(31)

EC82－43

推論向き高並列計算機システムの基本アーキテクチャ

後藤厚宏・相田　仁
田中英彦・元岡　達
（東　大）

1982年10月12日

社団法人 電子通信学会

EC82－43

推論向き高並列計算機システムの基本アーキテクチャ

THE BASIC ARCHITECTURE OF HIGHLY-PARALLEL PROCESSING SYSTEM FOR INFERENCE

後藤　厚宏 ，　相田　仁 ，　田中　英彦 ，　元岡　達

A. GOTO,　H. AIDA,　H. TANAKA,　T. MOTO-OKA

東京大学　　工学部

FACULTY OF ENGINEERING , UNIVERSITY OF TOKYO

1．　はじめに

現在、計算機システムの応用分野は、その問題の解決手順があらかじめ明確なものが主であり、計算機システムの開発目標は、与えられた問題を与えられた解決手順に従って最小時間で解くことである。現在盛んに研究が進められている関数型言語およびデータフローマシンは、問題解決の手順に余分な制限を持ち込まずに自然な形のまま処理を行なうことが最大の特徴となっている。

一方、比較的近い将来の計算機システムは知識情報処理の分野に適応する必要があることが指摘されている。知識情報処理では、与えられた問題と知識から問題解決の手順自体を生成しながら問題解決を行なう推論の機能が中心となり、これによって、問題解決の手順が不明確な分野まで計算機システムの応用分野を拡大することが可能となる。

Prolog は一階述語論理に基づいた言語であり、あらかじめ与えられた定義節（知識）と、入力されたゴール（ goal ）節との統合化（ unification：ユニフィケーション ）を繰り返し行なうことによって、問題解決を行なう言語システムである。

現在のProlog は実用面における問題も多く、このままの形で将来の知識システムの核となることは難しいと思われる。しかし、述語論理に基づいていることによる言語上の素質の良さ、及び宣言的（ declarative ），非決定的（ non-deterministic ），非数値処理向きという特徴は、知識システムの初期モデルとして大きな可能性を持つと考えられる。

そこで我々は、Prolog をベースとした推論向きシステムの開発をスタートポイントとし、将来の計算機システム像を考えることにした。

我々が想定する推論向きシステムの全体像を、図１に示す。

ＩＴＰ（Ｉntelligent　Terminal Processor）は、プログラマに対して高度なプログラミング環境を提供する。また、ＣＩＭ（Central　Inference　Machine）は、自然言語処理、自動設計等の高負荷な応用のための処理システムであり、複数のユーザに共有される。

現在、Prolog に関して、①応用との親和性、②プログラミング環境を含めた言語機能の拡充、　③Prolog 向き計算機アーキテクチャ、等の面から研究を進めている。①②の点についての考察は別の機会に譲ることとし、本稿では、Prolog （主に pure Prolog ）の実行モデルを示した後、Prolog の並列処理性に着目したＣＩＭ向きの計算機アーキテクチャについて検討する。

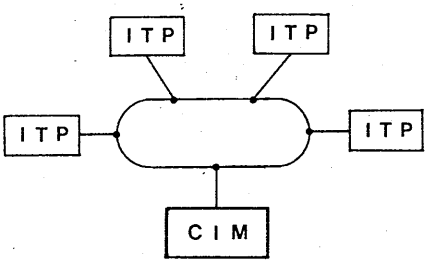

図１　推論向きシステムの全体像

２．　Prolog の実行モデル

（１）　探索木（Search Tree ）

Prolog の実行は、与えられた初期ゴール節と定義節との間でユニフィケーションを実行し、新たなゴール節を生成する input resolution である。初期ゴール節を出発点として次々と生成される全てのゴール節は、初期ゴール節を root node とする tree 状に配置でき、これは“探索木（Search Tree ）”と呼ばれる。探索木の形はユニフィケーションを行うゴール・リテラルの選択の仕方によって異なる。例えば ex.1 のプログラムを、通常の処理系のように左側のゴール・リテラルから実行した場合は、図２のようになる。

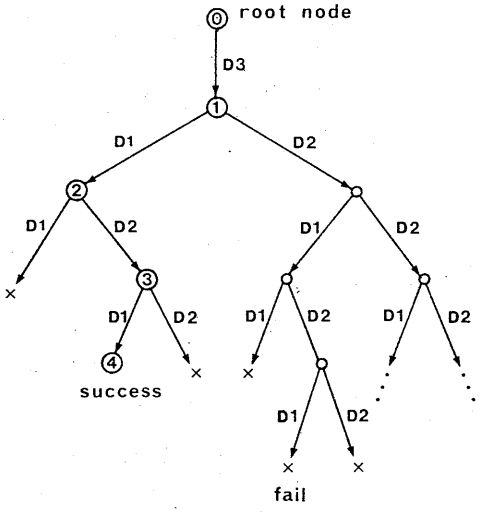

図２　ex. 1 を左から実行した時の探索木

注１）　←の横に、実行に用いた定義節名を示す
注２）　各 teminal node は root node からそこまでの derivation path が success／ failure であったことを示す

次に、探索木における各 node において root node からその node までの処理の中間結果を表現するデータ構造について考える。ここでは、Prolog プログラムの実行を表現する方法として、Proof Diagram を導入する。Proof Diagram は、Ferguson Diagram［１］に類似しているが、表現力においてより優れていると考える。

（２）　Proof Diagram

Proof Diagramでは ex.1 における各定義節Ｄ１～Ｄ３を図３のような定義節テンプレートで表現する。

ここで、定義節中の各リテラルは半円によって示される。ゴール・リテラルつまり手続き呼び出しは上半円（ call ）に対応し、頭部リテラル（ head literal ）は下半円（ head ）に対応する。両半円は、述語名とその引数に対応したポート（ argument-port：小さい円）を持つ。変数はアークで表現され、functor, atom または argument-port 間を結ぶ。

初期ゴール節は上半円つまり手続き呼び出しのみからなるテンプレートによって図４のように表わされる。

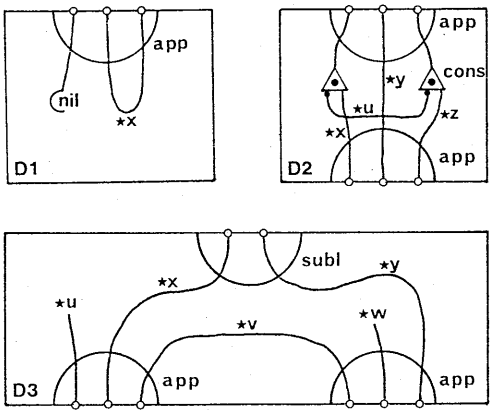

図３　定義節テンプレート

［ex. 1］

［D１］　append (nil, ＊x, ＊x) ←.
［D２］　append ((＊u ・ ＊x), ＊y, (＊u ・ ＊z)) ← append (＊x, ＊y, ＊z).
［D３］　sublist (＊x, ＊y) ← append (＊u, ＊x, ＊v) append (＊v, ＊w, ＊y).
［G］　← sublist ((a ・ ＊x), (＊y ・ nil)).

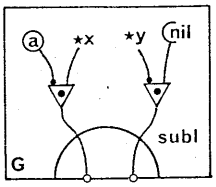

図４　初期ゴールのテンプレート

（３）　ゴールフレームとプログラムの実行

次にProof　Diagramを用いてプログラム（ ex.1 ）の実行を表現する。

探索木の各 node において、　root　node　からその node までの中間結果は“中間ゴール節とそれに付随したユニフィケーションの環境”であり、以降はこれを指して“ゴールフレーム（ goal-flame ）”と呼ぶことにする。ゴールフレームは、Proof　Diagram　においてテンプレートの組み合わせとして図式的に表現できる。

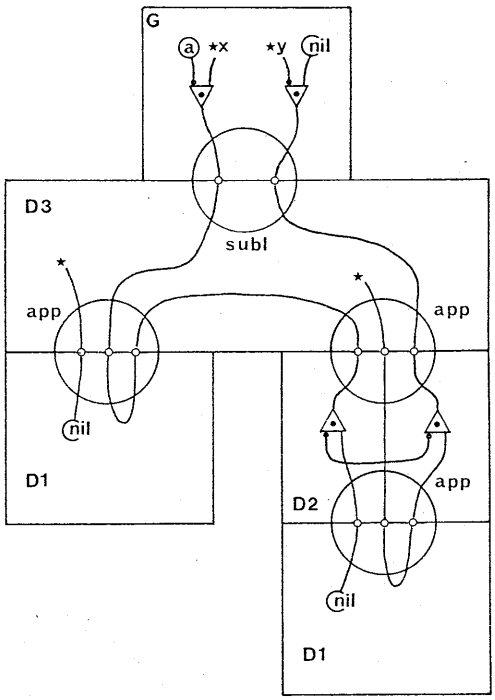

図５　success node　④における完了フレーム

初期ゴール節はひとつのゴールフレームであり、ユニフィケーションを必要とする手続き呼び出し“ sublist”（ゴールフレーム内の手続き呼び出しを active call と呼ぶ）を有する。ここでユニフィケーションは、ゴールフレーム内の active call　に対応する適当な定義節テンプレートをそのゴールフレームに組み合わせることに相当する。　例えば、図２において root node　から success node ④に到る実行の結果は図５のように表現される。ここでは、成功したユニフィケーションは円で表わされ、ゴールフレーム全体は矛盾がないものでなければならない。

success node におけるゴールフレーム（完了フレーム： complete flame と呼ぶ）は、その問題を解決するひとつの手順を示している。Prolog プログラムの実行は、初期ゴールフレームに対して定義節テンプレートを組み合わせ（ユニフィケーション）、active call （上半円）のない完了フレーム（図５）を生成することである。

他の言語との大きな違いは、“ゴールフレームに表現された問題解決の手順が決定的に得られるのではなく、それぞれの手続き適用（定義節テンプレートの組み合わせ）が非決定的であり、ゴールフレームを試行錯誤的に多数生成しながら問題を解く”ところにある。実行途中で生成されるゴールフレームの大部分は失敗に終り、成功するものも複数あり得る。

３．　並列処理方式の検討

（１）　Prolog の並列性

前述のように、Prolog プログラムの実行はテンプレートの組み合わせとしてとらえることができ、以下のような並列性が内在している。

〈Ｐ１〉　問題解決の手順（ゴールフレーム）の生成における並列性－ＯＲ並列性

〈Ｐ２〉　ゴールフレーム内の複数 active call間の並列性－ＡＮＤ並列性

〈Ｐ３〉　ユニフィケーションにおける引数間の並列性

〈Ｐ１〉は、定義節の適用における非決定性に対応するものであり、既に充分の並列性が抽出できることが判っている［４］。〈Ｐ２〉は、複数の active call が並列に unify できる可能性を持つことに基づくものである。ただし、並列に求めた解集合に対してＡＮＤ操作（通常Consistency Check と呼ぶ）を施す必要があり、実現は難しい。〈Ｐ３〉は、最大並列度が引数の数に限られる。しかしユニフィケーション操作は単なるパターン・マッチングではなく以降で述べるように負荷の重い操作である。そのためユニフィケーション向きハードウェアによって少ない並列性を生かすことが重要である。

（２） ゴールフレームの縮退とＯＲ並列

従来の逐次形Prolog システムは、ゴールフレームの生成におけるやり直し（ backtrack ）を効率的に行なうためにスタックを利用してきた。この場合、ゴールフレームは“テンプレートの組み合わせかた”としてスタック上に記憶される。

逆に、やり直しがなければゴールフレームを縮退させることが可能となる。例えば、図５のゴールフレームを縮退させながら求めると図６のようになる。注３） 完全なＯＲ並列処理〈Ｐ１〉では手続きの選択枝のすべてが並列に適用されるためゴールフレームの縮退が可能であり、縮退の結果、ゴールフレームのデータ量は大幅に減少する。これはＯＲ並列処理におけるコピーのオーバヘッドがそれほど大きくないことを意味しており、ＯＲ並列処理が有望であることがわかる。

また、ＯＲ並列処理におけるユニフィケーションは、パターン・マッチングによるテンプレート同志の組み合わせの操作と生成されたゴールフレームの縮退の操作を合わせたものであると言える。

注３） 手続きの適用自体が決定的である場合も縮退できる。

ａ．手続き定義がたたひとつの場合（静的決定性）

ｂ．ユニフィケーションが成功する選択枝がただひとつ（動的決定性）

次節以降において検討するＣＩＭ向きアーキテクチャは、マクロ並列性として〈Ｐ１〉を活かすものである。〈Ｐ２〉については、〈Ｐ１〉がすでに十分な並列性を持つと考えるため現時点での検討は見送る。またミクロ並列性〈Ｐ３〉については〈Ｐ１〉の要素として取り入れることを目標に現在検討中である。

４． ＯＲ並列とその制御

（１） ＯＲ並列処理の問題点

ＯＲ並列は、ゴールフレームの生成における並列性であり、手続き定義の選択枝全てを ゴールフレームに適用する。本節では、ＯＲ並列を実現するために必要な制御方式について検討する。以下の議論では、図７に示すプロセッサ（ＵＰ：Unify Processor）を想定する。

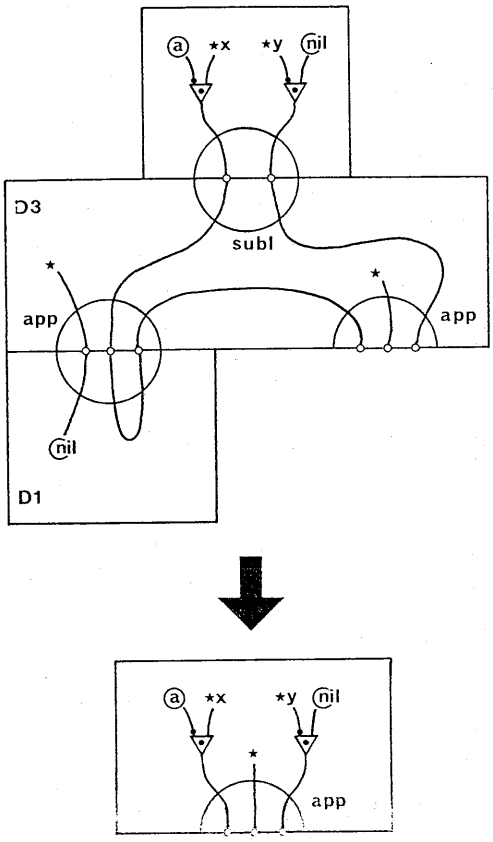

図６ ゴールフレームの縮退

UPは、あるゴールフレーム（GF）を入力すると、その中の active call のひとつについて対応するすべての定義節とのユニフィケーションを実行し、新たなゴールフレームを（複数）出力するものである。マシンは、多数台のUPと、 ゴールフレームのプール（ゴール・プール： goal pool ）を持つ。各UPは、プール内のゴールフレームを取出し、新たなゴールフレームをプールに戻すという動作を繰り返して処理を進める。（図8）

OR並列処理における並列度は、探索木の横幅に相当し、通常十分大きい。但し、実行中のある時点におけるゴールフレームのすべてが成功に到る（すなわち有効である）わけではなく、多くのものは失敗の確認に終る。

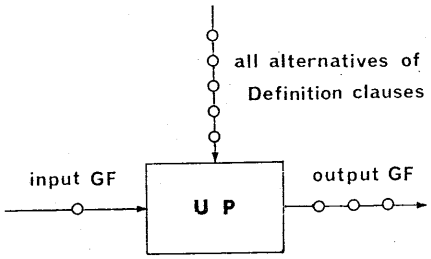

図7 Unify Processorの動き

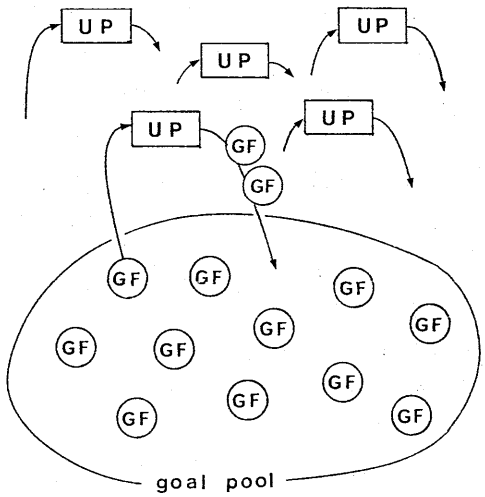

図8 OR並列処理イメージ

結局、実現上の問題は、“有限資源のもとで、より早く解を求めるために、いかにして、有効な並列性を抽出するか” にある。この問題に対処する方法としては次の3つが考えられる。

① 解が求まりやすいゴールフレームから優先してプロセッサーを割り当てる。

② 解を持たないことがあらかじめ判定可能なゴールフレームを早く見つけ出し、消滅させる。

③ 数の爆発が起きないように、ゴールフレーム内では決定性の高い active call を優先する。

以下、これらの点について検討する。

（2） ゴールフレームへのプロセッサの割り当て

本項では、「プロセッサ（UP）数＜ゴールフレーム数」という環境における、プロセッサの割り当てストラテジィ（ strategy ）について検討する。以下ゴールフレームの管理を行なうモジュールをアクティビィティ・コントローラと呼ぶことにする。

A. root node からの深さに基づくストラテジィ

深いものを優先した場合は depth-first処理（図9）、浅いものを優先した場合は breadth-first処理となる。

一般的に、解がひとつでよい時は depth-firstが、全部の解を求める時は breadth-firstが適する。ただし例外もあり得る。また breadth-first 処理は、適当なステップ数毎に先に伸びた枝を一時停止させる方法に変更することにより、暴走枝の回避に用いることができる。（図10）

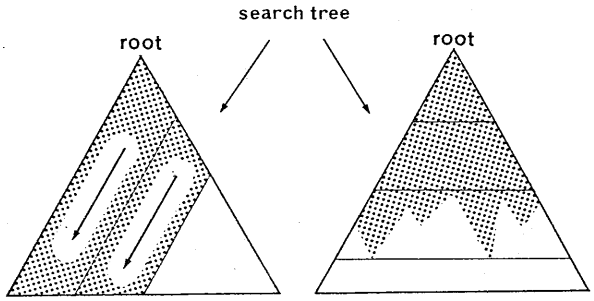

図9 Depth-first処理　　図10 暴走枝の回避

システムが動的に両者を使いわけることができればよいが、実現は難しい。このため、このような探索ストラテジィはプログラマによるメタな制御に依存することになると思われる。

B．決着の付き易いゴールフレームの優先

決着の付き易いゴールフレームを優先して処理することは

・早く最初の解を求める

・資源を解放する

という点から望ましい。

ある実行ステップにおいて、図１１に示すＧＦ１というゴールフレーム（Ａ～Ｄは active calls ）があったとする。 active call の実行順序として“左から”という規則がある時、ＧＦ１からはＧＦ２，ＧＦ３のようなゴールフレームの子孫が派生すると考えられる。この場合、ＧＦ１で導入された active call の生き残りの数を比較するとＧＦ２の方が少ない。このためＧＦ２の方がＧＦ３に比べて success／failure の決着が付き易いと考え、優先して処理することができる。この手法は、各 active call に、それを導入したユニフィケーションのステップ番号を付けることによって実現できる。しかし後述する active call の選択順序を最適化する手法と共に用いることは難しい。

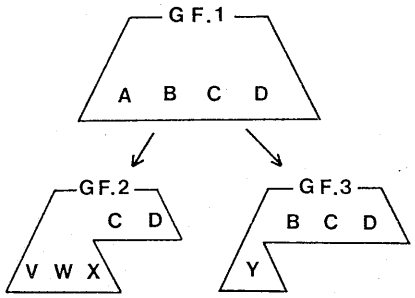

図１１　ゴールフレームの優先度の例

（３）　不要なゴールフレームの消去

ゴール・プール内のゴールフレームはユニフィケーションの実行なしに消去可能な場合がある。

ひとつは、求める解がただひとつでよい場合である。あるゴールフレームの実行が成功すると、その結果として他の選択枝の実行が不要になる。不要ゴールフレームの消去は初期ゴールの名前を各ゴールフレームに付けることによって比較的容易に実現できる。

もうひとつは失敗経験の利用によるものであり、逐次形システムにおける知的後戻り（ Intelligent Backtrack ）に相当するものである。知的後戻りは、失敗原因の分析により失敗が運命付けられている選択枝の実行を避けるものであり、並列処理への適用が望まれるものである。

知的後戻りをＯＲ並列の環境下に適用すると次のようになる。あるプロセッサ（ＵＰ）において全ての選択枝に対するユニフィケーションが直接的／間接的に失敗した時、プロセッサは失敗の原因を調べる。次に失敗したゴールフレームを祖先に溯り、失敗の原因が最後に揃ったゴールフレーム（ destined ＧＦ）を得る。プロセッサは destined ＧＦから派生したゴールフレームの消去をActivity Controller に依頼する。ただし逐次処理の場合と異なり、消去されるゴールフレームは他のプロセッサによって実行が進んでおり、既に失敗が確認されている可能性もある。

このような失敗経験の利用には、失敗原因の分析のためにどの時点において active call 及び変数の代入が導入されたかの記録が必要である。更に、多数のゴールフレームの中から destined ＧＦおよびその子孫のゴールフレームを見つけ出すためには、縮退させない形でゴールフレームを保存するか、またはゴールフレームに対して導出経過が連想できる名前付けが必要となる。

ＯＲ並列処理の環境下において失敗経験を利用する場合は、その効果と上述のようなオーバヘッドとのトレードオフが問題となる。

（4） ゴールフレーム内の active call の選択

Pure Prolog に限った場合、AND関係にあるゴールフレーム内の active call には順序関係がない。つまり各 call の選択順序を替えても完了ゴールフレームの生成に必要なユニフィケーションのステップ数は同じである。しかし、OR並列の環境では実行途中におけるゴールフレームの数が異なる。（すなわち探索木の形状が異なる。）有限資源の環境において資源の節約を計る為には、枝が大きく拡がり過ぎないようにゴールフレーム内の active call の選択を最適化することが有効である。

このような active call の選択の最適化は逐次形システムにおける sidetracking としてPereira らによって検討されている［２］。

最適化の効果は、問題によっては知的後戻りと同程度の効果があると考えられる。また、前項で述べたようにOR並列処理の環境下での失敗経験の利用には難点が多いため、本マシンでは効率的な選択法の検討に重点を置くことにする。

A． 手続き呼び出しの決定性

ユニフィケーションの結果、出力されるゴールフレームの数が１または０（ failure ）である call を決定的な call 、複数のゴールフレームを出力するものを非決定的な call と呼ぶ。 プロセッサが入力したゴールフレーム内に決定的な call と非決定的な call とがある場合、前者を優先して実行することにより失敗の認識が早まり、不要となるゴールフレームの発生も少なくできる。

プログラム中の定義節の中には ex.1 の“sublist ”のように静的な決定性（注３）を有するものがある。当然このような call は優先されるが、割合も少ないため全体的にみて効果は小さい。

爆発的なゴールフレームの発生を積極的に制限するためには、引数の instantiate の状態に従った動的な決定性の判定が必要になる。そこで引数の instantiate の状態から各 call の優先度の評価を行なうことを考える。

B． 引数の instantiation と優先度

最初に次のふたつの場合について考える。

① 頭部パターンによる決定性の検出

「 ex.1 に用いた手続き append は、第一引数が instantiate されている時その適用が決定的になる。つまり第一引数が nilであればＤ１が、 cons であればＤ２が決定的に選択される。」

② call の導入状況による優先付け

「 ex.2 における grand-parent において、

← grand-parent （A，＊z）

という呼び出し時にはａを、

← grand-parent （＊x，B）

という呼び出し時にはｂを先に実行するとゴールフレームの絞り込みが早くなる。」

一般的に見て、各手続きは引数が instantiate された状態で呼び出されると決定的またはそれに近い動作をすると考えてよい。

［ex．２］

```
grand-parent (＊x, ＊z) ← parent (＊x, ＊y), parent (＊y, ＊z).
                          ……a……        ……b……
parent (A, B) ←.
parent (A, C) ←.
    ……
```

①は call 中のある引数が instantiateされたことにより、手続きの実行が決定的になる例である。手続き定義の中には、呼び出し側の引数状況にかかわらず本質的に非決定性を有するものもあるが、多くは呼び出し時の引数パターンによる場合分けによって手続き定義が行なわれている。実行が決定的になる可能性を持つ手続き定義については、入力時にその条件を抽出し、実行時にその条件を判定することにより、その手続き実行を積極的に行なうべきかどうかの判断を下すことが可能となる。

②は定義のボディ部にある call がゴールフレーム中に導入された時の状況によって call 間の相対的な優先度を判断するものである。これについても定義式の入力時に頭部からボディ部への引数受け渡しを行なう変数に着目することによって条件を設定することができる。

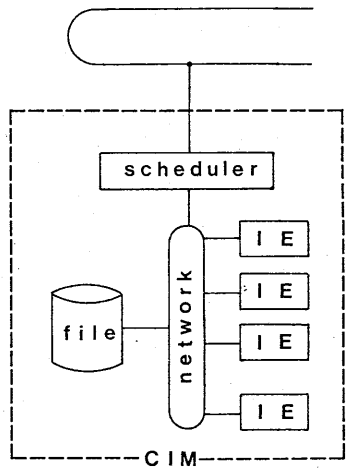

図１２　ＣＩＭの全体像

５．　ハードウェア構造の概要

（１）　全体像

ＣＩＭ向き高並列マシンの全体像を図１２に示す。

ＩＥ（Ｉnference　Ｅngine ／Ｅlement）は次項で述べる基本処理ユニットである。応用プログラムはスケジューラによって分割され、各ＩＥに割りつけられる。各ＩＥは、それぞれ独立した構造になっており、入出力等を除けば閉じた処理が可能である。

（２）　処理ユニットＩＥの概要

現在検討を進めている処理ユニット（ＩＥ）の内部構造を図１３に示す。

主な構成要素は、アクテイビィティ・コントローラ（ＡＣ：Activity Controller），多数のプロセッサ（ＵＰ：Unify　Processor　）および多数バンク構成による構造メモリ（ＳＭ：Structure　Memory ）であり、２種のネットワーク，トークン・ループ（Token Loop ）とＵＰ－ＳＭ　Net，によって結合される。

４節で述べたゴールフレームへのプロセッサ割り当てのために、トークンとアクティビィティ・コントローラを設ける。

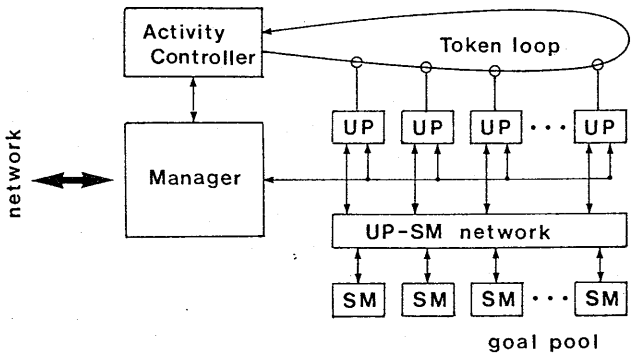

図１３　ＩＥの内部構造

トークンとは、各ゴールフレームにおける制御情報の内からゴールフレームへのプロセッサ割り当てに必要な情報をまとめたものである。トークンの主な情報としては次のものがある。

・ゴールフレーム名
・ジョブ名
・ユニフィケーションのステップ数
・優先度スコア
・構造メモリ内のゴールフレームへのポインタ

現在フォーマットの決定を急いでいる。

アクティビィティ・コントローラはＩＥ内のすべてのゴールフレームをトークンによって管理するモジュールである。アクティビィティ・コントローラでは、探索順序に関するメタな制御情報（ depth-first／breadth-first 等）と各トークンの優先度スコアによってプロセッサ（ＵＰ）に割りつけるゴールフレームが決定され、対応するトークンがトークン・ループへ送出される。

トークンの優先度スコアには次に述べるゴールフレーム内の active call の優先度スコアの総和に相当するものを用いる予定である。

プロセッサ（ＵＰ）は前節（図７）で想定したものである。ただし手続き定義はＵＰ内のプログラム・メモリに共通に与えられる。ＵＰはアクティビィティ・コントローラからトークン・ループ上に送出されたトークンを掴まえ、ゴール・プール（構造メモリ）から対応するゴールフレームを得る。 ＵＰは、ゴールフレーム内の各 call についての優先度スコアによって active call をひとつ選択し、対応する定義節とのユニフィケーションを実行する。ユニフィケーション操作が成功すると新たなゴールフレームが生成される。同時に、そのゴールフレーム内の各 active call について、 引数の instantiation の状況が調べられ、優先度評価のためのスコアがまとめられる。また、ゴールフレームに関する情報がトークンとしてまとめられ、アクティビィティ・コントローラに戻される。

構造メモリ（ＳＭ）は全体でゴール・プールを構成する。構造メモリの主な機能はガーベジ・コレクションである。

６． むすび

本稿では pure Prolog を中心に、その実行方式をProof Diagramを用いてモデル化した。さらにPrologのＯＲ並列処理に基づいた推論向き高い並列計算機を提案し、ＯＲ並列処理の有効性を高めるための制御方式について検討した。

４節においてハードウァア構造の概略を述べたが、検討事項は多く残されている。今後は本稿で述べた制御方式のシミュレーションによる評価を進めると共に、細部の検討を進めていく予定である。

制御方式のシミュレーションには本研究室で開発した手続きレベルのデータフローマシンＴＯＰＳＴＡＲ［３］と、その上に実装されているＯＲ並列Prolog システム“Paralog”［４］とを用いる予定である。現在、評価システムの準備を進めているので、結果が出しだい報告する。

謝辞

最後に当研究において有益な意見を戴いた丸山、湯原両君をはじめとする研究グループＳＩＧＩＥのメンバ諸氏に感謝の意を表します。

《参考文献》

［１］ M. H. van Emden, “An Algorithm for Interpreting PROLOG programs”, RRCS-81-28, Waterloo , Canada , Sept . 1981

［２］ L. M. Pereira et al. , “Intelligent backtracking and sidetracking in Horn clause programs — the theory”, CIUNL 2／ 79 Centro de Informatica da Universidade Nova de Lisboa , Oct. 1979

[3] T. Suzuki et al., "Procedure Level Data Flow Processing on Dynamic Structure Multiprocessors", JIP Vol. 5, No. 1, March, 1982

[4] 相田 他 「並列PROLOGシステム "Paralog" の性能測定」、第24回情処全大、1982

[5] 後藤 他 "ユニフィケーション向き計算機に関する一検討"、第24回情処全大、1982