TM-0145

# Message Based Module Structure for Parallel Logic Languages

岸下　誠 (富士通)
田中二郎 (ICOT)

November, 1985



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

# Message Based Module Structure for Parallel Logic Languages [1]

田中二郎　　岸下　誠
(ICOT)　　(富士通　国際研)

我々は、並列論理型言語のモジュール化構造について考察を行い、分散インプリメントに適した、メッセージ通信に基づくモジュール化構造を提案した。このモジュール化構造においては、super とpartの二種類の階層を定義することができ、モジュール間の連絡は主としてメッセージ交換で行われる。我々はこのモジュール化構造を、我々の開発したExtended Concurrent Prolog(ECP) の逐次処理系上に実現した。

## 1. 並列論理型言語

並列論理型言語とは論理型言語Prologを並列実行可能なように拡張したものである。近年、多数台のprocessor を結合させた、いわゆる非ノイマン型計算機の提案とともにParlog[Clark 85]、Concurrent Prolog[Shapiro 83a]、GHC[Ueda 85]などの並列論理型言語が提案され今日にいたっている。

これらの言語は、それぞれ微妙に異なっているが、概ねの言語仕様には共通点も多い。本稿ではこういった並列論理型言語のモジュール化構造について考察を行なう。

## 2. モジュール化

プログラムのモジュール化とは、プログラムをなんらかの形で分割して記述することにより、プログラムに局所性、階層性などを持ち込むことである。論理型プログラムのモジュール化についての概観、議論などに関しては[ICOT 84] などに詳しいが、プログラムのモジュール化の目的として以下のような要因が考えられる。

①ソフトウェア工学的側面。プログラムの作成、管理、保守などを容易にする。大規模プログラム、大人数プログラミングを可能にしたい。

②知識工学的要請。知識の構造化、多世界化など主としてプログラム能力の向上を目指すもの。

③分散処理的要因。プログラムを分散化された環境で実行しやすくするため。

これらの要因であるが、①のソフトウェア工学的側面については、いままで、『構造化プログラミング』として、『抽象データ型』とか『プログラムの部品化』とか様々な視点から考察がなされてきた。また②の知識工学的要請としては、『フレーム』に基づくシステムや『オブジェクト指向プログラミング』における、インヘリタンスの実現などで考察がなされてきた。また③の分散処理的要因であるが、これらについては、いままで論理型言語の分散処理が真面目に考えられていなかったせいもあり、あまり前例はない。しかしながらShapiro のLogix [Shapiro 85]など、最終的には分散インプリメントを目指すものと考えられ、今後、分散処理的側面からの考察は重要となってくると考えられる。

この①～③をモジュール化の実現方式という視点から考えると以下のようになる。

| | |
|---|---|
| ↑<静的> | ①ソフトウェア工学的側面 |
| | ②知識工学的要請 |
| ↓<動的> | ③分散処理的要因 |

すなわち①のソフトウェア工学的側面については、staticな、プログラム・コードのレベルの話であり、この段階のモジュール化はコンパイルを行なうことにより消滅する。②の知識工学的要請というのはstaticとdynamic の中間に位置するもので、例えばインヘリタンスなど、適当に前処理を行い、同時に実行時にもそのサポートを行なうのが普通である。③の分散処理的要因については、純粋にdynamic な、プログラムを実行時に、分散した環境のなかで、効率よく走らせるレベルの話である。

## 3. 並列論理型言語のモジュール化

さて、並列論理型言語であるが、並列論理型言語によるプログラミングについてはClark, Shapiro, 竹内 [Shapiro 83b]らにより、ゴール間の共有変数を通じてのstream通信に基づき、各種のプログラミング技法が開発されてきた。

したがって並列論理型言語のモジュール化についても、何かまったく新しい要素を言語仕様の中に持ち込むということではなく、stream通信によるメッセージ交換をもとに、モジュール化構造の在り方について、[Shapiro 85, Furukawa 84b]などを参考にしつつ考察することにする。

論理型言語ではないが、メッセージ交換に基づくモジュール化構造を持つものにsmalltalk [Xerox 81]などのオブ

---

[1] 本研究は第五世代コンピュータ・プロジェクトの一環として行なわれたものである。

ジェクト指向言語がある。本稿では、我々の提案する、並列論理型言語のモジュール階層について、オブジェクト指向言語の用語を使い、説明を行なう。

### 3.1 モジュール化構造

我々の提案するモジュール化構造によればモジュールは以下のように定義される。以下の定義について、{...} は、繰返しをあらわす。

```
"module"    モジュール名
  "part"        {部品のモジュール名}
  "super "      {上位のモジュール名}
  "global"      {グローバルな述語定義}
  "local "      {ローカルな述語定義}
"end."
```

ここで述語定義は関連あるものどうし、モジュールにたばねられ表現されている。また外部参照されないローカルな述語の定義ができるようになっている。すなわちモジュール一つで一つの世界を形成していると考えられる。

オブジェクト指向言語のクラス定義とくらべると、"part" はslotの宣言、"super " はインヘリタンスの宣言、また、述語定義はinstance method の宣言にあたっている。オブジェクト指向言語でいう、いわゆるclass method、class variableについては含まれていない。(class variable については特に不要であると考え、インプリメントしなかった。class methodについては、あればインスタンスの生成などを指示するのに便利である。インプリメントしなかったのは単にそれを怠っただけである。)

### 3.2 クラスとインスタンス

我々のモジュール化構造におけるクラスとインスタンスの関係であるが、まずソース・プログラムにあたるものがクラス定義である。すなわち、クラスとはstaticなものである。

一方、インスタンスとはdynamic なものであり、プログラムの実行時に動的に、create述語により生成される『世界』もしくは『場所』のようなものである(このインスタンスという言葉は、本来のオブジェクト指向的な意味と違って使われていることに留意されたい。オブジェクト指向言語的な意味でのインスタンスに近いのは、むしろ、モジュール内で起動されている述語(ゴール)であろう)。

本モジュール階層は分散インプリメントを『にらんで』設計されたため、特に生成される『場所』としてのインスタンスを強調したものになっている。分散環境上では一つのクラス定義より、幾つものインスタンス(実行場所)が形成され、それらが独自に処理を行なっていく。分散インプリメントを考えない場合、インスタンスはあまり大きな意味を持たない事に留意されたい。

### 3.3 super 階層

super 階層とはオブジェクト指向言語の用語でインヘリタンスと呼ばれるものである。インヘリタンスとは、あるモジュール内の計算において、定義されていない述語定義が現れたとき、より上位のモジュールの述語定義を捜し、それを使用する機能である。これは一種のscope ruleであり、知識工学などで、default の推論、概念を特殊化していく際などに使われるものである。

我々は知識工学などで一般化しているmultiple inheritanceを採用した。我々の言語は並列論理型言語であるので、定義されていない述語定義が現れたとき、super で宣言された上位のモジュールが同時に捜され、最初に見つかった定義が採用される。

問題は、これらのシステムにおいて述語定義の更新はどうなるかという問題である。我々は、述語定義の更新をインスタンスの中にのみ許容することにより有害な副作用の伝播を防いでいる。

### 3.4 part階層

partとは、知識工学の用語でいうスロットにあたるもので、モジュールの部品にあたるものである。

モジュールは仕事(ゴール)を受取ったとき、必要に応じ、その部品に仕事の一部(サブ・タスク)を渡す。

部品としては、二つの可能性がある。一つはモジュールを部品としてとる場合である。これはサブ・タスクの作業場所を定義することにあたる。

もう一つの可能性はpart-module を部品としてとる場合である。モジュールの中で状態を持っているのはモジュールの中のゴール(普通、並列論理型言語のゴールはperpetual なprocess であり、当分生き続ける)であり、モジュールそのものではない。したがって共有部品によりモジュール間で状態(変数)を共有したいときはこれでは困る。そこで考えられたのがこのpart-module である。

part-module はShapiro のLogix のmonitor にあたるもので[Shapiro 85]、この特殊なモジュールにおいては、module名と同じ名前の、たった一つの、グローバルな述語定義しか許されない。part-module には "part#Message" の形で、述語定義の中の変数に直接メッセージを送ることが出来る。

モジュール(インスタンス)が動的に生成されるとき、partの生成については3つの可能性がある。

①部品のモジュールを新たに生成する。

②すでに生成したモジュールを指定し、部品とする。

③partは空にしておく。

これらはすべてcreate述語により自由に指定することができる。

partは本体のモジュールより部品のモジュールへのリンク（ポインタ）と考えられる。従って、他のモジュールと部品を共有したり、特殊なデータ構造を作ったりすることができる。partの内容はset-part述語によりモジュール内部で動的に変更ができ、それを利用すればポインタを動的につけかえ、データ構造を動的に変化することも可能である。

## 4. Extended Concurrent Prologのモジュール化

我々は、1984年の核言語第１版概念仕様書[Furukawa 84a] に基づき、 AND並列及びOR並列機能、集合抽象化機能、メタ推論機能などを持つExtended Concurrent Prolog (ECP)を試作した[Fujitsu 85, Tanaka 85] 。

我々は、このECP に、前節でのべたモジュール化機能を実装した。ECP を選んだのには深い理由はなく、単に我々が今までECP をimplement してきたという歴史的な理由による。従ってGHC やParlogなどにおいても、このモジュール化は有効であろう。

### 4.1 モジュールの実装

我々は、モジュールの実装にあたっては、モジュール間のメッセージ交換をもとにモジュール化を実現した。

まずソース・プログラムは前処理され、各モジュールごとにmodule manager (mm) が付加され、systemへのoutput stream が陽に記述された形に書き直される。Shapiro のLogix [Shapiro 85]を参考に、述語定義の変換例を以下にしめす。（述語定義のガード部については簡単化のため省略してある。）

| 変換前 | 変換後 |
|---|---|
| p:- part#Mes. | p([Inst#Mes]):-<br>table(part,Inst). |
| p:- q. | p(S):- q(S). |
| p:- part#Mes,r. | p([Inst#Mes \| S]):-<br>table(part,Inst), r(S). |
| p:- q,r. | p(merge(S1) \| S2]):-<br>q(S1),r(S2). |

ここで、モジュールの外にmessage を送る可能性のあるゴールには、すべてsystemへのoutput stream が付加されている。

一つ一つのモジュールは、partと実際のinstanceとの対応table を持っており、partが呼ばれたときには対応table を引き、実際のinstanceにメッセージが送られる。

ある述語が二つ以上のゴールを呼ぶときには、その一つ一つにsystemへのoutput stream を付加してやらねばならない。ただし自分のもらっているstreamを分割したのでは段々とタコ足配線になるので、systemへお願いして新しいoutput stream をもらうようにしてある。

systemは、モジュールからのmessage を集め、適切なモジュールのmodule manager (mm) に配達する。mmは、systemからmessage を受取り、それを実行する。

一般にmodule managerはモジュールがcreateされた時点で自動的に起動される。part-module の場合にはmessageが述語の中の変数に直付になっており、part-module がcreateされた時点で述語が自動的に起動される。

こうしたモジュール構造の実行イメージを図１に示す。

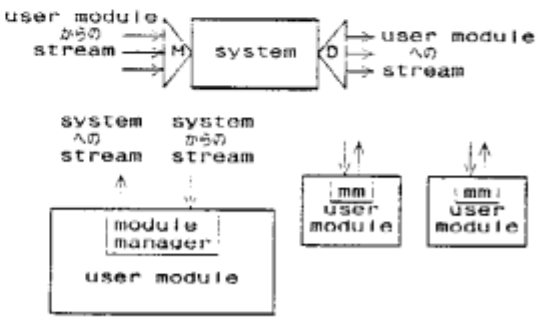


図１ モジュール構造の実行イメージ

### 4.2 階層構造の実現

我々のモジュール化においては二種類の階層構造が現れる。即ち“part”によって作られる“part-of ”の関係と、“super ”によって作られる“is-a”の関係である。

“part-of ”関係は、基本的には、あるモジュール（インスタンス）が部品(part)に仕事の一部(sub-task)を、メッセージの形で渡すことにより実現される。

一方、“is-a”関係は、モジュールの中に述語の定義がなされていないとき、より上位のクラスに定義を要求するという形で実現される。これは述語定義を自分の場所に引きいれるだけであり、あくまでも計算は自分の環境の中で進行していく。（なお現在のimplement では、ガード部の述語については、すべてローカルなものと想定し、inherit を許していない。これはECP が、普通、ガード部で環境を外に公開できないという言語仕様上の理由による。）

一般にモジュールのインスタンスはcreateされたとき、何の述語定義もassertされていない。計算が進んでいくに従い、インスタンスの中に述語定義が次第にふえていく(lazy fetch 方式）。

### 4.3 プログラムの起動

プログラムの起動は、まずtop level で使用するモジュ

ール( クラス) 群を宣言する事より始まる (この段階で使われるクラスのmodule managerは起動される )。つぎにそれらクラスのインスタンスをcreateし (ここでインスタンスのmodule managerが起動される )、それに幾つかgoalを投げ込んでやればよい。

## 5. まとめ

我々は、並列論理型言語のモジュール化について考察し、super とpartの二種類の階層を持つモジュール構造を提案した。並列論理型言語は逐次prologとは異なり、ゴールがperpetual process の形で、状態を持ち、生き続ける。従ってモジュール化もESP [Chikayama 84]などとは異なったものとなってくる。

我々の提案した“super ”と“part”の二種の階層はそれぞれ異なった性格を持つが、他のモジュールにある述語定義を使うという点で、よく似た面を持つことも事実である。同じようなものが二つあることは混乱の原因でもある。モジュール化の実用にあたっては、このECP のモジュール化を使っての経験及び一層の研究が必要であろう。

## 6. 分散インプリメントに向けて

現在ICOTでは、既に開発されたSIM(Sequential Inference Machine) をメッシュ状に結合し、Message 交換を行ないながら全体として計算を進めていくMulti-SIMを試作中である。(Multi-SIMの実行イメージを図11に示す。)

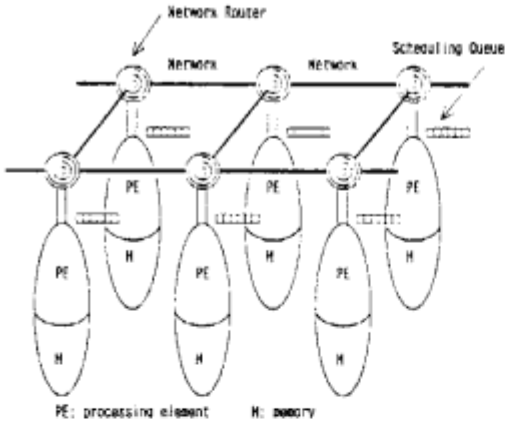


図11　Multi-SIM の実行イメージ

我々はこのMulti-SIM上で、分散インプリメントした並列論理型言語を走らせることを計画している。我々の提案したモジュール構造において、クラスとインスタンスを区別したのは、分散インプリメントをふまえての事に他ならない。このような分散環境下においては、ユニフィケイション一つにとっても、変数の値を得るのに時間がかかり、その間、そのプロセスはサスペンドするとみなければならない。分散インプリメントは既に[Hurakami 85] などで試みられているが、我々は、さらなる考察、実験、研究を行なっていく予定である。

## 7. 謝辞

本研究は、第５世代コンピュータ・プロジェクトの一環として行なわれたものである。本研究の一部は、ICOT第一研究室の竹内氏、上田氏、宮崎氏らと行なった核言語第一版のモジュール化の議論をふまえている。なお、ECP の試作は富士通、国際研の横森氏と共同で行なったものである。本研究の機会を与えて戴いたICOTの古川室長、富士通国際研の北川会長、橋本所長に感謝する。

## 「参考文献」

[Chikayama 84]T.Chikayama: ESP Reference Manual, ICOT Technical Report , TR-044, 1984.

[Clark 85]K.Clark and S.Gregory: PARLOG: Parallel Programming in Logic, Research Report DOC 84/4 , Dep. of Computing, Imperial College of Science and Technology, Revised 1985.

[Fujitsu 85]富士通: 59年度電子計算機基礎技術開発再委託成果報告書、5G核言語第１版の検証用ソフトウェア・詳細仕様書（第２版）及び試験評価データ Part I , 1985.

[Furukawa 84a]古川、国藤、竹内、上田: 核言語第１版概念仕様書、ICOT、1984年 3月.

[Furukawa 84b]K.Furukawa et al.: Mandala: A Logic Based Knowledge Programming System , 第五世代コンピュータ国際会議、 613-622、ICOT、1984.

[ICOI 84] 電子計算機基礎技術開発成果報告書、推論サブシステム編、 440-446、ICOI、1984年 3月.

[Murakami 85] 村上: マルチ・プロセッサにおけるユニファイヤ構成法の検討、Multi-SIM検討会内部資料、ICOT、1985.

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

[Shapiro 83b]E.Shapiro, A.Takeuchi: Object Oriented Programming in Concurrent Prolog, New Generation Computing, OHM-sha, Vol.1 No.1, 1983.

[Shapiro 85]E.Shapiro et al.: Logix User Manual for Release 1.1 , July 1985.

[Tanaka 85] 田中、横森、岸下: AND-OR-Queuing in Extended Concurrent Prolog, The Logic Programming Conference '85, 新世代コンピュータ技術開発機構、1985年 7月.

[Ueda 85] K.Ueda: Guarded Horn Clauses, ICOT Technical Report , TR-103, 1985.

[Xerox 81]The Xerox Learning Research Group: The smalltalk-80 system. BYTE, August 1981.