論理型言語によるソフトウェア設計

牧野　守邦
（慶応義塾大学・理工学部）

概要
共通問題（ネットワーク補修問題）の論理型言語Prologによる設計について述べる。Prologは、データ表現及び制御構造に対する自由度が高く、しかもパターンマッチングやバックトラッキングの機構に加え、これ等の表現には意味論を扱う道具としての一階述語論理を用いている。本論文では、この様な特徴が、実際のソフトウェア設計においてどういう効果を与えているか、設計者の立場から述べる。

§１　はじめに

ソフトウェア設計におけるシステム設計（外部設計）及びプログラム設計（内部設計）には、問題（要求）解決のためのアルゴリズム（実現手段）の提案と実現空間におけるモデル化が含まれる。この過程は、問題空間から実現空間への変換過程としてとらえることができる。［松本 '84］　この変換過程における労力を、プログラミング言語が進化することによって軽減できるとすれば、ソフトウェア設計の困難さを多少なりとも改善することになる。本論文では、論理型言語Prologを用いた例題［松本 '85］の設計について示し、この変換過程に対して論理型言語が与える影響を具体例を通して考察する。

§２　例題の設計

設計における特徴等について論じる前に、まず実際の設計について説明する。実装言語はC-Prolog 1.5KM (Keio Modified version)を使用した。

§２．１　モデルとデータ表現

要求から扱うべき情報としては、中継所（relay)、分配所（distributor)、輸送路（line)、発生源（source）、需要家（consumer）、接続器（connector)、最大許容量（max capacity）、優先度（priority）、需要家の要求量（demand）、輸送路に流れている流量（current volume)などがある。実現空間におけるモデルとしては、relay、distributor、source、consumer をネットワーク構成要素（node）としてとらえ、nodeの属性として次の様な情報をもたせることとした。

ｎｏｄｅ：　node自身を識別するための情報（通しNo. など）
　　nodeの種類を識別するための情報（rly/dstr/src/cnsm)
　　優先度
　　最大許容量
　　接続情報

このモデルは、ほとんどそのまま次の様にデータとして表現される。

```
node(Number,Id,Priority,MaxCap,ConnectionList).
```

ただし、接続情報は、接続しているnodeのNumberと現在の流量（nodeへの入力は負数で、出力は正数で表す）とを組にしたもののリスト（　[c(Number,Volume)・・・]　）である。nodeの種類によっては不必要な情報もあるので（例えば、src のPriorityやMaxCapなど）、そこへはanonymous variable(_)を代入しておく。又、cnsmのMaxCapへは要求量を代入する。

§２．２　アルゴリズムの提案

アルゴリズムの決定には、解の最適性と解法の効率を考慮する必要がある。最適な解を得ようとすると全数探索に近いアルゴリズムとなり、効率の上で問題となる場合がある。これとは逆に、問題自身をかなり単純化することによって効率を上げることができるが、得られる解の質に問題が残る。

例題の設計に当たっては、「人手で操作手順を決めるとしたらどう求めるか」という立場に立って、問題を単純化しているので最適性は保証されないものの、ある程度複雑な処理を行なうことにより良質な解を得るような手法を考えた。

まず、検討すべき状況として、図１の様な、あるnode(N) への供給(V) が停止した場合を考

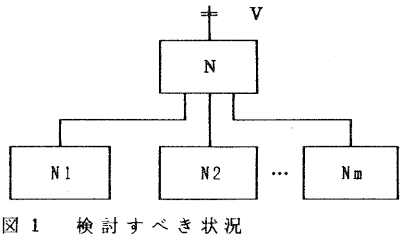

図１　検討すべき状況

える。補修対象が輸送路の場合は、即この状況を検討することになるし、node自体を補修する場合には、そのnodeと接続関係があるnodeに関してこの状況を検討することになる。そこで、適当な前処理を行なうことにより、補修対象が何であれ、問題は、この状況を検討することに帰着される。この状況を検討するroutine をexamine(node(N),V)とすると、手順決定は次の様にして行なわれる。

1)，別の供給源が確保できるかどうか調べ、できればそこから供給を受ける様にする。

2)，別の供給源が確保できない場合、このnode(N) と接続関係にある全てのnode(NN)についてその供給(VV)を止める。さらに、図１の様な状況となるものについては、再起的にexamine(node(NN),VV) を行なう。

1),は次の様な処理から成る。

①，新たな供給源となり得るnode（つながってはいるが、相互流量が０のnode）の集合（in-list)を作る。

- もし結合するとしたら必要となる流量と、供給源となるnodeのMaxCapとの差（Vover)も求めておく。

②，Vover ≦ ０とできるかどうか調べ、できるものは新たな供給源の一つとして登録する（another-supply）。

- 具体的には、in-list 中の各要素に対し、以下の処理を行なう。

a)，もとのnodeをN0、in-list 中の要素をN1とする。N0及びN1の出力node（流量Ｖ＞０のもの）を優先度の低い順にソートする。

b)，ソートした出力nodeの集合の先頭から一要素とり、Vover = Vover－V を求める。

c)，
- Vover ＞ ０ → ・出力nodeの集合が空でなければb)へ行く。
  ・そうでなければd)へ行く。
- Vover ≦ ０ → それまでにとった出力nodeの集合(N-List)と、最後にとった出力nodeの優先度(Pmax)、さらに現在のN1のNo. をanother-supply に記録する。

d)，次のin-list 中のN1についてa)から繰り返す。

③，another-supplyが空なら2)へ行き、そうでなければ、次の処理を行なう。

a)，another-supplyの要素のうち、min(Pmax) なものを求める。

b)，この要素のN-List中の全node(NN)について、その供給(VV)を止める。

c)，node(NN)のうち、図１の様な状況となるものについては、再びexamine(node(NN),VV) を行なう。

```
/* --- examine(Node,Volume) --- */

/* next node is a consumer */

examine(node(N0,I,P,M_C,[c(N1,0)|T]),V0) :-
        node(N1,cnsm,P1,M_C1,L1),examine(node(N0,I,P,M_C,T),V0).

/* next node is a source */

examine(node(N0,I,P,M_C,[c(N1,0)|T]),V0) :-
        node(N1,src,P1,M_C1,L1),not_lock(N1,N0),
        assertz(operate(N1,N0,close)),asserta(lock(N1,N0)).

/* next node is a relay or distributer */

examine(node(N0,rly,P,M_C,[c(N1,0)|T]),V0) :- not_lock(N0,N1),
        node(N1,rly,P1,M_C1,L1),mc_sum(L1,V0,Vt),Over is Vt - M_C1,
        mc_check(node(N0,rly,P,M_C,T),N1,V0,Over).

examine(node(N0,dstr,P,M_C,[c(N1,0)|T]),V0) :- not_lock(N0,N1),
        node(N1,rly,P1,M_C1,L1),mc_sum(L1,V0,Vt),
        Over is Vt - M_C1,mc_check(node(N0,dstr,P,M_C,T),N1,V0,Over).

/* examine connected node */

examine(node(N0,I,P,M_C,[]),V0) :- retract(in_list(L)),exam(L,N0,Ln0),
        exam_next(N0,Ln0).
```

```
/* open all the nodes of NO */

examine(node(NO,I,P,M_C,[]),VO) :- open_all(NO).

/* examine the next element of C_L */

examine(node(NO,I,P,M_C,[H|T]),VO) :- examine(node(NO,I,P,M_C,T),VO).
```

§２．３　全体の構成と実行結果

プログラム全体は、入力処理部、入力条件によりexamine routine の呼び出し方を決める部分、examine routine 、出力処理部、ソートroutine の様な各部での共有部分、から構成される。

全体の制御手順は、入力データを内部データに変換し、需要家の要求量と優先度からネットワークの各構成要素に関する流量と優先度を求め、それから補修手順の決定を行なう様になっている。以下にmain file の一部を示す。

```
/* ..................
   ... total control ...
   .................. */

:- [input,input_proc,operate_decision,examine,output_proc,common].

go :- input_proc,input_check,decide_volume,decide_priority,
      do_repair.
```

図２の様なネットワークに対する実行結果を図３に示す。

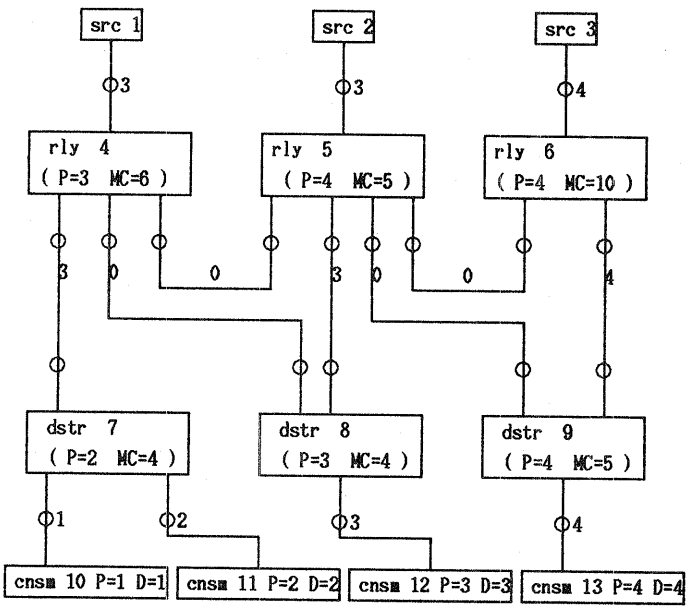

図2　入力として用いたネットワークの状態

```
**************************************************
  The operation sequence of repairing: 6
**************************************************

1     connector[3 -> 6]  open
2     connector[6 -> 9]  open
3     connector[9 -> 6]  open
4     connector[5 -> 8]  open
5     connector[8 -> 5]  open
6     connector[8 -> 4]  close
7     connector[4 -> 8]  close
8     connector[5 -> 9]  close
9     connector[9 -> 5]  close
```

a，中継所6を補修する場合の手順

```
**************************************************
  The operation sequence of repairing: line(3,6)
**************************************************

1     connector[3 -> 6]  open
2     connector[5 -> 8]  open
3     connector[8 -> 5]  open
4     connector[8 -> 4]  close
5     connector[4 -> 8]  close
6     connector[5 -> 6]  close
7     connector[6 -> 5]  close
```

b，輸送路（3－6）を補修する場合の手順

図3　出力結果

§3　論理型言語Prologのソフトウェア設計における特徴

Prologによる設計における特徴としては、データ表現と制御構造の自由度が高いことが挙げられる。さらにその背景には、意味論を扱う道具としての一階述語論理の効用があることも見のがせないであろう。§2で述べた設計例を参考にしながら、これ等の基本的な特徴について簡単に説明する。

§3．1　Prologにおけるデータ表現

Prologのプログラムは、全て項（term）から構成される。項とは定数か変数か構造（structure）のいずれかである。さらに構造は、関数子（functor)と要素（component)から成り、各要素も又項である。[Clocksin '83]

構造の次の様な特徴が、モデルの性質をデータとして表現することに高い自由度を与えている。

- 構造の要素がさらに又構造となっていてもかまわないこと。
- 関数子をフレーム名、各要素をスロットと考えると、構造によってフレーム的な表現ができること。

Prologでは述語論理は構造として表される。これをデータ表現の立場から眺めると、関数子は対応するモデルの名前を表し、そのモデルの性質は一階述語論理を用いて要素として表すことができるということになる。既にnodeのデータ表現にも現われているが、例えば、「本」をデータとして表現する場合は、述語として、

```
book(Title,author(Firstname,Secondname),PublishedYear)
```

の様に表せる。これは、述語表現による意味内容のシンボル化、パターン化とみなすことができ、単なる配列やリストの表現と比べて、データ表現の意味論的抽象化のレベルが高いと言える。このことはデータ表現の自由度が高いことと相まって、設計者個人のレベルに合わせたモデルの抽象化を可能とし、表現されたデータの「意味内容」を想起させることをその設計者に対して容易にさせていると考えられる。

§3．2　Prologにおける制御構造

Prologにおける基本的な制御構造は、以下の様な表現で実現される。

1)，再帰(recursion)

A：－　…，A，…．　自分自身の再帰的な呼び出し。

2)，reduction

A：－　…，B，…．　Bの部分の置き換え。

B：－　B1,B2,B3, …．

3), 逐次選択

⇓ A :- … . 同じ頭部（head）をもつ節（clause）が複数個ある場合は、上（先頭）に
A :- … . あるものから選択してゆく。

4), パターンマッチングによる選択

A (P1) :- … . 頭部の要素のパターンによって節を選択する。
A (P2) :- … .

5), 本体（body）におけるOR制御

A :- ((B1,B2,…) ; (C1,C2,…)) . まず先に（B1,B2,…）の方を試み、それが失敗する場合のみ（C1,C2,…）の方を行なう。

※これは次の様にも表現することが可能である。(3)と6)による)

A :- B1,B2,….
A :- C1,C2,….

6), 後戻り(backtracking)

A :- …, B(P), (fail) … . B(P1) を選んでもその直後の述語が失敗すると後戻りが生じ、3)により、次に選択可能なB(P2) を選び再び実行を試みる。
B(P1) :- … .
B(P2) :- … .

実際には、1)～6)を様々に組み合わせて制御を実現することができ、このことが制御構造に対して大きな自由度を与えていると言える。又、4), の様に、選択条件を論理式で'書く'のではなく、パターンとして'見る'ことができるのも大きな特徴となっている。さらに、assert,retract を組み合わせると、後戻りが生じるごとに副作用を残すことができ、一種のループ構造が実現される。この構造は、定型データの扱いに向いていると思われる。その様な例として、以下に、入力データを内部データに変換するroutine (input-proc)及び、ネットワークの流量を求めるroutine (decide-volume) を示す。

```
/* ..................................
   ... input format transformation ...
   .................................. */

input_proc :- retract(source(N)),assertz(node(N,src,_,_,_)),
        input_proc.
input_proc :- retract(relay(N,M_C,CL)),assertz(node(N,rly,P,M_C,CL)),
        input_proc.
input_proc :- retract(distributor(N,M_C,CL)),
        assertz(node(N,dstr,P,M_C,CL)),input_proc.
input_proc :- retract(consumer(N,P,V)),assertz(node(N,cnsm,P,V,_)),
        input_proc.
input_proc.

/* ........................
   ... decide the volume  ...
   ........................ */

/* 1 */ decide_volume :- node(N,cnsm,P,V,_),mk_cnsmlist(N,V),fail.
/* 2 */ decide_volume :- retract(cnsm_list(L)),dec_v_c(L),fail.
/* 3 */ decide_volume :- retract(dstr_list(L)),dec_v_d(L),fail.
/* 4 */ decide_volume :- retract(rly_list(L)),dec_v_r(L),fail.
/* 5 */ decide_volume :- retract(sv_list(L)),dec_v_s(L).
```

§3.3 Prologにおけるトップダウン設計とボトムアップ設計

Prologでは、アルゴリズム（又はモジュールに要求される機能）に従って、トップダウン的にもボトムアップ的にも設計を行なうことができる。

1), トップダウンな設計

- 必要な機能を分析／分割してゆくにつれて、reduction （§3.2 2),)によって徐々に詳細化してゆく。（ただし、繰り返しが必要となる場合や、引数の渡し方などによって、後で変更しなければならなくなることもある。）
- モジュールの機能がそれほど高くない場合には、比較的容易に設計できる。
- reduction による置き換えが深くなるにつれて制御構造が複雑になり、表現される意味内容がわかりにくくなる傾向がある。

2), ボトムアップな設計

- 機能の分析をまず一通り行ない、後で必要となるであろう部分を断片的に先に設計してしまう。次に、それ等を「部品」として用い、適当に修正しながら、必要となる機能を構成する。
- 述語ごとのモジュール性が高く、機能全体の意味内容の理解が容易となる。
- 始めから、必要となる部品を見抜くことは容易ではない。

ある意味において、トップダウン設計はアルゴリズムをプログラミング言語による表現向きに書き下してゆくことであり、ボトムアップ設計はプログラミング言語で表された意味内容をアルゴリズムのそれに向かって編集してゆくことであるととらえることもできる。実際は、この両者をうまく均衡をとって用いてゆくことになるが、その均衡のとり方が難しい。Prologでは、§3,2 2), で述べたreduction を1)の場合にも2)の場合にも用いることができる。例えば以下に示めす優先度決定routine の例の様に、書き下しにreduction を用いる場合、アルゴリズム（又は機能）の意味内容を一階述語論理で象徴的に表現しておき、後でそれ等を詳細化してゆくことができる。このことは、ユニフィケーションの機構のもとに条件をパターンで表現できること（§3.2 4),)と共に、制御構造における意味論的抽象化のレベルをも高くしていると言える。

```
/* ..................................................
   ... decide the priority of distributor and rely ...
   .................................................. */

decide_priority :- dec_p_d,dec_p_r.

/* setp 1 */

dec_p_d :- dstr_node(DNL),dec_p_d(DNL).
 dstr_node(L) :- mk_dstr_node,retract(dnl(L)).
  mk_dstr_node :- node(N,dstr,_,_,_),mk_dstr_node(N),fail.
  mk_dstr_node.

  mk_dstr_node(N) :- retract(dnl(L)),asserta(dnl([N|L])),!.
  mk_dstr_node(N) :- asserta(dnl([N])),!./* only for the first time */

dec_p_d([N|T]) :- retract(node(N,I,P,M_C,C_L)),dec_p_d(1,P,C_L),
        asserta(node(N,I,P,M_C,C_L)),dec_p_d(T).
dec_p_d([]).

dec_p_d(P,PP,[c(NN,V)|T]) :- node(NN,cnsm,P1,_,_),P1 > P,
        dec_p_d(P1,PP,T).
dec_p_d(P,PP,[H|T]) :- dec_p_d(P,PP,T).
dec_p_d(P,P,[]).

/* step 2 */

dec_p_r :- rly_node(RNL),dec_p_r(RNL).
 rly_node(L) :- mk_rly_node,retract(rnl(L)).
                .......
```

§4　おわりに

アセンブリコードでアルゴリズムを表現しようとすると、問題にとって自然であるようにではなく、機械が何をするかといった風に書かねばならない。最近のプログラミング言語でも、程度の差こそあれ、本質的には同じことを必要としているとは言えないだろうか。このためにアルゴリズムの変換（書き下し）が必要となるが、論理型言語ではデータ表現と制御構造の自由度の高さに加え、述語論理による意味論的抽象化の高さが、この変換過程の困難さを和らげていると考えられる。しかし、これ等の自由度は逆に、プログラムの再利用や保守、変更／拡張といったことに対して必ずしも良い方向に働くとは限らないことも見落してはなるまい。この点を改良するためには、自由度とのバランスの上に立って、論理型言語に適した設計技法としての表現上の何らかの「統制」（とりきめ）を設定することが必要となってくると考えられる。

参考文献

〔Clocksin '83〕　W.F.Clocksin，C.S.Mellish ；中村　克彦　訳：Prologプログラミング，マイクロソフトウェア，１９８３

〔松本 '84〕　松本　吉弘：ソフトウェア工学演習，朝倉書店，１９８４

〔松本 '85〕　松本　吉弘：新しいソフトウェア設計技法に関する試み，ソフトウェア基礎論研究会１３－７，１９８５，６