TM-0594

ＡＩ向きプログラミング「ＥＳＰ」

澤田秀穂，箕田依子（富士通）

2－3）ＥＳＰ

1　概要

ＥＳＰ(Extended Self-contained Prolog)は論理型言語Prologにオブジェクト指向パラダイムに基づくモジュール化機能を導入したプログラミング言語であり，逐次型推論マシンＰＳＩ(Personal Sequential Inference machine)のＯＳを含むシステム・プログラムおよびユーザ・プログラムの記述のために開発された．下記の特徴がある．

(1)　プログラムのモジュール化機能を利用したプログラムの部品化およびこれらの合成，再利用を容易に行うことで，大規模なプログラムを効率良く開発できる．

(2)　オブジェクト指向パラダイムによるクラス，継承の機能を利用した階層的な知識表現が可能である．

(3)　Prologのユニフィケーション，バックトラックの機能がある．

(4)　強力なマクロ展開機能が提供されており，プログラム記述の簡便化，読解性の向上をはかることができる．

現在，ＥＳＰは，ＳＩＭＰＯＳ(Sequential Inference Machine Programming & Operating System)のもとで動作する．

2　エキスパートシステムの実現

ここでは，ＩＦ－ＴＨＥＮ型のルールと，問題に現れる概念を定義したモデル（以下，概念モデルと呼ぶ）から成る知識ベース，知識を利用して問題解決を行う推論機構とワーキングメモリで構成されるエキスパートシステムを考える．例えば，設計型エキスパートシステムは，与えられた仕様をワーキングメモリ中の初期のデータとして，概念モデルを参照しながらデータにルールを前向きに適用し，ワーキングメモリの更新を繰り返すことで設計を行う．

以下，このようなエキスパートシステムのＥＳＰによる実現法の説明を通して，ＥＳＰの機能，特徴を述べる．

〔１〕知識ベース

(1)概念モデルとワーキングメモリの実現

問題に現れる各概念にクラスを対応させると，ＥＳＰの継承の機能，クラス―インスタンス関係を用いて階層的な概念モデルとワーキングメモリの構築を自然に行える．

ＥＳＰの「クラス」は，共通の性質，共通の振る舞いを持つオブジェクトをまとめて定義したものであり，この定義自体を「クラス・オブジェクト」という．概念モ

デルはクラス・オブジェクトの集りである．クラス・オブジェクトから生成されたオブジェクトを「インスタンス・オブジェクト」という．概念モデルを構成するクラス・オブジェクトから生成したインスタンス・オブジェクトがワーキングメモリのデータである．

例１：　ハードウェアにおけるレジスタという概念を定義する．レジスタは"レジスタ名"と"ビット範囲"という属性を持つ．

概念の属性は，「スロット」を用いてオブジェクトに格納できる．ＥＳＰのスロットには，「インスタンス・スロット」と「クラス・スロット」があり，それぞれインスタンス・オブジェクト，クラス・オブジェクトに対応して存在する．レジスタにおける"レジスタ名"と"ビット範囲"はレジスタ（クラスregister）の具体的な情報であるのでインスタンス・スロットを使用する．クラス・スロットは，インスタンス・オブジェクトを管理するために用いることが多い．例えば，ある時点でワーキングメモリの構成要素となっているそのクラスのすべてのインスタンス・オブジェクトを格納する．（図４クラスwm_class 参照）

ＥＳＰのスロットは，手続き型言語の変数に相当し，任意に値の変更が可能である．また，バックトラックにより値が解放されることがない．スロットには，キーワードattribute で宣言する「属性スロット」とキーワードcomponent で宣言する「要素スロット」の２種類がある．属性スロットは手続き型言語の大域変数に相当し，クラス定義の外からオブジェクトとスロット名の対を指定することで参照できる．要素スロットは局所変数に相当し，クラスの外から直接には参照できない．（ただし，スロット値をクラスの外から直接参照することはオブジェクト指向の特徴である情報隠蔽を壊すことになるので，参照にはメソッドを使用することが望ましい．）また，継承するクラス定義の間で同じ名称のスロットがあった場合に，属性スロットは継承により縮退してひとつのスロットとなるが，要素スロットは別々のままである．スロットを属性スロットとして定義するか，要素スロットとして定義するかはこれらの違いを考慮する．

概念を操作する手続きは，基本的にはPrologと同様なユニフィケーションとバックトラックの機能を持つ「述語」で記述する．ＥＳＰの述語は，クラス定義内に限って参照が可能な「ローカル述語」とクラスの外からも参照できる「メソッド」に分けられる．メソッドもまたスロットと同様に，クラス・オブジェクト，インスタンス

・オブジェクトのそれぞれに対応して「クラス・メソッド」，「インスタンス・メソッド」がある．Prologの述語呼び出しに相当するメソッド呼び出しは，第一引数で指定したクラス・オブジェクト，または，インスタンス・オブジェクトに，メソッド名と引数を渡すことである．オブジェクトは必要ならば他のオブジェクトにメソッド呼び出しを行うことでそのメソッドを実行する．

```
class  register  has
 :create(Class, BitRange, Name) :-                    ┐
    :new(Class, Ins),                                 ├ (a)
    :set__name(Ins, Name),                            │
    :set__range(Ins, BitRange);                       ┘
instance
attribute  name, bitRange;                            } (b)
  :set__name(Ins, Name) :- Ins!name := Name ;         ┐
  :set__range(Ins, Range) :-   ……;                    │
            ⋮                                         ├ (c)
            ⋮                                         │
  :name(Ins, Ins!Name);                               ┘
end.
```

図１　レジスタのクラス定義

図１において，(a)はクラス・メソッド，(b)はインスタンス・スロット，(c)はインスタンス・メソッドである．クラスregisterのインスタンス・メソッド set__name/2（述語名/Nは N個の引数を持つ述語を示す）のボディのIns!name:=Nameはオブジェクト Ins(ここではインスタンス・オブジェクト）のスロットname（インスタンス・スロット）へ値Nameを代入すること，同じくインスタンス・メソッドname/2のヘッドのIns!nameはオブジェクト Ins のスロットnameの値とユニファイすることを示すシステム定義のマクロである．

クラスregisterのクラス・オブジェクトに対して，create/3というメソッドの呼び出しを行うと，宣言されたレジスタ名とビット範囲を持つクラスregisterのインスタンス・オブジェクトが生成される．(new/2は，インスタンス・オブジェクトを生成するシステム定義のメソッドである．)

また，継承機構は，概念間の抽象―具象という階層関係の定義，プログラム記述量の削減，保守性の向上に有効である．

例２：　ハードウェアにおけるステートマシンを考える．そこに現れる条件を論理式で表すとき，論

理式のオペランドは，論理式，または，ステートである．

ステートは論理式であるというｉｓ－ａ階層を表す継承関係を用いてこれらの概念を表す．このときステートは論理式を継承するという．論理式（クラス logical_EXP)とステート（クラスstate)の継承関係とそれらのインスタンスの関係は以下の通りである．

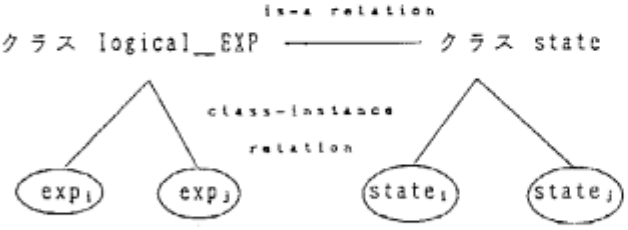

図２　クラスの継承とクラス―インスタンス

```
class  logical_EXP  has
instance
  :exclusive(Exp1, Exp2) :- …… ;
end.

class  state  has
nature   logical_EXP ;
instance
  :state(State) :- !;
  :exclusive(State1, State2) :-
      :state(State2),!, …… ;
end.
```

図３　論理式とステートの継承関係

継承はキーワードnatureを用いて宣言する．クラスを継承することによって，上位クラス(logical_EXP)に定義されたクラス・スロットとインスタンス・スロット，クラス・メソッドとインスタンス・メソッドが下位クラス(state) でも定義されたように扱われる．また，ＥＳＰは複数のクラスを継承することができる多重継承機構を備えている．

クラスstate のインスタンス state$_i$ に，exclusive/2 というメソッド呼び出しを行うには，exclusive/2 の第一引数をクラスstate のインスタンス state$_i$ にする．exclusive/2 の第二引数が exp$_j$ というクラス logical_EXP のインスタンスの場合には，クラスstate のインスタンス・メソッドexclusive/2 のボ

ディのインスタンス・メソッドstate/1 が失敗し，exclusive/2 は失敗する．すると，クラス logical__EXP に定義されたインスタンス・メソッドexclusive/2 がオルタナティブとして呼ばれる．クラス logical__EXP のインスタンス・メソッドexclusive/2 には，より一般的なexclusive/2 が定義されている．

継承の順序は，この例のように指定がない場合は自クラスを最優先し，以降，宣言の順序に従うが，自クラスを任意の優先順序にすることもできる．

クラス・オブジェクトの集まりである概念モデルに対して，それらのクラスから生成されたインスタンス・オブジェクトがワーキングメモリの要素となる．この方法は，ワーキングメモリを構造化する．ワーキングメモリ中のデータは，メソッド呼び出しにより参照される．

このとき，ルール中の手続き（メソッド呼び出しの第一引数が未定義でも，ルールを適用するときにオブジェクトに定義済であればよい）は同じでも，メソッドの呼び出し先が異なれば，異なる手続きとして実行される．

例３：　ワーキングメモリ中のデータを削除するという手続きを記述する．

```
class   wm__class  has
component
  (instances := L :- :create(#list,L));    ) (a)
instance
 after:delete(Obj) :-
    :class__object(Obj, Class),              } (b)
    :remove(Class!instances, Obj) ;
end.

class  register  has
nature  wm__class ;
instance
  :delete(Obj) :-   レジスタに固有の処理 ;
end.

class  logical__EXP  has
nature  wm__class ;
instance
  :delete(Obj) :-   論理式に固有の処理 ;
end.
```

図４　削除する手続きの記述

インスタンス・メソッドdelete/2の呼び出しをクラスlogical_EXP のインスタンスに対して行う場合とクラスregisterに対して行う場合は異なる処理が実行される．

(b)はクラスregisterとクラス logical_EXP に共通の処理である．共通の処理は継承を用いるのが自然である．ここで，after:delete/1 はインスタンス・メソッドdelete/1の「後デーモン述語」であり，delete/1が成功した後に起動される．デーモン述語には，この他，キーワードbeforeで宣言し，前処理を行う「前デーモン述語」がある．また，クラス・スロット(a)は，ＳＩＭＰＯＳが提供する「プール」（ＥＳＰのオブジェクト，データを格納する機能を持つ）を利用して，ワーキングメモリの要素である，そのクラスのインスタンス・オブジェクトを格納する．スロットには，スロット名:=初期値:-ボディという記述によりスロットの初期化を行うことができる．スロットの初期化は，オブジェクト生成時に行われる．後デーモン述語中のインスタンス・メソッドremove/2は前述のプールに定義された，プール中からオブジェクトを削除するメソッドである．

(2)ルールの実現

ステートマシンで実行する演算をコンポーネント（演算器）に割り付ける次のようなルールを考える．

例４：

〔演算Ａと演算Ｂが同時実行可能であれば，共通のコンポーネントに割り付ける．〕…… ①

〔同一のステートで実行され，同じ演算ならば，演算ＡとＢは同時実行可能である．〕…… ②

〔同種演算子で，対称（オペランドの置換えが可能であるような）演算子ならば，オペランドが同一であるとき，演算は等しい．〕…… ③

〔同種演算子で，非対称演算子ならば，位置を含めてオペランドが同一であるとき，演算は等しい．〕…… ④

①は条件部が成立するとき，ワーキングメモリを更新する．このような，前向きに推論を行うルールを前向き推論ルールと呼ぶ．前向き推論ルールの条件の検証には②，③，④が使われる．２つの演算について②が成り立つとき，①の条件部は成り立つ．②が成立するためには③または④が成り立てば良い．ルール中のＡ，ＢはＥＳＰの変数で，ワーキングメモリのデータ（オブジェクト）とユニファイした後，ルールの成否を調べられる．

②は後向きに推論を行う後向き推論ルールと呼ぶ．

例４の実現例を図５に示す．ここで，③，④は演算子

（クラス exp__Operator），演算（クラス logical__Exp）という概念を概念モデルに定義し，演算子とオペランドの関係としてクラス logical__Exp に定義することにする（図中(c)）．この他，①の実行部から呼び出される，共通のコンポーネントに割り付けるというアクションも，概念モデルの手続きとして記述する．

ユニフィケーション，バックトラックを利用してルールを記述すると，ＥＳＰの実行機構を使用できるので推論の高速化の助けになる．特に，後向き推論はバックトラックをそのまま使える．すなわち，ボディに記述した条件はワーキングメモリの事実に辿り着くまで，その条件をヘッドとする述語を繰り返し呼び出すことで，ヘッドに記述した結論の成否を調べる．前向き推論ルールは，ボディに，条件部と実行部を記述すると，first-hit を実現できる．

また，ルールがある特定の対象，または，ある過程に限って適用されるならば，探索空間を狭くするためにルールをモジュール化し，それぞれをクラスに割り当てる．前向き推論ルールは，クラス・メソッドとして定義する（(a)）．クラス・メソッドr1/3の第一引数は，クラス・オブジェクトを受け取るための変数である．この例のように特定の前向き推論ルールの条件の検証に限って用いる後向き推論ルールは，クラス内に限り呼び出されるローカル述語にする（(b)）．ローカル述語はメソッド呼び出しと比較して実行速度が速いからである．

```
class assign__component  has
  :r1(_,Exp1,Exp2) :-                                  }
    compatible(Exp1,Exp2), !,                          } (a)
    :assign($cmp__manager, [Exp1, Exp2] );             }
local
  compatible(A,B) :-                                   }
    :state(A,StateA), :state(B,StateB),                }
    :equal(StateA,StateB),                             } (b)
    :equal(A,B);                                       }
  compatible(A,B) :-  ····· ;                          }
end.

class  logical__Exp  has
instance
attribute   operator, pre, post, state ;
  :state(Exp,Exp!state) :- !;
  :equal(One,Other) :-                                 }
   :equal(One!operator,Other!operator),                }
   equal(One!operator,One!pre,One!post                 }
```

```
                    Other!pre, Other!post);  } (c)
local
  equal(Operator, Pre, Post, Post, Pre) :-
      :symmetric(Operator), !;
  equal(_, Pre, Post, Pre, Post) :- !;
end.

class  exp_Operator has
instance
attribute    name;
  :equal(Operator, Other) :-
    Operator!name == Other!name, !;
end.
```

図５　ルールの記述例

ここに示したモジュール分割は一例であり，この他にも様々な分割の方法が考えられる．

〔２〕推論機構

先に述べたように，後向き推論はＥＳＰの実行機構を使用すると特別なインタプリタを用意する必要はない．

前向き推論を起動するには，ルールを記述したクラスのクラス・オブジェクトにルール名とワーキングメモリのデータを与える．そこで，次の手続きを記述したクラスを設ける．

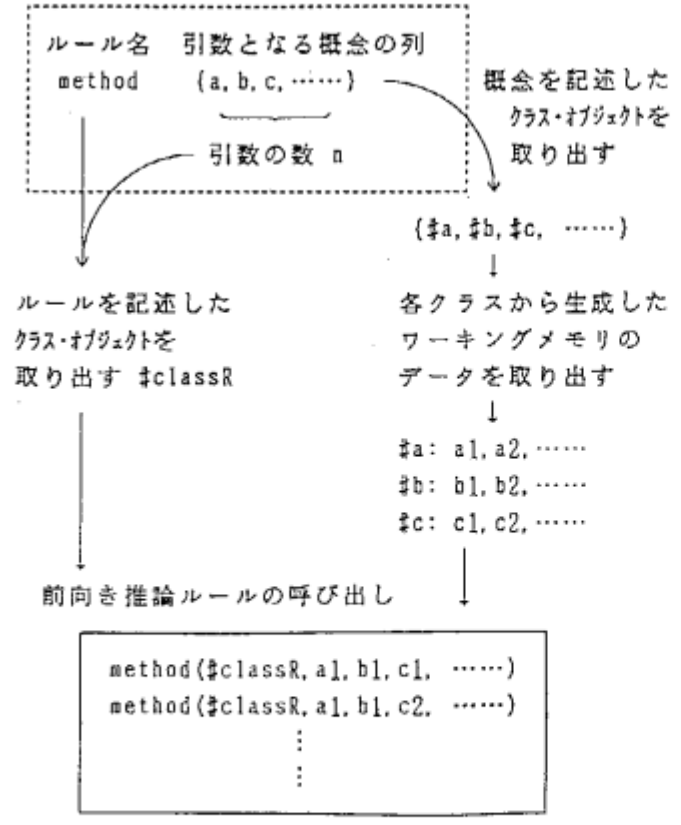

図６　前向き推論ルールの呼び出し

ルール名と引数となる概念の列が与えられているとき，ルール名と引数の数の組等，ルールを特定できる値から，ルールを表すメソッドを定義したオブジェクトを取り出す．メソッドの引数は，概念の列を参照してルールを起動する時点のワーキングメモリのデータを集める．このようにしてルールの起動に必要なメソッド名，オブジェクト，引数を得る．そして，システム定義のメソッド：refute(Object,Method,Args) を用いて，動的にメソッド呼び出しを行う．refute/3は，第一引数で指定したクラス・オブジェクト，または，インスタンス・オブジェクト(Object)に，第二引数で指定したメソッド名(Method)を持つメソッドを，第三引数で指定した引数ベクタ(Args)を引数として呼び出す．refute/3自身の引数は実行時に定義されていれば良いので，これによりルールのメタな制御を記述できる．

〔3〕 ルール記述

ルール記述中のマクロは，ルール作成を容易にし，概念モデルの変更の影響をルールに及ぼさない等の効果がある．

ＥＳＰの特徴であるマクロ展開機構は，単にマクロの対象パターンを展開結果に置き換えるだけでなく，それを含むゴールの前後に必要なゴール列を挿入したり，新たに生成した述語を付加する機能を備えている．例えば，ヘッドの各引数，ボディのゴールとその引数を対象としたものでは，展開パターンの他に，展開パターンの前に置かれる生成用ゴールの列，展開パターンの後に置かれる検査用のゴール列，指定した節のローカル述語への付加等の記述ができる．以上のようにマクロ展開は記述力が高く，ユーザが自由にマクロを定義でき，ユーザ・プログラムから動的に利用できるので，マクロを用いてトランスレータ等を記述するとシステムの開発が容易になる．

例５：引数で与えたリストのすべての要素がCondition を満たすことを記述するため，以下に示したforALLを定義する．

```
forALL(List, X -> Condition)
```

ただし，Listは任意のデータを要素とするリストとユニファイする変数，XはCondition の引数中のX とユニファイする変数，"->"は演算子，Condition は任意のゴール列である．

マクロを用いてforALLを定義した例を図６に示した．

```
macro_bank  rule_function has
  forALL(L,P) => :forALL(#r_func,L,P);                    } (a)
end.

macro_bank  expand_condition has
  Var -> Pred => cond(Var,Expand)                         } (b)
 :- :expand_condition(#expander,Pred,Expand);
end.

macro_bank  expand_rule has
 nature  rule_function,                                   } (c)
         expand_condition;
end.

class expander has
component (macro := Macro
       :- :new(#esp_macro_expander,Macro));
   :expand_condition(_,Pred,Expand):-
     translate(Pred,P),
     :expand_goals(#expander!macro,P,Expand),
     !;
local
   translate((G1,R1),(G2,R2)):-                           } (d)
     translate_goal(G1,G2),translate(R1,R2);
   translate(G1,G2):-
     translate_goal(G1,G2);
   translate_goal(forALL(X,Y),
                  ``:forALL(#r_func,X,Y)):- !;
          :
          :  ゴールのパターンによる置き換え
end.
```

図6　forALL のマクロ定義

(a)はルール記述に現れるforALL/2を，処理手続きを定義したクラスのメソッド呼び出しに置き換えるためのマクロ定義である．置き換えられたクラス r_funcのクラス・メソッド forALL/3 はリストの要素をひとつずつ取り出してCondition を検証する手続きである．(b)はCondition のゴールを展開する．(c)は(a)と(b)の展開順序を，継承宣言の順序関係を用いて定義している．(d)はforALL/2を適用するCondition 中のゴールを置き換える．translate_goal/2の第二引数の``はマクロ抑制を表す．

このマクロを使用してルールの展開を行うには，マクロバンク名をキーワードwith_macro を用いて宣言する．このときルールを記述したファイルには“－＞”という

オペレータの宣言がされているものとする．（マクロ定義を格納したファイルにもオペレータの宣言が必要である．）

```
class  ruleX with_macro expand_rule has
   :rule1(_,Exp1,Exp2) :-
       :primitive(Exp1,EXP1),
       :primitive(Exp2,EXP2),
       for_all(EXP1,X ->
         for_all(EXP2,Y -> :compatible(X,Y))), ①
           ..... ;
end.
```

図7　マクロパンクの使用方法

下線部①は，次のように展開される．

```
:for_all(#r_func,EXP1,
   cond(X,:for_all(#r_func,EXP2,
   cond(Y,:compatible(X,Y))))
```

（#クラス名はクラス・オブジェクトを表すマクロ表記であるためその展開も行われている．）

例えば，実行時にEXP1が〔 Exp_1 ,Exp_j 〕，EXP2が〔 Exp_i ,Exp_m ,Exp_n 〕であれば，

```
:compatible(Exp1 ,Expi )
:compatible(Exp1 ,Expm )
:compatible(Exp1 ,Expn )
:compatible(Expj ,Expi )
:compatible(Expj ,Expm )
:compatible(Expj ,Expn )
```

が呼び出される．

3　その他

ＥＳＰの開発，実行環境であるＳＩＭＰＯＳはＥＳＰでプログラミングを行うための様々な機能（エディタ，デバッガ，ライブラリ，ウィンドウ，プール等）を提供している．ＳＩＭＰＯＳを活用することで効率良いプログラミングを行うことができる．

4　参考文献

(1)ESP,SIMPOS,KLOの説明書　ＩＣＯＴ

(2)近山隆 ESP reference manual(TR-044 Feb.1984)　ＩＣＯＴ

(3)F.Heyes-Roth他編 AIUEO訳　エキスパート・システム　産業図書

(4)黒川利明　Prologのソフトウェア作法　岩波書店

⑸溝口文雄　ＥＳＰ
（コンピュータソフトウェア第５巻第１号pp.62-68）