ICOT Technical Memorandum: TM-0137

TM-0137

Prologを用いた論理設計支援エキスパートシステム

角田多苗子、丸山文宏、真野長男
林　　司、川戸信明、上原貴夫
(富士通)

October, 1985

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Ｐｒｏｌｏｇを用いた論理設計支援エキスパートシステム

角田　多苗子　　丸山　文宏　　真野　民男　　林　一司　　川戸　信明　　上原　貴夫
富士通株式会社

１．はじめに

ハードウェアを設計する場合，いまや計算機の支援は不可欠であり，ＣＡＤシステムの重要性は高まる一方である．これまでにも実装設計の部分では，配置・配線を行なうＣＡＤシステムなど実用化されているものも多い．しかし，与えられたハードウェアの仕様を，回路素子で実現される物理構造に変換していく論理設計の過程に関しては，設計の一部をサポートするＣＡＤプログラムしか実用化されていない．これは，論理設計者は過去の経験から得た知識を用いて設計を行なっており，従来この過程の支援あるいは自動化は困難であると考えられていたからである．本システム開発の目的は，このような設計者のノウハウを知識ベースのアプローチを採用して計算機に格納し，この有効利用により論理設計過程を計算機で支援することである．特に，従来ほとんど試みられていない，ハードウェアとしての動作を決定する機能設計の部分の支援もねらっている．

本システムは，設計するハードウェアでどのような処理を行なうかという仕様が，プログラミング言語 occamで記述した並列動作アルゴリズムの形で与えられると，ユーザとインタラクションを行ないながら，最終的にターゲットであるＣＭＯＳ回路を設計する．[1][2][3] 現在，有限状態同期マシンの概念に基づいたハードウェアの実現を対象としている．アルゴリズムレベルにおいては，レジスタ，クロックといったハードウェアの概念や使用する半導体テクノロジを意識することなく設計が行われる．本システムによればこの段階から支援を受けることができる．したがって，設計の生産性・信頼性の向上が期待できるばかりでなく，ハードウェア設計の熟練者でなくても，並列アルゴリズムの設計ができれば，このシステムを用いて比較的容易にハードウェアの開発ができると考えられる．

システムは論理型言語 Prolog で実現されている．また，ノウハウや設計データを表現するのにも Prolog の特徴をそのまま用いている．

本報告では，特に設計者のノウハウに大きく依存する機能設計フェイズを中心に，Prolog による知識ベース設計システムの構築について報告する．

２．システム概要

２．１　システム構成

論理設計の過程は，仕様からハードウェアとしての動作を決定する機能設計と，その機能設計に基づいて回路を設計する回路設計の二つの過程からなる．アルゴリズムとして与えられた仕様は，機能設計を行なうことによってハードウェアの概念が導入される．この機能記述は，回路設計を行なうにつれて，まず回路の構造に関する情報に変換される．そして最後にターゲットテクノロジに基づいた物理構造に変換される．

本システムではこのような過程を，図１に示すような構成で支援している．システムへの入力は，プログラミング言語 occam [4] で記述した，ハードウェアの並列動作アルゴリズムである．機能設計フェイズは，設計者の知識を利用し，ユーザと対話しながら，機能設計を行な

う．この結果は，レジスタ・トランスファ・レベルのハードウェア記述言語 DDLによる有限状態マシンとして表現する．回路設計フェイズは，まずトランスレータサブシステム，データパス設計サブシステム，及び制御回路設計サブシステムによって，DDL［5］による機能設計を構造的情報に変換する．このレベルまで，半導体テクノロジには依存しない．そして，基本セル割当サブシステム，関数分割サブシステム，複合セル設計サブシステム，及び回路最適化サブシステムを経て，ターゲットであるＣＭＯＳ回路を設計する．システムの最終出力はＣＭＯＳ基本セル，ＣＭＯＳ複合セル，及びそれらの接続関係である．

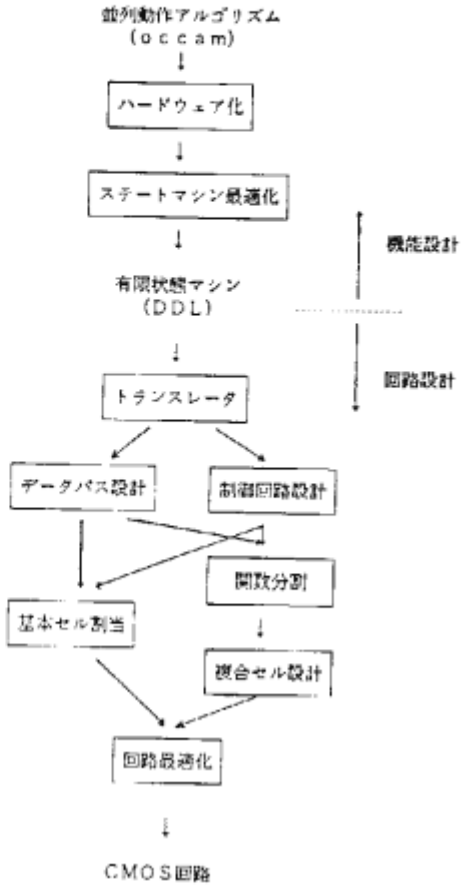

図１　　システム構成

２．２　実行例

このシステムを用いた論理設計の実行例を以下に示す．

設計対象はパターン・マッチャ［６］という，与えられた文字列（パターン）が別の文字列中に含まれているかどうかを検査するハードウェアである．パターン・マッチャは図２に示すように，上段のコンパレータと下段のアキュムレータから構成される．コンパレータはパター

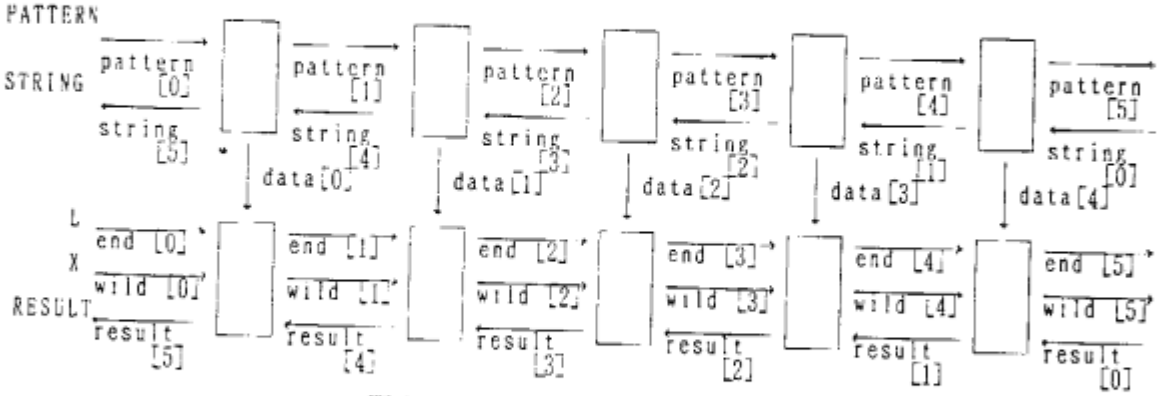

図２　パターン・マッチャ

ンを左側から，検査する文字列を右側から，それぞれ一文字づつ入力して比較し，その結果をアキュムレータに送る．アキュムレータはコンパレータから送られてきた個々の結果を蓄え，パターンの文字が全部比較されたとき，それまでの結果をまとめて，パターンの各文字がすべてマッチしたか否か，つまりパターンが含まれていたかどうかの結果を出力する．

ユーザはまずパターン・マッチャの動作アルゴリズムを occamで記述する（図3），システムを起動すると，機能設計が始まる．図4は機能設計フェイズにおける，ユーザとシステムのインタラクションの様子である．ユーザはシステムが発するいくつかの質問に対して応答を行う．システムはユーザからの答と知識ベースに蓄えられた知識を利用して，DDL で表わされた機能設計を生成する．図5は機能設計フェイズの生成した DDL記述である．

```
CHAN  pattern [6]:
CHAN  string [6]:
CHAN  data [5]:
CHAN  end [6]:
CHAN  wild [6]:
CHAN  result [6]:
PROC  comp (CHAN pin, sin, pout, sout, dout)=
  VAR  p, s:
  SEQ
    PAR
      p := 0
      s := 0
    WHILE  TRUE
      SEQ
        PAR
          pout ! p
          sout ! s
        PAR
          pin ? p
          sin ? s
        dout ! p = s:
PROC  acc (CHAN xin, lin, rin, din, xout, lout, rout)=
  VAR  d, x, l, r, t:
  SEQ
    PAR
      x := FALSE
      l := FALSE
      r := FALSE
      t := TRUE
    WHILE  TRUE
      SEQ
        PAR
          xout ! x
          lout ! l
          rout ! r
        PAR
          din ? d
          xin ? x
          lin ? l
          rin ? r
        IF
          l = TRUE
            SEQ
              r := t
              t := TRUE
          l = FALSE
            t := t /\ (x\/d):
PAR i =  [ 1 FOR 5 ]
  PAR
    comp (pattern[i-1], string[5-i],
          pattern[i], string[6-i], data[i-1])
    acc( wild[i-1], end[i-1], result[5-i],
         data[i-1], wild[i], end[i], result[6-i])
```

図3　パターン・マッチャの動作アルゴリズム（occam）

```
| ?- functional_design.
Parsing your specifications in occam ...

Implementing occam variables ...

Should variable x have only one bit? y/n
|: y

Should variable l have only one bit? y/n
|: y

Should variable r have only one bit? y/n
|: y

How many bits should variable s have?
|: 8

Should variable p have as many bits as variable s? y/n
|: y

Compressing a sequence of operations ...

Implementing inter-process communication ...

Can the entire system be controlled by a single clock? y/n
|: y

Would you prefer faster overall communication? y/n
|: y

Optimizing ...

Can pout always carry data? y/n
|: y

Can sout always carry data? y/n
|: y

Generating partial DDL descriptions ...

Constructing the final DDL code from partial DDL descriptions ...

yes
| ?-
```

図4　機能設計フェイズにおけるインタラクション

```
<system> pm.
  <time> clk.
  <entrance> pin(8), sin(8), xin, lin, rin, din.
  <exit> pout(8), sout(8), dout, xout, lout, rout.
  <terminal> send1.
  <automaton> comp: clk:
    <register> p(8), s(8).
    <states>
      init: p ← 0. s ← 0. → idle.
      idle: pout = p, sout = s,
            p ← pin, s ← sin, → state2.
      state2: send1 = 1, dout = (P:=s), → idle.
    <end>.
  <end> comp.
  <automaton> acc: clk:
    <register> d, x, l, r, t.
    <states>
      init: x ← 0. l ← 0. t ← 0. → idle.
      idle: send1: xout = x, lout = l, rout = r,
            x ← xin, l ← lin, r ← rin,
            d ← din, → state1.
      state1: |* l *| r ← t, t ← 1
                    ; t ← (t&(x|d)).. → idle.
    <end>.
  <end> acc.
<end> pm.
```

図5　DDL による機能設計

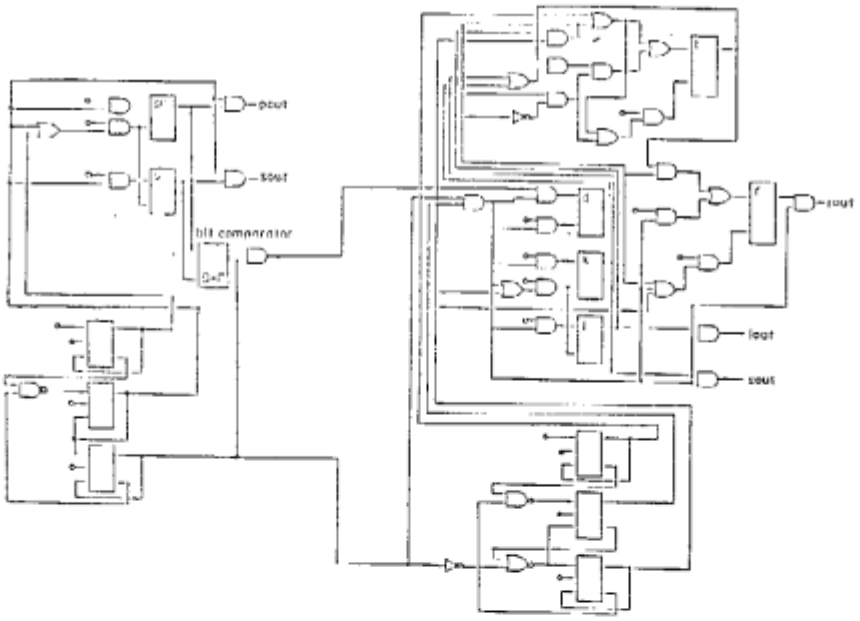

図6　　論理回路図

この記述は回路設計フェイズに入力され，カウンタ，加算器，レジスタ等の機能ブロックとそのまわりの組合せ回路が設計され，図6で示される論理回路図を得る．そして，機能ブロックは基本セルあるいはその組合せで実現される．また，組合せ回路は部分回路に分割され，個々に対して図7に示すような最適なCMOS複合セルとして実現される．

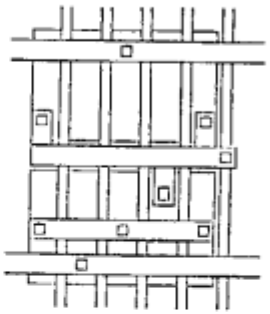

図7　　ＣＭＯＳ複合セル

3．推論メカニズムと知識表現形式

動作アルゴリズムからハードウェアの概念を導入して機能設計を行う過程は，従来設計者の頭の中で行われており，明白なアルゴリズムは存在せず，設計者のノウハウを引き出して，それらをうまく利用する必要がある．以下，機能設計フェイズに注目して，その推論メカニズムや知識表現法などについて述べる．

3．1　設計システムの性質

知識ベース設計システムを構築する際，設計という対象領域の性質に基づく，いくつかの考慮すべき点がある．

論理回路の設計は，基本的には与えられた仕様を徐々に詳細化していく過程であると考えられるが，ゲート数，性能等が満足できない場合は適切な個所に戻り，設計の修正を行う必要も生じる．本システムは，現在この詳細化の過程を支援している．設計者は全体の仕様を眺めながら部分部分に対して，どう設計するかという決定を下し，そのたびに設計状態が変更されていく．ある部分の設計は別の部分にも影響を与え，どの決定を最初に下すかによって設計の手間や質が大幅に変ることもある．また，設計における決定は複数個の対象間の関係を参照しながら進めていくので，知識表現のフレームワークは，対象間の関係を自然に表現できるものである必要がある．

3．2　推論メカニズム

基本的な推論メカニズムとして，前向き推論を用いる．前向き推論により各種のルールが初期の仕様から設計が完成されるまで，順次適用されていく．詳細化していく過程は，副作用を起こして環境を変化させることにより実現する．前向き推論では，ルールの前提部の条件がすべて満たされると結論部が起動され，ワーキング・メモリを更新する．この結果の新しいワーキング・メモリを参照しながら，新たなルールを起動する．このようにして処理が進む．

このような前向き推論が基本となって設計が行なわれるが，前向き推論の前提部の条件のチェックには，後向き推論メカニズムを使う．条件の中には，ワーキング・メモリの参照だけですむものもあれば，何段階かの推論が必要なものもある．この場合，結論部がその条件と一致する後向き推論のルールを見出し，今度はその

前提部の各条件のチェックに帰着させ，ワーキング・メモリの内容に行きつくまで繰り返すことで，条件のチェックの効率化を図る．

3．3　知識の表現形式

設計作業は対象間の関係を参照し利用しながら進められるので，知識の多くは対象間の関係を参照するものである．したがって，論理型言語の，述語がその引数間の関係を表わすという性質は，設計作業の知識を表わすのに適しており，また，節が宣言的に解釈できるのでルールや設計情報が自然に表現できる．我々はこれを，論理型言語による"関係指向パラダイム"と呼んでいる．本システムでは，前向き推論・後向き推論のルール，設計情報をすべて Prolog で表現する方針をとっている．

図8，図9に前向き推論のルール，後向き推論のルールの一般形を示す．

前向き推論では，前提部の条件1，・・・，条件nが満たされると，結論部であるアクション1，・・・，アクションmが起動され，assert，あるいは retract によって副作用をおこしてワーキング・メモリを更新する．条件のチェックに用いられる後向き推論では head が結論部，body が前提部を表わし，条件1，・・・，条件nが満たされたならば条件0が満たされることになる．後向き推論のルールが起動されても，ワーキング・メモリは更新されない．設計が進むにつれて，ワーキング・メモリ内の設計情報が充実していき，最終的にハードウェアの機能設計が生成される．

ワーキング・メモリに書き込まれる設計情報は，Prolog の fact の形で表され，その時点での環境を表している．設計情報を表すために約40個の述語が用意されているが，その述語名は設計者が通常用いる表現に近いものであるように選んでいる．

4．　知識の例

機能設計フェイズは，ソフトウェアとして記述された仕様にハードウェアの概念を導入していく過程であるので，ここで用いられる知識は，ソフトウェアとハードウェアを対応づけるものが中心となっている．知識には，以下に述べるような種類がある．

① occam に現れる変数をハードウェアで実現するための知識

② occam のオペレーションをハードウェアの並列性を利用できるように圧縮するための知識

③ occam に現れるチャネルをハードウェアで実現するための知識

④ 各オートマトンに対してステートの特徴付け（初期ステート，アイドルステート等）を行う知識

⑤ ステートマシンの最適化を行うための知識

```
（処理を規定する head）　:－
    （条件1），    ┐
       ・          │
       ・          ├ 前提部
       ・          │
    （条件n），    ┘
    （アクション1），┐
       ・           │
       ・           ├ 結論部
       ・           │
    （アクションm）．┘
```

図8　前向き推論のルールの一般形

```
（条件0）　:－            結論部
    （条件1），  ┐
       ・        │
       ・        ├ 前提部
       ・        │
    （条件n）．  ┘
```

図9　後向き推論のルールの一般形

図１０に示す知識は，occam 記述に現れた変数を具体的なハードウェアに変換するための知識である．ある occam変数 Var を実現する時，Var へチャネルを通じて入力されるソース(input_source)がすべて真理値(truth_value) で，Var へ代入されるソース(assigned_source) がすべて真理値で，その他の条件が満たされるならば，Var は１ビットレジスタで実現する，ことを示す前向き推論のルールである．最終行に現れる述語 implementation が，設計情報を表す述語の一つである．

```
implement_variable(Var,_) :-
   input_source(Var,Input_sources),
   truth_value(Input_sources),
   assigned_source(Var,Assigned_sources),
   truth_value(Assigned_sources),
                .
                .
                .
   assert(implementation(Var,1,
                register,･･･)).
```

図１０　変数実現のルール
（前向き推論ルール）

このルールは，パターン・マッチャの例では変数 t を実現するときに用いられている．

図１１に示すのは，ソフトウェアで逐次的に記述されたオペレーションがハードウェアでは並列に実行できることを表わすルールである．ハードウェアの最大の特徴は，その並列動作にあるので，ソフトウェアで表わされた仕様から，できるだけ並列性を引き出すことが望ましい．compatible は二つのオペレーション Operation1, Operation2 が，ともに代入文であり(store_operation)，Operation1 の直後に Operation2 が実行され(followd_by)，それぞれの変数が異なり (Var1\==Var2)，両変数ともレジスタで実現されることになっており (implementation)，直前のオペレーション Operation1 の変数がその直後のオペレーション Operation2 のソースに現れていなければ，これらは同時に実行可能である，ということを表わしている．

```
compatible(Operation1,Operation2) :-
   store_operation(Operation1,Var1,
                assign,Source1,･･･),
   store_operation(Operation2,Var2,
                assign,Source2,･･･),
   followd_by(Operation1,Operation2),
   Var1 \== Var2,
   implementation(Var1,_,register,･･･),
   implementation(Var2,_,register,･･･),
   not(referred_to(Var1,Source2)).
```

図１１　並列性をチェックするルール
（後向き推論のルール）

このルールを適用すると，occam 中の逐次的オペレーション（図３の点線部分）を DOLでは同一サイクルで実行できる（図５の点線部分）ことになる．

図１２に示す知識も，変数を具体的なハードウェアに変換するための前向き推論のルールであるが，ルールの中で直接に推論の順序を制御している．

この implement_variable は，ある変数を実現したいのだが，ビット幅に関する手掛りがない場合，似たような変数を探し，その変数を先に実現して，その結果を利用しようというルールである．後回しにした変数はリスト (Task_list)にして先に実現するときの引数として引き渡し，ループに陥ることを防いでいる．

```
implement_variable(Var,Task_list) :-
   looks_similar(Var,Another_var),
   not(member(Another_var,Task_list)),
   implement_variable(Another_var,
                [ Var|Task_list] ),
   implementation(Another_var,
                Bit_width,･･･),
                .
                .
                .
   assert(implementation(Var,Bit_width,
                register,･･･)).
```

図１２　局所的に推論の順序を制御するルール

似たような変数 (looks_similar)とは，ある変数を別の変数に代入していたり，二つの変数を比較するプロセスが存在する場合，これらの変数のことを言う．パターン・マッチャでは PROC compの中に現れる変数ｐとｓがこれに該当する．図４のインタラクションに示すようにまずシステムは変数ｓを実現しようとするが，

ビット幅に関する手掛りがないので，似たよう

な変数ｓを先に実現しようとする．この場合，ｓに関してもビット幅に関する情報がないのでユーザに質問してくるが，ｓが実現されるとｐに関しては，ｓと同じでよいかどうかの確認を行なうだけである．

5．まとめ

本論文では，Prolog を用いた論理設計支援エキスパートシステムについて，特に機能設計フェイズにおける推論メカニズム，知識などを中心に報告した．Prolog に代表される論理型言語は，"関係指向パラダイム"を提供するため，対象間の関係が表現しやすく，宣言的な読み方ができるため，知識を自然に表現できる．また，本システムは Prolog の実行メカニズムをそのまま利用して開発したが，ルールと手続きの表現形式が同じなので，アルゴリズミックな処理とルールを統一して扱えるなど，論理設計支援エキスパートシステムの構築に Prolog が有用であることが確認できた．

しかし，問題点として以下の点が挙げられる．まず，現在のシステムでは，ワーキング・メモリに書き込まれる設計情報は，Prolog の fact であり，その内容の更新には assert，retract を繰り返し用いねばならない．またこれらの操作手続きがすべてルールの中に現れてしまう．このためオブジェクト指向の概念を導入してワーキング・メモリ操作手続きとルールを分離することを考えている．

次に，現在のシステムではルールインタプリタを用いていないため，局所的推論の制御は可能であるがグローバルな推論の制御ができない．これに関しては，メタルールの導入などが考えられる．

設計の質の向上は，知識の量と質に大きく依存する．また，設計者の使っている設計に必要なさまざまな概念を引出し，それらに基づいてシステムを構築しなければならない．これらの概念を獲得し，知識ベースを充実させるために，知識獲得の研究が重要である．

謝辞

本研究は，第５世代コンピュータプロジェクトの一環として行なわれたものであり，御支援頂いたＩＣＯＴ第一研究室古川室長に深く感謝します．

［参考文献］

［１］Maruyama, F., Mano, T., Hayashi, K., Kakuda, T., Kawato, N. and Uehara, T., "Prolog-Based Expert System for Logic Design," International Conference on Fifth Generation Computer Systems 1984(FGCS'84), pp. 563-571, Nov. 1984

［２］Mano, T., Maruyama, F., Hayashi, K., Kakuda, T., Kawato, N. and Uehara, T., "OCCAM to CMOS Experimental Logic Design Support System," 7th Computer Hardware Description Languages and their Applications(CHDL85), pp. 381-390, Aug. 1985.

［３］林　一司，丸山　文宏，真野　民男，角田多苗子，川戸　信明，上原　貴夫：「論理型言語による論理設計支援システム」，設計自動化研究会　EC-85-28

［４］Taylor, R. and Willson, P., "OCCAM : Process-oriented language meets demands of distributed processing", Electronics, Nov. 30, 1983

［５］Dietmeyer, D.L., Logic Design of Digital System, Allyn and Bacon, 1971

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