done by my supervisor, Dr. John Jones, and other previous students, I have detected and eliminated the inconsistencies in the original prototype knowledge base to support Stirling engine heat exchanger design. My research work mainly focused on developing the *factual* knowledge base. In doing so, I had to test the knowledge base for many, if not all, combinations of possible inputs and if there are inconsistencies in the knowledge, as there are in most cases, I had to determine what causes these inconsistencies and correct them. The knowledge base has been extended and tested and we have used

it to design several different heat exchangers successfully. The knowledge base now comprises about six thousand lines of Echidna code and can be used as **alpha-test** software by interested third parties.

7.2 Future Work

Our knowledge-based system is still a long way from being exploited by commerce and industry. Rather, our hope is more to elucidate the theory of design than to produce a tool that can be used in industry, at least in the short term.

Several issues have to be resolved in the future. Our development tool, Echidna, is still being developed. The Expert Systems Laboratory is improving the performance of Echidna. We have been working closely with the system developers to add desired features to Echidna.

In order to make our knowledge base complete, more details are needed. For example, the present library of solid materials only has aluminum, iron and steel. We could expand it to include many other materials. Also we need to incorporate the manufacturing methods in our knowledge base.

One of the contributions of this research is the idea of scaling. There are two directions in which we intend to develop this work. Firstly, the classification of devices on the basis of scale can be supplemented by classification schemes based on suitable dimensionless numbers. For example, a knowledge base describing turbomachinery might classify devices into their homologous series. Secondly, the use of fixed boundaries between different size classes might be replaced by the use of fuzzy numbers,

reflecting the imprecision naturally present in such expressions as a *large* engine.

The reader can see from the design results shown in chapter 6 that they are a large amount of data. The ideal design should look like the engineering model with two or three dimensional images to show the real object, and other design specification can be shown on the screen as context. This is not a trivial task as we don't know how to write rules to represent geometric design description.

Another difficult task is geometric reasoning since we need to use explicit methods to represent the shape, size, and location (relative or absolute, same or different coordinate systems) of the objects, and we need to incorporate knowledge of spatial reasoning to check whether the components can fit into the whole system, whether they interfere. This is an interesting and under-studied area.

We hope to develop an intelligent design system which can not only produce a design to meet given requirements, based on the expert's knowledge, but also produce some innovative designs which is beyond the expert's present knowledge. Again this is a difficult task.

A more interactive user interface is desired. Right now we type all the input into a data file and load it to the Echidna interface. Ideally, we could have a menu to select these data. The interface should be able to show the inheritance hierarchy, the proof tree and the constraint network.

As there will always be relevant knowledge that is not included in the knowledge base, there is no prospect of eliminating the human designer at any time. Some requirements cannot be stated formally in a knowledge base – for example, we want an

artifact to have a *pleasing appearance.* We should always remember that engineering design is a *Mixed-Initiative* activity. Our aim is to develop intelligent computer systems that can cooperate with the human designers, alternatively taking and relinquishing the initiative. There is a lot of work to do to perform the *Mixed-Initiative* design well. For example, if the human designer wants to elaborate a design which the system has just been working on, he or she must understand what the system has done, and why. He or she must also know which features of the design can be changed readily, and which are constrained by the design requirements. How do we enable the system to convey such information?

Appendix A

Sample Files

As we mentioned in chapter 3, we have two kinds of knowledge bases: *facts* and *strategies*. Here we show one sample from each kind. reg.kb and agentMat.kb. We also include two testing programs, namely testHX.db and testHX.kb.

```
***********************************************************
**            Name:      testHX.db                        **
***********************************************************

set pretty on
set failbreak on
catch unify clause off
load declarations.kb
load global.kb
load material.kb
load geometry.kb
load tube.kb
load fin.kb
load matrix.kb
load capsule.kb
load agentMat.kb
load heat.kb
load reg.kb
load testHX.kb
precision(24).
heatTest H isa heatTest.
H:design.
```

```
***********************************************************
**            Name:       testHX.kb                      **
***********************************************************

schema heatTest
{
   real               AtmosphericPressure.
   capsule            Capsule.
   interval           ColdMassFlowRate.
   interval           ColdVolumetricFlow.
   fluidFluid         Cooler.
   interval           CPS.
   interval           Diameter.
   gases              Gas.
   interval           HeatFlow.
   fluidFluid         Heater.
   interval           HotMassFlowRate.
   interval           HotVolumetricFlow.
   interval           Length.
   materialChooser    MaterialChooser.
   [0.00001, 0.001]     PistonClearance.
   interval           Pm.
   powerRange         Power.
   pressureRange      Pressure.
   real               ReferenceTemperature.
```

```
regenerator         Regenerator.
interval            SinkTemperature.
interval            SourceTemperature.
sizeRange           Size.
interval            Stroke.
temperatureRange    Temperature.
interval            Ve.
interval            WallThickness.


heatTest:-
   AtmosphericPressure is 100000,
   ReferenceTemperature is 273,
   MaterialChooser isa materialChooser.


design:-
   SourceTemperature =:= 900,
   SinkTemperature =:= 300,
   CPS =:= 1,
   PistonClearance =:= 0.001,
   Diameter =:= 0.5,
   Stroke   =:= 0.5,
   Pm  =:= 100000,
   WallThickness =:= 0.01,
   Gas isa gases,
   Gas:species(nitrogen),
```

```
     Gas:specificHeat([500,50000] SpecificHeat),
     Gas:density([0.05, 5.0] GasDensity),
    interval  HotDensity =:= GasDensity * Pm * ReferenceTemperature
                             /(AtmosphericPressure * SourceTemperature),
    interval  ColdDensity =:= GasDensity * Pm * ReferenceTemperature
                             /(AtmosphericPressure * SinkTemperature),
     Ve  =:= 3.1415*0.25* Diameter**2 * Stroke,
     HotVolumetricFlow  =:= Ve * CPS,
     ColdVolumetricFlow =:= Ve * CPS,
     HotMassFlowRate    =:= HotVolumetricFlow * HotDensity,
     ColdMassFlowRate   =:= ColdVolumetricFlow * ColdDensity,
     HeatFlow        =:= HotMassFlowRate * SpecificHeat * 100,
     estimateSize(Ve, Size),
     estimatePower(HeatFlow, Power),
     estimatePressure(Pm, Pressure),
     estimateTemperature(SourceTemperature, Temperature),
     designHeater,
     designCooler,
     designRegenerator.

 designHeater:-
    [0.2, 200]   DeltaT < 100,
          chooseFF(HeatFlow, heating).


 designCooler:-
```

```
  [0.2, 200]   DeltaT < 50,
  chooseFF(HeatFlow, cooling).

 designRegenerator:-
    capsuleDesign CapsuleDesign isa capsuleDesign(Capsule, Pm, Pressure,
    Size, SourceTemperature,Temperature, WallThickness, Diameter, Length),
    matrixDesign  MatrixDesign  isa matrixDesign(Matrix,SourceTemperature),
    Regenerator isa regenerator(Gas,Capsule,HeatFlow,HotMassFlowRate,
                          Matrix,NumRegenerators,Pm,Power,Pressure,
                          Size,SinkTemperature,SourceTemperature,Ve),
    indomain(NumRegenerators),
    CapsuleDesign:chooseValues(Capsule),
    MatrixDesign:chooseValues(Matrix).

order chooseFF.

  chooseFF(HeatFlow, functionType Function):-
      chooseFFAccordingToHeatFlow(HeatFlow, Function).
  chooseFF(HeatFlow, Function):-
     chooseFFOtherwise(HeatFlow, Function).

  chooseFFAccordingToHeatFlow(HeatFlow, Function):-
     HeatFlow > 1000,    %HeatFlow>1000
     createHX(tubularFF, Function).
  chooseFFAccordingToHeatFlow(HeatFlow,Function):-
```

```
    HeatFlow < 1000,
    HeatFlow > 10,  %10<=HeatFlow<=1000
    createHX(finnedFF, Function).
chooseFFAccordingToHeatFlow(HeatFlow, Function):-
    HeatFlow < 10,   %HeatFlow<100
    createHX(plainFF, Function).

chooseFFOtherwise(HeatFlow, Function):-
    HeatFlow < 1000,   %not HeatFlow>500
    createHX(tubularFF, Function).
chooseFFOtherwise(HeatFlow, Function):-
  HeatFlow < 10,  %not (100<=HeatFlow<=500)
    createHX(finnedFF, Function).
chooseFFOtherwise(HeatFlow, Function):-
    HeatFlow > 10,   %not HeatFlow<100
    createHX(plainFF, Function).

createHX(finnedFF, heating):-
    finChoose({50..150} NumberOfFins),
    MaterialChooser:choosePVMaterial(Diameter, WallThickness, Pm,
                    SourceTemperature,solids HotMetal),
    FinDesigner isa finDesigner(Diameter, Stroke,
                   SourceTemperature,fin HotFin),
    Heater:fin(HotFin),
    HotFin:choose,
```

```
    Heater isa finned(heating,CPS,Diameter,Gas,HeatFlow,HotMetal,
         NumberOfFins,Pm, Power,Pressure,Size,Stroke,Temperature,
         SourceTemperature,HotVolumetricFlow),
    Heater:choose.

createHX(finnedFF, cooling):-
    finChoose({50..150} NumberOfFins),
    MaterialChooser:choosePVMaterial(Diameter, WallThickness,
                        Pm,SinkTemperature,solids ColdMetal),
    FinDesigner isa finDesigner(Diameter, Stroke, SinkTemperature,
                   fin ColdFin),
    Cooler:fin(ColdFin),
    ColdFin:choose,
    Cooler isa finned(cooling,CPS,Diameter,Gas,HeatFlow,ColdMetal,
                  NumberOfFins,Pm,Power,Pressure,Size,Stroke,
                  Temperature,SinkTemperature,ColdVolumetricFlow),
    Cooler:choose.

createHX(tubularFF, heating):-
    tubeChoose([50,150] NumberOfHotTubes),
    interval HotTubeFlow =:= HotVolumetricFlow/NumberOfHotTubes,
    TubeDesigner isa tubeDesigner(Pm,Pressure,Size,SourceTemperature,
                                            HotTubeFlow,tube HotTube),
    MaterialChooser:choosePVMaterial(Diameter, WallThickness, Pm,
                              SourceTemperature,solids HotMetal),
```

```
    Heater isa tubular(heating,Gas,HeatFlow,HotMetal,HotTube,
              HotMassFlowRate,NumberOfHotTubes,Pm,Power,Pressure,
              Size,SourceTemperature,HotVolumetricFlow),
    HotTube:choose.

createHX(tubularFF, cooling):-
    tubeChoose([50,150] NumberOfColdTubes),
    interval ColdTubeFlow =:= ColdVolumetricFlow/NumberOfColdTubes,
    TubeDesigner isa tubeDesigner(Pm,Pressure,Size,SinkTemperature,
                                          ColdTubeFlow,tube ColdTube),
    MaterialChooser:choosePVMaterial(Diameter, WallThickness, Pm,
                                     SinkTemperature,solids ColdMetal),
    Cooler isa tubular(cooling,Gas,HeatFlow,ColdMetal,ColdTube,
                      ColdMassFlowRate,NumberOfColdTubes,Pm,Power,
                 Pressure,Size,SinkTemperature,ColdVolumetricFlow),
    ColdTube:choose.

createHX(plainFF, heating):-
    MaterialChooser:choosePVMaterial(Diameter, WallThickness, Pm,
                                   SourceTemperature,solids HotMetal),
    Heater isa plain(heating,CPS,Diameter,Gas,HeatFlow,HotMassFlowRate,
                    HotMetal,PistonClearance, Pm,Power,Pressure,Size,
                         Strok,SourceTemperature,HotVolumetricFlow),
    Heater:choose.
```

```
createHX(plainFF, cooling):-
   MaterialChooser:choosePVMaterial(Diameter, WallThickness, Pm,
                                   SinkTemperature,solids ColdMetal),
   Cooler isa plain(cooling,CPS,Diameter,Gas,HeatFlow,ColdMassFlowRate,
                              ColdMetal,PistonClearance,Pm,Power,Pressure,
                         Size,Stroke,SinkTemperature,ColdVolumetricFlow),
   Cooler:choose.

order tubeChoose.      % Follow the simple principle of
tubeChoose(50).        % 'the fewer the better'
tubeChoose(60).
tubeChoose(70).
tubeChoose(80).
tubeChoose(90).
tubeChoose(100).
tubeChoose(110).
tubeChoose(120).
tubeChoose(130).
tubeChoose(140).
tubeChoose(150).

order finChoose.
finChoose(50).
finChoose(60).
finChoose(70).
```

```
 finChoose(80).
 finChoose(90).
 finChoose(100).
 finChoose(110).
 finChoose(120).
 finChoose(130).
 finChoose(140).
 finChoose(150).

estimatePower(HeatFlow, veryLow):-
    HeatFlow > 0,
    HeatFlow < 1.0.
estimatePower(HeatFlow, lowPower):-
    HeatFlow > 0.1,
    HeatFlow < 100.0.
estimatePower(HeatFlow, medPower):-
    HeatFlow > 10.0000,
    HeatFlow < 1000.0.
estimatePower(HeatFlow, highPower):-
    HeatFlow > 100.0,
    HeatFlow < 100000.0.
estimatePower(HeatFlow, veryHighPower):-
    HeatFlow > 100000.0.

estimatePressure(Pm, low):-
```

```
    Pm < 1000000.
estimatePressure(Pm, mediumPr):-
    Pm > 1000000,
    Pm < 10000000.
estimatePressure(Pm, high):-
    Pm > 10000000.

estimateSize(Ve, verySmall):-
    Ve   > 1.0e-09,
    Ve   < 0.000001.
estimateSize(Ve, small):-
    Ve   > 5.0e-07,
    Ve   < 0.0001.
estimateSize(Ve, medium):-
    Ve   > 5.0e-05,
    Ve   < 0.001.
estimateSize(Ve, big):-
    Ve   > 0.0005,
    Ve   < 0.1.
 estimateSize(Ve, veryBig):-
    Ve   > 0.05,
    Ve   < 10.

 estimateTemperature(SourceTemperature, cryogenic):-
    SourceTemperature < 250.
```

```
   estimateTemperature(SourceTemperature, roomTemperature):-
      SourceTemperature > 200,
      SourceTemperature < 400.
   estimateTemperature(SourceTemperature, hot):-
      SourceTemperature > 300,
      SourceTemperature < 1000.
   estimateTemperature(SourceTemperature, veryHot):-
      SourceTemperature > 1000.
}

schema tubeDesigner
{
  materialChooser      MaterialChooser.

  tubeDesigner(interval Pm,pressureRange Pressure,sizeRange
               Size,[20,2000] T, _ ,tube Tube):-
      MaterialChooser isa materialChooser,
      MaterialChooser:choosePVMaterial(interval InternalDiameter,
                       WallThickness, Pm,T, solids TubeMetal),
      Tube isa tube(_, InternalDiameter, _ ,TubeMetal,Pm,
                     Pressure,Size,WallThickness).
}

schema finDesigner
{
```

```
  materialChooser       MaterialChooser.
  sizeRange             Size.

 finDesigner(interval Diameter, interval FinLength,
             interval T, fin Fin):-
     MaterialChooser isa materialChooser,
     MaterialChooser:chooseMaterial(T, solids FinMetal),
     chooseSize(Diameter, Size),
     Fin isa fin(Diameter, _,FinLength,_, FinMetal, Size).

   chooseSize(interval Diameter,verySmall):-
      Diameter < 0.01.
   chooseSize(interval Diameter,small):-
      Diameter < 0.05.
   chooseSize(interval Diameter,medium):-
      Diameter < 0.5.
   chooseSize(interval Diameter,big):-
      Diameter < 1.0.
   chooseSize(interval Diameter,veryBig):-
      Diameter < 5.0.
   chooseSize(_,_).
}

schema capsuleDesign
{
```

```
   materialChooser       MaterialChooser.
   sizeRange             Size.
   pressureRange         Pressure.
   temperatureRange      Temperature.

   capsuleDesign(capsule Capsule,interval Pm, Pressure, Size, interval
      SourceTemperature, Temperature, WallThickness, Diameter, Length):-
      MaterialChooser isa materialChooser,
      MaterialChooser:choosePVMaterial(Diameter,WallThickness, Pm,
                       SourceTemperature, solids Metal),
      Capsule isa capsule(Diameter, _, Length, Metal, Pm, Pressure,
                 SourceTemperature, Temperature,Size, WallThickness).

   chooseValues(capsule Capsule):-
       Capsule:chooseLength.
}

schema matrixDesign
{
   materialChooser       MaterialChooser.

   matrixDesign(Matrix,SourceTemperature):-
      MaterialChooser isa materialChooser,
      MaterialChooser:chooseMaterial(SourceTemperature, solids Metal),
      Matrix isa matrix(Metal , _).
```

```
   chooseValues(matrix Matrix):-
      indomain(matrixType    Type),
      Matrix:type(Type),
      Matrix:setPorosity(Type, _),
      Matrix:chooseValues.
}
```

```
#define pi 3.1415926

**************************************************************
**          Name:    agentMat.kb                            **
**************************************************************

schema gasChooser
{
   chooseGas(speedRange Speed, gasSpecies Species):-
      chooseGasAccordingToSpeed(Speed, Species).

   chooseGas(Speed, Species):-
      chooseGasOtherwise(Speed, Species).

   chooseGasAccordingToSpeed(fast, hydrogen).
   chooseGasAccordingToSpeed(mediumSp, helium).
   chooseGasAccordingToSpeed(slow, nitrogen).

   chooseGasOtherwise(fast,helium).
   chooseGasOtherwise(mediumSp,hydrogen).
   chooseGasOtherwise(slow,helium).
}
```

```
schema materialChooser
{
    choosePVMaterial(interval Diameter, interval WallThickness,
              interval Pm, interval Temperature, solids Metal):-
       Metal isa solids,
       indomain(solidSpecies Species),
       Metal:species(Species),
       Metal:setProps(Species),
       Metal:stressLimit([100000000,1000000000]*3 StressLimit),
       [100000000,1000000000]*3 HoopStress =:= StressLimit * 0.8,
       WallThickness > Pm * Diameter * 0.5 / HoopStress.

    chooseMaterial(interval  Temperature,solids Metal):-
       indomain(solidSpecies Species),
       Metal isa solids,
       Metal:species(Species),
       Metal:setProps(Species).
}
```

```
#define pi 3.1416926
**********************************************************
**          Name:          reg.kb                        **
**********************************************************

schema regenerator:heatExchange
{
   interval   EnthalpyLoss.  % Mean heat leak through regenerator (W)
   interval   MatrixMass.
   [1,4]      NumRegenerators.
   interval   TidalThermalMass.
   interval   Ve.
   capsule    Capsule.
   matrix     Matrix.

   regenerator(Gas,Capsule,HeatFlow,MassFlowRate,Matrix,NumRegenerators,
         Pm,Power,Pressure,Size,SinkTemperature,SourceTemperature,Ve):-
      setSize,
      heatExchange,
      Type = regeneratorHE,
      Capsule:geometry(capsuleType Geometry),
      Matrix:reynoldsDiameter(ReynoldsDiameter),
      Matrix:reynoldsNumber(R),
      Matrix:getCorrelations(A,C,M,N),
      calculateCrossArea,
```

```
    calculateArea,
    calculateDiameter,
    calculateLength,
    calculateMass,
    calculateT(SourceTemperature,SinkTemperature),
    calculateVolume,
    calculateEnthalpyLoss.

  calculateCrossArea:-
    Matrix:porosity([0.5,0.9] Porosity),
    Capsule:crossArea(interval CapsuleCrossArea),
    CrossArea =:= NumRegenerators * Porosity * CapsuleCrossArea.

  calculateArea:-
    Capsule:volume(interval  CapsuleVolume),
    Matrix:specificArea([10,100000] SpecificArea),
    indomain(NumRegenerators),
    Area =:= NumRegenerators * CapsuleVolume * SpecificArea.

  calculateDiameter:-
    Capsule:externalDiameter(Diameter).

  calculateT(SourceTemperature,SinkTemperature):-
    T =:= (SourceTemperature - SinkTemperature)
          /log(1.61,SourceTemperature/SinkTemperature).
```

```
calculateEnthalpyLoss:-
   Gas:specificHeat([500,50000] SpecificHeat),
   indomain(H),
   [0.001,1000] Lambda =:= H * Area / (SpecificHeat * MassFlowRate),
   TidalThermalMass =:= Ve * SpecificHeat * Density,
   Matrix:material(solids MatrixMetal),
   MatrixMetal:specificHeat([100, 1000] MatrixSpecificHeat),
   MatrixThermalMass   =:=  MatrixMass * MatrixSpecificHeat,
   [0.00001, 1] ThermalMassRatio =:= TidalThermalMass/ MatrixThermalMass ,
   [0.01,1] Effectiveness   =:= Lambda/(Lambda + 2),
   [0.01,1] AdjustedEffectiveness =:= Effectiveness
           * (1 - (ThermalMassRatio*ThermalMassRatio)/ 9),
   [0.001, 1] Ineffectiveness =:= 1 - AdjustedEffectiveness,
   EnthalpyLoss =:= HeatFlow * Ineffectiveness.

calculateLength:-
   Capsule:length(Length).

calculateMass:-
   Capsule:mass(interval CapsuleMass),
   Capsule:volLimits(real VolumeLow, real VolumeHigh),
   interval CapsuleVolume in [VolumeLow, VolumeHigh],
   Capsule:volume(CapsuleVolume),
   Matrix:density(interval  MatrixDensity),
```

```
    MatrixMass =:= MatrixDensity * CapsuleVolume,
    Mass =:= CapsuleMass + MatrixMass.

calculateVolume :-
    Volume =:= CrossArea * Length.

setSize:-
    Gas:species(Species),
    Gas:setConductivity(Species, real Conductivity_r),
    Gas:setViscosity(Species, real Viscosity_r),
    Matrix:type(matrixType MatrixType),
    Matrix:setHeatCorrelations(MatrixType,real C_r, real N_r),
    real HLow               is  5,
    real HHigh              is  10000,
    H                       in  [HLow, HHigh],
    ReLow                   is  10,
    ReHigh                  is  10000,
    R                       in  [ReLow, ReHigh],
    real GLow               = 0,
    real GHigh is Viscosity_r * HHigh * ReHigh
                  /ReHigh**N_r / (C_r * Conductivity_r),
    G                       in   [GLow, GHigh],
    real SpecificConductanceHigh   is  500,
    real SpecificConductanceLow    is  1,
    setPower(Power,real PowerLow,real PowerHigh),
```

```
   real EnthalpyLossLow           is PowerLow * 0.01,
   real EnthalpyLossHigh          is PowerHigh * 0.1,
   EnthalpyLoss      in [EnthalpyLossLow, EnthalpyLossHigh],
   real MassLow            is PowerLow /SpecificConductanceHigh,
   real MatrixMassLow      is MassLow * 0.001,
   real MassHigh           is PowerHigh/SpecificConductanceLow,
   Mass                    in [MassLow, MassHigh],
   MatrixMass                     in [MatrixMassLow, MassHigh],
   real TidalThermalMassLow       is PowerLow / 100000,
   real TidalThermalMassHigh      is PowerHigh /10,
   TidalThermalMass  in [TidalThermalMassLow, TidalThermalMassHigh],
   real AreaLow                   is PowerLow /HHigh,
   real AreaHigh                  is PowerHigh/HLow,
   Area                      in [AreaLow, AreaHigh],
   FrictionLow                    is PowerLow * 0.00001,
   FrictionHigh                   is PowerHigh * 0.1,
   FrictionalPower           in [FrictionLow, FrictionHigh],
   HeatFlow                 in [PowerLow, PowerHigh].

setPower(veryLow, real PowerLow, real PowerHigh):-
   PowerLow  is  0.01,
   PowerHigh is  1.00.
setPower(lowPower, real PowerLow, real PowerHigh):-
   PowerLow  is  0.1,
   PowerHigh is  100.00.
```

```
  setPower(medPower, real PowerLow, real PowerHigh):-
     PowerLow  is   10.0,
     PowerHigh is   1000.00.
  setPower(highPower, real PowerLow, real PowerHigh):-
     PowerLow  is   100.0,
     PowerHigh is   100000.00.
  setPower(veryHighPower, real PowerLow, real PowerHigh):-
     PowerLow  is   100000.0,
     PowerHigh is   1000000000.00.

  numRegenerators(NumRegenerators).
  capsule(Capsule).
  matrix(Matrix).
}
```

Appendix B

Echidna's Output of The Example in Chapter 6

```
wein dongli <17> echidna testHX.db
                    Echidna Version 0.947 beta
                Compiled: Wed Feb 24 15:56:31 PST 1993
                (c) Copyright Expert Systems Lab.
                  Simon Fraser University, 1991, 1992
                        All rights reserved
                        (Expires: 25-Nov-93)
loading data base file "testHX.db" ...
loading knowledge base file "declarations.kb" ...
loading knowledge base file "global.kb" ...
loading knowledge base file "material.kb" ...
loading knowledge base file "geometry.kb" ...
```

```
loading knowledge base file "tube.kb" ...
loading knowledge base file "fin.kb" ...
loading knowledge base file "matrix.kb" ...
loading knowledge base file "capsule.kb" ...
loading knowledge base file "agentMat.kb" ...
loading knowledge base file "heat.kb" ...
loading knowledge base file "reg.kb" ...
loading knowledge base file "testHX.kb" ..
query #0 "precision(24)." issued
done #0 precision(24).
query #1 "heatTest H isa heatTest." issued
done #1 H isa heatTest,
        H:heatTest.
query #2 "H:design." issued
echidna 23> set failbreak off
echidna 24> c
done #2 H:design.
echidna 25> p H
heatTest.0 = {
    real AtmosphericPressure = 100000.
    capsule Capsule = capsule.0.
    interval ColdMassFlowRate = [0.1018435, 0.1018435].
    interval ColdVolumetricFlow = 0.09817187.
    fluidFluid Cooler = tubular.1.
    interval CPS = 1.
```

```
    interval Diameter = 0.5.
    gases Gas = gases.0.
    interval HeatFlow = [3394.774, 3394.797].
    fluidFluid Heater = tubular.0.
    interval HotMassFlowRate = [0.03394783, 0.03394784].
    interval HotVolumetricFlow = 0.09817187.
    interval Length = [0.1, 0.1000001].
    materialChooser MaterialChooser = materialChooser.0.
    PistonClearance = [0.0009999999, 0.001].
    interval Pm = 100000.
    powerRange Power = highPower.
    pressureRange Pressure = low.
    real ReferenceTemperature = 273.
    regenerator Regenerator = regenerator.0.
    interval SinkTemperature = 300.
    interval SourceTemperature = 900.
    sizeRange Size = big.
    interval Stroke = 0.5.
    temperatureRange Temperature = hot.
    interval Ve = 0.09817187.
    interval WallThickness = 0.01.
}
echidna 26> p tubular.0
tubular.0 = {
    heType Type = fluidFluidHE.
```

```
A = [0.04569292, 0.04570484].

interval Area = [0.002056285, 0.002062397].

real AtmosphericPressure = 100000.

C = [0.02299988, 0.023].

interval CrossArea = [6.788815e-05, 6.795683e-05].

DeltaT = [127.1014, 127.6736].

Density = [0.3457992, 0.3458231].

F = [0.003521323, 0.003522992].

Flux = [1646032, 1650938].

interval FrictionalPower = [2.591413, 2.682012].

real FrictionHigh = 10000.

real FrictionLow = 0.0009999999.

G = [499.5499, 500].

gases Gas = gases.0.

interval H = [12934.64, 12946.86].

interval HeatFlow = [3394.774, 3394.797].

interval Length = [0.009999999, 0.01000012].

M = [-0.2000001, -0.2].

M1 = [0.03457069, 0.03576279].

interval Mass = [0.2403686, 0.2522895].

interval MassFlowRate = [0.03394783, 0.03394784].

solids Metal = solids.1.

N = [0.8, 0.8000002].

Nu = [651.5148, 652.0632].

interval Pm = 100000.
```

```
    powerRange Power = highPower.
    interval PressureDrop = [26.39731, 27.31163].
    pressureRange Pressure = low.
    interval R = [367497.6, 367869.8].
    real ReHigh = 500000.
    real ReLow = 1000.
    real ReferenceTemperature = 273.
    ReynoldsDiameter = [0.001309487, 0.001309607].
    sizeRange Size = big.
    temperatureRange Temperature = hot.
    T = [899.9999, 900].
    interval Volume = [3.877904e-07, 2.743187e-06].
    interval VolumetricFlow = 0.09817187.
    fluidFluidType Geometry = tubularFF.
    functionType Function = heating.
    real HHigh = 100000.
    real HLow = 1000.
    NumberOfTubes = [50, 50.00002].
    tube Tube = tube.0.
}
echidna 27> p tube.0
tube.0 = {
    interval ExternalArea = [0.0004840977, 0.000484104].
    interval ExternalCrossArea = [0.0001864908, 0.0001864925].
    interval ExternalDiameter = [0.01540934, 0.01540939].
```

```
    interval InternalArea = [4.114031e-05, 4.114307e-05].
    interval InternalCrossArea = [1.346967e-06, 1.34708e-06].
    interval InternalDiameter = [0.001309562, 0.001309592].
    interval Length = [0.009999999, 0.01000006].
    interval WallThickness = [0.007049871, 0.007049871].
    cylinder Void = cylinder.0.
    interval VoidVolume = [7.755808e-09, 3.11625e-08].
    interval ExternalVolume = [1.86481e-06, 1.865034e-06].
    cylinder Wall = cylinder.1.
    interval WallVolume = [1.851458e-06, 1.851854e-06].
    real DensityHigh = 20000.
    real DensityLow = 1000.
    interval Mass = [0.004999997, 0.005000593].
    solids Metal = solids.0.
    interval MetalDensity = [2699.998, 2700.001].
    interval Pm = 100000.
    pressureRange Pressure = low.
    sizeRange Size = big.
    real VolumeLow = 7.853983e-09.
    real VolumeHigh = 0.1963495.
}
echidna 28> p gases.0
gases.0 = {
    materialState State = {solid, fluid}.
    interval Conductivity = 0.026.
```

```
    interval SpecificHeat = 1000.
    interval Density = 1.14.
    fluidType Type = gas.
    Viscosity = [1.779999e-06, 1.780011e-06].
    gasSpecies Species = nitrogen.
}
echidna 29> p tubular.1
tubular.1 = {
    heType Type = fluidFluidHE.
    A = [0.04569292, 0.04570484].
    interval Area = [0.004565691, 0.004571703].
    real AtmosphericPressure = 100000.
    C = [0.02299988, 0.023].
    interval CrossArea = [0.0003329978, 0.0003331926].
    DeltaT = [99.70232, 99.91213].
    Density = [1.037377, 1.037424].
    F = [0.003311694, 0.003313005].
    Flux = [742561.8, 743545.3].
    interval FrictionalPower = [0.6423409, 0.7245953].
    real FrictionHigh = 10000.
    real FrictionLow = 0.0009999999.
    G = [305.6594, 305.836].
    gases Gas = gases.0.
    interval H = [7442.618, 7446.937].
    interval HeatFlow = [3394.774, 3394.796].
```

```
interval Length = [0.009999999, 0.01000012].
M = [-0.2000001, -0.2].
M1 = [0.01907349, 0.02145767].
interval Mass = [0.2403686, 0.2522895].
interval MassFlowRate = [0.1018435, 0.1018435].
solids Metal = solids.3.
N = [0.8, 0.8000002].
Nu = [833.1009, 833.5062].
interval Pm = 100000.
powerRange Power = highPower.
interval PressureDrop = [6.552734, 7.374078].
pressureRange Pressure = low.
interval R = [499711.3, 500000].
real ReHigh = 500000.
real ReLow = 1000.
real ReferenceTemperature = 273.
ReynoldsDiameter = [0.002910095, 0.002910333].
sizeRange Size = big.
temperatureRange Temperature = roomTemperature.
T = [300, 300.0001].
interval Volume = [1.555625e-06, 5.071631e-06].
interval VolumetricFlow = 0.09817187.
fluidFluidType Geometry = tubularFF.
functionType Function = cooling.
real HHigh = 100000.
```

```
    real HLow = 1000.
    NumberOfTubes = [50, 50.00002].
    tube Tube = tube.1.
}
echidna 30> p tube.1
tube.1 = {
    interval ExternalArea = [0.0004909599, 0.0004909663].
    interval ExternalCrossArea = [0.0001918154, 0.0001918171].
    interval ExternalDiameter = [0.01562777, 0.01562783].
    interval InternalArea = [9.142571e-05, 9.142877e-05].
    interval InternalCrossArea = [6.65178e-06, 6.651962e-06].
    interval InternalDiameter = [0.002910196, 0.002910226].
    interval Length = [0.009999999, 0.01000006].
    interval WallThickness = [0.00635877, 0.00635877].
    cylinder Void = cylinder.2.
    interval VoidVolume = [5.456919e-08, 7.797588e-08].
    interval ExternalVolume = [1.918056e-06, 1.91828e-06].
    cylinder Wall = cylinder.3.
    interval WallVolume = [1.851654e-06, 1.851854e-06].
    real DensityHigh = 20000.
    real DensityLow = 1000.
    interval Mass = [0.004999997, 0.005000593].
    solids Metal = solids.2.
    interval MetalDensity = [2699.998, 2700.001].
    interval Pm = 100000.
```

```
    pressureRange Pressure = low.
    sizeRange Size = big.
    real VolumeLow = 7.853983e-09.
    real VolumeHigh = 0.1963495.
}
echidna 31> p regenerator.0
regenerator.0 = {
    heType Type = regeneratorHE.
    A = [19.99998, 20.00002].
    interval Area = [7.067176, 7.069561].
    real AtmosphericPressure = 100000.
    C = [0.9999998, 1].
    interval CrossArea = [0.1767145, 0.1767146].
    DeltaT = [5.622372, 5.631909].
    Density = [1.196568, 1.196592].
    interval Diameter = [0.5199999, 0.5200001].
    F = [0.06515193, 0.0651561].
    Flux = [479.2213, 480.4134].
    interval FrictionalPower = [0.005763371, 0.01053174].
    real FrictionHigh = 10000.
    real FrictionLow = 0.0009999999.
    G = [0.1920885, 0.1921482].
    gases Gas = gases.0.
    interval H = [85.37003, 85.37063].
    interval HeatFlow = [3394.774, 3394.796].
```

```
interval Length = [0.1, 0.1000001].
M = [-0.8200002, -0.8199999].
M1 = [0.002384186, 0.003576279].
interval Mass = [9.617465, 9.641307].
interval MassFlowRate = [0.03394783, 0.03394784].
solids Metal = solids.4.
N = [0.4999995, 0.5000005].
Nu = [32.83719, 32.84911].
interval Pm = 100000.
powerRange Power = highPower.
interval PressureDrop = [0.2382814, 0.3586816].
pressureRange Pressure = low.
interval R = [1078.994, 1079.073].
real ReHigh = 10000.
real ReLow = 10.
real ReferenceTemperature = 273.
ReynoldsDiameter = [0.009999996, 0.01000012].
sizeRange Size = big.
temperatureRange Temperature = roomTemperature.
T = [260.0922, 260.0923].
interval Volume = [0.01767145, 0.01767147].
interval VolumetricFlow = [0.02836487, 0.02837679].
interval EnthalpyLoss = [343.8331, 345.9035].
interval MatrixMass = [5.293042, 5.304963].
NumRegenerators = [1, 1].
```

```
    interval TidalThermalMass = [117.4686, 117.4722].
    interval Ve = 0.09817187.
    capsule Capsule = capsule.0.
    matrix Matrix = matrix.0.
}
echidna 32> p capsule.0
capsule.0 = {
    capsuleType Geometry = cylindrical.
    cylindricalCapsule CCapsule = cylindricalCapsule.0.
    annularCapsule ACapsule = unbound(8756512).
    interval Length = [0.1, 0.1000001].
    solids Metal = solids.5.
    pressureRange Pressure = low.
    sizeRange Size = big.
    temperatureRange Temperature = hot.
}
echidna 33> p matrix.0
matrix.0 = {
    A = [19.99999, 20].
    C = [0.9999999, 1].
    M = [-0.8200001, -0.82].
    N = [0.4999998, 0.5000002].
    Porosity = [0.8999999, 0.9].
    Density = [269.9991, 270.0014].
    matrixType Type = meshMatrix.
```

```
    solids Metal = solids.6.
    SpecificArea = [359.9992, 360.0112].
    ReynoldsDiameter = [0.009999995, 0.01].
    R = [1078.994, 1079.073].
    FibreDiameter = [2e-05, 0.002].
    SphereRadius = [2e-05, 0.001].
    WireDiameter = [0.001111077, 0.001111077].
}
```

REFERENCES

[Agogino 87] A.M. Agogino and A.S. Almgren, **Techniques For Integrating Quantative Reasoning and Symbolic Computation In Engineering Optimization.** *Engineering Optimization*, 1987, Vol. 12, pp 117-135.

[Ashley 92] S. Ashley, **Engineous Explores the Design Space.** *Mechanical Engineering*, February 1992, pp 49-52.

[Brown 89] D.C. Brown and B. Chandrasekaran, **Design Problem Solving: Knowledge Structures and Control Strategies**, Morgan Kauffmann, 1989.

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

[Cutkosky 90] M.R. Cutkosky and J.M. Tenenbaum, **Research in Computational Design at Stanford.** *Research in Engineering Design* 2(1), 1990, pp 53-59.

[Davis 84] R. Davis, **Diagnostic Reasoning Based on Structure and Behaviour.** *Artificial Intelligence*, 24, 1984, pp 347-410.

[Dechter 89] R. Dechter and I. Meiri, 1989, **Experimental Evaluation of Preprocessing Techniques in Constraint-Satisfaction Problems.** In *Preceeding of the Eleventh International Joint Conference on Artificial Intelligence*, Menlo Park, California, 1989, pp 290-296.

[Doyle 79] J. Doyle, **A Truth Maintenance System.** *Artificial Intelligence*, 12, 1979, pp 127-272.

[Duda 79] R. Duda, P. Hart, K. Konolige and R. Reboh, **A Computer-Based Consultant for Mineral Exploration**, SRI International, 1979.

[Haralick 80] R. Haralick and G. Elliot, **Increasing Tree Search Efficiency for Constraint-Satisfaction Problems**. *Artificial Intelligence* 14(3), 1980, pp 263-313.

[Havens 90] W.S. Havens, **Echidna Constraint Reasoning System: Next-Generation Expert System Technology**, Technical Report CSS-IS TR 90-09. The Expert Systems Laboratory, Centre for Systems Science, Simon Fraser University, Burnaby, British Columbia, Canada.

[Hayes-Roth 83] F. Hayes-Roth, D.A. Waterman and D.B. Lenat, **Building Expert Systems**, Addison-Wesley, 1983.

[Hentenryck 89] P.V. Hentenryck, **Constraint Satisfaction in Logic Programming**, The MIT Press, Cambridge, MA, 1989.

[Jones 82] J.D. Jones, **Thermodynamic Design of The Stirling Engine**, Ph.D. Thesis, The University of Reading, England, 1982.

[Jones 93] J.D. Jones and D. Li, **A Sense of Scale**, to appear in IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, May 1993.

[Kumar 92] V. Kumar, **Algorithms for Constraint-Satisfaction Problems: A Survey**. *AI Magazine*, 5, 1992, pp 32-44.

[Luger 89] G.F. Luger and W.A. Stubblefield, **Artificial Intelligence and The Design of Expert Systems**, Benjamin/Cummings, 1989.

[Mackworth 77] A.K. Mackworth, **Consistency in Networks of Relations**. *Artificial Intelligence*, 8, 1977, pp. 99-118.

[Martin 71] W.A. Martin and R.J. Fateman, **The MACSYMA System**. *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, California, 1971, pp 59-75.

[McDermott 91] D. McDermott, **A General Framework for Reason Maintenance**. *Artificial Intelligence* 50, 1991, pp 289-329.

[Mittal 86] S. Mittal, C.L. Dym and M. Morjaria, **PRIDE: An Expert System For The Design of Paper Handling Systems.** *Computer*, July, 1986, pp 102-114.

[Mittal 90] S. Mittal and B. Falkenhainer, **Dynamic Constraint-Satisfaction Problems.** In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Menlo Park, California, 1990, pp 25-32.

[O'Connor 92] L. O'Connor, **Stirling Machines**. *Mechanical Engineering*, June 1992, pp. 75-79.

[Papalambros 87] P.Y. Papalambros, **Knowledge-based Systems in Optimal Design**, NATO ASI Series, Vol. F27, 1987.

[Papalambros 88] P.Y. Papalambros and D.J. Wilde, **Principles of Optimal Design**, Cambridge University Press, Cambridge, 1988.

[Prosser 91] P. Prosser, **Hybrid Algorithms for The Constraint-Satisfication Problem**, Research Report, AISL 46-91, Computer Science Dept., Univ. of Strathclyde.

[Reichgelt 91] H. Reichgel, **Knowledge Representation : An AI Perspective**, Ablex Publishing Corporation, 1991, pp 168-171.

[Rinderle 91] J.R. Rinderle and L. Balasubramaniam, **Automated Bond Graph Modeling and Simplication to Support Design**. *Journal of Dynamic Systems Measurement and Control*, EDRC 24-54-91.

[Ross 77] A. Ross, **Stirling Cycle Engines**, Solar Engines/Phoenix, 1977.

[Sell 85] P.S. Sell, **Expert Systems - A Practical Introduction**, MacMillan Publisher Ltd., 1985.

[Sidebottom 92] S. Sidebottom, W.S. Havens and S. Kindersley, **Echidna Constraint Reasoning System (Version 1): Programming Manual**, Expert Systems Laboratory, Centre for Systems Science, Simon Fraser University, Burnaby, British Columbia, Canada, December 1992.

[Stallman 77] R. Stallman and G.J. Sussman, **Forward Reasoning and Dependency-Directed Backtracking**. *Artificial Intelligence* 3(5), 1977, pp 135-196.

[Suh 90] N.P. Suh, **The Principles of Design**, Oxford University Press, 1990.

[Sidebottom 91] G. Sidebottom and W.S. Havens, **Hierarchical Arc Consistency Applied to Numeric Processing in Constraint Logic Programming**, Technical Report CSS-IS TR 91-06. The Expert Systems Laboratory, Centre for Systems Science, Simon Fraser University, Burnaby, British Columbia, Canada.

[Walker 80] G. Walker, **Stirling Engines**, Oxford University Press, 1980.

[Ward 87] A. Ward and W. Seering, **Representing Component Types for Design**, Design Automation Conference, Boston, MA, September, 1987.

[Zeigler 90] B.P. Zeigler, **Object-Oriented Simulation With Hierarchical Structures**, Academic Press Inc., 1990.