ICOT Technical Memorandum: TM-0287

TM-0287

並列フレーム

松本裕治

March, 1987

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

並列フレーム

Parallel Frames

松本裕治（新世代コンピュータ技術開発機構）

並列論理型言語による実現を考えたフレーム型知識表現法について述べる。フレーム内の個々のスロットが並列に動くプロセスとして定義されている。ここでは、その基本的な考え方を示し、フレーム型知識表現言語に必要な諸機能を並列論理型言語によっていかに実現すればよいかについて考察する。

1．はじめに

並列論理型言語によるフレーム型知識表現について述べる。並列論理型現語の知識情報処理への応用を考えると、知識の表現法をこの枠組の中で考えることは、非常に重要である。

ＧＨＣ[Ueda 85]、Parlog[Clark 84]、Concurrent Prolog[Shapiro 83]などのいわゆるCommitted-choice Languageによってオブジェクト指向言語を実現しようと言う試みがいくつかある[Shapiro & Takeuchi 83][Kahn 86][Ohki 87]。実際、これらの並列論理型言語によるオブジェクトの記述は、明瞭かつ自然であり、オブジェクトをストリーム通信によって並列に動くプロセスとしてうまく記述できている。ただし、オブジェクト指向言語の一つの特徴であるクラス間の継承階層については、様々な提案があるものの、どの方式でもうまく扱えているとは言い難い。

ここでは、オブジェクト指向言語というよりはむしろ知識の表現としてのフレームを並列に実現することに焦点を当てる。知識表現にとっては、概念の継承階層は基本的な性質で、上で挙げたオブジェクト指向言語のように継承階層の重い実現は、フレームのような知識表現言語としてはふさわしくないと言わざるを得ない。また、フレームにおけるスロットは、これらのオブジェクト指向言語のインスタンス変数とは異なり単なる変数以上の情報を持つことが必要である。ここで提案するのは、フレームの各要素を、並列に動作できるプロセスとして表現する方法である。具体的には、フレームを構成する各スロットを並列に動作できるプロセスとして表現する訳である。こうすることによる最も重要な特徴は、継承階層の実現が簡単になることであり、同時に、継承階層の表現法とその実現の対応がより明確になることである。また、スロット一つ一つをプロセスとして扱っているため、個々のスロット毎に性質を記述するのではなくて、スロットとは何かという大局的な記述を与えるだけでフレーム型の知識表現を実現することができる。

繰り返し言うが、ここでの目的は、オブジェクト指向言語を設計することではなくて、並列論理型言語を用いて知識表現のための基本的な機構を実現することである。ではあるが、本文では、比較のために、まず、[Shapiro & Takeuchi 83]、[Kahn 86]流の並列論理型言語によるオブジェクト指向言語の実現法について考察する。次に、フレーム型知識表現を実現するためのここでの基本的な考え方を説明し、さらにその問題点の整理と対処法について考察する。ただし、ここで述べられているのは、並列論理型言語によるフレーム型知識表現法についての我々の基本的な考え方であって、詳細はまだ未確定である。

2．並列論理型言語によるオブジェクト指向言語

並列論理型言語のパーペチュアルプロセス(perpetual process)を用いてオブジェクトを自然に記述できることが知られている。ここでは、我々の方法との類似性のために、[Shapiro & Takeuchi 83]、[Kahn 86]の方法について考える。オブジェクトは、再帰的に定義された節の集合として記述され、インスタンス変数は、頭部述語の引数として定義される。また、一つ一つの節がメソッドの定義に対応する。例えば、メソッドの定義は次のような具合である。

```
window([moveBy(Dx,Dy)|NewWindow],X,Y,
            Width,Height,Contents) :- true |
      plus(X,Dx,NewX),
      plus(Y,Dy,NewY),
      window(NewWindow,NewX,NewY,
            Width,Height,Contents).
```

ここに、windowの第一引数は、このオブジェクトへのメッセージの受取口である。第二引数以下は、このオブジェクト固有のインスタンス変数である。送られてきたメッセージに対応する処理が本体部で行われて、新しいwindowが再帰的に呼ばれている。この場合は、moveByというメッセージによって、windowの座標が変化させられている。このように、オブジェクトの簡潔な定義が可能であるが、この表現法では、クラス間の継承関係の記述に難点がある。継承関係の実現のために、主に、次の二つの方法が考えられている。

（1）コピー方式：上位概念クラスとして定義されているクラスのメソッドおよびインスタンス変数を下位のクラスにすべてコピーする方法。クラスの定義を並列論理型言語のプログラムに変換する際にコピーが行われるわけであるが、コピー量が膨大になる恐れがあり、現実的な方法ではない。
（2）委任法(delegation to parts)：下位のクラスのオブジェクトが上位のクラスのオブジェクトを部品として持ち、自分の処理できないメッセージを上位の部品に委せてしまう方法。上位のオブジェクトへのメッセージの送信を下位のオブジェクトが扱う点にオバーヘッドがあるが、コピー方式のようなスペースの無駄はない。

3．並列フレーム

3．1　並列フレームの基本的考え方

前節では、並列論理型言語によるオブジェクトの表現に関して、メソッドと継承関係の実現法について述べた。

フレームのような知識表現においては、概念間の継承関係は、基本的な性質である。また、フレームの意味というのは、オブジェクト指向言語の立場とは異なり、その構造自体が重要であって、その手続き的な意味はフレームを扱うインタプリタによって記述されるのが普通である。つまり、オブジェクトの記述とは異なり、フレームの意味は、その構成要素の組み合せによって決まる。更に、前節で紹介した並列論理型言語によるオブジェクト指向言語のインスタンス変数は、単なる論理変数として実現されているだけであり、その指示対象はオブジェクトではなかった。これに反して、フレームのスロットは、単なる値を保持するためだけの変数であってよいわけではない。一般にスロットが指示する対象は、他のフレームのインスタンスであり、それ以外にも、そのスロットが指示する対象についての制約やその他の条件などを記述することができなくてはならない。

ここでは、フレームとはそれを構成するスロットの集合によって意味付けされるものと見なす。更に、スロット一つ一つが並列に動くことのできるプロセスとして定義されているというモデルを考える。各スロットはメッセージを受け取ってはそれに応じた処理を行う訳で、具体的には、それぞれがperpetual processとして実現される。個々のスロットにとって必要な情報はスロット自身に与えられており、スロットの動作に関する記述は、大局的に定義されることになる。表現自体は非常に簡潔である。図1のmailのフレームの記述の例を見よう。楕円で囲まれているのは概念(concept)を表し、一つのフレームに相当する。小円はフレームが持つスロットに対応し、スロット名と共に記述されている。mailフレームは次のようなＧＨＣ節によって表現される。この例は並列フレームの基本的な考え方を示すためのものであって、後でわかるように、これが最終的な記述と言う訳ではない。

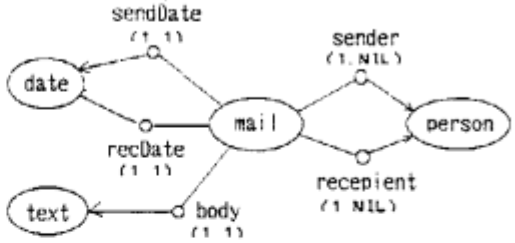

図1．フレームの例[Brachman 85]

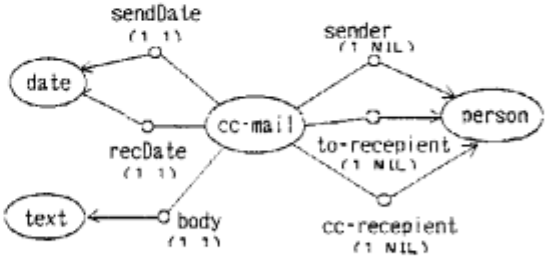

図2．cc-mailの例

```
mail(Mail) :- true |
    slot(sendDate,Mail,(1,1),date,unknown),
    slot(recDate,Mail,(1,1),date,unknown),
    slot(sender,Mail,(1,NIL),person,unknown),
    slot(recep,Mail,(1,NIL),person,unknown),
    slot(body,Mail,(1,1),text,unknown).
```

mailは、五つのスロットを持つフレームとして定義されている。上の例で、slotの引数は、順に、スロットの名前、メッセージの受取口、そのスロットの性質を持つ要素の個数（この例では最小値と最大値の対。NILは無限を表す）、スロットを埋めるべき要素の属性概念の名称、スロットを埋める要素(filler)へのメッセージの送り口、を表している。fillerとなるべき対象ははじめは未定であるから、初期値はunknownとなっている。

この定義からもわかるように、フレームのインスタンスを特徴付けているのは、入力ストリームを共有したスロットの集合である。つまり、その入力ストリームそのものがフレームのインスタンスにアクセスするための唯一の手掛りである。スロットのfillerの初期値unknownが書き換えられるのは、そのスロットが指示すべきインスタンスへの入力ストリームがわかったとき、または、そのスロットを埋めるべきインスタンスを自ら生成するに足るデータが与えられたときなどである。また、スロットが指示

しているインスタンスが何かを尋ねるメッセージが来ることがあり、その場合、スロットは、fillerが指しているインスタンスの入力ストリームを返せばよいわけであるが、自分が持っているストリームをそのまま渡す訳には行かない。つまり、同じインスタンスへの入力ストリームを二つのプロセスが共有することはできず、スロットはmergeを用いてそのストリームへの入力を二つに分け、その一方を質問を発したプロセスに渡すようにすればよい。

ここで示した並列フレーム表現の特徴は、次のように整理できる。
（１）継承関係の記述がほとんど何のオバーヘッドも要しない。
（２）上位概念のスロットの修正や削除、また、スロットの追加などが容易である。
（３）スロット自身に複雑な情報を記述し、管理できる。
（４）スロットが共通に持つ性質や処理すべき仕事を統一的に記述できる。

次に、上位概念について考える。例えば、mailを上位概念に持つcc_mailを定義してみよう。

```
cc_mail(Mail) :- true |
  slot(to_rec,Mail,(1,NIL),person,unknown),
  slot(cc_rec,Mail,(1,NIL),person,unknown),
  mail([erase(recep)|Mail]).
```

cc_mailが呼ばれると、mailが展開されて、mailのスロットが、cc_mailのスロットとして生まれるが、その時、mailに渡されるメッセージerase(recep)によって、recepientに対応するスロットが自ら消滅する。実際、cc_mailのインスタンスが作られたときには、それはあたかも、図２のようなフレームが構成されたのと同じことになる。

ここでは、recepientに対応するスロットが削除されて、新たなスロットが二つ加わるものとしてcc_mailを定義した。しかし、今の場合、新しく生まれたスロットは、元々recepientとして存在したものが二つに分岐したと見る方が自然であるかも知れない。フレームの記述としてここで参考にしたＫＬ－ＯＮＥ[Brachman 85]にはRoleSet differentiationといって、スロットを分岐させるような記述が定義されている。スロットの性質としてそのような記述を許すならば、例えば、cc_mailの定義として、次のような記述を考えることもできる。

```
cc_mail(Mail) :- true |
  mail([split(recep,[to_rec,cc_rec])|Mail]).
```

splitというメッセージがrecepに対応するslotに解釈されて、to_recとcc_recという二つのスロットに分岐する。もし、スロットの名称以外にも変更すべき点があるならば、その情報が一緒に送られる。例えば、受手がただ一人からなるmailをprivate_mailと定義するには次のようにすればよい。

```
provate_mail(Mail) :- true |
  mail([number_restrict(recep,(1,1))|Mail]).
```

このような記述は、mailとcc_mail,private_mailとの関係を自然に表現していると思うがどうであろうか。

ここには、記述されていないが、mailの上位概念があれば、それに含まれているスロットも同様に展開されることになる。つまり、この方法によると、インスタンスが生成される際に、上位概念のスロットがダイナミックに生み出されて来るわけである。同様に、例外の記述などによる上位の概念のスロットの修正(modification)や取消(cancellation)等が簡単にできるのも明らかであろう。

さて、これらの利点に対して、この表現で問題になる点を考えてみよう。第一は、すべてのスロットによって入力ストリームが共有されていることである。フレームへのメッセージが何らかの返答を要するならば、それはバックコミュニケーションを用いて行われる。つまり、メッセージを発する側が返答を置うべき変数もしくは返答用のテンプレートを一緒に送る訳である。このとき、二つ以上のスロットが同じメッセージに対して、異なる返答を返してはならない。つまり、フレームへ渡されるメッセージは、ただ一つのスロットに対してのみ処理されなければならない。また、第二点としては、今の定義のままでは、スロットからフレーム自身へのアクセスの道が用意されていない。

第二点の問題に対処する方法としては、フレーム内のスロットからの出力ストリームを、マージを介して自分自身に送れるようにする方法がある。例えば、cc_mailの最初の定義の場合については、それを次のように定義し直せばよい。

```
cc_mail(Mail,Self) :- true |
   slot(to_rec,Mail, ... ,Self1),
   slot(cc_rec,Mail, ... ,Self2),
   mail([erase(recep)|Mail]Self3),
   merge3(Self1,Self2,Self3,Self).
```

merge3は３入力マージである。この定義からわかるように、上位概念のスロットから自分自身へのメッセージもmergeを経由して送られて来ている。インスタンスへの入力メッセージと自分のスロットからのメッセージの接点は、インスタンスが作られるときに設定される。例えば、フレームのインスタン

スを作り、そのインスタンスに入力ストリームを与えるための節createが次のように定義されていると思えばよい。inputStreamがここで作られるインスタンスに本来渡されるべきストリーム、Selfが自分自身のスロットから渡されるメッセージを運ぶストリームである。

```
create(cc_mail,InputStream) :- true |
      cc_mail(CC_mail,Self),
      merge(InputStream,Self,CC_mail).
```

以後、自分自身への通信の機構はデフォルトとしてフレームが持つことと仮定し、Self変数やmerge等は陽には記述しないことにする。

さて、次節では、上で述べた第一点の問題を解決するための方法について考察しよう。

３．２　分配器を持つ並列フレーム

前節の基本的な並列フレームでは、フレームのインスタンスへの入力メッセージが全てのスロットに渡されていた。そのため、そのメッセージが有効なスロットは基本的に一つでなければならなかった。

ここでは、各フレームの定義に分配器(distributor)を配置して、入力メッセージをどのスロットへ渡すかを決定させることを考える。分配器は、それぞれのスロットへの入力を管理しており、通常同じメッセージが二つ以上のスロットに渡らないようにする。もし二つ以上のスロットへ返答を要するメッセージを送る必要がある場合には、それぞれのスロットからの返答を受け取るための変数を別々に用意する。勿論、分配器もまたperpetual processとして定義される。例えば、mailの定義は次のように記述し直すことができる。

```
mail(Mail) :- true |
    mail_dstr(Mail,[SD,RD,Sr,Rec,Body]),
    slot(sendDate,SD,(1,1),date,unknown),
    slot(recDate,RD,(1,1),date,unknown),
    slot(sender,Sr,(1,NIL),person,unknown),
    slot(recep,Rec,(1,NIL),person,unknown),
    slot(body,Body,(1,1),text,unknown).
```

mail_dstrが分配器に相当し、第二引数に、それぞれのスロットへの送信口を持っている。これらがリストによって束ねられているのは、スロットの追加、消去に対応するためのもので、そういう事態が起こらない保証があるのなら、それぞれ独立の引数として表現されていてもよい。分配器は上でも述べたように、perpetual processとして定義されている。具体的には書きにくいが、例えば、次のような形の節の集合で定義される。

```
mail_dstr([MessiMail],[SD,RD,Rec,Body]) :-
    messToSender(Mess) |
    SD = [MessiNewSD],
    mail_dstr(Mail,[NewSD,RD,Rec,Body]).
```

これは、senderへ送るべきメッセージが来れば、それに対応するスロットへそのメッセージを送っているだけである。また、上位概念を持つフレーム、例えば、cc_mailは次のように表現される。

```
cc_mail(CC_mail) :- true |
    cc_mail_dstr(CC_mail,[Trec,Crec,Mail]),
    slot(to_rec,Trec,(1,NIL),person,unknown),
    slot(cc_rec,Crec,(1,NIL),person,unknown),
    mail([erase(recep)|Mail]).
```

cc_mail_dstrは、To_recにもCC_recにも関係のないメッセージをMailへ送る。

分配器を持つフレームでは、前節の基本的なフレームに比べると、スロットの行う処理が簡単になっている。入力メッセージがどのスロットに受け付けられるべきかの判断を分配器が引き受けてくれるからである。ただし、分配器がメッセージに対してどれだけの処理を行えばよいかは、まだ流動的である。両極端の立場が考えられると思う。

消極的な立場は、分配器が入力メッセージを処理する際に、そのメッセージに無関係であるとわかっているスロット以外の全てのスロットに対してそのメッセージを送ることである。ただし、返答が必要なメッセージを二つ以上のスロットに送る場合には、それぞれのメッセージに異なる返答用の変数を渡してやる必要がある。

積極的な立場は、分配器がメッセージのスロットへの割当のほとんどを行い、スロットの処理の負担を極力軽減することである。

いずれの場合も、もし分配器の行う処理が一般的で、個々のフレームについて固有に定義される必要がないならば、分配器の定義はどのフレームにも共通なものすることもできる。

フレームが分配器を持つことの利点としては、これ以外に次の三点が考えられる。

（１）自分自身へのメッセージの一部をSelfを介さずに分配器自身が扱うことができる。つまり、分配器は自分が管理しているスロットへのメッセージの振り分けを行っており、その間もしくはその後自分自身へメッセージを送りたくなったときには、再帰的に呼び出す分配器の入力ストリームの先頭にそのメッセージを書き込むことによって、自分自身にメッセージを送ることができる。

（２）一つのスロットだけには答えられないような質問を処理することができる。例えば、フレームが持っているスロットの名称の一覧についてとか、い

くつかのスロットの内容を比較することなどの必要な質問に対しても、分配器が個々のスロットに質問を出し、その解答をまとめ挙げることによって答えることができる。

（3）インスタンスに対するスロットの追加は、以前の定義では、インスタンスの生成時にしかできなかった。分配器がある場合には、スロットの動的な生成を分配器に行わせることができる。例えば、次のような具合である。

```
mail_dstr([add_slot(via,_,[surface,air])
                |Mail],Slots) :- true |
    slot(via,Via,_,[surface,air],unknown),
    mail_dstr(Mail,[Via|Slots]).
```

この例では、viaというスロットが生み出されると共に、そのスロットへの入力ストリームがmail_dstrに割り当てられている。実は、このような動的なスロットの追加が起こる場合には、分配器は自分のスロットへのストリームを単なるリストとしてまとめて持っているだけではまずい。リスト内の位置関係によって、どのストリーム変数がどのスロットに対応しているかを同定できないからである。この場合には、分配器は、自分が管理しているスロットへのストリーム変数とそのスロットの名前を対にして持つなどの工夫が必要になる。

4．その他の機能と問題について

前章までで、並列フレームの基本的な考え方と特徴について説明した。本章では、知識表現システムに必要な他の諸機能をどのように並列フレームに取り込めばよいかについて考察する。ただし、ここで考える問題のほとんどは、未解決もしくはよい解決策のない問題で、今後の課題として残されているものである。

4．1　インスタンスの管理者について

フレームシステムにおいては、現在生きているインスタンスに対して、グローバルにアクセスしたいという状況が起こり得る。そのため、新しいインスタンスが生まれる度にそれを登録して管理しているプロセス(instance manager)を設定する必要がある。また、インスタンスからの色々な要望に答えるために、この管理者は、インスタンスをその属性や発生順序（時間）などの情報と共に記録しておかなくてはならない。

しかし、並列論理型言語では、プロセスへのアクセスを行うには、そのプロセスへのアクセスの入口（入力ストリーム）を捉えていなければならない。従って、すべてのインスタンスが、生成時に管理者への入力ストリームを貰わなければならないことになる。不可能なことではないが、管理者への通信のネックが全体の処理に影響を与える恐れがあり、よい解決策とは言えない。

4．2　手続き付加

今までの例には示さなかったが、フレームシステムの基本的な機能として、スロットに対する手続きの付加がある。これに関してはまだ深い考察は行っていないが、基本的にメタコールを用いて実現することを考えている。スロットのfillerに関するデフォルト値やデモンなども同様である。

ＧＨＣ自身にメタコールの定まった仕様がないので、具体的にはどのようなメタコールを用いるか決まっていないが、手続き付加について現在想定しているのは、何のコントロール情報も返さない最も簡単なメタコールで、それで充分であると考えている。success、failなどのコントロール情報を返すメタコールは、むしろ、分配器がメッセージをスロットに割り当てる計算をする際のガード部分の実行に必要かも知れない。

4．3　多重継承について

今のところ、多重継承については、決定的な解は得ていない。勿論、多重継承を禁止するというのも一つの解であるが、もし、多重継承を許すとするならば、二通りの対策を考えている。

一つは、多重継承による多重な解の調停を分配者に委せてしまう方法である。あるフレームが複数のスロットを持つとき、そのフレームのインスタンスの生成時に、両方の上位概念をかまわず展開してしまう。分配者は、入力メッセージを受け取ったときにそれが自分の管理しているスロット宛でないことがわかると、両方の上位概念にそのメッセージを送る。ただし、そのメッセージが返答を要するメッセージである場合には、返答を返すべき変数を別の変数に置き換えて送らなければならない。この後、両方のメッセージに対する返答の取扱いは分配者に委ねられる。両方の返答をリストによってまとめ上げてもよいし、先に返ってきた返答のみを解と考えてもよい。

もう一つの方法は、インスタンスの生成時に複数の上位概念をもつ分配者がある場合、その分配者が両方の上位概念に対してそれらが持っているスロット名の一覧を要求することである。分配者の節で説明したように、これは可能である。もし、両者で重複しているスロットがあることがわかれば、どちらかのスロットに対してeraseメッセージを送って消去してやればよい。この方法は、インスタンスの生成時に要する処理の負担が大きいが、一旦インスタンスが作られてしまえば、そのインスタンスには無駄がなく、結果的にはよいインスタンスの記述を与えてくれる。

4．4　メタ機能

インスタンスの動的な状態を見るのは、前節の説明でもわかるように、分配者に頼むことによって可能である。静的なフレームの記述を見ることは、この問題もまたＧＨＣにどのようなメタ機能を持たせればよいかという問題と係わるので、何とも言えない。ただ一つ言えるのは、ここで紹介したＧＨＣによるフレームの記述は、フレームのスロットおよび上位概念の構造を非常に素直に表現している。従って、ＧＨＣにプログラム節へのアクセスの道ができるなら、簡単なインタプリタによってフレームの静的な記述をデータとして得ることができる。

ただし、インスタンスが他のインスタンスをコントロールするようなメタな機能についてはまだ考察を行っていない。

5．あとがき

並列論理型言語による実現を目指した知識表現法並列フレームについて述べた。フレームのスロット一つ一つが並列に動作可能なプロセスとみなす表現法を提案した。フレームはスロットの集合によって特徴付けられると考えられ、フレームのインスタンスは入力ストリームを共有するスロットの集合として実現された。その主な特徴としては、

（１）フレームのあらゆる要素が並列に動き得る。
（２）継承関係の記述と実現が容易である。
（３）スロットが共通にもつ性質を大局的、統一的に記述できる。
（４）フレームを解釈するインタプリタがなく、並列論理型言語にコンパイルされている。

などが挙げられる。さらに、フレームのインスタンスに入力ストリームの分配者を割り当てることにより、インスタンスに対してメタな処理を行うことができる

前半で、並列論理型言語によるオブジェクト指向言語との比較をしたが、我々の目的はあくまで知識表現である。そのためには、スロットが主体性を持ち、また、多くの情報を保持することができる我々の並列フレームが知識表現言語としてのポテンシャルが高いと思われる。ただし、前章でも述べたように、知識表現の枠組として完成するためには解決すべき問題が多く残されている。

また、並列フレームの設計を通じて、我々のプロジェクトの核言語の詳細仕様決定へフィードバックさせることもこの研究の目的の一つである。

【謝辞】

本テーマについて議論またはコメントをいただいた次の方々に特に感謝したい。ＩＣＯＴ古川康一次長、大木優氏、宮崎敏彦氏、吉田かおる嬢。また、伊藤英則室長はじめＩＣＯＴ第一研究室諸氏の日頃討論並びにサポートに感謝したい。

【参考文献】

[Brachman 85] Brachman,R.J. and Schmolze,J.G., "An Overview of the KL-ONE Knowledge Representation System," Cognitive Science Vol.9, No.2, pp.171-216, 1985.

[Clark 84] Clark,K. and Gregory,S., "PARLOG: Parallel Programming in Logic," Research Report 84/4, Department of Computing, Imperial College, London, 1984.

[Kahn 86] Kahn,K. et al., "Vulcan: Logical Concurrent Objects," Xerox Palo Alto Research Center, 1986.

[Minsky 75] Minsky,M., "A Framework for Representing Knowledge," in 'The Psychology of Computer Vision, P.H.Winston(ed.), pp.211-277, McGraw-Hill, New York, 1975.

[大木 87] 大木優、竹内彰一、古川康一、「並列論理型プログラミング言語ＫＬＩを基にしたオブジェクト指向プログラミング言語」本書内.

[Shapiro 83] Shapiro,E., "A Subset of Concurrent Prolog and its Interpreter," ICOT Technical Report, TR-003, 1983.

[Shapiro & Takeuchi 83] Shapiro,E. and Takeuchi,A., "Object Oriented Programming in Concurrent Prolog," New Generation Computing, Vol.1, No.1, 1983.

[Ueda 85] Ueda,K., "Guraded Horn Clauses," Proc. The Logic Programming Conference, ICOT, 1985.