ソフトウェア基礎論　23－3
(1987. 12. 10)

並列オブジェクト指向言語処理系の実現方式の導出

－－プログラム変換に基づくアプローチ－－

柴山悦哉

東京工業大学理学部情報科学科
東京都目黒区大岡山2－12－1

並列分散処理環境下における並列オブジェクト指向言語処理系の実現方式を検証する一手法を提案する。この手法はプログラム変換技法に基づくもので、まず、単純な処理系実現方式をモデル化する並列オブジェクト系を構成する。次に、この系にプログラム変換をほどこし、現実的処理系実現方式をモデル化する並列オブジェクト系を導く。このプログラム変換の過程は意味を保存するものであり、導出された現実的処理系実現方式が、最初に与えた単純な方式と意味的に同等であることが示される。

我々のプログラム変換技法は、主として並列オブジェクトの融合と分割に基づくものである。各変換規則の正当性は形式的に証明される。

How to Invent Distributed Implementation Schemes for Object-Oriented Concurrent Languages

— A Transformational Approach —

Etsuya SHIBAYAMA
Department of Information Science,
Tokyo Institute of Technology,
Oh-Okayama, Meguro-ku, Tokyo, Japan, 152

Abstract: A verification technique for implementation schemes on distributed environment is presented. In this technique, first, naive implementation schemes are modelled by concurrent object systems, that is, those which are constituted of computational agents with capability of concurrent execution and message passing. Then, each of these concurrent object systems are transformed into another concurrent object system which models a more sophisticated implementation scheme.

Our transformation technique is mainly based on fusing and splitting concurrent objects. The correctness of transformation rules can be proven in a formal manner.

1 はじめに

並列に動くオブジェクトから構成される系の最適化・検証等を考える時、このような系に対して、その意味を不変に保つ代数的操作（変換）が行えると便利である。もし、この代数的操作によって系の効率が向上すれば、それはすなわち最適化である。また、効率は悪くてもより単純な系に変換することができれば、検証の手間を削減できるであろう。

並列環境で効率よく実行するために複雑な実現方法を用いているが、逐次的な実現は比較的簡単な問題がよくある。このような時、代数的操作による系の単純化が検証・バグの発見等に威力を発揮するものと思われる。

我々は［柴山 87］において、並列オブジェクト指向言語で書かれたプログラムに対するいくつかの変換技法とその適用例を示した。具体的には、並列オブジェクトの融合と分離を基本とし、さらに簡略化を組み合わせた手法である。また、並列オブジェクト指向言語に対するプログラム変換の正当性を形式的に定義し、この正当性の定義に照らし合わせて、変換規則の正当性を証明した。

本稿では、［柴山 87］で示したプログラム変換技法の実際的適用例として、アクターモデルを分散システム上に実現する方式の検証を試みる。

2 計算モデル

本稿で対象とする計算モデルとしては、我々が提唱した並列オブジェクト指向計算モデルＡＢＣＭ／１［米澤他 86］とほぼ同様のものとする。ただし簡単のため、ＡＢＣＭ／１のいくつかの機能は削除して考える。

我々の計算モデルにおいて、各オブジェクトは独自の計算能力を持ち、他のオブジェクトと並列に仕事を行えるものとする。共有メモリや大域的な時計の存在は一切仮定しない。オブジェクトは疎に結合された系を構成するものと考えられる。

各オブジェクトは、そのオブジェクトだけから参照・更新できる恒久的内部メモリを持つ。オブジェクトはメッセージを受理することにより次の基本動作の列を実行することができる。

- ・メッセージの送信
- ・メッセージに対する返事の送信
- ・返事の受信
- ・オブジェクトの生成
- ・内部メモリの参照／更新と逐次的計算

オブジェクト内部において、これらの基本動作は、逐次的に実行される。

我々の計算モデルにおいては、メッセージの受理（acceptance）と受信（reception）は異なる概念である。メッセージの受信とは、そのメッセージが、オブジェクトのメッセージキューに到着したことを意味し、メッセージの受理とは、そのメッセージの処理が開始されたことを意味する。メッセージの受信は基本動作と非同期的に行われるが、メッセージの受理はそうではない。デッドロック状態にあるオブジェクトでもメッセージの受信はできるが、受理はできない。

本稿で対象とする計算モデルにおいては、非同期型と同期型の二種類のメッセージ送信が存在する。非同期型のメッセージを送ったオブジェクトは、返事を待たずに続きの仕事を行うことができる。この型のメッセージ送信は、ＡＣＴＯＲモデルにおけるメッセージ送信と類似の性質を持つ。

一方、同期型メッセージを送信したオブジェクトは、そのメッセージに対する返事が返って来るのを待ってから、続きの仕事を行う。これは、同期型の遠隔手続き呼び出し（synchronous remote procedure call）と類似の機構である。ただし、同期型メッセージ送信は、通常の遠隔手続き呼び出しと異なり、メッセージを受信したオブジェクトが他のオブジェクトに返事を返す作業を依頼することを許す。

オブジェクトは、二種類のポート（port）を持つものとする。一つは、メッセージ・ポートと呼ばれ、通常のメッセージが到着するメッセージ・キューのことである。もう一方は、リプライ・ポートと呼ばれ、これは、同期型メッセージに対する返事が到着するポートである。我々の計算モデルでは、メッセージに対する返事もまたメッセージとして実現される。両者の相

違点は、送り先の住所がメッセージ・ポートであるか、リプライ・ポートであるかという点だけである。

我々の計算モデルでは、次のメッセージ送受信順序の保存則を満たすメッセージと満たさないメッセージが共存する。

メッセージ送受信順序の保存則

オブジェクトＡからＢへ二つのメッセージＭとＭ’が送られた時、送信される順番（Ａの局所時計で計る）と受信される順番（Ｂの局所時計で計る）は一致する。

ＡＢＣＭ／１では、このメッセージ送受信順序の保存則をすべてのメッセージに対して仮定する。この仮定により、プログラムの記述が簡潔になることがよくある。しかし、そのために必要以上の決定性が入り込み、プログラム変換による検証を考える際の障害となる場合もまたある。そこで、本稿の計算モデルでは、メッセージの送受信順序は本当に必要な場合だけ保存されるように指定できるものとする。なお、受信されたメッセージは、受信された順番に受理される。

ここで扱う計算モデルは、ＡＣＴＯＲモデルに、(1) 同期型のメッセージ送信機能、(2) 恒久的内部記憶、(3) 送受信順序の保存則をつけ加えたものと考えて差し支えない。

3　並列オブジェクト指向言語

以下では、オブジェクトの記述に言語ＡＢＣＬ／１ [米澤他 86]の記法を使用する。言語ＡＢＣＬ／１では、オブジェクトを図１の構文で定義する。オブジェクトの局所メモリは状態変数として表現される。あるメッセージパターンにマッチするメッセージを受理すると、オブジェクトは記述された基本動作列を実行する。

```
[object <オブジェクト名>
  (state <状態変数宣言>)
  (script
    (=> <メッセージパターン>
        <基本動作列の記述>)
          :
          :
    (=> <メッセージパターン>
        <基本動作列の記述>)))]
```

図１　オブジェクトの定義

基本動作のうち、局所メモリ（状態変数）の参照と逐次的計算は Lisp の表記法を用いて表す。局所メモリの更新は次の代入構文により表現される。

[<変数> := <値>]

同期メッセージに対する返事は、

!<値>

という式により表現される。メッセージ送信に関しては、

[オブジェクト <- メッセージ]

または、

[オブジェクト <= メッセージ]

という構文で、非同期型のメッセージ送信を表現する。二番目の構文で送信されたメッセージおよび同期型メッセージの間では、送受信順序の保存則が成り立つものとする。また、

[オブジェクト <== メッセージ]

という構文で、同期型のメッセージ送信を表現する。さらに、この式は同期メッセージにたいする返事の値をも表現する。同期メッセージ間には、必然的に送受信順序の保存則が成り立つ。

4　並列オブジェクトの変換規則

この章では、本稿で用いるプログラム変換規則を与える。一部の規則の正当性については、[柴山 87] で証明されている。残りの規則に関しても同様の証明を行うことができる。これらの変換規則は、いずれも双方向に適用することができる。

変換規則１

二つのオブジェクトＡとＢが次の条件：

(1) Ａが送ったメッセージは、すべてＢのメッセージ・ポートに到着する。
(2) Ｂのメッセージ・ポートに到着するメッセージはすべてＡから送られる。
(3) Ａが送ったすべての同期型メッセージに対して、Ｂは（デッドロック状態になければ）返事を返す。
(4) ＡからＢへ送られるメッセージ間では、送受信順序の保存則が満たされる。

を満たす（図２）時、ＡとＢを次のようなオブジェクトＡ＋Ｂに置き換える（図３）。

(a) Ａ＋Ｂの局所メモリは、Ａの局所メモリとＢの局所メモリの和集合である。
(b) Ａ＋ＢはＡと同じメッセージパタンを持つ。
(c) Ａ＋Ｂの動作は、Ａの動作中の

[B <= M] と [B <== M]

を「ＢがＭを受理した時の動作」から返事を返す動作を省いたものに、

[x := [B <== M]]

を「ＢがＭを受理した時の動作」中の最初に実行される返事を返す式を x への代入式で置き換えたものに置き換えることにより得られる。

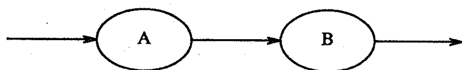

図２　変換規則１適用前

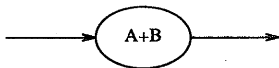

図３　変換規則１適用後

プログラム１は generator と filter という二つのオブジェクトよりなる系を表す。プログラム２は、プログラム１の generator と filter に変換規則１を適用したものである。この場合、generator をＡ、filter をＢと考えると、変換規則１の条件 (1), (3), (4) が満たされることは、簡単にわかる。(2) に関しては、filter が generator 以外のオブジェクトからメッセージを受け取らないという保証が必要であり、filter という名前のスコープをプログラム１の内部だけに限るという仮定が必要となる。

```
[object generator
  (state [n := 1])
  (script
    (=> :start
        (loop
          [filter <= n]
          [n := (+ n 1)]))))]

[object filter
  (script
    (=> number
        (if (evenp number)
            [output <= number]))))]
```

プログラム１

```
[object generator+filter
  (state [n := 1])
  (script
    (=> :start
        (loop
          (if (evenp n) [output <= n])
          [n := (+ n 1)]))))]
```

プログラム２

変換規則２

二つのオブジェクトＡとＢが次の条件：

(1) Ｂのメッセージ・ポートに到着するメッセージはすべてＡから送られる。
(2) Ｂが送るメッセージは、たかだかＡに対する返事だけである。
(3) Ａが送ったすべての同期型メッセージに対して、Ｂは返事を返す。
(4) ＡからＢへ送られるメッセージ間では、送受信順序の保存則が満たされる。

を満たす（図４）時、変換規則１と同様の方法により得られるオブジェクトＡ＋Ｂで、ＡとＢ

を置き換える（図５）。

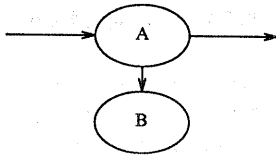

図４　変換規則２適用前

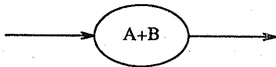

図５　変換規則２適用後

プログラム３は generator と evenp という二つのオブジェクトよりなる系を表す。プログラム４は、プログラム３の generator と evenp に変換規則２を適用したものである。この場合、generator をＡ、evenp をＢと考えると、変換規則２の条件 (2), (3), (4) が満たされることは、簡単にわかる。(1) に関しては、プログラム１の場合と同様、evenp という名前のスコープをプログラム２の内部だけに限るという仮定が必要となる。

```
[object generator
  (state [n := 1])
  (script
    (=> :start
        (loop
          (if [evenp <= n]
              [n := (+ n 1)])))))]

[object evenp
  (script
    (=> number
        !(evenp number)))]
```

プログラム３

```
[object generator+evenp
  (state [n := 1])
  (script
    (=> :start
        (loop
          (if (evenp number)
              [n := (+ n 1)])))))]
```

プログラム４

変換規則３

三つのオブジェクトＡ，Ｂ，Ｃが次の条件：

(1) ＡはＢに送受信順序の保存則を満たす非同期メッセージのみを送る。
(2) ＢはＡからしかメッセージを受け取らない。
(3) ＢはＡからメッセージを受け取るたびにただ一つだけ送受信順序を保存しない非同期メッセージを送る。
(4) Ｂは書き込み可能な恒久的内部メモリを持たない。

を満たす時、ＡとＢを次のように変形しても構わない。

(a) ＡからＢへ送信する非同期メッセージをすべて送受信順序の保存則を満たさないものに変える。
(b) ＢからＣへ送信する非同期メッセージをすべて送受信順序の保存則を満たすものに変える。

変換規則４

二つのオブジェクトＡとＡ’が次の条件：

(1) ＡとＡ’の一時記憶以外のメモリ領域（状態変数）は読みだし専用である。
(2) ＡとＡ’の定義は同等のものである。
(3) ＡとＡ’が送受信するメッセージに関しては送受信順序の保存則は仮定されない。

を満たすとき、ＡとＡ’をただ一つのオブジェクトＡで置き換える。

５　ＡＣＴＯＲモデルの分散実現

この章では、ＡＣＴＯＲよりなる系を分散システム上で実現する方式の検証をプログラム変換技法を用いて行う。

５．１　ＡＣＴＯＲモデル

本稿で採用した計算モデルは、自然にＡＣＴＯＲモデルの拡張と考えることができる。ＡＣ

ＴＯＲモデルにおいては、各アクターは、非同期型のメッセージ送信しか行わず、しかも、あるメッセージの処理中に送信されるメッセージの総数は高々有限個である。また、各アクターは、書き込み可能な恒久的内部メモリを持たない。さらに、送受信順序の保存則は一切仮定されない。

今、簡単のため実行時における動的なアクターの生成は考えないものとする（動的にアクターを生成する変わりに、すでに存在するが活性化されていないアクターに、初期化メッセージを送ると考えても、問題ない場合が多い）。

以降では、あるアクターのプログラムを固定して考える。このプログラムに出現するアクターの集合を Act、メッセージの集合を Mess で表す。この時、アクターの動作は、次の条件を満たす関数 $f: Act \rightarrow Mess \rightarrow (Act \times Mess)*$ によって、完全に表現される。

$$f(A)(M) = [[A_1\ M_1]\ [A_2\ M_2]\ ...\ [A_n\ M_n]]$$

iff

アクター A はメッセージ M の処理中に、A_1, A_2, ..., A_n の各アクターに、それぞれ、M_1, M_2, ..., M_n の各メッセージを送る。

アクター A の挙動をＡＢＣＬ／１で記述するとプログラム５のようになる。

```
[object Actor-A
  (script
    (=> Message
        (foreach [Actor Message']
          in f(A)(Message) do
          [Actor <- Message']))))]
```

プログラム５

ここでは、関数 f が定義中に用いられている。また、アクターが送信しうるのは、非同期のメッセージで、しかも、送受信順序の保存則を満たさないようなものだけである。したがって、「<=」ではなく「<-」を用いてメッセージは送られる。

５．２　静的なプロセッサ割付手法

ＡＣＴＯＲモデルの分散実現方式としては、静的にプロセッサに割り付ける方法が、一番簡単なものあろう。このような実現方式は、プログラム変換を用いて簡単に導き出すことができる。

まず、プログラム５のアクター A は、変換規則２を逆向きに用いることにより、プログラム６のような二つのオブジェクトに分割することができる。

```
[object Actor-A
  (script
    (=> Message
        (foreach [Actor Message']
          in [behavior-of-A <== Message] do
          [Actor <- Message']))))]

[object behavior-of-A
  (script
    (=> Message
        !f(A)(Message)))]
```

プログラム６

プログラム６において、Actor-A はアクターのためのインタプリタであり、behavior-of-A という記述を解釈実行していると考えることもできる。したがって、プログラム６をプログラム５に変換する過程は、behavior-of-A という記述のコンパイルと考えることができる。あるいは、Actor-A というインタプリタに部分計算を適用する過程と言ってもいいであろう。

さて、プログラム中に出現するアクターは、すべて、A のように、インタプリタと動作の記述に分解されたものとする。ここで、f と類似の関数 f' を次のように定義する。

$$f(A)(M) = [[A_1\ M_1]\ [A_2\ M_2]\ ...\ [A_n\ M_n]]$$

iff

$$f'(A)(M) = [[[A_1\ A_1.behavior]\ M_1]\ [[A_2\ A_2.behavior]\ M_2]\ ...\ [[A_n\ A_n.behavior]\ M_n]]$$

ただし、A_i.behavior は、アクター Ai の記述を表現するオブジェクトを表すものとする。すなわち、これからはプログラム６のような二つのオブジェクトの対でアクターを表現するとい

うことである。

この時、プログラム６はプログラム７のように変換することができる。この変換の正当性を、厳密に示すことは可能であるが、結果がトリビアルなわりには過程が長くなるので、ここでは省略する。

```
[object Actor-A
  (script
    (=> Message
      (foreach [[Actor Behavior] Message']
         in [behavior-of-A <== Message] do
         [Actor <- Message']))))]

[object behavior-of-A
  (script
    (=> Message
        !f'(A)(Message)))]
```

プログラム７

A の他にもう一つ B というアクターがあるものとすれば、B は、プログラム８のような形に変形される。

```
[object Actor-B
  (script
    (=> Message
      (foreach [[Actor Behavior] Message']
         in [behavior-of-B <== Message] do
         [Actor <- Message']))))]

[object behavior-of-B
  (script
    (=> Message
        !f'(B)(Message)))]
```

プログラム８

ここで、プログラム７の Actor-A のようなすべてのオブジェクトに変換規則１、３、４を適用することによりプログラム９のような定義が得られる。詳細は省略する。

```
[object Actor-A
  (script
    (=> [Behavior Message]
     (foreach [[Actor Behavior'] Message']
        in [Behavior <== Message] do
        [Actor <- [Behavior' Message']]))))]
```

プログラム９

この段階までくると、Actor-A と Actor-B の定義は同等となり、Actor-A と Actor-Bを融合して一つのオブジェクトとすることができる。ただし、この変換の正当性を証明するにあたっては、[柴山 87] で定義した restriction を、集合 {Actor-A, Actor-B} に対してではなく、集合 {Actor-A, behavior-of-A, Actor-B, behavior-of-B} に対してかける必要がある。

図６は、Actor-A と Actor-B を融合したあとの状況を図示したものである。

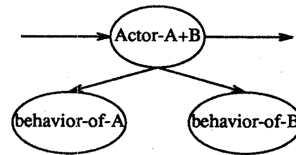

図６

図６に出現する三つのオブジェクト間の通信はすべて同期型のものである。したがって、これらのオブジェクトは完全に逐次的に実行される。この図は次のように解釈することができる：あるプロセッサ内部において、二つのアクターの実行のシミュレーションが behavior-of-A と behavior-of-B によって行われ、外部のプロセッサとの通信サービスを Actor-A+B が行っている。

ここで重要なのは、このような方法で二つ（一般には複数）のアクターを一つのプロセッサに割り当てたとしても、その結果、実行の非決定性が減少することはないという点である。この性質は、プログラム変換に使った各変換規則がすべて非決定性を保つようなものであることから容易に結論できる。

５．３　フォーワード・アドレス

あるプロセス（並列オブジェクトと考えてもよい）を割り付けるプロセッサを動的に変更するための手法としてフォーワード・アドレス技法が知られている。この手法の基本的なアイデアは次のようなものである。

フォーワード・アドレス技法

プロセスＰがあるプロセッサＰｉから別のプロセッサＰｊに移った時、その情報をＰｉは覚えておく。以降、ＰｉはＰに対するメッセージを受け取ると、ただちにＰｊにそのメッセージを配送する。

これは、実生活において、誰かが住所を変更した時、郵便局が手紙の配送を行うのと基本的には同じ原理である。

このようなメカニズムもオブジェクトの融合と分離に基づくプログラム変換の技法をもちいて検証することができる。まず、簡単のため三つのアクター A, B, C のみが存在するものとする（図７）。

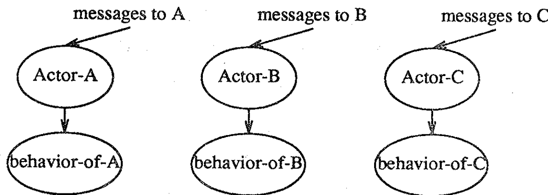

図７

図７の、Actor-A, Actor-B, Actor-C はプログラム９のように定義されたオブジェクトとする。このとき、Actor-B の定義を、プログラム１０のように二つのオブジェクトに分割する。

```
[object Actor-B
  (script
    (=> [Behavior Message]
        [Actor-B' <= [Behavior Message]])))

[object Actor-B'
  (script
    (=> [Behavior Message]
     (foreach [[Actor Behavior'] Message']
        in [Behavior <== Message] do
        [Actor <- [Behavior' Message']]))))
```

プログラム１０

プログラム１０において、Actor-B は送られたメッセージをそのまま Actor-B' に送るパイプのようなオブジェクトである。

プログラム１０のような変換をほどこしたのち、Actor-B' と Actor-C は容易に融合できる（図８）。

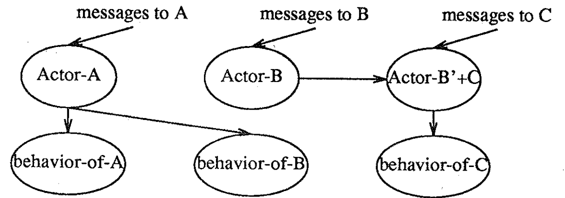

図８

さらに、Actor-A とプログラム１０の Actor-B を融合してプログラム１１のようなオブジェクトを作る。この時の系全体の状態は、図９のようなものとなる。これは、フォーワード・アドレス技法の実現例に他ならない。

```
[object Actor-A+B
  (script
    (=> [Behavior Message]
        where (eq Behavior behavior-of-B)
       [Actor-B' <= [Behavior Message]])
    (=> [Behavior Message]
        where (eq Behavior behavior-of-A)
     (foreach [[Actor Behavior'] Message']
        in [Behavior <== Message] do
        [Actor <- [Behavior' Message']]))))
```

プログラム１１

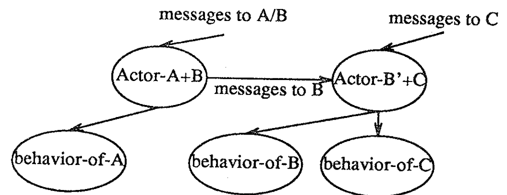

図９

この最後の変換が非決定性を保存するかあるいは減少させるかという問題に関しては、我々も確信をもって答えることができない。ただ、この変換が仮に非決定性を減少させたとしても、「実現方式」としては問題ないであろう。

7 まとめ

本稿では、並列オブジェクト指向言語に対するプログラム変換技法を用いて、アクターモデルの分散実現方式を導出する方法について議論した。この方法は、アクターモデルの単純な実現方式から意味を不変に保って複雑な実現方式を導くものであった。この導出過程は可逆的であり、本稿は逆方向の導出を試みることも当然考えられる。

近年、ユーザからみると単純だが、その実現は複雑なシステムが増えている。例えば、ネットワーク透過性を持ったOSは、ユーザから見れば巨大な単一計算機と変わりない。しかし、その実現は単一計算機上のOSより複雑になっている。また、キャッシュ・メモリや仮想メモリは、（スピードを無視すれば）メインメモリと同じ仕様を持つはずであるが、実現はやはり複雑である。このような状況を考えると、本稿で行ったような検証技法の重要性が理解される。

謝辞

本研究を行うにあたって東京工業大学の米澤助教授に貴重な意見をいただいた。ここに謝意を表します。

参考文献

[Agha 84] Agha, G.: Actors: Amodel of Concurrent Computation in distrivbuted Systems, MIT Press, 1986.

[America他 85] America, P. et al.: Operational Semantics of a Parallel Object-Oriented Language, Repost CS-R8515, Dept. of Computer Science, Centre for Mathematics and Science, 1985.

[Burstall & Darlington 77] Burstall, R. M. and Darlington, J.: A Transformation System for Developing Recursive Programs, JACM, Vol.24, No.1, 1977.

[Clinger 81] Clinger, W.: Foundations of Actor Semantics, Ph.D Thesis, Dept. of Math., MIT, 1981.

[Milner 80] Milner, R.: A Calculus of Communicating Systems, Vol. 92, Lecture Notes in Computer Science, Springer-Verlag, 1980.

[Milner 85] Milner, R.: A Calculus of Communicating Systems, Lecture Notes in Computer Sceience, Vol. 197, pp. 197-220, Springer-Verlag, 1985.

[Pepper 84] Pepper, P.(ed): Program Transformation and Programming Environments, Springer-Verlag, 1984.

[Shibayama & Yonezawa 86] Shibayama, E. and Yonezawa, A.: ABCL/1 User's Guide, ABCL Project, 1986.

[柴山 87] 柴山悦哉：並列オブジェクト系の変換とその応用，情報処理学会ソフトウェア基礎論研究会資料，87-SF-22, 1987.

[Tamaki & Sato 84] Tamaki, H. and Sato, T.: Unfold/Fold Transformation of Logic Programs, Proc. 2nd Int. Logic Programming Conf., Uppsla, Sweden, 1984.

[米澤他 86] 米澤明憲，柴山悦哉，J.-P. Briot，本田康晃，高田敏弘：オブジェクト指向に基づく並列処理モデルABCM/1とその記述言語ABCL/1，コンピュータソフトウェア，Vol. 3, No. 3, 1986.