1987

A knowledge-based system for integrated circuit package design /

Robert Steven Voros
Lehigh University

A KNOWLEDGE-BASED SYSTEM

FOR

INTEGRATED CIRCUIT

PACKAGE DESIGN

by

Robert Steven Voros

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

1987

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

May 12, 1987
(date)

Donald J. Hillman
Professor In Charge

L. J. Vanier
Chairman of Department

Acknowledgments

The writer would like to thank his parents, William and Alice, for their love and support. He is also very thankful for having an uncle and an aunt, Franklin and Renie Voorhees, who made his Lehigh career possible. He would like to express his gratitude to Kitty Liebhardt who encouraged him to continue on with his education. He would also like to show his appreciation to the Semiconductor Research Corporation for their financial support. And finally, he would like to thank Dr. Donald J. Hillman for his guidance through his graduate program.

Table of Contents

Abstract

This paper describes a knowledge-based system for integrated circuit package design. First, the Prolog programming language is described, and the concept of relational databases in the language is introduced. Next, fast retrieval methods such as indexing and inverted files in Prolog are presented. An expert system shell, ***LUESS*** is then described. This shell is unique because it generates a menu-driven expert system from simple English-like rules. Finally all the ideas introduced in the paper are pulled together and used in a prototype expert system for the Semiconductor Research Corporation Project.

1 The Semiconductor Research Corporation (SRC) Project

In the past decade, semiconductor technology has moved toward large-scale integration (LSI) and very large-scale integration (VLSI) of integrated circuit chips. With this advancing trend of semiconductor technology, considerable research is needed in the packaging of semiconductor components. Because of the increasing knowledge, the process of package design is fragmented among many individuals -- each having his own specialty. Currently we are building large materials databases and learning the latest techniques in package design from experts in the field. The primary goal of this project is the total integration of these databases and design tools into a unified knowledge base that will provide expert packaging advice via a user-cordial interface. Eventually the power of the expert system will be teamed with the power of CAD and finite element analysis to totally automate the design and testing of integrated circuit packages.

1.1 Packages

A package has to fulfill many needs. A semiconductor component, or silicon die, needs electronic connections to outside logic

for it to function. It needs an enclosure to protect it from hostile environments and to facilitate mounting to circuit boards. The semicondutor's package must also keep it cool as it dissipates heat while performing its function. In addition, since most silicon components have a finite failure rate, the package must allow for easy removal and replacement from a PC board.

The central focus of this thesis is on the application of Artificial Intelligence in the Semiconductor Research Corporation Project. The goal of the project is the development of a knowledge-based system for integrated circuit package design. This paper is divided into six chapters. The first chapter describes databases in Prolog. The next chapter shows how to speed up searching in Prolog. Chapter Three describes ***LUESS***, an expert system shell. The fourth chapter goes into more depth about packages and demonstrates the use of ***LUESS*** with simple packaging rules. The last chapter lists conclusions and future developments and enhancements to the system.

2 Prolog's Database

Prolog can be used to implement a relational database. There are three characteristics that make Prolog relational. First, facts are perceived by the user as a collection of tables. Next, the Prolog language is utilized as a powerful query language. Finally, all possible solutions can be found via backtracking. Backtracking is the process of re-satisfying queries which is described below.

As with relational databases, information is regarded as a set of tables in Prolog. A Prolog table consists of one or more facts, each having a predicate and a collection of arguments. There is generally no limit to the number of tables that can be asserted in a Prolog database. Nor is there a limit to the number of arguments in a Prolog predicate. The following is a sample table in Prolog:

```
part(shovel,[top_handle_assbly,
                    scoop_shaft_connector,
                    shaft, nail, rivet, scoop_assembly]).
part(top_handle_assbly,[top_handle,nail,bracket]).
part(scoop_assembly,[scoop,blade,rivet]).
```

The Prolog language can be used as a powerful query language for databases. Typically, the user has a number of different facts asserted into RAM, and Prolog questions are formulated to select any of these facts. A question in Prolog consists of a predicate name

and one or more uninstantiated arguments. For example, a question for the database above could be ***part***(shovel,X), where X is an uninstantiated variable. Upon execution of this question, Prolog will match "shovel" in the database and instantiate X with a list of component parts. The process of "matching" that has been referred to above is called unification. "A set of atomic formulae are ***unifiable*** if, as Prolog structures, they can be matched together." [1] Thus, in Prolog, unification takes care of assigning values to variables, accessing data structures via a general pattern-matching mechanism, and certain kinds of tests for equality. [2]

```
***part***(shovel, X).

***returns:*** X = [top_handle_assbly,
                    scoop_shaft_connector,
                    shaft, nail, rivet, scoop_assembly]).
```

Finally, Prolog searches for all possible solutions to a question. It does this with the use of backtracking. "Backtracking consists of reviewing what has been done, attempting to re-satisfy

---

1. W.F. Clocksin and C.S. Mellish, Programming in Prolog, New York: Springer-Verlag, 1984, p. 247

2. _____, Turbo Prolog User's Manual, Borland International, 1986, p. 54

the goals by finding an alternative way to satisfying them."[3] Backtracking can be initiated by typing a semicolon after Prolog returns a solution:

```
part(X,Y).

returns:

X = shovel,
Y = [top_handle_assbly,
     scoop_shaft_connector,
     shaft, nail, rivet, scoop_assembly]) ;

X = top_handle_assbly,
Y = [top_handle,nail,bracket]) ;

X = scoop_assembly,
Y = [scoop,blade,rivet])
```

Prolog allows facts to be written to and read from external files much like facts asserted in *RAM* databases. There are many benefits in using external files. For example, the programmer is no longer restricted by the size of available *RAM* for his database -- only by the size of available disk space. Secondly, with external disk files, heap overflows are not likely to occur due to a decrease in need for internal memory. Finally, different searching techniques can be employed to speed up access to the facts in the database. This is particularly important for large databases (2000+ entries), for

---

3. W.F. Clocksin and C.S. Mellish, Programming in Prolog, New York: Springer-Verlag, 1984, p. 37

using Prolog's pattern matching facilities may take too long for real time applications (See Expert Knowledge Management Systems).

2.1 Prolog Database Clauses: Explanation

The last three pages of this section contain a group of clauses that are used to implement a database in Turbo Prolog. Most of these clauses appear on pages 143-144 of the Turbo Prolog manual; however, they are slightly modified. The listed clauses require the user to enter an index file and data file. They also allow for the user to directly access specific facts in these files.

The first two clauses assert facts into a database on disk. If the user types in the following goal:

```
dbass(alloy(1,aluminum),
   "indexfile","datafile").
```

Prolog would execute the first dbass clause. To prevent overwriting existing files, dbass checks to see if the files have already been created. ***Existfile*** performs this function. If the two files have been previously created, dbass will open the files by calling ***openappend*** and append new facts to the end of them.

Appending is done by calling ***writedevice***, which tells Turbo Prolog that the device that is to be written to is the data file. ***Filepos***(datafile,Pos,0) returns an integer value in ***Pos*** which is the location of the file position pointer in the data file. Finally the new fact is written into the data file, and the file is closed. The same sequence is repeated for the index file: in this case the file position pointer, ***Pos***, is added to the end of the index file. The second dbass clause performs the same sequence of events; however, if the two files do not exist, they are created.

Dbread opens both the ***datafile*** and the ***indexfile*** for reading and calls the dbaaccess clause. If the user types:

```
dbread(alloy(X,Y),
  "datafile","indexfile").
```

all the facts in the database will be returned, because both X and Y are uninstantiated. If the user enters the following Prolog query:

```
dbread(alloy(1,Y),
 "datafile","indexfile").
```

dbaaccess will read the facts sequentially until it finds one that matches the 1 in the first attribute. ***dbaaccess*** does this by recursively calling itself.

Dbret retracts facts from the database. ***Dbret*** does not actually remove facts from the database file, it just replaces the facts'index numbers with -1's so that the facts are overlooked by dbaaccess.

The last two clauses access the database randomly instead of sequentially. ***Dbread2*** opens both the datafile and the ***indexfile*** and calls ***dbaccess*** (NOTE: different from above). Because the data file positions are formatted into a field of 7 as they are placed into the index file, access into the data file can be done very quickly. If the 52nd fact is sought in the data file, dbaccess calculates the position into the index file with the following equation:

Index3 = (Index - 1) * 9.

In our case, Index = 52, and 9 is the field length (7) plus <cr> and <lf>. The fileposition now points to the 52nd integer in the index file. This integer is a pointer that points to the 52nd fact in the data file. Using this file position, the fact is read from the data file and returned to the user.

The clauses introduced in this section allow Prolog to be a very powerful database system. Since random access can be performed, facts can be retrieved very quickly from the database. And by making simple modifications, hashing and binary searches are possible. The next

chapter describes indexing and inverted files, which greatly decrease Prolog's search time.

2.2 PrologBase: Excerpts from Source [4]

```
/*****************************************************************/
/*                                                               */
/*                        PrologBase                             */
/*                                                               */
/*                     Robert Steven Voros                       */
/*                                                               */
/* This group of Prolog clauses allow Prolog to have external    */
/* database files.                                               */
/*                                                               */
/*****************************************************************/

DOMAINS
   file = datafile; indexfile

DATABASE
   alloy(integer,string)

PREDICATES
   dbass(dbasedom, string, string)
   dbaaccess(dbasedom, real)
   dbret(dbasedom, string, string)
   dbret1(dbasedom, real)
   dbread(dbasedom, string, string)
   dbread2(dbasedom, integer, string, string)
   dbaccess(dbasedom, integer, integer, real)

CLAUSES

/********************** Dbassert ***********************************/

dbass(Term, IndexFile, DataFile) :-
   existfile( DataFile ), existfile( IndexFile ), !,
     openappend( datafile, Datafile ),

----------
```

4. _____, Turbo Prolog Manual, Borland International, 1986, pp. 143-144

```
 writedevice (datafile ),
  filepos( datafile, Pos, 0 ),
 write( Term ), nl,
closefile( datafile ),
openappend( indexfile, IndexFile ),
  writedevice( indexfile ),
  writef( "%7.0\n", Pos),
  closefile( indexfile ).

dbass( Term, IndexFile, DataFile ) :-
   openwrite( datafile, Datafile ),
     writedevice( datafile ),
       filepos( datafile, Pos, 0 ),
     write( Term ), nl,
   closefile( datafile ),
   openwrite( indexfile, IndexFile ),
      writedevice( indexfile ),
      writef( "%7.0\n", Pos),
   closefile( indexFile).

/********************** Dbread ************************************/

dbread( Term, IndexFile, DataFile ) :-
   openread( datafile, DataFile ),
     openread( indexfile, IndexFile ),
   dbaaccess( Term, -1 ).

dbread( _,_,_) :-
   closefile( datafile ), closefile( indexfile ), fail.

/********************** Dbaccess ***********************************/

dbaaccess( Term, Datpos ) :-
   Datpos >= 0,
     filepos( datafile, Datpos, 0 ),
     readdevice( datafile ),
   readterm( Dbasedom, Term ).

dbaaccess( Term, _ ) :-
   readdevice( indexfile ),
     readreal( Datpos1),
   dbaaccess( Term, Datpos1 ).

/********************** Dbretract ***********************************/

dbret( Term, Indexfile, Datafile) :-
```

```
   openread( datafile, DataFile ),
     openmodify( indexfile, IndexFile ),
   dbret1( Term, -1 ).

dbret1( Term, Datpos ) :-
   Datpos >= 0,
    filepos( datafile, Datpos, 0),
     readdevice( datafile ),
       readterm( Dbasedom, Term ),!,
         filepos( indexfile, -9, 1 ),
       flush(indexfile),
      writedevice(indexfile),
   writef( "7.0\n",-1),
  writedevice(screen).

dbret1(Term, _) :-
   readdevice(indexfile),
     readreal( Datpos1 ),
   dbret1( Term, Datpos1).

/********************** Dbread2 ***********************************/

dbread2(Term, Index, Indexfile, Datafile) :-
  openread(datafile, Datafile),
   openread(indexfile,Indexfile),
    dbaccess(Term,Index,0,0),
   closefile(indexfile),
  closefile(datafile),  !.

/********************** Dbaccess2 *********************************/

dbaccess(Term, Index, Index2, Datpos) :-
   Index3 = (Index - 1) * 9,
     filepos( indexfile, Index3, 0),
      readdevice(indexfile),
       readreal(Datpos1),
      filepos( datafile, Datpos1, 0),
     readdevice(datafile),
   readterm(Dbasedom,Term), !.
```

3 Prolog Databases: Expert Knowledge Management Systems

Many times expert systems lack the means to provide efficient knowledge bases, and database technology lacks knowledge representation schemes and reasoning capabilities. An expert knowledge management system, however, integrates artificial intelligence with data processing techniques. As mentioned earlier, Prolog provides database facilities much like those in relational database programs, but searching tends to be very slow. If a database of 500 facts were asserted into RAM, typical searches would take four to seven minutes; searches would take even longer in external files. In query-oriented environments, there is not a lot of time available for scanning entire files for desired attributes. To speed up retrieval, data processing techniques have to be applied. The techniques used in this research are ***indexing*** and ***file inversion***.

A typical Prolog database file consists of many facts, each being a predicate with one or more attributes. Below is a sample of a record in a Prolog database. In this case each record is split across three files: alloy, alloy2, and comments.

```
alloy(1, "ALUMINUM", "6 . 24", "A91060", "", "AA1060",
      "10 KSI", "A", "", "4 KSI", "A", "A1060 (89)", "43 %",
      "E", "1A", "19 HB", "X", "0.0000", "0", "B", "", "B",
      "0").
```

```
alloy2(1, ["STRESS CORROSION RESISTANT"],
          ["WROUGHT", "FORGING", "EXTRUSION", "SHAPES",
           "SHEET", "PLATE", "BAR", "WIRE", "TUBE"]).

comments(1, ["Applications requiring very good resistanc to
              corrosion and", "good formability, but tolerate
              low strength.  Chemical pro-", "cess equipment is
              typical."]).
```

An inverted list is a list of all records having a given value of some primary or secondary key. Let us take the above example. The database contains 500 metal alloys, and these alloys can be of certain classes, certain forms and certain properties. These facts can be represented as the following:

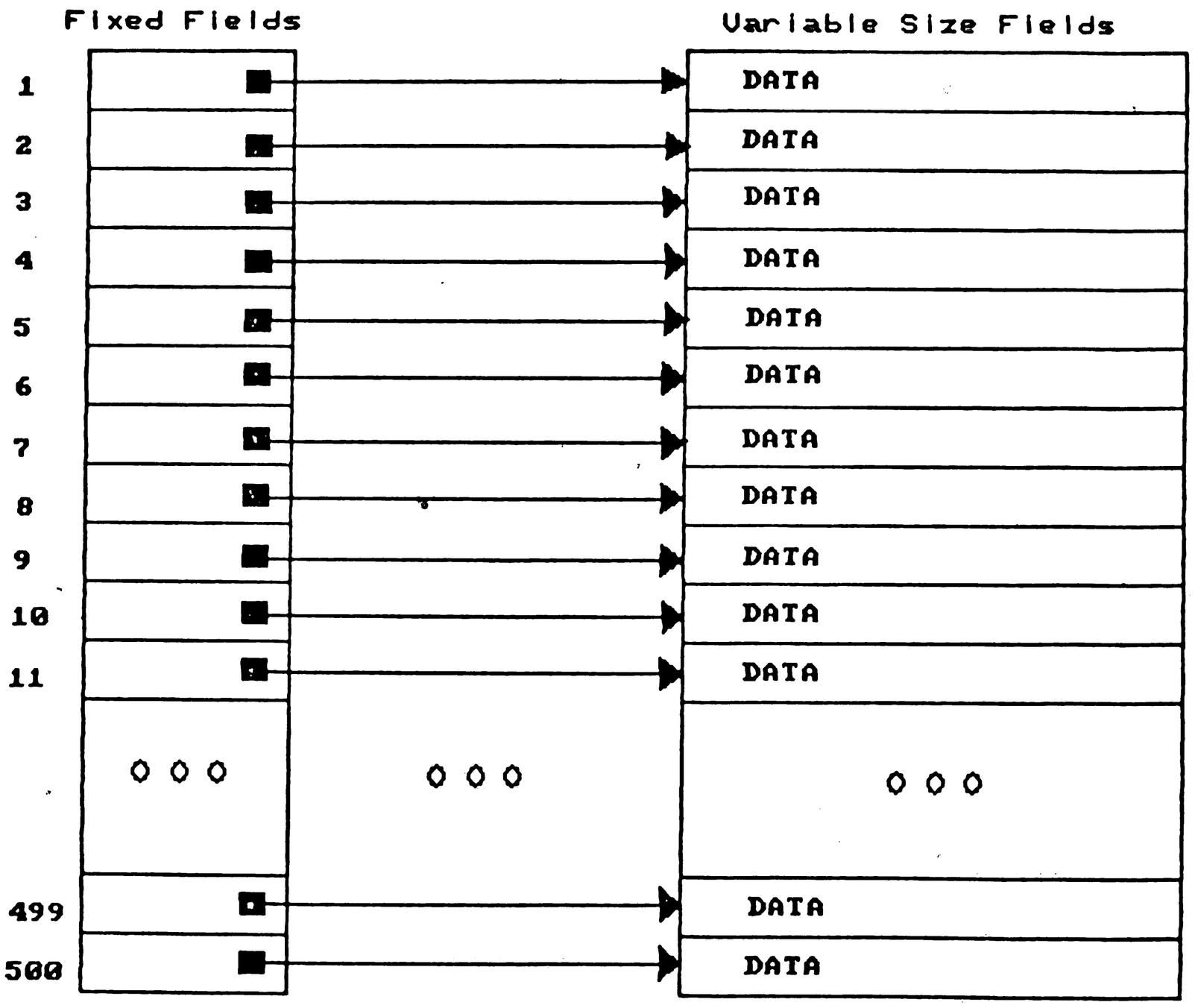
```
inverted_CLASS("ALUMINUM",[1,2,3,4,5,6,7,8, ..., 60]).
inverted_CLASS("TIN",[200,203,205,206, ..., 215]).

inverted_FORMS("WROUGHT",[25,30,31,32,89,100]).
inverted_FORMS("FORGING",[230,231,400,423,424,425]).

inverted_PROPS("STRESS CORROSION RESISTANT",[34,35,60,67]).
inverted_PROPS("ELECTRICAL MATERIALS",[401,402,403,404]).
```

As seen above, an inverted file contains unique keys with lists of index numbers. Each index number corresponds to an alloy in the main metals database. Remember from chapter 1, two files are created: a ***data file*** and an ***index file***.

The ***data file*** consists of the facts in Prolog. The ***index file***, on the other hand, consists of pointers into the ***data file***. The following figure shows this relationship between the files:


Fixed Fields
Variable Size Fields
1
2
3
4
5
6
7
8
9
10
11
499
500
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA
DATA

Because the index file has fixed fields, pointers can be retreived directly by multiplying index-1 by nine (field length + <cr> + <lf>), and reading the file position.

Now that there is an efficient way of retrieving facts from a database, simple Prolog predicates can be used for database queries. Remember, no facts are asserted into RAM: just lists of index numbers. Therefore, only list manipulation clauses are needed. The most common list manipulation clauses are *intersection* and *union*. [5]

```
member(Element,[Element|_]).
member(Element,[_|Tail]) :- member(Element,Tail).

intersection([],X,[]).
intersection([X|R],Y,[X|Z]) :-
                member(X,Y),
                !,
                intersection(R,Y,Z).
intersection([X|R],Y,Z) :- intersection(R,Y,Z).

union([],X,X).
union([X|R],Y,Z) :-
                member(X,Y),
                !,
                union(R,Y,Z).
union([X|R],Y,[X|Z]) :- union(R,Y,Z).
```

If the user wants to find all those alloys that contain aluminum or tin, a union is taken between the lists associated with the

---

5. W.F. Clocksin and C.S. Mellish, Programming in Prolog, New York: Springer-Verlag, 1984, p. 154

aluminum and tin classes in the inverted file. If the user desires to view all those metals that are composed of aluminum and are stress corrosion resistant, an intersection can be taken between the lists.

As will be seen in the last sections of this paper, the techniques developed here in expert knowledge management systems are applied to various applications.

4 LUESS Version 1.0

Expert system "shells" are becoming more popular as research in Artificial Intelligence escalates. Shells are problem-independent tools used to facilitate the design of expert systems. Much like a language, shells interpret rules that are entered by the user.

Expert system tools such as shells furnish the power of Artificial Intelligence in simple English-like style notation. Knowledge is acquired from seasoned experts. However, many times experts do not understand the representational mechanisms of common Artificial Intelligence languages. There are two alternatives for knowledge acquisition. The first is not to give the expert direct access to the system; rather, use a computer scientist as a buffer. The second alternative is to use a shell that takes as input simple "if-then" rules. It is this second approach that I employ in my system. The Lehigh University Expert System Shell (***LUESS***) was created for the Semiconductor Research Corporation. The shell compiles simple English-like rules into Prolog predicates which are then used to build a menu-driven expert system.

One of the key features of ***LUESS*** is that it is a menu-driven expert system. Many expert system shells require the user to do a considerable amount of typing. And other systems allow the user to

answer only "yes" or "no". ***LUESS***, on the other hand, prompts the user with questions and menus. After the initial knowledge is entered into the system, users need only point to the answers to the questions.

4.1 Rule Syntax

The syntax of the rules is quite simple. There are currently nine key words: ***RULE***, ***IF***, ***IS***, ***IN***, ***ARE***, ***AND***, ***OR***, ***QUESTION***, ***RANGE***. And there are four Key punctuation marks: ' " ', ' ; ', ' : ', and '.'.

The following is the syntax diagram for a rule:

```
RULE : goal IS value IF
                [AND clauses] ;
                [OR  clauses] ;
                QUESTION: [question] .

AND clauses --> [goal IS value [AND]]*
                [goal IS IN RANGE number TO number
                     [AND]]*
                [goal IS number [units] [AND]]*

OR  clauses --> [goal IS value [OR]]*
                [goal IS IN RANGE number TO number
                     [OR]]*
                [goal IS number [units] [OR]]*
```

Question --> " [string] "

Every rule must begin with the keyword ***RULE*** followed by a colon. During compilation, the shell attaches unique integers to each rule. These values will be displayed in future edits of the rulebase.

The first line of a rule is the "***goal***". That is, if all the conditions are satisfied, the goal will be proven true. The conditions are themselves goals, but they are goals of other rules that are generated by the system. The following is a sample goal line:

RULE : package ***IS*** pga ***IF***
. . .

As mentioned above, condition lines follow the goal line. These lines are either ***ANDed*** or ***ORed***. An ***AND*** clause consists of one or more conditions ***ANDed*** together, and all the conditions must be proven true for it to succeed. A typical ***AND*** clause is displayed below. In this case, in order for the package to be a pin grid array, the clock rate has to be high, the dielectric has to be ceramic and the there should be between 200 and 1000 pins.

```
RULE : package IS pga IF
   clock_rate IS high AND
   dielectric IS ceramic AND
   pins ARE IN RANGE 200 TO 1000 ;
```

QUESTION: "" .

An ***OR*** clause is exactly the same as an ***AND*** clause, but in this case the lines are ***ORed***. At least one condition must be proven true for an ***OR*** clause to succeed. Below, in order for the clock rate to be high, the chip technology must be cmos or ecl.

```
RULE : clock_rate IS high IF
   chip_technology IS cmos OR
   chip_technology IS ecl ;
   QUESTION: "What is the clock rate?"
```

Many of the rules will not contain any conditions. They will consists only of a goal line and a ***QUESTION*** line. These rules are the "end of the line" in the inference engine's recursion. An inference engine can be thought of as a rule interpreter. It is the job of the inference engine to decide what rules are to be applied. The strategy that the inference engine uses is called the control strategy. In ***LUESS*** a goal driven control strategy is used: the system starts with an initial goal, breaks this goal into a number of subgoals, and these subgoals are broken into more subgoals. The inference engine will work its way down this tree of goals until it arrives at a point were it can go no further. ***LUESS*** will then search the rulebase for any other rules with the same rule name and create a selection menu. The user will then be prompted with a question, and will be asked to

select one of the items in the selection menu. The following is an example of a "goal-question rule":

RULE : low_cost ***IS*** yes ;
QUESTION: "Is a low cost package required?" .

4.2 Knowledge Representation in *LUESS*

LUESS compiles the rules down into Prolog predicates, which are then used as the system's rulebase. Because the number of rules may run into the thousands, database techniques are used to retrieve only those rules that are needed (See Databases in Turbo Prolog). There exists seven slots in the rule predicate: Rule_Number, Rule_Question, Rule_Name, Value, Range, OR_List, and AND list. The following shows the syntax of a rule in the rulebase of ***LUESS***:

```
rule(Rule_Number, Rule_Question, Rule_Name, Value, Range,
                OR_List, AND_List).
```

Each rule contains a unique rule number. As mentioned earlier, unique rule numbers are generated during compilation. In addition, each rule has an optional question. Questions are optional because rules that contain ***AND*** clauses or ***OR*** clauses never prompt the user

with a selection menu. Only "goal-question" type rules need questions.

Each rule has a rule name and rule value. Both these fields do not have to be unique. Rule_Name is a goal and Rule_Value is the value of the goal. For example, if we had "*RULE* : package *IS* pga *IF*', package would be the Rule_Name and pga would be the Rule_Value. If the above were the goal of a "goal-question" type rule, *LUESS* would search for all those rules with the same Rule_Name and generate a selection menu containing the Rule_Values from each.

During compilation, if a rule is encountered that contains a *RANGE*, the key word *range* will be placed in the Rule_Value slot. The *range* slot will then contain a set of real numbers. If the keyword *IS* is encountered followed by a numeric value, Rule_Value will take the value *is* and the range slot will contain a single real number.

Finally, each rule contains a list of *ORed* rules and a list of *ANDed* rules. These lists are lists of integers, each associated with other rule numbers generated by the compiler. Below is a set of rules as they would appear in an *LUESS* rulebase.

```
rule(1, package, "", [], [], [9, 10, 12]).
rule(9, clock_rate, "What is the clock rate?", high,
```

```
        [], [24, 26], []).

rule(24, chip_technology, "What is the chip technology of the chip?",
        cmos, [], [], []).

rule(26, chip_technology, "What is the chip technology of the chip?",
        ecl, [], [], []).

rule(10, dielectric, "What is the dielectric material?", ceramic,
        [], [], [17, 18]).

rule(12, pins, "What range is the number of pins in?", range,
        [1000, 200], [], []).
```

4.3 The Inference Engine: A description of how it works

The inference engine of *LUESS* is very simple. Like Prolog *LUESS* implements a goal-driven strategy based upon resolution. Resolution is the process of starting with an initial goal statement and resolve it with one of the hypotheses to create a new clause. This new clause is then resolved with one of the hypotheses to give another new clause. This process continues until it reaches a point where the new clause can be conclusively answered. In Prolog terms, goals lead to sub-goals, and sub-goals lead to other sub-goals, etc. This process continues until the sub-goals are satisfied.[6]

---

6. W.F. Clocksin and C.S. Mellish, Programming in Prolog, New York: Springer-Verlag, 1984, pp. 250-252

The inference engine is displayed below. ***Check*** has two parameters: Rule_No and a List_of_rules that have to be proven. If the list of rules is empty, the rule can be evaluated by calling ***evaluate_rule***. If the list of rules is not empty ***check*** is called recursively, this time with the first element of the new list of rules. ***LUESS*** continues to creep down the left side of the tree until a rule can be evaluated. If it succeeds it is asserted into RAM with those rules that have been proven so far. If it fails, the system will backtrack until another rule can be evaluated. ***LUESS*** will continue to execute until the goal has been proven or disproven.

```
check(Rule_No, []) :- evaluate_rule(Rule_No).
check(Rule_No, [BNO|REST] ) :-
      get_rule(BNO, Rule_Quest, Rule_Name, Value, Range, Any, All),
      check(BNO, All),
      asserta_if_necessary(BNO, Rule_Name, Value, Range, Any, All),
      check(0, REST).
```

4.4 Main Menu

4.4.1 Consultation

After a rulebase is loaded into ***LUESS***, the system is ready for consultation. The consultation screen consists of three windows: a

trace window, a selection window, and a dialog window. The Trace window displays *LUESS's* reasoning; that is, the facts that have been inferred. Menus generated by *LUESS* are displayed in the selection window. And finally the dialog window contains questions that are asked of the user.

4.4.2 Read Knowledge

By selecting this choice in the main menu, the user is able to read in a knowledge base. Knowledge bases have ".dba" extensions and are generated by compiling the user's source code.

4.4.3 List Knowledge

List Knowledge allows the user to display the knowledge from a rulebase. The user should have loaded or compiled his source code before using this facility.

4.4.4 Compile Knowledge

If knowledge was edited on an outside editor, it can be loaded into ***LUESS*** by using this option.

4.4.5 Push to DOS

The user can temporarily leave ***LUESS*** using this option. To return, type "Exit" at the DOS prompt.

4.4.6 Exit the system

This option allows the user to halt execution of the ***LUESS***.

4.4.7 Edit Knowledge

Edit Knowledge allows the user to edit the knowledge in his rulebase. Upon completion, the knowledge is recompiled.

Summary of Editor Keystrokes[7]

Keys	*Purpose*
Esc or F10	Exit the editor Arrow
keys, PgUp, PgDn, Home, End	Move the cursor
Del	Delete the character at the cursor
Ctrl-K B	Mark the beginning of a block
Ctrl-K K	Mark the end of a block
Ctrl-K H	Un-mark a block
Ctrl-K C	Copy a marked block to the position indicated by the cursor
Ctrl-K Y	Delete a marked block
Ctrl-K V	Move a marked block to the position indicated by the current cursor posit
F1	Help information
F5	Copy block
F6	Move block
F7	Delete block
F3	Search
Shift-F3	Repeat last search
F4	Search and replace
Shift-F4	Repeat last search and replace

---

7. _____, Turbo Prolog Manual, Borland International, 1986, p. 13.

5 Application: An Expert System for Package Design

As described at the beginning of this paper, a package has to fulfill many needs. A semiconductor component, or silicon die, needs electronic connections to outside logic for it to function. It needs an enclosure to protect it from hostile environments and to facilitate mounting to circuit boards. The semicondutor's package must also keep it cool as it dissipates heat while performing its function. In addition, since most silicon components have a finite failure rate, the package must allow for easy removal and replacement from a PC board. To give the reader an idea of the vast amount of knowledge that is needed for package design, the next two sections introduce the types of materials and typical guidelines for VLSI package design.

5.1 Materials

Typical desirable properties in a package are hermeticity, high thermal conductivity, low dielectric constants, high mechanical strength, thermal expansion that matches other components, low sintering temperatures, and low cost.

Hermeticity is an important property in a semiconductor component's package. The package must be impervious to water vapor, air, etc. This is particularly important for high reliability and for optical component packages with thermoelectric coolers. Possible failure points in a package that would destroy its hermeticity are pores, delaminations, vias, ceramic-metal interfaces, and seals.

High Thermal Conductivity is another important property in a package. As the density of semiconductor chips gets greater, the thermal load in the chip becomes very large. And as packages get larger, the temperature constraints become more severe. It is therefore desirable to utilize a high thermal conductive material. Conditions for high thermal conductivity are low average atomic mass, strong interatomic bonding, and a simple crystal structure. Materials with high thermal conductivities are SiC (270 W/mK), BeO (250 W/mK), and AlN (60-170 W/mK).

Thermal expansion is also important. As the semiconductor chip expands due to heat, the package should also expand, preferably matching the thermal expansion of the chip it is encasing.

Finally, low sintering temperatures are desirable. The sintering process becomes increasingly expensive as the temperature increases.

Lower sintering temperatures also allow for the use of less resistive metals. Typical sintering temperatures are the following: Glass bonded ceramics (500-1200), Borosilicate (1000), SiO2 (1000-1200), and SiC (2100).

5.2 Typical VLSI Package Design Guidelines

Below is a list of package design guidelines. It is by no means complete; however, it demonstrates the type of knowledge that is needed in package design. It is this type of knowledge that will make up the SRC rulebase.

Skeleton List [8]

Die Characteristics

Size_____ I/O Pads_____ Thickness_____ Power_____

Cavity Design (for eutectic attach with scrub)

Size (add .05" to .08" per side to die size)_____
Length_____ Width_____
Depth (add .005" to die thickness)_____
Die pad flatness .003 max., .002 preferred.

---

8. John A. Nelson, VLSI Package Design Considerations, IEEE VLSI Computer-Aided Design Testing, and Packaging, 1982, pp. 320-324

Outer Wire Bond Ledge Design

-Cavity Length

-Package suppliers prefer line widths and spaces of .010".

-Calculated by: L = (N X WL) + [(N+1) X WS]
N = number of bonding pads per cavity side.
WL = the width of the bonding pad.
WS = the width of the space.

Wire Bond Ledge Length

L = 2H + .010"
H = thickness of the top layer of ceramic

Sealing Area Design

A minimum of .08" per side should be allowed.

Electrical Requirements

Resistance of special leads_____ milliohms maximum.
Resistance of all others _____ milliohms maximum.
Capacitance: Specify lead to ground or lead to lead as required.
Inductance: Low inductance grounds or other conductors may require special design enhancements.

Thermal Design

Maximum die power _____.
Desired maximum junction temperature TJ = _____.
Maximum ambient temperature TA = _____.
ThetaJ. Cooling media = $\frac{TJ - TA}{\text{max power}}$

This chapter was to have given the reader an idea of the complexities involved in designing packages for integrated circuits.

It introduced many properties and design criteria that are used, but was by no means a complete list.

6 Enhancements and Future Directions

6.1 The SRC Expert System

An expert system is currently being designed for the Semiconductor Research Corporation. Its goal is to decide what kinds of packages would be best for various semiconductor components. If a package does not exist, the expert system will aid the user in the design of a new package using information from a database that contains instantiated packages. The system is currently being developed using the Lehigh University Expert System Shell (***LUESS***) on the Personal Computer. The expert system will eventually use a CAD system for designing packages, ANSYS for finite element analysis, and ***LUESS*** written in Quintus Prolog as the inference engine. The Appendix contains a prototype system developed in ***LUESS***. This last chapter describes enhancements of ***LUESS*** that are currently being added.

Currently, many databases have been created, and user friendly routines have been developed to retrieve the information. However, much of the knowledge in these databases is not pertinent to package design; for example, information on cast iron or magnesium is not

needed. Therefore, the next goal for the project is to prune the databases and make them specific to package design only.

Next the databases need to be integrated with the shell. The most feasible method of integration would be to build a database language into the rule language. The most common and most widely used database language is SQL. SQL is a relational language used by many of the most popular database systems. This querying system was developed to be used by both technical and nontechnical people. In the case of ***LUESS***, only a subset of SQL needs to be implemented, for the expert system shell will not be updating the databases, just retrieving information (at this time). The integration of such a language would increase the power of ***LUESS*** tremendously. Instead of the expert entering all the pertinent information into the rulebase, the needed information can be retrieved from existing databases and used as if they were part of the rulebase.

The most common SQL command is ***SELECT***. ***Select*** has the following format:

```
SELECT field(s)
FROM table
WHERE predicate
```

An example of ***SELECT*** is the following:

```
SELECT *
FROM  metals
WHERE property =
    stress_corrosion_resistant
```

The above syntax structure for ***SELECT*** can easily be added to ***LUESS***'s rule language. The internal representation for rules with imbedded SQL can be represented as the following:

```
rule(Rule_Number, Rule_Question, Rule_Name, Value, Range,
          OR_List, AND_List, SQL).

SQL --> [database_name, [[where predicate],[where predicate]]]
```

All that is needed in the internal representation of rules is a new field called ***SQL***. When Rule_Name is "database", this SQL slot is filled with a list that contains the database query. An example of a rule being represented internally is the following:

```
RULE : "database" IS "" IF
    SELECT *
    FROM metals
    WHERE property = "stress_corrosion_resistant"
```

```
        AND         property = "electrical";
        QUESTION:   "" .

rule(12, "database", "", "", [], [], [], [metals,
        [[property,[stress_corrosion_resistant]],
        [[property,[electrical]]])
```

The database query will then be performed by a Prolog clause that opens the database and sifts through the properties of the metals. The only command that is needed in *LUESS* at this time is *SELECT*. But in the future, other SQL commands will be added that will allow for database updates and insertions.

Another goal of the SRC project is to integrate CAD into the system so that a package design can be displayed on the screen. With this feature, finite element analysis routines will be built in, providing information on the thermal properties of the package.

6.2 Summary

This paper introduced a knowledge-based system for integrated circuit package design. The system currently developed is still a prototype, but eventually it will be fully implemented and will contain design rules of many experts. The concept of relational databases in the Prolog programming language was introduced, and fast

retrieval methods such as indexing and inverted files were described. An expert system shell, *LUESS* was then presented. This shell uniquely generates menu-driven expert systems from simple English-like rules. The project will continue for the next two years, and will continue to grow and change. And in the future, this integrated expert system might help lead the American semiconductor companies back to the front of this rapidly changing technology.

7 Bibliography

Clocksin, W.F., and Mellish, C.S., Programming in Prolog, 2nd ed., New York: Springer-Verlag, 1981.

Date, C.J., An Introduction to Database Systems, 4th ed. revised. Reading, Massachusetts: Addison-Wesley Publishing Company, 1986.

Harmon, Paul, and King, David, Expert Systems, New York: John Wiley & Sons, Inc., 1985.

Kroenke, David, Database Processing, 2nd ed., Chicago: Science Research Associates. Inc., 1983.

Rich, Elaine, Artificial Intelligence, New York: McGraw-Hill Book company, 1983.

Winston, Patric Henry, Artificial Intelligence, 2nd ed., Reading, Massachusetts: Addison-Wesley Publishing Co., 1984.

_____. Turbo Prolog, Borland International, 1986.

Nelson, John A., "VLSI Package Design Considerations", IEEE VLSI Computer-Aided Design, Testing and Packaging, IEEE (1982), pp 320-325.

8 Appendix

8.1 Sample Expert System

The following is a short example of an expert system created in *LUESS*. In this case it is a prototype expert system that helps recommend packages for integrated circuits.

```
Rule 1: package IS pga IF
          clock_rate IS high AND
          dielectric IS ceramic AND
          pins IS IN RANGE 200 TO 10000 ;
          QUESTION: "" .

Rule 2: package IS dip IF
          clock_rate IS low AND
          pins IS IN RANGE 0 TO 100 AND
          dissipation IS IN RANGE 0 TO 0.001 ;
          QUESTION: "" .

Rule 3: package IS soic IF
          clock_rate IS medium AND
          pins IS IN RANGE 0 TO 100 AND
          dissipation IS IN RANGE 0 TO 0.001 ;
          QUESTION: "" .

Rule 4: package IS plcc IF
          clock_rate IS high AND
          pins IS IN RANGE 99 TO 201 ;
          QUESTION: "" .

Rule 5: package IS clcc IF
          clock_rate IS high AND
          pins IS IN RANGE 0 TO 100 ;
```

```
        QUESTION: "" .

Rule 6: package IS lccc IF
        clock_rate IS high AND
        dielectric IS ceramic AND
        pins IS IN RANGE 99 TO 201 ;
        QUESTION: "" .

Rule 7: clock_rate IS low IF
        chip_technology IS pmos ;
        QUESTION: "What is the clock rate?" .

Rule 8: clock_rate IS medium IF
        chip_technology IS nmos OR
        chip_technology IS ttl OR
        chip_technology IS lsttl OR
        chip_technology IS i2l ;
        QUESTION: "What is the clock rate?" .

Rule 9: clock_rate IS high IF
        chip_technology IS cmos OR
        chip_technology IS ecl ;
        QUESTION: "What is the clock rate?" .

Rule 10: dielectric IS ceramic IF
        low_cost IS no AND
        high_reliability IS yes ;
        QUESTION: "What is the dielectric material?" .

Rule 11: dielectric IS plastic IF
        low_cost IS yes AND
        military IS no AND
        high_reliability IS yes ;
        QUESTION: "What is the dielectric material?" .

Rule 12: pins IS IN RANGE 200 TO 10000;
        QUESTION: "What range do the pins fall in?" .

Rule 13: pins IS IN RANGE 0 TO 100;
        QUESTION: "What range do the pins fall in?" .

Rule 14: pins IS IN RANGE 99 TO 201;
        QUESTION: "What range do the pins fall in?" .

Rule 15: dissipation IS IN RANGE 0 TO 0.001;
        QUESTION: "What is the range of heat dissipation?" .
```

```
Rule 16: low_cost IS yes;
         QUESTION: "Is a low cost package required?" .

Rule 17: low_cost IS no;
         QUESTION: "Is a low cost package required?" .

Rule 18: high_reliability IS yes;
         QUESTION: "Does the chip require a highly reliable package?" .

Rule 19: high_reliability IS no;
         QUESTION: "Does the chip require a highly reliable package?" .

Rule 20: military IS yes;
         QUESTION: "Will the chip be used in military applications?" .

Rule 21: military IS no;
         QUESTION: "Will the chip be used in military applications?" .

Rule 22: chip_technology IS nmos;
         QUESTION: "What is the chip technology?" .

Rule 23: chip_technology IS pmos;
         QUESTION: "What is the chip technology?" .

Rule 24: chip_technology IS cmos;
         QUESTION: "What is the chip technology?" .

Rule 25: chip_technology IS ttl;
         QUESTION: "What is the chip technology?" .

Rule 26: chip_technology IS ecl;
         QUESTION: "What is the chip technology?" .

Rule 27: chip_technology IS lsttl;
         QUESTION: "What is the chip technology?" .

Rule 28: chip_technology IS i2l;
         QUESTION: "What is the chip technology?" .
```

8.2 Vita

Robert Voros, son of William and Alice Voros, was born in Bethlehem, Pennsylvania February 6, 1963. In 1985, he received a Bachelor of Science degree in computer engineering from Lehigh University. Robert then continued at Lehigh for his Master of Science degree in computer science. For his first year in graduate school, he served as computer assistant to the office of the Vice President and Provost at Lehigh. Later he became a research assistant in the Computer Science and Electrical Engineering department where he did research in the area of Artificial Intelligence for the Semiconductor Research Corporation. Currently, Robert is managing the Artificial Intelligence Laboratory at Lehigh and is pursuing a doctoral degree in Computer Science at Lehigh.