TR-023

拡張ＯＲ並列 ＰＲＯＬＯＧシステム

－ＸＰ’Ｓ－

麻生　盛敏　　尾内　理紀夫

August, 1983

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

拡張OR並列PROLOGシステム
-XP'S-

麻生　盛敏　　尾内　理紀夫

（新世代コンピュータ技術開発機構）

XP'S : AN EXTENDED OR-PARALLEL PROLOG SYSTEM

Moritoshi Asou & Rikio Onai

ICOT Research Center

Technical Report, August 25 1983

ABSTRACT

This paper describes an Extended OR-Parallel Prolog System (called " XP'S "), its computational model and the XP'S-Interpreter.

This system features the following facilities :

(1) OR-Parallel Reduction,
(2) Modularization (or Multi-World Expression),
(3) Preceded Implication (Control for searching AND-OR tree),
(4) Extended Guard (Control for searching AND-OR tree),
(5) Synchronization of Parallel Processes.

XP'S makes it possible for us to control the search of the AND-OR tree and program the synchronization of parallel processes using the facilities cited above.

In addition, XP'S can be used for applications regarding " Multi-World Expression " and " Object-Oriented Programming ".

拡張OR並列PROLOGシステム-XP'S-

麻生　盛敏　　尾内　理紀夫
（新世代コンピュータ技術開発機構）

1．はじめに

現在の計算機システムの応用分野は，与えられた問題の解決手段があらかじめ明確なものが中心であり，その開発目標はいかに効率よくこの問題を解くかということになろう．

一方，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迫まられており，この知識情報処理システムを構築する上で，その中核言語として論理型言語，特にその種々の特徴からPROLOGが脚光を浴びている．

知識情報処理では与えられた問題と知識から問題解決手順を試行錯誤的に発見する推論機能が中心となり，このような機能を言語自体に備えたPROLOGを直接実行する計算機システムを実現することは，より高度な知識情報処理システムを構築するための第一歩となろう．

さて，このPROLOGに代表される述語論理型プログラムの実行過程は定理証明のための推論過程であり，これをいかに効率よく実現するかが問題となる．
定理証明の推論を高速に実行する方法としては，

(a) 推論アルゴリズムの効率化
(b) 推論の並列化

などが考えられる[1]．

(a) はいいかえると，無駄な探索を少なくして，解のありそうな所を優先的に試行するということである．

問題はこの“解がありそう”ということをどのようにして判断するかであり，
従来の逐次型PROLOGは，プログラムの文法的構造，すなわちPROLOGがHorn節の集合であることを利用して，これにSNL(Selective Negative Linear) 導出を適用したものである[2]．

また，PROLOGではカットオペレータにより無駄な探索を少なくしている．

(b) については，定理証明の推論過程を表わす証明木をAND-OR木の形式で展開することが効率的であるため，これに基づいた並列化など種々の並列化が現在提案されている[3]～[9]．

本レポートではこれらの現状を鑑み以下の機能を備えた拡張OR並列PROLOGシステム
(EXtended OR-Parallel Prolog System ：以下XP'Sと呼ぶ）とその計算モデルおよびXP'Sインタプリータについて述べる．

－XP'Sの特徴

(1) OR並列推論処理機能
(2) モジュール化機能
(3) 推論の優先度制御機能
(Preceded Implication の導入)
(4) 拡張ガード機能
(5) 並列プロセスの同期制御機能

2．論理型プログラムの実行モデルと並列処理

PROLOGプログラムは，基本的にHorn節の集合であり，H ：-B1 ，B2，…，Bn(n≧0)という形のHorn節から成る手続き本体とH 部分（ヘッド部分）のないゴール文（Goal Statement ）からなる．
プログラムの実行過程は質問として与えられるゴール文で始まり，空なゴール文（Empty Goal Statement）で終るゴール文の列の導出過程であり，その基本的実行サイクルは次の２ステップからなる．

(i)　現在のゴール文（A1，A2，…，An）からリテラルAiを選択する．
一般にこれらリテラルは論理的にAND の関係にありどれを選択してもよいが，現在の逐次型PROLOGでは左から順に選択する．

(ii)　次に(i) で得られたAiと統一化可能（Unifiable ）なヘッドを持つ節
B ：-B1 ，B2，…，Bm　を選択し，新しいゴール文（A1，…，Ai-1，B1，…，Bm，Ai+1，…，　An)θを導出する．
ここでθは統一置換（Most General Unifier）である．
一般にAiと統一化可能なヘッドを持つ節は複数存在し，
この非決定性（Non-determinism ）はOR木として表現される．現在の逐次型PROLOGではこの非決定的処理を節の書かれた順番に従った後戻り処理（Back tracking）により実現している．

さて，以上の説明からもわかるようにPROLOGの実行過程はAND-OR木として展開，表現することが出来る．
従って，おのずからこのAND-OR木に基づいた並列処理というものが考えられる．
PROLOGにおける並列処理はこのAND-OR木に基づいた並列処理と統一化の並列処理とを併せて，大きく次の３つに分類できる．

①AND 並列処理
②OR並列処理
③並列ユニフィケーション

以下，これらの並列処理について簡単に述べる．

(1) AND 並列処理
前述の(i) で述べたゴール文中の複数のリテラルを同時に並列に処理するのがAND 並列処理である．
一般に，このAND 並列処理においては次のような問題が生じる．すなわち，AND 並列に

処理を実行しているリテラル間に共有変数が存在する場合には，これらリテラルの並列処理結果がその共有変数に関して矛盾をおこしていないかどうかを確認しなければならない．この無矛盾性チェック（Consistency Check ）は，単純に考えると次々と並列に求まる全ての解の組み合わせについて確認する必要がある．従って，このオーバーヘッドのために，並列処理の効果が低減される可能性がある．

(2) OR並列処理

(ii)で述べたような，現在の逐次型PROLOGで後戻り処理により逐次制御している部分を並列に実行するのが，OR並列処理である．

単純には統一化を行なうユニット（Unification Unit）を複数用意しておけばよく，AND 並列処理に比べるとその実現は容易であるが並列プロセス数が爆発的に増加する可能性があり，この制御機構が必要となる．

また，[7] で述べられているゴールフレーム間並列処理も一種のOR並列処理と考えられる．

(3) 並列ユニフィケーション

並列ユニフィケーションとは文字通りユニフィケーション自体を並列に実行するものである．この場合にも，リテラル内の複数の引数の間で共有されている変数については無矛盾性チェックを行う必要があり，その意味でAND 並列の一種とみなすこともできる．

但し，この場合には(1) のように多数の組み合せについて考える必要はなく，単に矛盾がないかどうかを調べるだけでよい．

次の章では，OR並列処理に前章の(1) ～(5) の機能を備えた拡張OR並列PROLOGシステムについて述べる．

3．拡張OR並列PROLOGシステム

拡張OR並列PROLOGシステム（XP'S）は現在の逐次型PROLOGを完全にそのサブセットとして含み，それに第１章で述べたような種々の機能を導入したシステムである．

本章では，このXP'Sのシンタックスとその特徴について，プログラム例を用いて説明する．

なお，現在XP'Sインタプリータが DEC-10 PROLOG 上に試作されており，例として用いたプログラムはこの上で実行できる．

3.1 シンタックス

XP'Sのシンタックスは以下に述べる点を除くと全てDEC-10 PROLOG と同じであり，プログラムはOR並列に処理される．

(1) モジュール定義

XP'Sのプログラムは Example 3-1に示すようにdefineおよびend により定義されたいくつかのモジュールからなる．

このモジュール内でのシンタックスはDEC-10 PROLOG と同じである．

(2) ゴールフレーム

従来のゴール文G はG=G1, G2, …, GnのようなゴールリテラルGiの列であったが，XP'Sではこれを Example 3-1のようにゴールフレーム文goal_frame(Module_name, G)として与える． プログラムの各モジュールは１つのワールドを定義すると考えられ，XP'Sではこのゴールフレームを Module_nameで定義されたワールドに浮かぶ，G という初期状態を持ったプロセスとして実現する．

(3) プロセス生成述語

この述語は，あるモジュール内で他のモジュール（同じモジュールでもよい）に新しいゴールフレーム（プロセス）を生成したい場合に用いる．

Example 3-4のようにgenerate_PROCESS(Module_name, G)によりModule_nameで定義されたワールド内に初期状態G のプロセスが生成される．

(4) Preceded Implication

DEC-10 PROLOG における“:-”オペレータは，数理論理学における含意記号（Implication ）に相当する．

XP'Sではこれに優先度をつけたオペレータ（Preceded Implicationと呼ぶことにする）を導入し，AND-OR木の探索の制御を可能にしている．これにより第１章で述べたような，解

のありそうな所を優先的に試行するプログラムの作成が可能になる．

現在のXP'Sインタプリータでは高優先度と低優先度の２レベルをサポートしており，Example 3-1の“::- ”が後者（低優先度）に相当する．

このオペレータを持った節はOR並列処理を行なう時に，“:-”を持った節よりも優先度の低いプロセスとして処理される．

Fig.3-2(a)は，Example 3-1のプログラムを実行した場合に，探索木が拡大していく様子を示したものである．一方同図(b) は優先度をつけない場合のものである．

(5) 拡張ガード

XP'Sでは従来のDEC-10 PROLOG のカットオペレータに相当する機能を持ったオペレータとして，拡張ガード（Extended Guard）“!N”（N ≧ 1）をサポートしている．

これは，CONCURRENT PROLOG [10]におけるガードの拡張概念である．

この拡張ガードの“N ”の意味は２通りに解釈できる．

１つの解釈は，単にAND-OR木のOR分岐の所で，高々N 個の節の非決定的なOR選択を許すと解釈するものであり，この解釈のもとではN=1 の場合がCONCURRENT PROLOG におけるガードに相当する．

もう１つの解釈は，N はその節に負荷された重みを表わすと解釈するものであり，あるOR分岐点で，この重みの総和が一定の値に達した場合には，他の節のOR選択を捨てさるというものである．

XP'Sインタプリータはどちらのモードでも実行できる．これにより第１章で述べたように無駄な探索を少なくできる．

Example 3-3 にそのプログラム例を示す．

(6) 並列プロセス同期用述語

一般的に並列実行をサポートする上で，並列プロセス間の通信と同期のための機構が必要である．XP'Sではこれを実現するために，従来のセマフォアの概念にほぼ準じた同期用述語“sync”を導入している．

これによりプロセスの相互関係が明確に記述できるようになる．

Example 3-4 は単純な生産者・消費者問題のプログラム例である．

ヘッド部が producer(N)と consumer の２つの節は，それぞれ２つの並列プロセスとして実現され，

sync(identifier , N^) （送信側）と

sync(identifier , OUT?) （受信側）との間で同期がとられる．

3.2 応用

本節ではXP'Sの応用について簡単に述べる．ただし，ここではその可能性を述べるだけ

にとどめ，詳細は今後の課題とする．

(1) 多世界表現

前節でも述べたように，XP'Sのモジュールは１つの世界を定義していると考えられる．また，プロセス生成述語により各世界間にまたがった関係も記述できる．

さらに，現在のXP'Sでは各世界の間には階層構造はないが，これにその上下関係を記述する機能，これはホーン論理で記述できる，を取り入れることにより階層構造をもった世界の記述が可能となると考えられる．

(2) オブジェクト指向プログラミング

プログラミング手法の１つの流れとして，オブジェクト指向プログラミングがある．このオブジェクト指向プログラミングを実現する上で，まず何をオブジェクトにするかが問題となるが，プロセスをオブジェクトと考えればXP'S上で素直に実現できる．

例えば，その実現例として Example 3-5にカウンターの記述を示す．

動作はプログラムを見れば一目瞭然と思われる．

その他 Property Inheritance 等の問題もあるが，ここではこれ以上深入りはしないことにする．

なお，Example 3-5 の 6～ 9行目までは本来なら次のようにプログラムすべきである．

```
show(N)      :-sync(id,[show,N]^) , display(N), ttynl ,
counter(OLD):-sync(id,COM?) , state_transition(OLD,COM,NEW) ,
                generate_PROCESS(counter,counter(NEW)) .
```

これは，現在のPUROLOG ではグローバル変数が扱えないためで，次のバージョンではこれもサポートする予定である．

(3) その他の応用

XP'Sでは Preceded Implocation と Extended Guard によりAND-OR木の探索の制御を可能にしている．

従来よりAND-OR木の探索戦略は種々のものが研究されており，この２つの機能を用いた有効な戦略も考えられよう．

```
( Example 3-1 )

 /*01*/ define example3_1.
 /*02*/   factorial(X,Y) :-  fact(0,X,Y),display(Y).
 /*03*/   fact(X,X,1)    ::- true.
 /*04*/   fact(X,Y,Y)    ::- Y is X+1.
 /*05*/   fact(X,Y,Z)    :-  Y>X+1, MID is (X+Y)/2,
                              fact(X,MID,Z1),fact(MID,Y,Z2),Z is Z1*Z2.
 /*06*/ end.

        ********** LOGGING LIST **********

| ?- compile(xps).

xps compiled: 293 words,     0.63 sec

yes
| ?- run.
INPUT No
 ** 1 : XPS.TRANSLATOR
 ** 2 : XPS.INITIALISER
 No=2

xps.initialiser compiled: 665 words,     0.87 sec.
 :
xps.built_in compiled: 2096 words,     3.74 sec.
 PROGRAM FILE NAME [ FORMAT : '******.***'. ] = 'test.out'.

test.out consulted   233 words      0.17 sec.
 NEXT FILE ? (Y/N) = N
[ PROCESS_POOL_No. , PROCESSING_UNIT_No. ] =[2,3].
*** IF NECESSARY, SET MODE ***
yes
| ?- debug(1),debug(2),debug(3),xmode(guard_mode(1)),run_xps.
INPUT GOAL [ FORMAT : goal_frame(WORLD,(GOAL)). ]
>>goal_frame(example3_1,factorial(3,X)).
PROCESS_POOL_#1
     PROCESS(priority #1,ready) = 1
     PROCESS(priority #2,ready) = 0
     PROCESS(priority #1, wait) = 0
     PROCESS(priority #2, wait) = 0
PROCESS_POOL_#2
     PROCESS(priority #1,ready) = 0
     PROCESS(priority #2,ready) = 0
     PROCESS(priority #1, wait) = 1
     PROCESS(priority #2, wait) = 0
PARENT   : [process([1,test,1,[1,1,1]],[0,1],1,wait,query,[factorial(3,_57
POCKET_1 : [process([2,test,1,[2,1,0]],1,_640,ready,factorial(3,_609),[fac
,_609),display(_609)])]
n(NEXT)  = 1
:
:
[process([3,test,2,[3,1,1]],2,0,ready,fact(0,3,3),[fail_in_built_in_predicate_i
s])]
:
6
:
INPUT GOAL [ FORMAT : goal_frame(WORLD,(GOAL)). ]
```

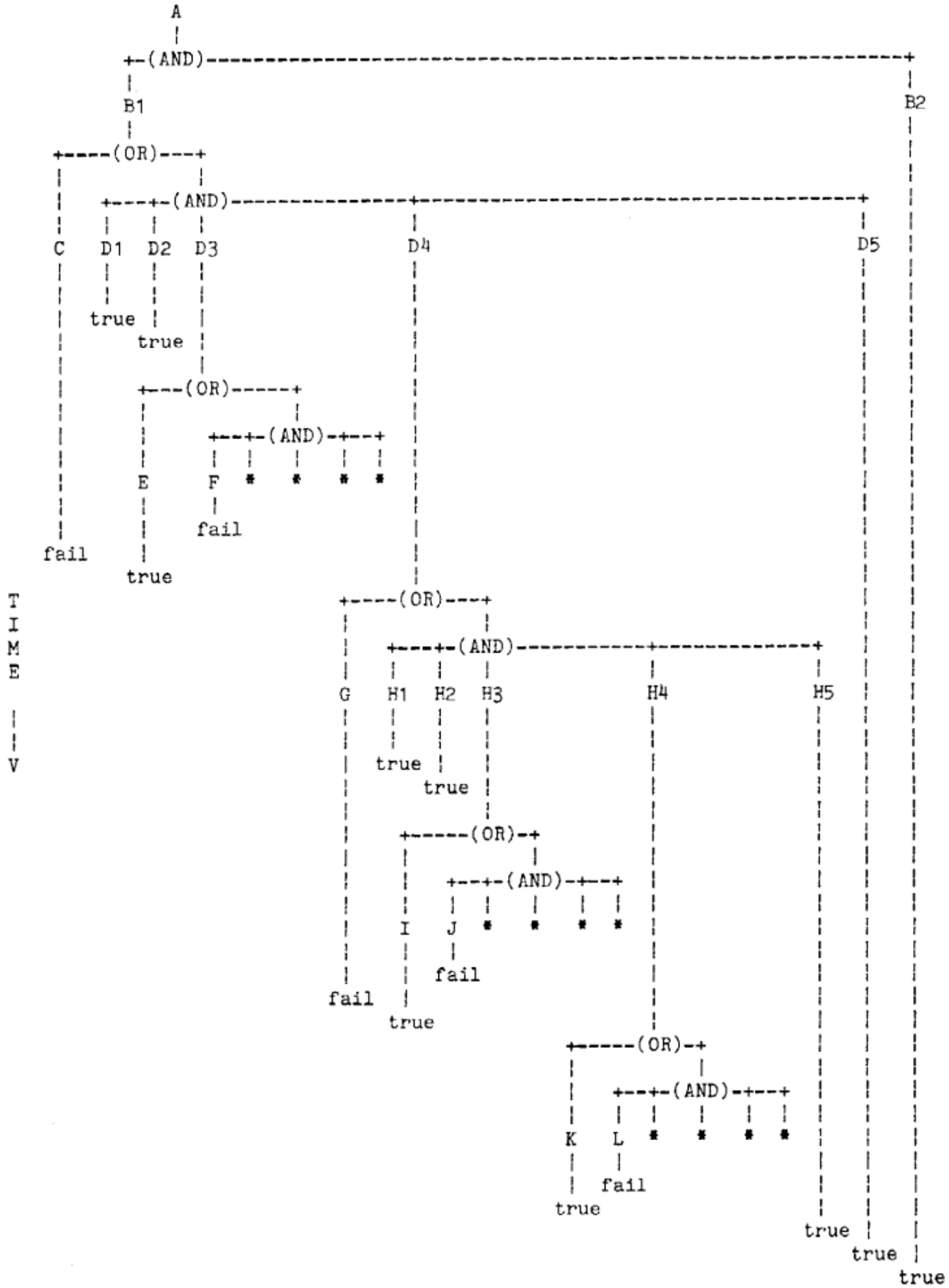

Fig. 3-2 (a) AND-OR Tree of Example 3-1.

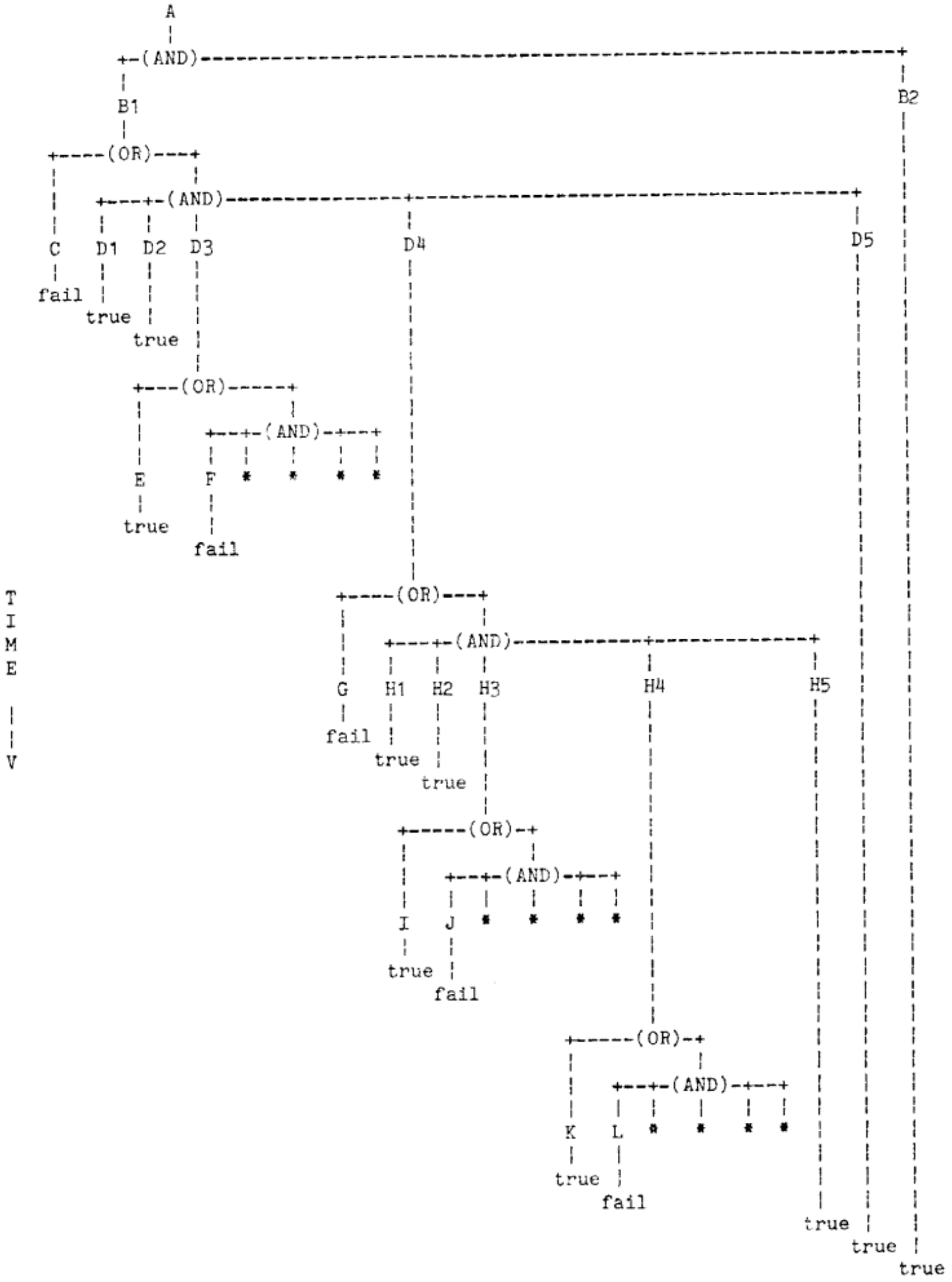

Fig. 3-2 (b) AND-OR Tree of Example 3-1 (No Precedence).

(Example 3-3)

```
/*01*/ define example3_3.
/*02*/   go(X)  :- p(X,Y),write(Y).
/*03*/   p(X,1) :- a(X),b(X),!1.
/*04*/   p(X,2) :- a(X),!2,b(X).
/*05*/   p(X,3) :- a(X),!3,b(X).
/*06*/   p(X,4) :- a(X),b(X),!4.
/*07*/   a(1).
/*08*/   a(2).
/*09*/   b(1).
/*10*/ end.
```

(Example 3-4)

```
/*01*/ define example3_4.
/*02*/   go :- producer(0).
/*03*/   go :- consumer.
/*04*/   producer(M) :- M<10, N is M+1,sync(identifier,N^),
                         generate_PROCESS(example3_4,producer(N)).
/*05*/   consumer    :- sync(identifier,OUT?),OUT<10,write(OUT),
                         generate_PROCESS(example3_4,consumer).
/*06*/ end.
```

(Example 3-5)

```
/*01*/ define counter.
/*02*/   create(N)     :- integer(N),generate_PROCESS(counter,counter(N)).
/*03*/   clear(N)      :- integer(N),sync(id,[clear,N]^).
/*04*/   up(N)         :- integer(N),sync(id,[up,N]^).
/*05*/   down(N)       :- integer(N),sync(id,[down,N]^).
/*06*/   show(N)       :- sync(id,[show,_]^),sync(rec(id),N?),display(N),ttynl.
/*07*/   counter(OLD) :- sync(id,COM?),state_transition(OLD,COM,NEW),
                          send(COM,NEW),generate_PROCESS(counter,counter(NEW)).
/*08*/   send([show,_],NEW) :- sync(rec(id),NEW^).
/*09*/   send([COM,_],_)    :- \+(COM=show).
/*10*/   state_transition(_,[clear,N],N).
/*11*/   state_transition(OLD,[up,N],NEW)   :- NEW is OLD+N.
/*12*/   state_transition(OLD,[down,N],NEW) :- NEW is OLD-N.
/*13*/   state_transition(OLD,[show,_],OLD).
/*14*/ end.
```

4．計算モデル

本章では，XP'Sの計算モデルについて述べる．
第２章で述べたように，PROLOGプログラムの実行過程は空なゴール文で終るゴール文の例の導出過程である．
その基本的実行サイクル［第２章参照］を図示したのがFig. 4-1である．

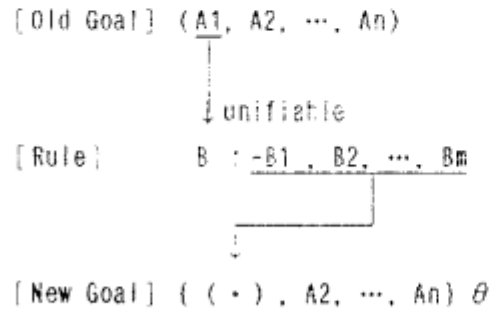

Fig. 4-1 PROLOGの基本的実行サイクル

XP'Sでは，これをFig. 4-2に示すようなプロセスモデルとして実現している．
また，Fig. 4-3はそのアーキテクチャの概念図を示したものであり，各々のプロセスはプロセス・プールに蓄えられる．
各々のプロセシング・ユニットはネットワークを介して適当なプロセス・プールに浮いているレディ状態のプロセスをロードし，このプロセスが解くべきゴールの最初のリテラルと統一化可能なルールをクローズ・プールより選択し，ついで，これをもとに新しいプロセスを（並列に）Forkする．
親プロセスとこのForkされたプロセスは再びネットワークを介して適当なプロセス・プールに戻される．
このとき，親プロセスはForkした子プロセスの数を記憶し，この子プロセスからメッセージが戻ってくるまでウェイト状態となる．
一つの子プロセスから（success ）メッセージが返ってくると，これをもとに新しいプロセスをつくる［第６章(9) の(a) 参照］．
この時の状態の一例を図示したのがFig. 4-2である．
また，拡張ガードや並列プロセス間の同期をとるための機能は，Fig. 4-3のゲート・プールで実行される．

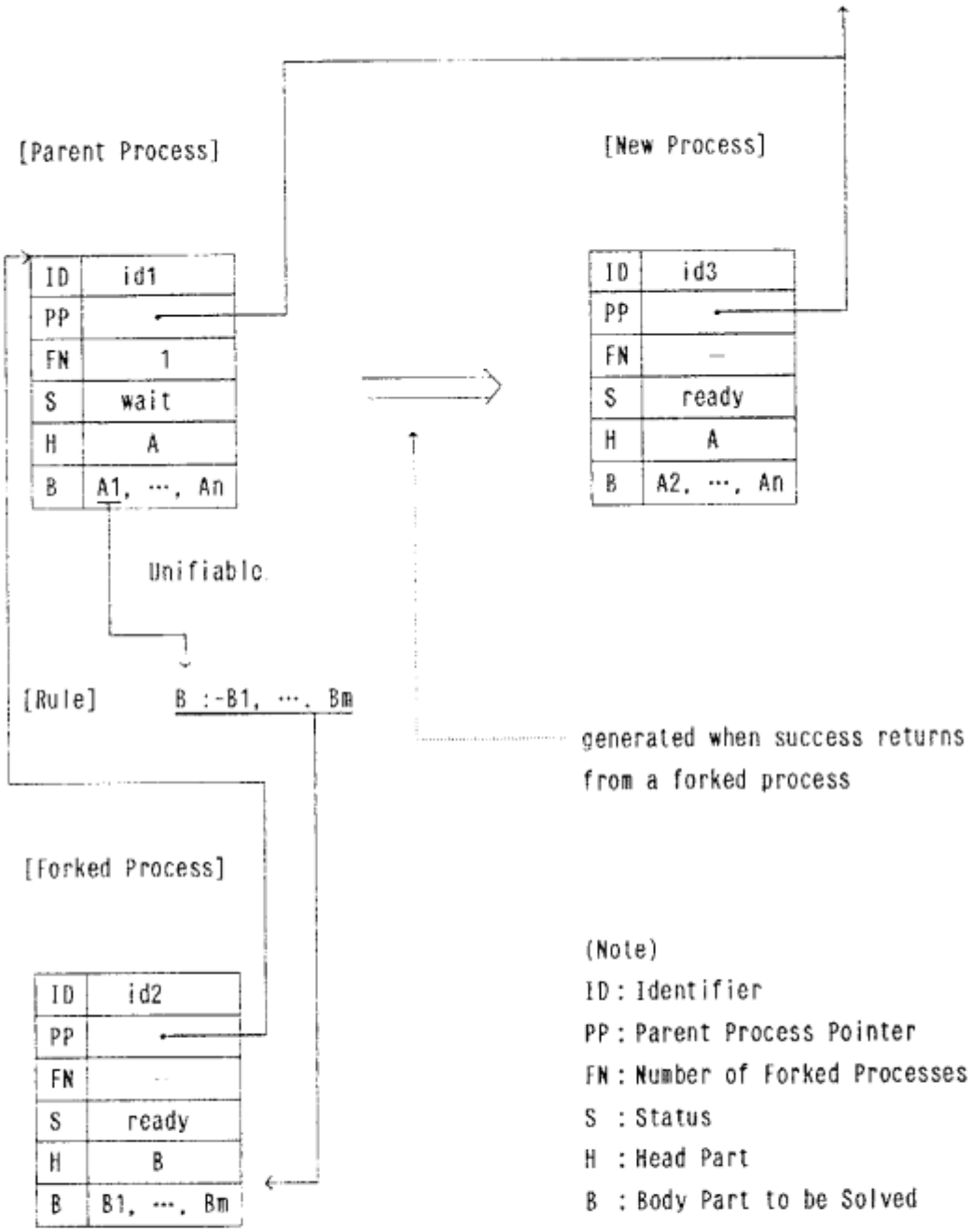

Fig. 4-2 XP'Sにおけるプロセス・モデル

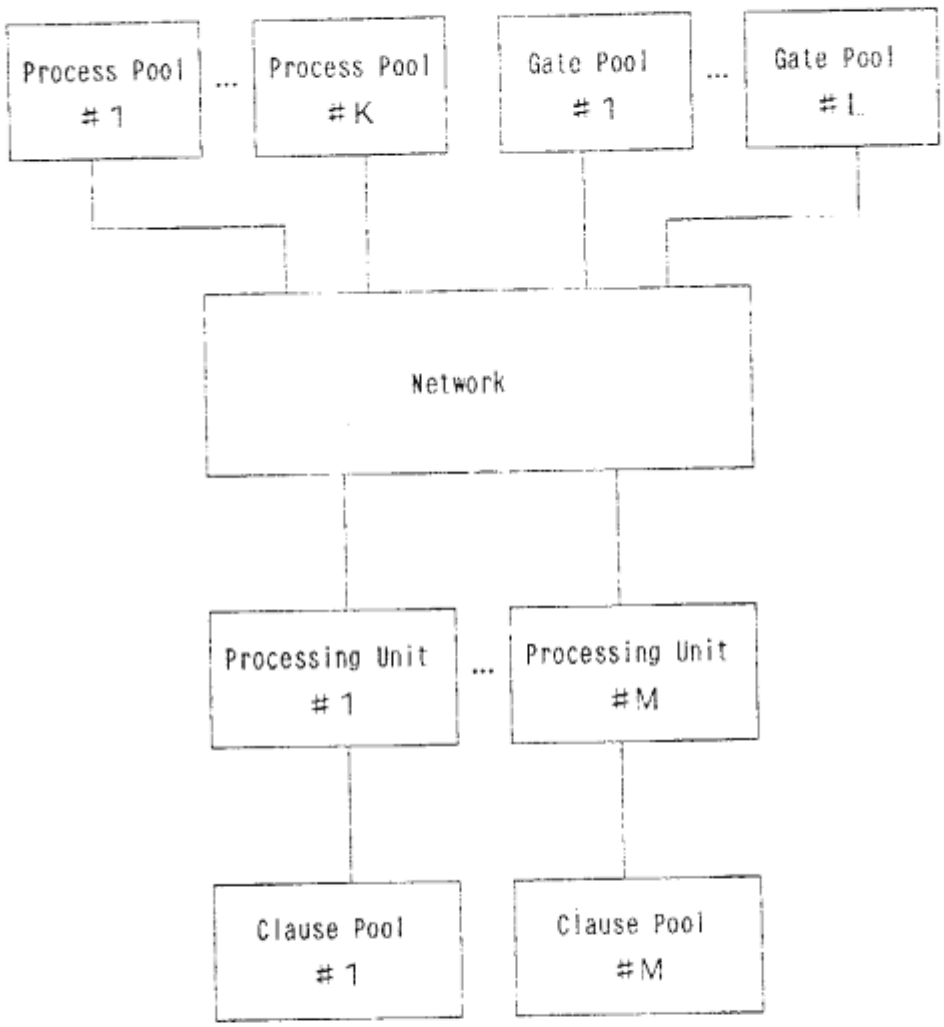

Fig. 4-3 アーキテクチャの概念図

5．XP'Sインタプリータのモジュールの構成と機能

XP'Sインタプリータは DEC-10 PROLOG 上に試作されており，その上で実行される．
XP'Sインタプリータは，次の３つのデータ構造と14個のモジュールから構成される．

(1) データ構造

① Clause Pool
② Process Pool
③ Gate Pool

(2) モジュール

① XPS.
② XPS.TRANSLATOR
③ XPS.REPLACE
④ XPS.INITIALISER
⑤ XPS.TIMER
⑥ XPS.DISPATCHER
⑦ XPS.GATEKEEPER
⑧ XPS.GENERATOR
⑨ XPS.SYSTEM
⑩ XPS.AWAKER
⑪ XPS.UNIFIER
⑫ XPS.LOADER
⑬ XPS.UTILITIES
⑭ XPS.BUILT _IN

以下，それぞれの基本的な構成と機能を説明する．

5.1 データ構造

前述の３つのデータ構造について説明する．
これらは第４章，Fig. 4-3のアーキテクチャの概念図において，それぞれに対応するユニットを表したものである．

(1) Claus Pool
Claus PoolはXP'Sのプログラムであり，１つの節（Clause）は次のような述語として表

現される.

- xclause([MN,P], Head, Body)
  ① MN : モジュール名
  ② P : 優先度
  ③ Head : 節のヘッド部
  ④ Body : 節のボディ部

(2) Process Pool

XP'S インタプリータでは，第4章で述べたプロセスを次のような述語として表わす．Process Poolはこの述語の集合である．

- PROCESS(ID, PP, FN, S , Head, Body)
- ID= [PPN , PN, MN, P , [OL, ON, AL]]
  ① PPN : プロセス・プール番号
  ② PN : プロセス番号
  ③ MN : モジュール名
  ④ P : 優先度
  ⑤ OL : ORレベル
  ⑥ ON : OR番号
  ⑦ AL : AND レベル
  ⑧ PP : 親プロセスへのポインタ
  ⑨ FN : Forkした子プロセスの数
  ⑩ S : 本プロセスの状態
  ⑪ Head : 親プロセスへ返す情報
  ⑫ Body : 本プロセスの解くべきゴール

(3) Gate Pool

XP'Sでは第３章で述べたように各種の拡張機能を取り入れているが，この内の拡張ガードと並列プロセスの同期を制御するのがこのGate Pool である．

Gate Pool は次の３つの述語の集合からなる．

- gurard(ID , N)
- semaphore(ID, (DQHP, DQTP) , (PQHP, PQTP))
- message(ID, DATA)
  ① ID : 各々の識別子
  ② N : 拡張ガードで設定される値

③ DQHP： データキュー・ヘッドポインタ
④ DQTP： データキュー・テイルポインタ
⑤ PQHP： プロセスキュー・ヘッドポインタ
⑥ PQTP： プロセスキュー・テイルポインタ
⑦ DATA： 送信用データ

5.2 モジュール

XP'Sインタプリータの14個のモジュールの関係をFig. 5-1に示す.

(1) XPS
本モジュールは，ユーザの指定により必要なモジュールをコンパイルして，XPS.TRANSLATORもしくはXPS.INITIALISER を起動する.

(2) XPS.TRANSLATORおよびXPS.REPLACE
本モジュールは，ユーザの指定したプログラム・ファイルを入力し，これに 5.1(1) で述べたような変換を施した後，出力ファイルとして出力する.
実際の変換はXPS.REPLACE が行なう.

(3) XPS.INITIALISER
本モジュールは次の処理を行なった後，XPS.TIMER を起動する.
(i) プロセス・プール，プロセシング・ユニットの数，およびその他のモードなどの初期設定.
(ii) (2) で変換されたプログラム・ファイルの入力.
(iii) ゴールフレーム（質問文）の入力.

なお，XPS.TIMER の処理が終了したあとは，ふたたび(iii) の状態となる.

(4) XPS.TIMER
XPS.DISPATCHERおよびその配下のモジュールが実行することは，ちょうど１つのプロセシング・ユニットの機能に相当する.
本モジュールは，(3) で設定されたプロセシング・ユニットの数だけXPS.DISPATCHER以下のモジュールの実行をくり返す.

(5) XPS.DISPATCHER
本モジュールは，XPS.LOADERを介してプロセス・プールよりレディ状態のプロセスを選択し，そのプロセスの実行すべきゴールに従って以下で述べる各モジュールを起動する.

(6) XPS.GATEKEEPER

本モジュールは，プロセスの実行すべき最初のゴールが拡張ガードもしくは同期用述語“sync”の時，XPS.DISPATCHERにより起動される．

(a) 拡張ガードのとき

(i) 拡張ガードは，論理的にORの関係にある兄弟プロセスにおいては，同一の識別子を持つように実行時に変換される．

もし，同一の識別子を持つ述語“guard ”がGate Pool になければ，XPS.LOADERを介してGate Pool に新しくこれを作り，ゴールから拡張ガードを取り除く．

(ii)すでにGate Pool に同一の識別子を持つ“guard ”が存在すれば，この“guard ”にセットされている値をチェックする．

3.1(5) で述べた解釈に従ってチェックを行い，セットされている値を更新する．新しいゴールは拡張ガードを取り除いたものとなる．

もし，このチェックでエラーとなった場合は，新しいゴールを“fail”とする．

(b) 同期用述語のとき

Gate Pool に同じ識別子を持つ述語“semaphore ”が存在しなければ，(a) と同様に新しくこれをつくる．

存在すればこれを読み出し，同期用述語“sync”の実行を行う．

この“sync”の機能はほぼ従来のセマフォア・システムのP 操作，V 操作と同じである．

新しいゴールは“sync”を取り除いたものとなる．但し，受信（P 操作）の時で，データがまだ送信されてきていない時にはこのプロセスはキューにつながれる．

(a)，(b) いずれの場合にも，前記受信時の場合を除いて，新しいゴールを持ったプロセスをXPS.LOADERを介してプロセス・プールに作る．

(7) XPS.GENERATOR

本モジュールは，プロセスの実行すべき最初のゴールがプロセス生成述語の時，XPS.DISPATCHERにより起動される．

3.1(3) に述べたようにプロセスを生成した後，(b) と同様に新しいプロセスをプロセス・プールに作る．

(8) XPS.BUILT _ INおよびXPS.SYSTEM

XPS.BUILT _INは実行すべきゴールリテラルが組み込み述語であるか否かをチェックするモジュールである.

XPS.DISPATCHERは，実行すべき最初のゴールが組み込み述語であることをこのモジュールで確認した後，XPS.SYSTEMを起動する.

XPS.SYSTEMではこの組み込み述語を実行し，新しいゴールを持ったプロセスをXPS.LOADERを介してプロセス・プールに作る.

新しいゴールは，組み込み述語の実行が成功した場合は，この組み込み述語を取り除いた残りのゴール，失敗した場合は“fail”となる.

(9) XPS.AWAKER

本モジュールは，プロセスの実行すべきゴールが全て実行し終った時，もしくは“fail”の時に，XPS.DISPATCHERにより起動される.

(a) ゴールの実行を全て終了したとき

第４章で述べたように，このプロセスが親プロセスへ返すべき情報と，親プロセス・ポインタで示される親プロセスから新しいプロセスを生成する.

この時，親プロセスは新しいプロセスを生成するためのひな型であり，親プロセスの持つ“fork”した子プロセスの数”であるFNを 1減ずる［ 6.1(2) 参照］.

同時に，親プロセスの親プロセスに対して新しいプロセスができたことを通知する．具体的にはそのプロセスのFNに 1加える.

(b) failのとき

(a) と同様にFNを 1減ずる.

(a)，(b) いずれの場合にも，XPS.DISPATCHERよりその処理を引き継いだプロセスはガベージとなる．また，FNが 0となった親プロセスも同様である.

このガベージの処理も本モジュールで行なっている.

(10) XPS.UNIFIER

本モジュールは，プロセスの実行すべき最初のゴールが上記で述べたいずれでもない時，XPS.DISPATCER により起動される.

この最初のゴールと統一化可能なヘッド部を持つ節を，XPS.LOADERを介してクローズ・プールより選択し，プロセス・プールに子プロセスを生成する.

親プロセスは，子プロセスから解が送信されてくるまでウェイト状態となる.

また，前記のような節がみつからなかった時には，このプロセスのゴールは“fail”となりプロセス・プールにもどされる．

(11) XPS.LOADER
本モジュールは，他の各モジュールと各プールの間でのデータの送受信を行なう．

(12) XPS.UTILITIES
各種ユーティリティの集りである．

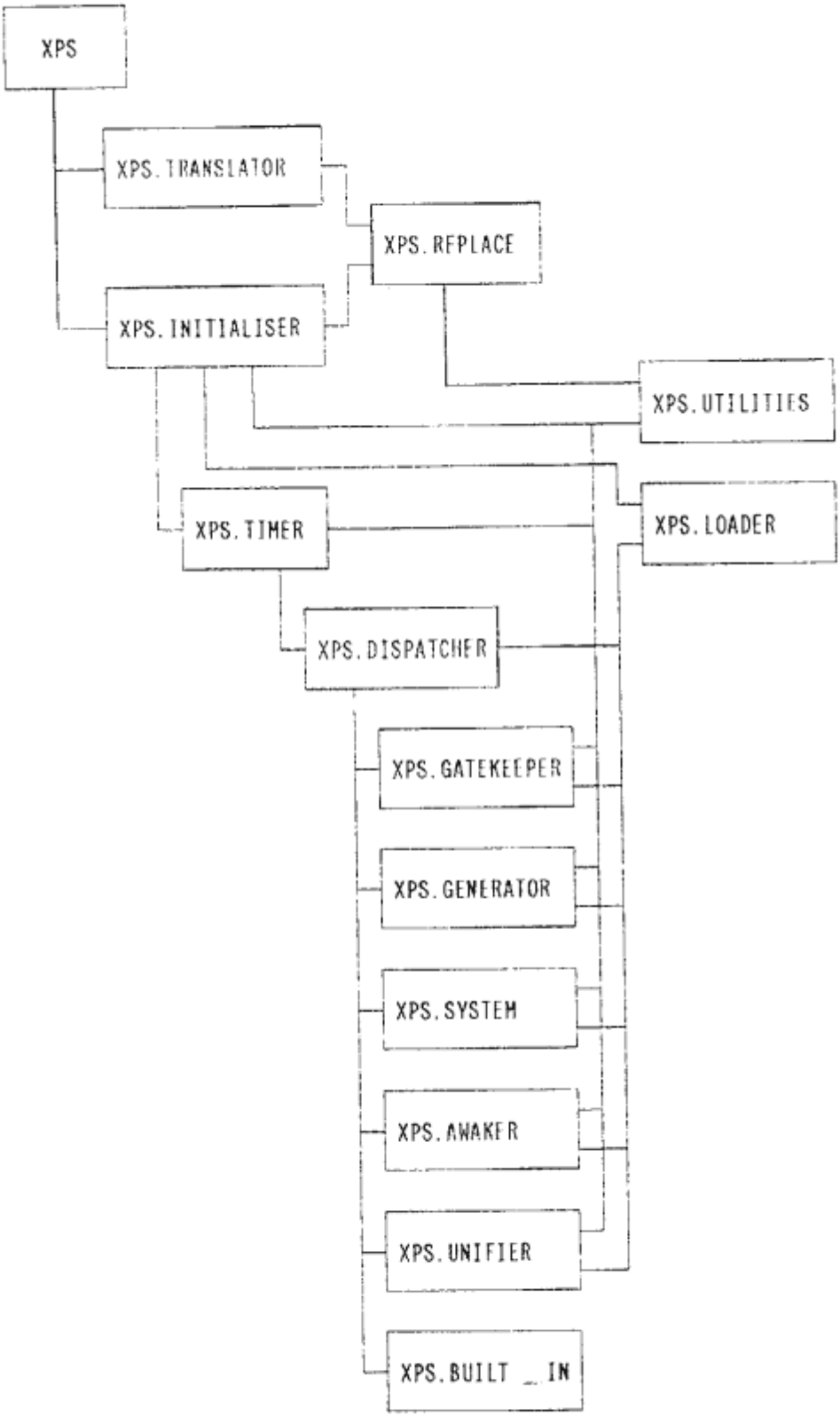

Fig. 5-1 XP'Sインタプリータのモジュールの関係

6．おわりに

本論文では以下の機能を備えた拡張OR並列PROLOGシステム（XP'S）を提案した．

(1) OR並列推論処理機能
(2) モジュール化機能
(3) 推論の優先度制御機能
　(Preceded Implication の導入)
(4) 拡張ガード機能
(5) 並列プロセスの同期制御機能

これらの機能の導入により，XP'SではAND-OR木の探索制御が可能となり，自然な形で並列プロセスの記述が可能となっている．
さらに，多世界表現，オブジェクト指向プログラミング等が実現できると考えられる．
これらについては今後さらに検討していく予定である．
最後に日頃御指導いただく村上国男第一研究室長はじめ第一研究室諸氏に深謝する．

［参考文献］

[1] 電子計算機基礎技術開発調査報告書，アーキテクチャ技術編，
日本情報処理開発協会，(1982)
[2] 佐藤泰介，導出原理による定理証明，情報処理，Vol.22，No.11 ，(1981)
[3] 雨宮，長谷川，データフロー制御による論理プログラムの実行機構，
Proceedings of The Logic Programming Conference '83,Tokyo,(1983)
[4] 伊藤，尾内他，データフロー方式の並列Prologマシン，
Proceedings of The Logic Programming Conference '83,Tokyo,(1983)
[5] 安原，小松，OR並列モデル　：ORBIT について，
Proceedings of The Logic Programming Conference '83,Tokyo,(1983)
[6] 梅山，田村，論理プログラムのためのOR並列実行モデル，
Proceedings of The Logic Programming Conference '83,Tokyo,(1983)
[7] 後藤，相田他，高並列推論エンジンPIE について，
Proceedings of The Logic Programming Conference '83,Tokyo,(1983)
[8] 田村，松田他，K-Prolog（並列Prolog）の実現方法について，
Proceedings of The Logic Programming Conference '83,Tokyo,(1983)
[9] 尾内，清水他，リダクション機構に基づくPrologマシンの一構成法，
情報処理学会第26回全国大会，(1983)
[10] E.Y.Shapiro , A Subset of Concurrent Prolog and Its Interpreter ,
ICOT Technical Report TR-003, (1983)