北海道大学工学部研究報告
第167号（平成6年）

Bulletin of the Faculty of Engineering
Hokkaido University, No. 167 (1994)

# Language Embeddingによる並行オブジェクトモデルの分析

渡辺　慎哉　赤間　清　宮本　衛市

（平成5年8月31日受理）

# Analyzing Object-Based Concurrent Models using Language Embedding

Shin-ya WATANABE Kiyoshi AKAMA and Eiich MIYAMOTO

(Received September 31, 1993)

**Abstract**

We have tried to construct a translator which compiles Actor programs into programs based on the Theory of Generalized Logic Programs. It is a part of the trial that integrates a variety of programming languages onto one general theory.

This translator enables us to understand the concept of concurrentprogrammings from the viewpoint of GLP theory. Moreover, it enables us to compare the concepts of Actor with those of other concurrent modelson GLP theory, and also enables us to apply the theory of program transformation to Actor programs.

## 1．はじめに

近年，並行プログラミングの分野において並行オブジェクト指向計算モデルの研究が脚光を浴びている。これは，オブジェクトの持つ独立性が，大規模かつ柔軟な並行計算に適しているためである。さらに，オブジェクト指向モデルの性質であるイベント駆動の概念が，並行計算モデルにおけるメッセージ通信の概念と親和性が高く，並行計算への拡張が容易であったことも理由の一つであると考えられる。

しかし，並行オブジェクト指向計算モデルに与えられた性質は直観的なものが多く，また，オブジェクト合成などを行なうためのプログラム変換の理論も論理型言語の分野に較べて大きく遅れている。従って，既存のモデルのより厳密な比較や分類，および理論の構築は，並行オブジェクト指向計算の分野においては非常に重要な課題である。それによって，通信とは何か，計算とは何か，というような，本質であるにもかかわらず従来は常識として直観的に捉えていた問題を，より厳密に議論し，将来の研究への指針とすることができるようになる。

本研究は，並行オブジェクトモデルの基礎であるActorモデル[2]を，一般化論理プログラム(GLP)の理論に基礎を置くプログラミング言語UL/$\alpha$の上に投影し，並行性・通信などの概念の明確化，およびUL/$\alpha$上での他の並行計算モデルとの比較を試みる。また，さらに，GLPの理論の一般性を活かした並行計算系の可能性，およびオブジェクト合成の有効性についても検討を行う。

情報工学科　言語情報工学講座

## 2．背景と目的

Shapiro は，主に論理型の並行プログラミング言語を対象として Language Embedding を行い，各言語が持つ概念同士の包含関係の調査，分類を試みている[7]。これは各言語相互の1対1の埋め込み関係に基づいたものであるため，言語間の包含関係などを議論することはできるが，並行計算において何が共通の基礎概念なのかを議論することは難しい。共通の基礎概念を議論するためには，一般性の高い理論を通して多くの並行計算系を理解することが重要である。

また，オブジェクトの合成などに必要なプログラム変換の理論に関していえば，既存の並行プログラミング言語の分野は論理型言語の分野に対して大きく遅れている。これは，既存の並行計算モデルの大部分が，経験則に基づいた操作的な意味付けのみを有しており，プログラムの宣言的意味を構築することが困難であることに起因していると考えられる。この遅れを解消する一つの方法は，広範でかつ厳密な理論体系を持つ系に並行計算モデルを投影し，その上でプログラム変換等の操作を行なうことである（図1）。

埋め込みを行なうことのもう一つの利点は，様々な言語の埋め込みの分析によって，より有効な新しい言語やモデルを発案する基礎を与えることであろう（図2）。本研究では，幾つかの言語の通信に関する埋め込みの結果から，より有効な通信の可能性を論じている。

本研究では，このような埋め込みに適したものとして GLP の理論に基づくプログラミング言語 UL/$\alpha$ を選択した。GLP の理論は，Prolog，クラスを扱う Prolog，CLP 言語，ユニフィケーション文法などを統一的観点から論じることのできる広範な理論体系である。我々は，既存の様々なプログラミング言語を，GLP の理論という視点から統一的に説明する試みの一環として，Lisp，関数型プログラミング言語である Miranda，およびオブジェクト指向プログラミング言語である Smalltalk-80の UL/$\alpha$ への宣言的埋め込みを行ってきた[4)5)8)]。その結果，それぞれが持つプログラミングパラダイムが，一般化論理オブジェクトおよび拡張されたユニフィケーションにより，明快に説明され得ることが明らかになりつつある。本研究は，この試みを並行プログラムの世界にも適用し，並行プログラムの様々な概念を GLP の理論の上で明らかにしようとするものである。

## 3．一般化論理プログラム

### 3.1 GLP の理論

GLP の理論は，従来の LP (Logic Programming) の理論を大きく拡張したものとして捉えることができる。LP では，項だけが論理オブジェクトとして許されていた。GLP の理論では，これをプログラミングの世界に登場する任意の構造を許すように一般化し（一般化論理オブジェクト），さらに，それを用いたプログラムに宣言的意味を与えることができる枠組である。

このため GLP の理論は，複雑な対象を扱う広範な問題領域（従来は問題領域毎に個別に対象の複雑さに対応していた）に対して，統一的な理論的基礎を与えることができる。

### 3.2 プログラミング言語 UL/$\alpha$

UL/$\alpha$ は，GLP の理論をベースに考案されたプログラミング言語である。この言語は S 式記法を採用しており，プログラムの節は，

(HEAD <- BODY1 BODY2... BODYn)

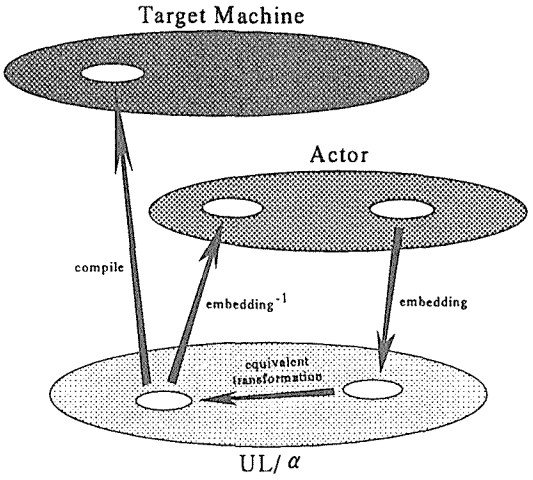


図1：UL/α 上でのプログラム変換

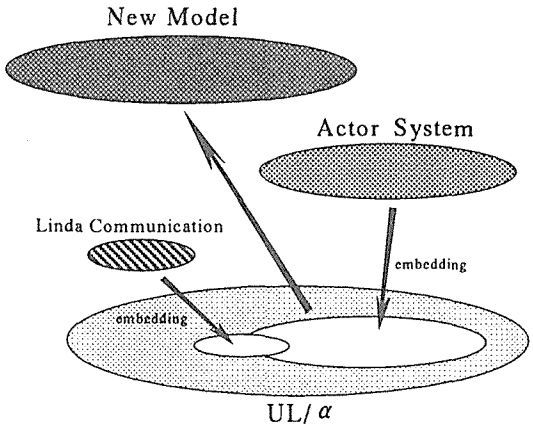


図2：UL/α 上での新しいモデルの構築

という形をしている。変数はシンボルの先頭に'*'が付いたものか，'?'(無名変数)である。また，この言語ではPrologの論理オブジェクトに加えて「情報付き変数」をも論理オブジェクトとして扱う。情報付き変数は，〈変数〉~〈情報〉のように，通常の変数に任意の情報が付加された形のものである。プログラム中に現れる様々な対象は，この情報付き変数を用いて表現される。表現可能な対象は，例えば，

```
*x~dog
```

のような単純なものから，

```
*y~((CLASS Integer)(VALUE 10))
```

のような複雑なもの，また，

```
*p~(PROC *e~(EVENT 1) *p)
```

のような無限構造[6]など広範に及ぶ。

情報付き変数に意味を与えるのが，defUnifyである。これは，情報付き変数同士のユニフィケーションを定義するものであり，これによって，情報付き変数同士の関係が規定される。例えば，

```
((defUnify dog animal dog) <-- )
```

は，?~dogと?~animalはユニファイ可能であり，ユニファイした結果，?~dogになることを表現している。

プログラムに登場する論理オブジェクト，およびこれらの間の「ユニファイ可能」という関係は，プログラムの扱う対象によって異なって然るべきである。UL/αの情報付き変数とdefUnifyは，問題領域に最もふさわしい論理オブジェクトを用いてプログラムを書くための手段をプログラマに提供する。

## 4. 埋め込み

この節では，Actor モデルの構造や動作を示し，それらを保存したまま UL/$\alpha$ のプログラムに投影する。

### 4.1 Actor の構造

Actor は，queue/state/behavior から構成される。このうち，state と behavior は *replacement* によって変化する。queue は identity (*mail address*) を持ち，他の Actor からの参照に利用される。この Actor の構造を UL/$\alpha$ の一般化論理オブジェクトを用いて表現すると，

〈Actor〉 ::= ?~(ACTOR 〈class〉 〈queue〉 〈state〉)
〈queue〉 ::= ?~(QUEUE 〈qid〉 〈contents〉)

のようになる。behavior は実際のプログラム部分に相当するので，UL/$\alpha$ の節形式に展開される。

### 4.2 Actor プログラムの動作の解釈

ある Actor プログラムを実行したときのある時点のスナップショットを *configuration* と呼ぶ。*configuration* はメッセージ（*task*）と Actor の集合である。このとき，Actor プログラムの実行は *configuration* の遷移として定義されている[1]。*configuration* の集合を $C$, *task* の集合を $T$, *Actor* の集合を $A$ としたとき，ある *configuration* における *task* の集合，および actor の集合を求める関数を，

$$tasks : C \rightarrow T$$
$$actors : C \rightarrow A$$

とする。Actor プログラムが $c_1$ という *configuration* にあるとき，*task* $\tau$ を実行して *configuration* $c_2$ に遷移することを，

$$c_1,\ c_2 \in C,$$
$$c_1 \xrightarrow{\tau} c_2 \ \textit{if}\ \tau \in tasks(c_1)$$

と表現する。これは UL/$\alpha$ では，

```
(transit *c1 *c2)
```

と表現することができる。この述語は，*configuration* *c1 の中から実行可能な *task* を見つけ出し，その *task* を実行し，新たな *configuration* *c2 を生成する。

ここで示した動作は並行プロセスのスケジューリングに相当するものである。UL/$\alpha$ では，集合構造を一般化論理オブジェクトとして扱うことにより，スケジューリングを集合のユニフィケーションとして捉えることができる。

UL/$\alpha$ の defUnify を用いて，集合のユニファイアは，

```
((defUnify (SET . *x) (SET . *y) (SET . *x)) <--
        (cut)
        (length *x *lx)
```

```
        (length *y *ly)
        (if (>= *lx *ly)
            (unifySet *x *y)
            (unifySet *y *x)))

    ((unifySet *x *y) <-- (== *x *y) (cut))
    ((unifySet *x *y) <-- (var *x) (cut) (= *x *y))
    ((unifySet *x *y) <-- (var *y) (cut) (= *x *y))
    ((unifySet *x (*ya . *yd)) <-- (extract *ya *x *) (unifySet * *yd))
```

のように表すことができる。この集合構造を用いることにより，本来，集合の意味を持つ *configuration* *c1，*c2 をそのままの形で表現できるようになる。

遷移の具体的内容は，Actor では

$$tasks(c_2) = tasks(c_1 - \{\tau\}) \cup T$$
$$actors(c_2) = actors(c_1 - \{\alpha\}) \cup A \cup \{\alpha'\}$$

と表される。第1の式は，メッセージの受理に関するものであり，あるメッセージキューからメッセージを取り出して実行し，その結果0個以上のメッセージが生成されることを示している。$\tau$ はそのときの *task* であり，はその *task* により新たに生成された *task* の集合である。また，第2の式は，実行による新しい Actor の生成と実行した Actor の *replacement* による Actor 集合の変化を示したものである。ここで，$\alpha$，$\alpha' \in A$，$A \in A$ である。

これらの制約を基に，実際の transit の実現は次のようになる。

```
((transit
    *c1~ (SET *a1~ (ACTOR *class
                          ?~ (QUEUE ?~ (QID *qid)
                                    ((*head . *rest) *tail))
                           . *state)
                   . *config)
    *c2~ (SET *a2~ (ACTOR *newClass
                          ?~ (QUEUE ?~ (QID *qid)
                                    (*rest *tail))
                           . *newState)
                   . *newConfig))
<--
    (*head *a1 *newClass *newState *newActors *targetActors)
    (setMinus *config *targetActors *fixedActors)
    (append *newActors *fixedActors *newConfig))
```

*configuration* *c1 の中から *a1 にユニファイする Actor を選びだし，そのキューの第1要素に相当する behavior を実行する。その後，*configuration* は *c1 に変化する。このとき，*a1 から *a2 への変化は，behavior 中の become 命令による *replacement* を表している。すなわち，上の

式の $\alpha$ から $\alpha'$ への変化である。また，*newActors は $A$ に相当し，*targetActors は $T$ を受理した Actor の集合に相当する。

以上のように，UL/$\alpha$ への埋め込みによって，スケジューリングが集合のユニフィケーションとして定義できることが明らかになった。

### 4.3 Behavior の埋め込み

#### 4.3.1 Actor の文法

今回埋め込みの対象とした Actor 言語の文法を以下に示す。

```
<act program>              ::= <behavior definition>* (<command>*)
<behavior definition>      ::= (define (<id> <(with <pattern>)>*)
                                <communication handler>*)
<communication handler>    ::= (Is-Communication (a <pattern>) do <command>*)
<command>                  ::=   <let command> |
                                 <conditional command> |
                                 <send command> |
                                 <become command> |
                                 <new command>
```

#### 4.3.2 send

send 命令はある Actor にメッセージを送る命令である。具体的には標的となる Actor のキューにメッセージを格納する。従って，task 集合に変化を与えることになる。この命令は，次の述語に変換される。

(enqueue *targetActor *enqueuedActor)

また，さらに 1 つの behavior に複数の send 命令が存在する場合があるため，メッセージが格納された Actor をリストにまとめる述語も同時に生成される。

#### 4.3.3 become

become 命令はそれを実行した Actor の *replacement* を行なう。このとき変化するのは所属クラスと状態である。mail address は変化しない。従って，次のような 2 つの述語に変換される。

(= *newClass <class>)
(= *newState <patterns>)

#### 4.3.4 new

new 命令は Actor 集合に新しい Actor を加える。これは，次の述語に変換される。

(new *actor <class> <patterns>)

この述語は <class> と <patterns> を持った新しい Actor を生成し，*actor に返す働きをする。さらに，1 つの behavior に複数の new 命令が存在する場合があるため，生成された Actor をリストにまとめる述語も同時に生成される。

#### 4.3.5 例

ここでは簡単な例を用いて変換を行なってみる。次のプログラムはスタックを表している（紙面の都合上一部だけを記載している)。

```
(define (stack (with content *c)
               (with nextNode *n))
  (Is-Communication (a push (with content *v) do
      (become
        (stack (with content *v)
               (with nextNode
                     (new stack (with content *c)
                                (with nextNode *n)))))))))
```

このプログラムを変換する場合，まず，スタックの構造が記憶される。

```
((stack (with content *c) (with nextNode *n)) <-- )
```

behavior は上で述べた各命令の変換方針に従って UL/$\alpha$ の節形式に展開される。

```
((push ((with content *v))
       ?~ (ACTOR stack ? (with content ?))
       *newClass
       *newState
       *newActors)
  <--
    (new *actor stack (with content *c) (with nextNode *n))
    (= *newClass stack)
    (= *newState ((with content *v) (with nextNode *actor)))
    (= *newActors (*actor)))
```

## 5．考　　察

### 5.1 Actor の合成

Actor に代表される Object-Based Concurrent Model では，インスタンスが実行の単位である。従って，効率の向上のためのオブジェクト合成を目指すならば，インスタンスの合成/分解を考える必要がある。このとき問題となるのが *mail address* の変化である。いま，2 つの Actor $a_1$ と $a_2$ を考えた時，これらの Actor はそれぞれ別の *mail address* $m_1$，$m_2$ を持ち，他の Actor から参照されている。この 2 つの Actor の合成が他の Actor に影響を及ぼさないためには，合成された新たな Actor $a_{12}$ が $m_1$，$m_2$ の両方の *mail address* を持たなければならない。

UL/$\alpha$ へ変換されたプログラムの実行においては，GLP のユニフィケーションによってこの問題を簡単に解決することができる。例えば，それぞれ?~ (QID 1) と?~ (QID 2) という *mail address* を持った 2 つの Actor を合成した時，それによってできる新たな Actor の *mail address* を?~ (QID 1 2) とする。このとき，?~ (QID 1 2) と?~ (QID 1) または?~ (QID 2) がユニファイするようにユニフ

ァイアを設定すればよい。

((defUnify (QID . *a) (QID . *b) (QID . *a)) <-- (subset *a *b))

### 5.2 他の並行計算モデルとの比較

ここでは,Actor と他の並行計算モデルとの比較を通信を中心に議論する。他の並行計算モデルの埋め込みにより,通信は通信相手の選択,および情報の伝搬の2つに分離されることが分かった。その観点から考察すると,Actor モデルは

**通信相手の選択** *mail address* による一意性
**情報の伝搬** パターンマッチ

という特徴を持つ。これに対して Linda は,タプルを用いることにより,相手の選択をパターンマッチで決定できるという多様性を持っている。この点で,Actor モデルはアルゴリズムのしっかりした問題に対しては対処できるが,より柔軟な構造を有する問題には適用が難しいと考えられる。

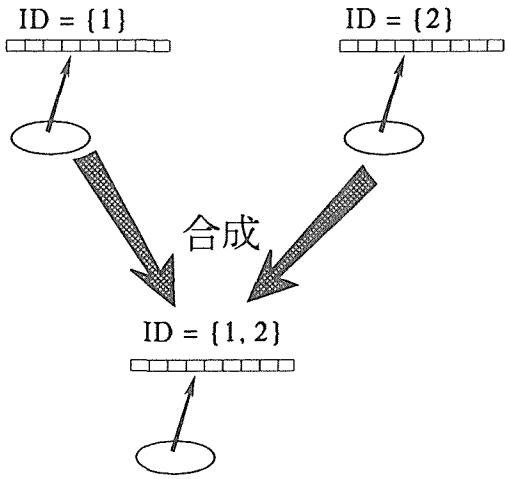


図3 : 集合型の ID による Actor の合成

この議論をふまえることにより,Actor モデルの通信に関する拡張を考えることができる。すなわち,並行オブジェクトモデルと Linda の通信の融合である。さらには,UL/$\alpha$ の持つ拡張されたユニフィケーションに基づき,より高機能な通信概念を導入することも可能である。そのモデルは現在考案中であり,次の機会に報告する予定である。

## 6. おわりに

本研究では,Actor モデルを UL/$\alpha$ で表現することにより,並行計算系の理論的な把握を試みた。それによって,Actor の合成が GLP の理論の枠組で行なえる可能性を検討できた。また,他の並行計算モデルとの通信の表現力の差異が把握できた。また,複雑なオブジェクトを扱えることは,問題の記述力を高めることも確認できた。

今後は,Actor モデルの合成やプログラム変換に対する考察を行ない,並行論理モデルなどの埋め込みなど,より広い範囲での並行計算系の定式化を行っていきたい。

## 参考文献

1) G. Agha and C. Hewitt: Concurrent Programming Using Actors, Object-Oriented Concurrent Programming, MIT Press, pp. 37-54 (1988).
2) C. Hewitt: Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence, Vol.8, No.3, pp. 323-364 (1977).
3) Kiyoshi Akama: Sufficient Conditions of Two Inference Rules for Generalized Logic Programs, Proc of LPC'91, pp. 161-170 (1991).
4) 繁田良則，赤間清，宮本衛市：関数型言語から論理型言語への変換について，日本ソフトウェア科学会第8会大会論文集 (1991).
5) 繁田良則，赤間清，宮本衛市：ユニフィケーションによる関数型プログラムの実行，情報処理学会研究報告92-PRG-6 (1992).
6) 繁田良則，赤間清，宮本衛市：UL/$\alpha$ における無限データ構造，日本ソフトウェア科学会第9会大会論文集 (1992).
7) E. Shapiro: Separating Concurrent Languages with Categories of Language Embeddings, Technical Report CS91-05, The Weizmann Institute of Science (1991).
8) 渡辺慎哉，赤間清，宮本衛市：オブジェクト指向プログラムから一般化論理プログラムへの変換，情報処理学会プログラミング言語・理論・基礎実践研究会7-6 (1992).
9) N. Carriero and D. Gelernter: Linda in Context, CACM, Vol. 32, No. 4, pp. 444-458 (1989).
10) C. A. R. Hoare: Communicating Sequential Processes, Prentice-Hall (1985).