PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

PROLOG-BASED EXPERT SYSTEM FOR LOGIC DESIGN

Fumihiro Maruyama, Tamio Mano, Kazushi Hayashi, Taeko Kakuda,
Nobuaki Kawato, and Takao Uehara

FUJITSU LIMITED
Kawasaki, Japan

ABSTRACT

Building an expert system to assist in hardware logic design is one of the activities undertaken in the Fifth Generation Computer Systems (FGCS) Project. This paper describes the current status of such an expert system.

Logic design involves a variety of aspects ranging from purely algorithmic processing to tasks that are heavily dependent on expert knowledge. This system explores those aspects by incorporating designers' expertise, for instance.

Given a concurrent algorithm described in OCCAM, the system designs a CMOS circuit to aid the designer in the logic design process. OCCAM is a programming language characterized by its treatment of concurrency. It enables the user to specify concurrent algorithms with great ease. The result of functional design, the first half of the logic design process, is a finite-state machine description in DDL, a hardware description language. This is the first level at which the correspondence to hardware concepts emerges. Circuit design, followed by CMOS design, the second half of the logic design process, transforms the finite-state machine description into a CMOS circuit.

A prototype has been implemented in Prolog. In the course of implementation, we evaluated Prolog for its effectiveness as an implementation language for a new generation of CAD systems.

1. INTRODUCTION

The Fifth Generation Computer Systems (FGCS) Project has undertaken research on knowledge-based systems in some application areas. Among these is hardware logic design. Previous work in this area includes the Palladio system, developed at Stanford University (Brown et al. 1983). Palladio is an attempt to create an integrated design environment. Its main concern is to provide compatible design tools ranging from simulators to layout generators, to permit specification of digital systems from architecture to layout in compatible languages, and to provide the means for explicitly representing, constructing, and testing such design tools and languages.

We chose hardware logic design as an application area for the following three reasons. First of all, the application must be in an area in which human expertise plays a significant role. In hardware logic design, only experienced designers can achieve good results. Secondly, our target is a design system rather than a diagnostic system. While there are a number of successful knowledge-based diagnostic systems, few knowledge-based design systems are in practical use. This fact poses the challenge of exploring the many unknown factors that such a system involves. Lastly, hardware logic design encompasses a variety of aspects; many types of knowledge contributes to expertise in this domain. Thus, it is possible for us to explore the use of expert knowledge from various angles.

The implementation language is Prolog, which is also used as the underlying knowledge-representation language. Knowledge representation is an important issue. A single multi-purpose framework for knowledge representation would be simplest. However, as described above, hardware logic design employs various kinds of knowledge, and design data must be represented as well. In addition, human designers switch from one representation to another in the course of their work. For these reasons, we have not adopted any particular existing tool for knowledge representation. We hope that our approach will result in an effective new knowledge-representation framework.

2. OVERVIEW

The system covers the entire design processes from specifications, described in OCCAM (Taylor and Wilson 1983), to complete CMOS circuits. OCCAM is a programming language characterized by its treatment of

concurrency. (Strictly speaking, we use OCCAM-S, the "s-expression" version of OCCAM, for internal expressions.) It enables the user to easily specify concurrent algorithms. However, specifications are not necessarily hardware-oriented. In other words, the user is not required to describe specifications based on hardware concepts.

Between the initial stage of OCCAM design specifications and the final stage, in which CMOS circuits are produced, finite-state machine description is generated in DDL, a hardware description language. In this intermediate design stage the correspondence with hardware concepts first emerges. The system's final output is CMOS basic cells, functional cells, and the connections between them.

The system consists of ten subsystems. Figure 1 shows how the subsystems are related to one another. All the subsystems appear in the figure with the exception of the editor subsystem. The top two subsystems are responsible for functional design. The functional design subsystem determines the application of hardware concepts in implementing the concurrent algorithms, and produces the finite-state machine description in DDL (Dietmeyer 1971). The state machine optimization subsystem inspects the finite-state machine description and makes modifications to refine it.

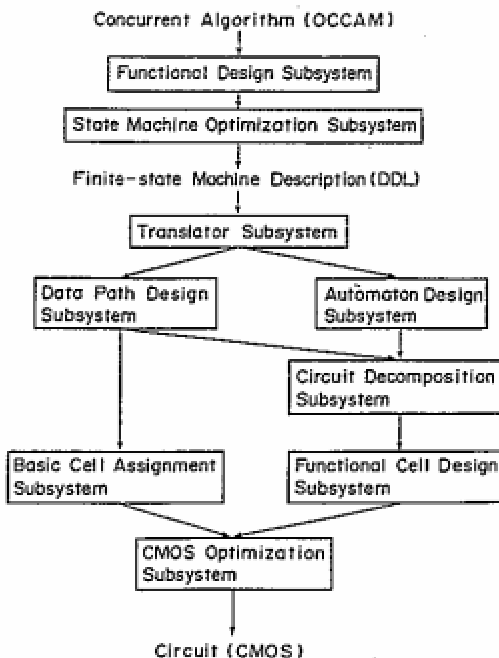


Figure 1. System configuration

The finite-state machine description in DDL is functional, not structural. (Strictly speaking, we use DDL-S, the "s-expression" version of DDL, for internal expressions.) In order to design circuits, we need information about hardware structure; this means that functional descriptions must be transformed into structural descriptions. Here, the translator subsystem plays its part. It generates two kinds of design information: that concerning data paths and that concerning control circuits.

The control circuit design subsystem implements automata having the appropriate states using flip-flops. It designs a control circuit around these flip-flops according to information on state transition supplied by the translator subsystem. The data path design subsystem allocates data paths around functional components, such as registers, memories, adders, and decoders.

Both the data path design subsystem and the control circuit design subsystem generate logical expressions, which are then implemented as combinational circuits using CMOS functional cells. It is not always possible to implement a given combinational circuit using a single functional cell, because large cells fail to meet the high performance requirements. The circuit decomposition subsystem takes a logical expression and breaks it down into subexpressions in such a way that each subexpression can be implemented by a single cell satisfying the performance requirements. These subexpressions are passed to the functional cell design subsystem, which creates a functional cell for each subexpression.

On the other hand, functional components, such as registers, memories, adders, decoders, and I/O pins, are designed by the basic cell assignment subsystem. The subsystem searches the basic cell library for the appropriate cell. If one is found, it is assigned to the hardware component, possibly with slight modification. Otherwise, the subsystem either assembles a cell using basic cells in the library as components, or it attempts to design one from scratch.

The system provides a facility that optimizes the entire CMOS circuit after the basic cells and the functional cells have been completed. It also provides a user interface facility, which is used throughout the design process under control of the editor subsystem.

In the design process the system explores a variety of design aspects. In the course of this exploration, the techniques applied range from algorithmic approaches to knowledge-based approaches. We describe several of these techniques in the following sections.

3. FUNCTIONAL DESIGN

Functional design can be thought of as the phase of design that determines the type of hardware components required and describes their interactive behavior. The primary functions of the functional design subsystem are as follows:

1) Implementing variables described in OCCAM-S using hardware elements (registers, etc.).

2) Designing hardware control mechanisms for "constructs" of OCCAM-S. ("seq" for sequential processes, "par" for parallel processes, etc.)

3) Implementing communication between processes described in OCCAM-S. ("input" for inputting a value from a channel, "output" for outputting a value to a channel)

The end result of the functional design subsystem is a finite-state machine description, which is further refined by the state machine optimization subsystem.

The functional design subsystem is one of the most knowledge-intensive parts of the system. Figure 2 shows its four design levels and the four processes linking them.

1) Cognitive Process

This process takes the entire specifications level into account, makes deductions concerning the intension of the concurrent algorithm, and stores these deductions in the working memory in the form of high-level concepts described in Prolog.

2) Checking Process

The above deductions are generic in that the cognitive process does not take context into account. The checking process makes a conjecture in reference to the working memory, goes back to the specifications level, and checks whether it is true. If so, the checking process puts the assertion into the working memory.

3) Instantiation Process

This process instantiates high-level concepts in the working memory and puts partial DDL-S descriptions into the draft.

4) Construction Process

The partial DDL-S descriptions in the draft are finally assembled into DDL-S code.

The following is a rough sketch of how this subsystem works. As an example, we use the pattern-matching chip proposed by M. J. Foster and H. T. Kung (Foster and Kung 1979). Figure 3 shows part of its algorithm. Here, concentrate on a part of it:

```
[par,        /* following input processes are
                executed in parallel */
  [input,pin,p],    /* from the channel pin to
                       the variable p */
  [input,sin,s]]                       ----- (1).
```

```
PROC comp(CHAN pin, sin, pout, sout, dout)=
  VAR p,s:
  SEQ
    PAR
      p:=O
      s:=O
    WHILE TRUE
      SEQ
        PAR
          pout ! p
          sout ! s
        PAR
          pin ? p
          sin ? s
        dout ! p=s:
```

(a) in OCCAM

```
[proc, comp, [[chan, pin, sin, pout, sout, dout]],
  [[var, p, s],
   [seq,
    [par,
      [assign, p, O],
      [assign, s, O]],
    [while, true,
      [seq,
        [par,
          [output, pout, p],
          [output, sout, s]],
        [par,
          [input, pin, p],
          [input, sin, s]],
        [output, dout, [equal, p, s]]]]]]]
```

(b) in OCCAM-S

Figure 3. Algorithm for the pattern-matching chip (comparator)

The cognitive process takes the entire structure of the algorithm into account from a hardware perspective. It deduces, for instance, that (1) ought to be implemented as a state. In due course, [input, pin, p] is recognized as an input operation, and further, as a passive operation, as is [input, sin, s]. The following two deductions are put into the working memory as Prolog facts:

```
passive_operation([input, pin, p],...).
passive_operation([input, sin, s],...).
```

By "passive operation" we mean an operation that is not invoked internally.

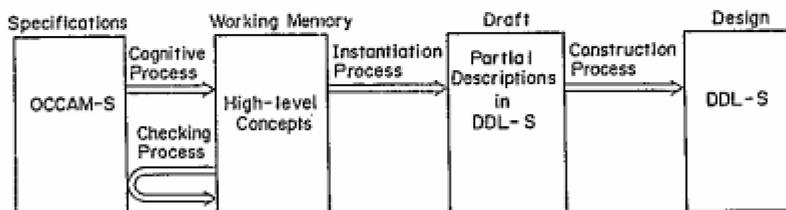


Figure 2. Overview of the functional design subsystem

On the other hand, the checking process makes the conjecture that (1) may be an idle state, by which we mean a state in which the automaton waits for a signal from the outside every time it returns after a series of operations; this will be referred to as the idling condition. Using the knowledge that a state can be an idle state if it is the only passive state, the checking process checks whether (1) is a passive state in reference to the following knowledge:

```
  passive_state([par,X|Y],...):-
    passive_operation(X,...), !.
  passive_state([par,X|Y],...):-
    passive_state([par|Y],...).
```

In this way, the checking process finds that (1) is an idle state:

```
  idle_state([par, [input, pin,  p],  [input,
sin, s]],...).
```

Once the working memory is completed, the instantiation process begins to generate partial DDL-S descriptions. (1) is transformed into a partial DDL-S description, the definition of a state, using the following instantiation knowledge:

```
  idle_state_instantiation(X,...):-
    idling_condition(X,Y,...),
    action(X,Z,...),
    assert(state([idle,Y,Z],...)).
```

The construction process assembles the partial DDL-S descriptions stored in the draft and produces the final DDL-S code.

The above sketch illustrates that hardware concepts can be represented explicitly and clearly by Prolog predicates. Prolog constitutes a "concept-oriented" paradigm.

4. CIRCUIT DESIGN

Circuit design stands between functional design and CMOS design, and provides all the information necessary for designing CMOS functional cells and assigning basic cells. This section illustrates this using several examples.

4.1 Translator

The translator subsystem transforms the DDL-S finite-state machine description into design information for the circuit design process. It gathers and edits conditions for terminal connection, register transfer and state transition operations; it then organizes this data in a frame-like structure.

The translator subsystem, which was implemented in Prolog, generates this information while parsing the DDL-S code. If we had used a language like PL/I instead of Prolog, a translator generator would have been indispensable. Using Prolog as the implementation language saved a considerable amount of work. Roughly speaking, all we had to do was to write Horn clauses in keeping with the DDL-S grammar. The extracted information is classified into eight categories of hardware components, arithmetic data, and data about logical expressions.

All logical expressions are given unique names to prevent their arbitrary duplication by combinational circuits. The occurrences of each logical expression are counted and used to determine which logical expression to implement as a CMOS functional cell.

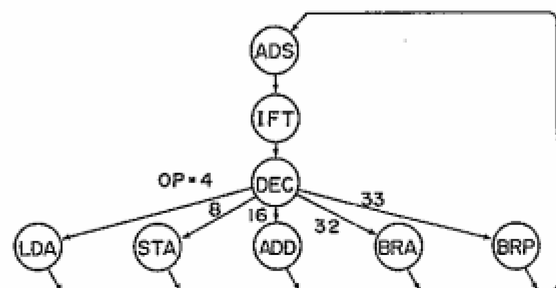
4.2 Control Circuit Design

In this section, we discuss the implementation of automata.

We use a very simple computer as an example. The DDL description of this machine is shown in Figure 4. Figure 5 is a skeleton state diagram for an automaton CPU.

```
<SYSTEM> SIMPLE:
    <TIME> CLK <(10)>.
    <STORAGE> M(1024, 16).
    <REGISTER> ACC(16), IR(16), MAR(10),IAR(10).
    <AUTOMATON> CPU: CLK:
      <STATES>
         ADS: MAR <- IAR, IAR <- IAR +1, -> IFT.
         IFT: IR <-M(MAR, 0:16), -> DEC.
         DEC: MAR <-ADR,
              ?IR(0:5) # 4 -> LDA
                       # 8 -> STA
                       #16 -> ADD
                       #32 -> BRA
                       #33 -> BRP
         LDA: ACC <- M(MAR,.0: 16), -> ADS.
         STA: M (MAR, 0:16) <- ACC, -> ADS.
         ADD: ACC <- ACC + M(MAR, 0:16), -> ADS.
         BRA: IAR <- ADR, -> ADS.
         BRP: |* ~ACC(0) *| IAR <-ADR., -> ADS.
      <END>.
    <END>CPU.
<END> SYSTEM
```

Figure 4. Simple computer



ADS : address set
IFT : instruction fetch
DEC : decode
LDA : load address
STA : store address
ADD : add
BRA : branch
BRP : branch-positive

Figure 5. State diagram

There are several approaches to implementing this 8-state machine. At least three flip-flops are required, although this point is also subject to discussion, as we shall presently see. Coding eight states into three flip-flops might produce the most desirable control characteristics. Figure 6 shows an instance of such a control circuit. At the other extreme, we might use eight flip-flops, one for each state. This state assignment greatly simplifies the design process, but this approach is probably uneconomical.

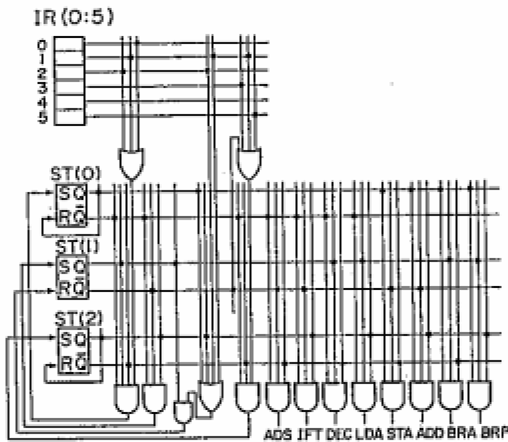


Figure 6. Control circuit

Still another approach recognizes that the settings of the high-order six bits of the instruction register (IR) reflect a specific value for each instruction type and, thus, can be used to distinguish the execution states of the CPU. We might build a 4-state 2-flip-flop machine with states ADS, IFT, DEC, and a new state, EXC. When this 2-flip-flop machine is in the EXC state, the actual state of the CPU is determined by the settings of the high-order six bits of the IR. In this case, the state diagram is modified as shown in Figure 7 and the control circuit is modified as shown in Figure 8.

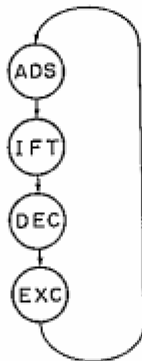


Figure 7. Improved state diagram

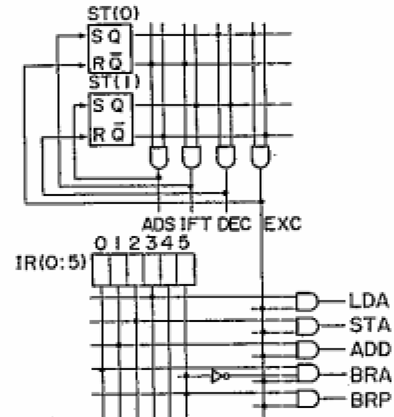


Figure 8. Improved control circuit

This approach, described by Dietmeyer (Dietmeyer 1971), is often very effective. Circuit designers deal with a great deal of such kinds of knowledge.

This knowledge, however, is not always applicable. For example, the last approach is applicable if and only if the following four conditions are satisfied:

1) If a state has two or more branches, the transition between the current state and each of its subsequent states is indicated by the value in the same register (here, IR).

2) Each subsequent state has no other predecessor.

3) When the state transition occurs, the register value is unaffected.

4) In each subsequent state, there is no recursive transition that changes the contents of the register.

A Prolog implementation of the last approach is shown in Figure 9. The program structure parallels the above conditions, as indicated by the corresponding numbers. In the program, the "for" predicate is similar to the LISP "map" function, and allows us to legibly express processing on all the elements of a list.

```
branch_state_reduction(State) :-
    next_states(State, Next_states),
    for(X in Next_states,
        set_of(Y, (transition condition(State, Next_states, Register), ...... (1)
                   unique_predecessor(X, State), .......................... (2)
                   unaffected(State, X, Register), ........................ (3)
                   unaffected(X, X, Register), ............................ (4)
                   Y = (X, Condition)),
              Z)),
    update_table(State, Z).
```

Figure 9. Prolog program

4.3 Design of Functional Components

There are two extreme automatic allocators for functional components: a distributed allocator and a central allocator (Thomas et al. 1983). The distributed allocator adds a new functional component for each unique reference in the functional description. However, this design is inefficient. The central allocator tries to map all references onto a structure with a single functional component. While such an approach might be adequate for a simple computer, it is not optimal for a large system.

Our system is capable of determining whether to add a new functional component or to map the reference onto a structure with a single functional component on a case-by-case basis, by checking whether a component can be requested from more than one operation at the same time.

Now, we describe the design of an actual functional component, taking a decoder as an example. The specifications for a decoder consist of two items: input and the values into which it is to be decoded (that is, the values that correspond to the output lines on a one-to-one basis). In general, an n-bit decoder needs an n-input gate for each output line, and the size of a full n-bit decoder is proportional to 2^n. Whether we design an n-bit decoder using n-input gates, or assign a basic cell to it, it is often much larger than necessary. Under certain conditions, we can make it considerably smaller.

The following rule deals with such a case. The idea here is that, if the values into which the input is to be decoded are few and the other values can be ignored, they need only be distinguished from one another.

IF

The number of values to be taken into account does not exceed the number of bits in the input,

THEN

1) For each value that is the only value with the ith bit on (for some i), connect its output line with the ith input line.

2) For the others, connect their output lines with gates so that they can be distinguished from the others.

Figure 8 gives an example. An IR decoder is shown at the bottom of the figure. 1), above, is applied to the values, 4, 8, 16, and 33; 2) is applied to 32.

5. CMOS DESIGN

This section discusses the implementation of a random logic function on an array of CMOS transistors. A heuristic algorithm that minimizes the array size is presented.

The basic layout of a functional cell is illustrated in Figure 10, starting from the AND/OR (sum of products) logic specification. AND/OR gates in the logic diagram correspond to the series/parallel connections in the circuit diagram. It is clear that for every AND/OR specification of a Boolean function, one can obtain a series/parallel implementation in CMOS technology, in which the p-MOS side and n-MOS side are each other's dual.

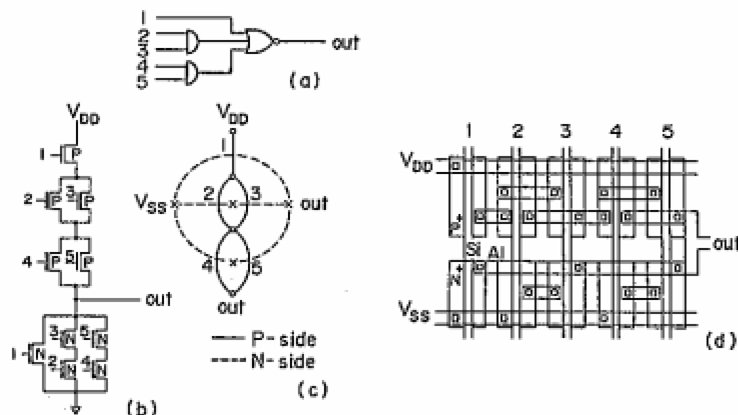


Figure 10. Basic layout of the functional cell
(a) Logic diagram (b) Circuit (c) Graph model (d) Layout

Physically adjacent gates can be connected by a diffusion area. The layout can be further improved by judicious pairing of sources and drains. Separation is required when there is no connection between physically adjacent transistors, as shown in Figure 11. However, the best results are obtained using the alternative circuit shown in Figure 12(b). This circuit is logically equivalent to the one shown in Figure 10(b). Since both the cell height and the basic grid size are functions of the technology employed, an optimal layout is obtained by minimizing the number of separations. Finally, the layout of the functional cell can be optimized as shown in Figure 12(d). The size of this array is smaller than that of the basic layout by almost 50%.

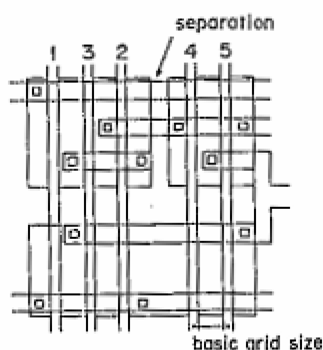


Figure 11. Optimization of layout

In order to reduce the array size, it is necessary to find a pair of Euler paths (an Euler path is an edge chain that contains all the edges of the graph model) having the same sequence of labels on the dual graph model, because p-type and n-type gates corresponding to the same input signal have the same horizontal position in the CMOS array. Since the graph-theoretical algorithm to obtain the best solution is exhaustive, the following heuristic algorithm has been proposed (Uehara and vanCleemput 1981):

Step 1) To every gate with an even number of inputs add a "pseudo" input.

Step 2) Add this new input to the gate in such a way that the planar representation of the logic diagram shows a minimal interlace of "pseudo" and real inputs. The vertical order of inputs on the planar logic diagram produces an optimal gate sequence layout. "Pseudo" inputs, except for those at the top or bottom, correspond to separation areas.

The minimization of the separation areas can be performed using a logic diagram, which clearly shows the structure of the series/parallel graph.

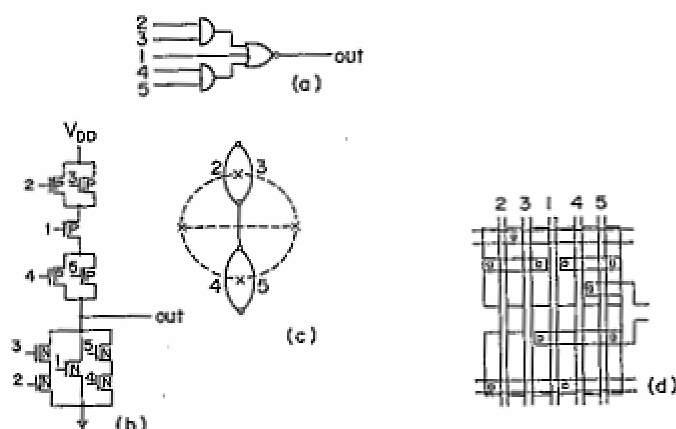


Figure 12. An alternative circuit and optimal layout
(a) Logic diagram (b) Circuit (c) Graph model (d) Layout

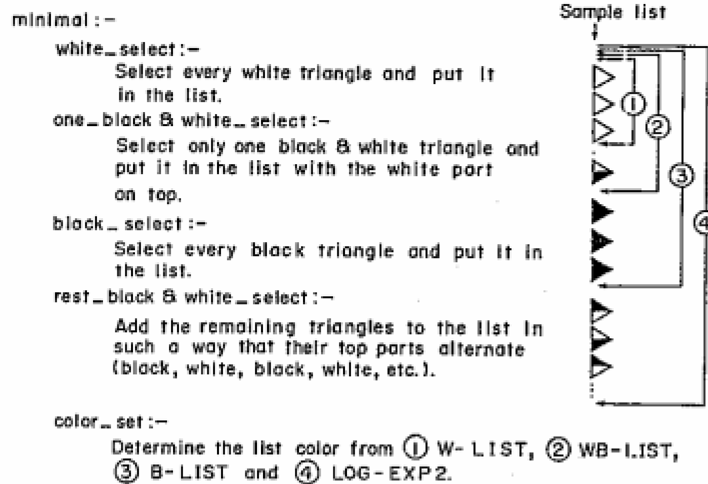


Figure 13. Minimal interlace algorithm

An algorithm for constructing a minimal interlace is implemented in Prolog, as outlined in Figure 13. Figure 14(b) is a conceptual model of the logic diagram shown in Figure 14(a). The black and white triangles correspond to real and "pseudo" inputs, respectively.

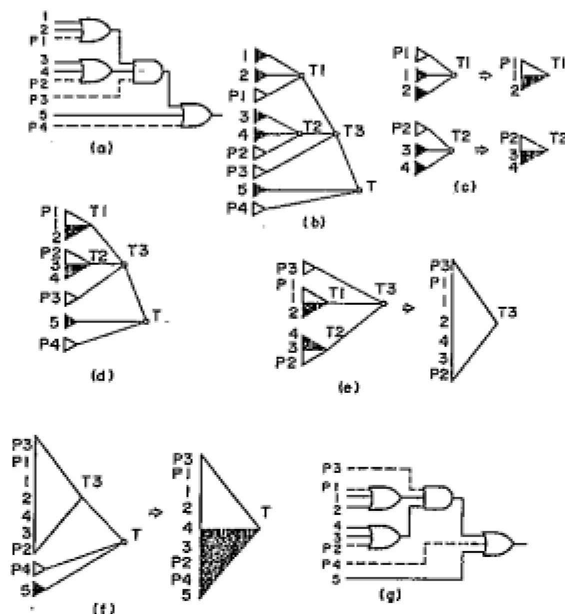


Figure 14. Example of applying the minimal interlace algorithm

Triangles 1, 2 and p1 in subtree T1 are rearranged by the algorithm. The result is represented by a single triangle with a white top and black bottom("white-black"), because the color of the top triangle, p1, is white and the color of the bottom triangle, 2, is black. T2 is similarly represented by a new triangle. A new model is then obtained as illustrated in Figure 14(d). The arrangement of subtree T3 is shown in Figure 14(e). (Note that T3 is represented by a white triangle because the top triangle, p3, is white and so is the bottom of triangle T2.) The final rearrangement of the tree is represented in Figure 14(f). In the end, we obtain a logic diagram with an input sequence characterized by minimal interlacing, as shown in Figure 14(g) [[1,2,4,3],[5]]. This sequence shows the separation between the two sublists.

Part of the minimal interlace algorithm, which is implemented in Prolog, is shown in Figure 15. The goal, minimal(LOG_EXP1, LOG_EXP2, COLOR) means that applying the minimal interlace algorithm to the given logical expression, LOG_EXP1, yields the logical output expression, LOG_EXP2, and its color, COLOR. Rules (2), (3) and (4) are used to choose an element whose color is "white-black". The goal, minimal, in rule (3), checks whether the color of an element is "white-black". If the color is not "white-black", rule (4) is applied, which selects a "white-black" element from among the rest.

We have presented a systematic method of implementing random logic functions using functional cells. Components such as registers, memories, decoders, adders, and I/O pins are assembled from a library of basic cells. Since these cells are of the same height, and have the same power connections and standardized connection points, they can be readily incorporated into existing automated layout systems, as shown in Figure 16.

```
minimal(LOG_EXP1, LOG_EXP2, COLOR):-
      white_select(LOG_EXP1, [] , W_LIST, REST1),
      one_black & white_select(REST1, W_LIST, WB_LIST, REST2),
      black_select(REST2, WB_LIST, B_LIST, REST3),
      rest_black & white_select(REST3, B_LIST, LOG_EXP2), ↑ ,
      color_set(W_LIST, WB_LIST, B_LIST, LOG_EXP2, COLOR).····· (1)
                          x Unify at this point ─────────────┘

   one_black & white_select( [], [], LIST, LIST).·························· (2)
   one_black & white_select([H|T], LIST, NEW_LIST, T):-
         minimal(H, NEW_H, white_black),
         append(LIST, (NEW_H), NEW_LIST).······························ (3)
   one_black & white_select([H|T], LIST, NEW_LIST, [H|TAIL]):-
         one_black & white_select(T, LIST, NEW_LIST, TAIL).··········· (4)
```

Figure 15. Prolog implementation of minimal interlace algorithm

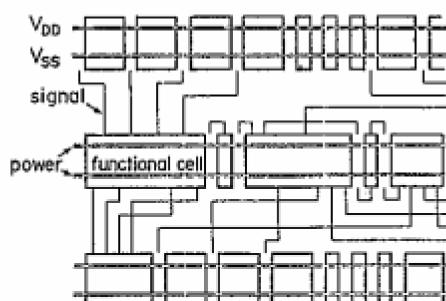


Figure 16. Example of a row-based layout scheme

6. CONCLUSION

Using Prolog, we have implemented a prototype expert system for logic design. From this experience, we have learned that it is possible to construct an intelligent CAD system using a knowledge-based approach. Moreover, Prolog appears to have the ability both to succinctly express algorithms and to effectively represent knowledge.

ACKNOWLEDGEMENTS

This work is based on the results of the R & D activities of the Fifth Generation Computer Systems Project. The authors would like to thank Dr. K. Furukawa of ICOT (Institute for New Generation Computer Technology) for his encouragement and support.

REFERENCES

Brown, H., Tong, C., and Foyster, G. Palladio: An Exploratory Environment for Circuit Design, COMPUTER, Vol.16, No.12, 1983.

Taylor, R. and Wilson, P. OCCAM: Process-oriented language meets demands of distributed processing, Electronics, Nov. 30, 1983.

Dietmeyer, D. L. Logic Design of Digital Systems, Allyn and Bacon, 1971.

Foster, M. J. and Kung, H. T. Design of Special-Purpose VLSI Chips: Example and Opinions, CMU-CS-79-147, 1979.

Thomas, D. E. et al. Automatic Data Path Synthesis, COMPUTER, Vol.16, No.12, 1983.

Uehara, T. and vanCleemput, W. M. Optimal Layout of CMOS Functional Arrays, IEEE Trans. Vol.C-30, No.5, 1981.