並列システムの階層的記述に適した通信機構

福井 眞吾

日本電気(株) C&C システム研究所

並列性を内在したシステムは動作が複雑なため、階層化によって、より単純なサブ問題に分割して記述する必要がある。階層化した場合、親コンポーネントは「タスクのサブタスクへの分割」「サブタスクの実行順序管理」「処理結果の収集」という作業をになう。しかし、従来の並列オブジェクト指向言語が提供するメッセージ通信機構ではこれらの作業を明確に分離して記述できない。そこで、情報送受の指定と実行順序の指定を独立に記述できるメッセージ通信機構を提案し、ロボットアーム制御問題を記述することでその有効性を示す。

Communication Mechanism for Hierarchical Concurrent Systems

Shingo Fukui

C&C Systems Research Laboratories, NEC Corporation
1-1, Miyazaki 4-Chome, Miyamae-Ku, Kawasaki, Kanagawa 216, Japan

Since the behavior of concurrent systems is very complicated, it is important to describe them hierarchically by dividing them into simple and small sub-systems. Parent components in hierarchical systems take charge of the following jobs.

1. Division of tasks
2. Management of sub-task execution order
3. Collection of sub-task results

The message passing mechanism proposed until now can not give a well separated description of these jobs. We propose new message passing mechanism which allows us to describe information exchanges and execution order separately. Program examples of the robot arm control with this mechanism are presented.

1 はじめに

大規模で複雑な問題を解くには、その問題をより小規模で単純なサブ問題に分割し、それらを個別に解くことによって全体の解を得る「分割統治法」が有効である。ソフトウェアの開発でもこの方式は有効である。大規模で複雑なシステムを記述する場合、機能によっていくつかのモジュールに分割し、そのモジュールをさらに分割していく「モジュール分割」が行われる。この手法で得られるソフトウェアは、モジュール分割に対応した階層構造を持つことになる。

多くのプログラミング言語は、階層的なモジュール構造を記述するためのメカニズムを提供している。たとえば、C,Pascal に代表される逐次型の手続き型言語では、サブルーチンによってモジュールを表現することができる。サブルーチンは、別のサブルーチンの呼び出しの集まりとして構成されるので、自然にシステムの階層性を表せる。関数型言語でもサブルーチンと同じように関数で階層構造を表現できる。

逐次型のオブジェクト指向言語 [4] においては、データ構造と手続きを一体化したオブジェクトが唯一与えられた構造である。プログラムはメッセージによって互いに通信するオブジェクトの集まりで記述される。通常、オブジェクトはそれ以上分割できない単位であり、オブジェクトをオブジェクトの集まりとして表現することはできない。したがって、与えられた問題を階層的に記述することが困難に思われる。しかし、実際には、メッセージ通信がサブルーチンコールと同じメカニズムによって実現されているため、逐次型の手続き型言語と同じように階層を表現することができる。

これに対し、並列オブジェクト指向言語においては、メッセージ通信はサブルーチンコールと異なるメカニズムによって実現されている [1][2][5][9] ので、従来の手法に基づいて階層を表現す ることはできない。

しかし、階層構造の表現力は、複雑な問題を記述する上で有効であるのみならず、デバッグ [6]、資源配分、ガベージ・コレクションなどにおいても役立つと考えられるので、階層表現機構を検討する必要がある。

本論文では、まず、並列オブジェクト指向言語で階層構造を表現する場合の問題点を考察し、親オブジェクトが子オブジェクトを管理する上で処理の終了を認識することが重要であることを述べる。次に、処理の終了を扱う通信機構を備えた並列オブジェクト指向言語を提案する。この言語を用いて、階層性が内在すると考えられるロボットアームの制御問題を記述し、この言語の有効性と限界について考察する。

2 並列オブジェクト指向言語による階層表現

本節では、従来提案されている並列オブジェクト指向言語で階層構造を記述する場合の問題点を考察し、記述のために必要と考えられる言語機能を明らかにする。

階層構造の最も単純な例として、図 1に示すような 2 層構造を記述することを考える。全体は、a というモジュールであり、a は b, c というモジュールから構成されている。

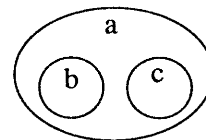


図 1: 単純な階層の例

この構造を 1 対 1 にオブジェクトにマッピングするには、全体に対応する a というオブジェクトと、内部のモジュールに対応する b, c というオブジェクトを準備すればよい。(図 2)

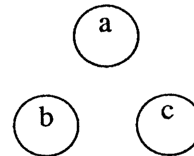


図 2: オブジェクトによる階層の表現

ここで、オブジェクト a に注目してみよう。a は次の 2 つの働きがあると考えられる。

1. a が含まれる階層の中でのオブジェクトとしての仕事
2. b, c の親としてそれらの動きを制御、管理する仕事

階層構造を表現するには特に 2. が重要である。オブジェクト a は外部から依頼された仕事を b, c に割当てて実行させなければならない。また、b, c は並列に動作するので、その実行順序を制御しなければならない。

並列オブジェクト指向言語においては、別のオブジェクトに処理を依頼することは非常に簡単である。単にそのオブジェクトにメッセージを送ればよい。しかし、依頼した処理がどうなっているか、あるいは処理の結果を知ることは容易ではない。基本的にはそのオブジェクトからメッセージを受け取ることでこれを認識しなければならない。

しかし、この方式では終了通知を受け取るためのメソッドを、受け取りたい終了メッセージの種類だけ準備しなければならず、オブジェクトの記述が繁雑になる [2]。それを回避する手段として、多くの並列オブジェクト指向言語は遠隔手続き呼び出し型 (RPC 型) のメッセージ通信を用意している [9]。RPC 型通信を用いると相手オブジェクトの処理が終了したことと、処理結果 (返事) を同時に知ることができる。

RPC を行っている間は、送信者は待ち状態になり作業を行うことができなくなる。例えば a が b の処理終了を認識したい場合、a は b に RPC すればよいが、その場合 a の処理は中断される (図 3)。この方式では b,c の両方に処理をおこなわせ、両者の終了を待つということができなくなる。

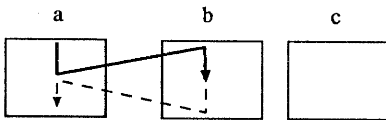


図 3: RPC 型通信による待ち状態

この困難を解決する方法として、値の非同期受取を可能にする future が提案されている [2][9]。これを使えば、複数のオブジェクトにメッセージを送信しておき、必要になった時点で返事を待つことで同期を実現できる。

しかし、future を用いて子オブジェクトの実行を管理しようとすると、終了を知らせてもらうための future をメッセージ引数に加えて各子オブジェクトに送り、その future の値が確定するのを待つコーディングになり、大変煩雑で全体の処理の流れが分かりにくいプログラムになる。

future を使う方式では、返事を受け取る副作用として処理終了を認識することになる。見通しのよいプログラムにするには、返事の受取とは独立に、かつ、明示的に処理終了を表現できる構文を用意する必要がある。

3 実行順序制御機構を強化した並列オブジェクト指向言語

ここでは、処理終了を明示的に扱える通信機構を備えた並列オブジェクト指向言語を提案する。この言語はメッセージ送信によるメソッドの起動と、future による値の非同期な受取りを基本としている。

メッセージ送信文 (send 文) は次のような形をしている。

```
[send target message]
```

target を評価して得られたオブジェクトに対して、message を評価した結果を送信するする。たとえば、

```
[send stack (list :put 3)]
```

は、stack という変数 (通常はインスタンス変数) に束縛されているオブジェクトに対して (:put 3) というメッセージを送ることを表現している。

オブジェクトの振舞いはクラスで定義する。クラス定義の概略は次の通りである。

```
(class クラス名 (インスタンス変数 ...)
    メソッド定義 ...)
```

メソッド定義は

```
(パターン　文 ...)
```

という形で記述する。例えば 2 次元座標を表現するクラス Point の定義は次のようになる。

```
(class Point (x y)
    ((:set xx yy)
        (assign x xx)
        (assign y yy))
    ((:add dx dy)
        (assign x (+ x dx))
        (assign y (+ y dy))))
```

メッセージを受信すると、そのメッセージにマッチするパターンを持つメソッドを探し、実行する。メソッド内の文はシーケンシャルに実行される。文には次の種類がある。

assign 文	(assign 変数 式)
if 文	(if 条件式 then 文... else 文...)
let 文	(let (変数宣言) 文...)
case 文	(case 式 (is パターン 文...)...)
parallel 文	(parallel 文...)
sequential 文	(sequential 文...)
send 文	[send 送信先 メッセージ]
	(send 送信先 メッセージ)
	<send 送信先 メッセージ>

parallel 文ではその中に並べられた文を並行に実行し、全ての文の処理が終了したときに parallel 文の実行が終了する。これに対し、sequential 文では、その中に並ばれた文をシーケンシャルに実行する。最後の文が終了したとき sequential 文が終了する。

次に future を用いた値の受信を例を用いて説明する (図 4)。左側のオブジェクト a が右側のオブジェクト b に

(:get 返答先)

というメッセージを送り、答をインスタンス変数 x に束縛するプログラムである。

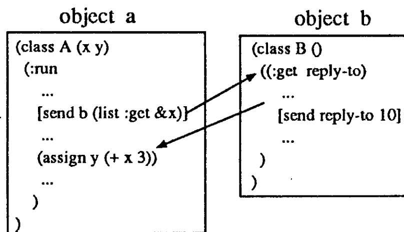


図 4: future による値の受信

& は future 生成子である。&x を評価すると新しい future (一種のメイルボックス) が生成され、それの書き込み用アドレスが評価値としてかえる。それと同時にインスタンス変数 x には生成した future (メイルボックス) の読み出しアドレスが束縛される。future の書き込みアドレスを宛先として send 文を実行するとメッセージが future に渡される (メイルボックスに放りこまれる) 。future の読み出しアドレスが束縛されているインスタンス変数を参照しようとすると (図 4では (assign y (+ x 3)) の部分での x の参照)、すでに future に値が送信されている場合には future の値がその変数に束縛される。future にまだ値が送信されていない場合は、値が送信されて確定するまで参照を行った文の実行がサスペンドされる。値が確定すればサスペンドは自動的に解除される。

send 文には

<send 送信先　メッセージ>
[send 送信先　メッセージ]
(send 送信先　メッセージ)

の 3 種類がある。これは、send 文が終了したと判断する時点の違いを表している。

<send 送信先　メッセージ>	送信が終了した時点
[send 送信先　メッセージ]	メッセージが相手オブジェクトのメッセージ・キューに並んだ時点
(send 送信先　メッセージ)	対応するメソッドの処理が終了した時点

2 番目と 3 番目の終了待ちは暗黙の future を用いて実現されている。メッセージ送信時に、到着時とメソッド終了時を判定するための future 引数が暗黙のうちに付加されて相手オブジェクトに送信されている。メッセージを送信した側はこれらの暗黙の future を参照することによって、メッセージの到着、メソッドの終了を知ることができる。メッセージ到着の通知は、言語のメッセージ転送機構によって実現する。メソッド終了の通知は、メソッドを前処理して通知のための send 文を付加することで実現する。2、3 番目の send 文の、送信側での実現方式を図 5に示す。

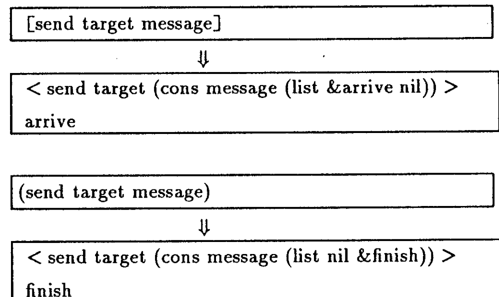


図 5: 終了待ちの実現方式

send 文の待ち合わせ機構と、parallel 文,sequential 文を組み合わせることによって、処理の実行順序を明示的に表現することができる。また、future による値の受渡しを採用しているので、実行順序と値の受け取りを独立に記述できる。

4　例題の記述

本節では前節で提案した並列オブジェクト指向言語を用いて例題の記述をおこなう。

4.1　2腕ロボット制御問題

例題として次に述べる「2腕ロボット制御問題」を選んだ。

> ロボットは2本の腕を持っており、各々独立に動作させることができる。腕は、手と腕木の2つの要素から構成される。手は開閉することで物体をつかめる。腕木は関節を制御することによって手の位置を変えることができる。腕木の位置決めと手の開閉は並列に行える。このロボットには足はなく場所の移動はできない。また、体の方向を変えることもできない。ロボットの前にはテーブルが2つあり、その上に物体がいくつか置かれている。(図6)
>
> このロボットに対して「地点(x,y)にある物体を地点(x',y')に移動せよ」という命令を与える。各腕には到達可能領域があり、指示された2地点が両方とも一方の腕の到達可能領域に属していれば、片腕で物体を移動する。2地点がそれぞれ右腕と左腕の到達可能領域にそれぞれ属している場合には両腕を使って物体を動かす。右腕と左腕の到達可能領域には重なりがあり、その場所で物体を一方の腕からもう一方の腕に渡すことができる。ただし、重なりあっている部分にはテーブルがないので、手から手へ直接渡さなければならない。このロボットに対して次々に移動命令を与える。なるべく効率よく各腕を制御することが課題である。

この問題は、

1. ロボット、腕、腕木、手というように階層性がある。
2. 上の階層に属するオブジェクトと下の階層に属するオブジェクトが並列に動作する必要がある。
3. 構成要素間のインタラクション(右腕と左腕の連携動作)がある。

という特徴をもち、並列システムの階層的記述の検討に適していると考えられる。また、

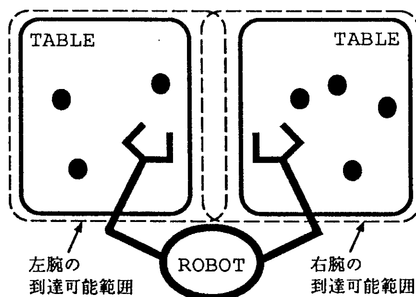


図6: ロボットを上から見た図

1. 腕を増やす。
2. 運んだ物体の重さをカウントする。
3. 重過ぎる場合には運搬に失敗する。

などの条件を増やすことが容易であり、高度な機能の検討にも使用できる。

4.2　構成要素とオブジェクトの対応関係

ロボットを構成するオブジェクトを次のように定めた。

Hand	手を表すオブジェクト。開閉が可能である。
Arm	腕木を表すオブジェクト。位置決めが可能である。
Hand-and-arm	腕を表すオブジェクト。手(hand)と腕木(arm)から構成される。
Robot	ロボット全体を表すオブジェクト。2本の腕(hand-and-arm)を持ている。

オブジェクトの階層関係を図7に示す。

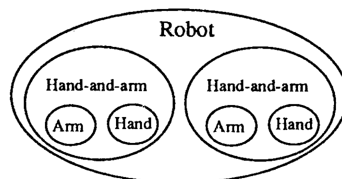


図7: オブジェクトの階層関係

4.3 Hand と Arm

Hand オブジェクトと Arm オブジェクトの仕様は非常に単純である。各オブジェクトが受信可能なメッセージは次の通りである。

Hand

:open	手を開く
:close	手を閉じる

Arm

(:positioning x y)	腕木を (x,y) に位置決めする

Hand と Arm は、実際には歯車やモータなどのより下部の要素から構成されていると考えられるが、本論文ではそれらの機能はあらかじめ与えられていると考えてこれ以上の機能の分割はおこなわない。オブジェクトの外部インタフェースのみを記述することとする。

Hand オブジェクトの定義は次のようになる。

```
(class Hand (status)
  (:init
    **initialize primitives**
    (assing status 'closed))
  (:open
    (if (eq status 'closed) then
      (parallel
        ** call the primitive :open **
        (assign status 'open))))
  (:close
    (if (eq status 'open) then
      (parallel
        ** call primitive :close **
        (assign status 'closed)))))
```

Hand オブジェクトのメソッド :open では手を開く動作と、インスタンス変数「status」を更新する作業を並行に行う。作業が両方終了した時点でこのメソッドが終了する。メソッド :close もほぼ同じ動作をする。

Arm オブジェクトの定義は次のようになる。

```
(class Arm ()
  (:init
    ** initialize primitives **)
  ((:positioning x y)
    ** call primitive (:positioning x y) **))
```

メソッド (:positioning x y) は位置決め動作が終了したときに終了する。

4.4 Hand-and-arm

Hand-and-arm は、手と腕木から構成される腕全体を表すオブジェクトである。このオブジェクトが受信可能なメッセージは次の通りである。

Hand-and-arm

(:take-from x y :to xx yy)	地点 (x,y) にある物体を 地点 (xx,yy) に移動する

Hand-and-arm オブジェクトは、このメッセージを受信すると構成要素である Hand と Arm を起動して機能を実現する。Hand-and-arm オブジェクトの定義は次のようになる。

```
(class Hand-and-arm (hand arm)
  (:init
    (assing hand (new Hand))
    (assign arm  (new Arm)))
  ((:take-from x y :to xx yy)
    (parallel
      (send hand :open)
      (send arm (list :positioning x y)))
    (send hand :close)
    (send arm (list :positioning xx yy))
    (send hand :open)))
```

メソッド :init では、インスタンス変数 hand, arm にそれぞれ Hand, Arm のインスタンスを代入している。

メソッド (:take-from x y :to xx yy) では、まず手を開く動作と腕木の位置決めを並列に実行する。メッセージ送信式が両方とも ( ) で囲まれているので、これはメソッドの終了を待ち合わせることを意味している。2つの動作が終了すると parallel 文が終了する。次に手を閉じ、腕木を目的地点 (xx,yy) に位置決めし、手を開く。これらのメッセージ送信もすべて ( ) で囲まれているので終了を待つ型である。従って全ての動作が終了した時点でこのメソッドが終了する。動作の実行順序を図 8 に示す。

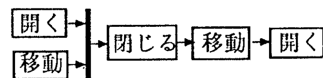


図 8: Hand-and-arm の実行順序

4.5 Robot

Robot オブジェクトは 2 本の腕 (Hand-and-arm) を制御している。外部から与えられた移動命令に従って腕を動かす。従って Robot が処理するメッセージは次のとおりである。

Robot

(:move-from x y :to xx yy)　地点 (x,y) にある物体を地点 (xx,yy) に移動する

まず、移動命令が片方の腕だけで解決できる場合を想定して Robot の定義を考える。

```
(class Robot (left-arm right-arm)
  (:init
    (assign left-arm  (new Hand-and-arm))
    (assign right-arm (new Hand-and-arm)))
  ((:move-from x y :to xx yy)
    (case (which-arm x y xx yy)
      (is left-arm-only
        [send left-arm
              (list :take-from x y :to xx yy)])
      (is right-arm-only
        [send right-arm
              (list :take-from x y :to xx yy)])
      (is from-left-arm-to-right-arm
        ;; to be filled later ;;)
      (is from-right-arm-to-left-arm
        ;; to be filled later ;;))))
```

メソッド (:move-from x y :to xx yy) では、まず、2地点の座標に従って場合分けをおこない動作させる腕 (Hand-and-arm) を決定する。腕に対して (:take-from x y :to xx yy) というメッセージを送信する。この send 文は [ ] で囲まれているので腕がこのメッセージを受け取った段階で send 文が終了する。すると、(:move-from x y :to xx yy) というメソッド自身も終了する。従って、腕が物体を運んでいる最中に、Robot は次の移動依頼メッセージを処理することができる。運よくそれがもう一方の腕で処理できる内容ならば、並列に実行される。

以上のプログラムで、片腕だけ使用する動作はうまく記述できるようになった。次に両腕の協調動作が必要な場合のプログラムを考える。単純に考えると2つの移動動作を逐次的に行う次のようなプログラムで解決するように思える。

```
(is from-left-arm-to-right-arm
  (send left-arm
        (list :take-from x y
              :to 'center-x 'center-y))
  (send right-arm
        (list :take-from 'center-x 'center-y
              :to xx yy)))
```

しかし、このプログラムだと左腕は右腕が物体をつかむ前に手を開いてしまうので物体を落としてしまう。また、右腕は左腕が物体を中央に持って来るまではなにもしないが、実際には左腕が運んでいる間に並列に動作して中央に移動することが可能なはずである。これをおこなわせるには腕の同期と並列実行を記述する必要がある。

２つの腕の動作過程を図示すると図9のようになる（左腕が最初に運んで次に右腕が運ぶ場合）。

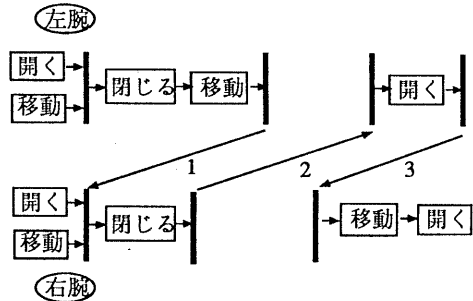


図 9: 協調動作の過程

両腕は同時に動作を開始することができる。左腕が物体を中央に持ってきて、かつ、右腕が中央に到着すれば、右腕は手を閉じることができる。↓1はこの同期を表現している。閉じ終わると左腕は手を開くことができる。これが↑2の同期である。↓3は開いたのを確認してから右腕が移動を開始するための同期である。

このような同期ポイントを持ったメソッドを Hand-and-arm に定義する。これは同期のないメソッドに同期のための待ちとメッセージ送信を加えた形になる。同期ポイントの場所と方向が異なるので、最初に移動するためのメソッドと次に移動するためのメソッドを両方用意する。Hand-and-arm の定義は次の通りである。

```
(class Hand-and-arm (hand arm)
  (:init
     前と同じ)
  ((:take-from x y :to xx yy)
     前と同じ)
  ((:take-firstly-from x y :to xx yy
                         :sync-points a d e)
     (let (wait_d)
         [send d &waid_d]   ---(1)
         (parallel
           (send hand :open)
           (send arm (list :positioning x y)))
         (send hand :close)
         (send arm (list :positioning xx yy))
         [send a :ok]  ---(2)
         wait_d        ---(3)
         (send hand :open)
         [send e :ok]))---(4)
  ((:take-secondly-from x y :to xx yy
                         :sync-points b c f)
```

```
(let (wait_b wait_f)
    [send b &wait_b]
    [send f &wait_f]
    (parallel
      (send hand :open)
      (send arm (list :positioning x y))
      wait_b)
    (send hand :close)
    [send c :ok]
    wait_f
    (send arm (list :positioning xx yy))
    (send hand :open)))
)
```

同期ポイントはfutureを用いて実現されるので、メッセージの引数としオブジェクトに渡される。最初に運ぶ側のメソッドにはa,d,eという3つの引数が与えられる。

(:take-firstly-from x y :to xx yy :sync-points a d e)

この中でa,eは同期信号を送るためのアドレスである。dは同期信号を受け取るための引数であり、この引数が示すアドレスにfutureを渡す必要がある。(1)でそれを行っている。同期信号はwait_dというインスタンス変数に将来束縛されることになる。(2)のメッセージ送信文は↓1の同期をインプリメントしている。同様に(3)は↑2の同期、(4)は↓3の同期をイプリメントしている。

2番目に物体を移動する場合のメソッド

(:take-secongly-from x y :to xx yy :sync-points b c f)

も同様である。ただし、この場合は同期待ちが2個(b,f)で、同期通信を送るのが1個(c)である。

このように定義されたHand-and-armを利用するには次のようなプログラムを実行すればよい。また、Robotオブジェクトを含めた全体のメッセージの流れは図10に示したようになる。

```
(class Cooperator (a b c d e f)
  ((:cooperate ha1 ha2 x y xx yy)
    (let ((aa &a)(bb &b)(cc &c)
          (dd &d)(ee &e)(ff &f))   ---(1)
      (parallel
        (send ha1
              (list :take-firstly-from x y
                    :to 'center-x 'centery
                    :sync-points aa dd ee))
        [send ha2
              (list :take-secondly-from
                    'center-x 'center-y
                    :sync-points bb cc ff)]
        [send b a]  ---(2)
        [send d c]  ---(3)
        [send f e]  ---(4)
        ))))
```

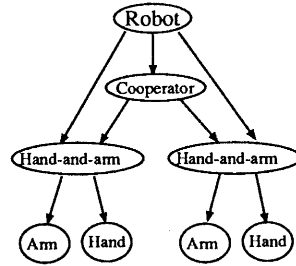


図 10: メッセージの流れ

同期ポイントを実現するためのインスタンス変数を(1)で用意し、そのfutureをHand-and-armに送る。メッセージ送信文(2)(3)(4)で2つの同期ポイントを結びつけている。ha1への送信文は( )で囲まれており、ha2への送信文は[ ]で囲まれているので、このcooperatorのメソッドは、最初に持って来る方の腕の動作が終了した時点で終了する。この時点以降では、2つの腕が両方とも占有されることがなくなるのでデッドロックの可能性がなく、他の移動要求を受け付けることが可能である。

このCooperatorを用いてRobotオブジェクトを記述すると次のようになる。

```
(class Robot (left-arm right-arm)
  (:init
    前と同じ)
  ((:move-from x y :to xx yy)
    (case (which-arm x y xx yy)
      (is left-arm-only
        前と同じ)
      (is right-arm-only
        前と同じ)
      (is from-left-arm-to-right-arm
        (send (new Cooperator)
              (list :cooperate left-arm right-arm
                    x y xx yy))) --(1)
      (is from-right-arm-to-left-arm
        (send (new Cooperator)
              (list :cooperate right-arm left-arm
                    x y xx yy)))))) --(2)
```

両手を使う処理が同時に重なるとデッドロックする可能性があるので、メッセージ送信文(1)(2)は終了待ち形になっている。

5 例題プログラムの検討

ロボットに命令を送る側（以後要求者と呼ぶ）の事情を考えてみよう。要求者は

(:move-from x y :to xx yy)

というメッセージをロボットに送ることで作業を依頼できる。たとえば、

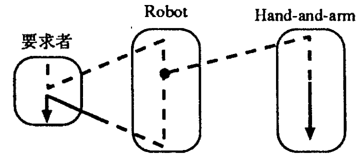
(send robot (list :move-from 1 2 :to 15 7))

というメッセージ送信文を実行すればよい。この文は ( ) で囲まれているのでメソッドの終了を待つことを意味している。要求者は移動という動作の完了を知りたいと考えられる。しかし、Robot は Hand-and-arm に移動メッセージを送るとその終了を待たずにメソッドを終了してしまう。したがって、要求者は Robot レベルでの作業終了を知ることはできるが、本当の移動作業の終了を知ることはできない。Robot のメソッド内のメッセージ送信文

[send left-arm (list :take-from x y :to xx yy)]

の [ ] を ( ) に改めれば、移動作業の終了を正しく知ることができるが、この場合 Robot は逐次的にしか移動命令を処理できなくなてしまう。

Robot と Hand-and-arm が何の関係もない独立のオブジェクトだと考える場合には、Robot での処理終了が要求者からの処理の終了と考えることは妥当だが、Hand-and-arm が Robot の内部構造であり外部からは直接にはうかがい知れないオブジェクトだと考えた場合には不自然である。Hand-and-arm という内部構造を導入したのは Robot オブジェクトを定義した人の裁量範囲であり、外から利用する人は内部構造まで意識する必要はないはずである。この問題点を図示すると図 11のようになる。要求者は Robot というオブジェクトの動作に関心があるのではなく、Robot オブジェクトが代表している構造全体の動作に関心がある。(図 12)



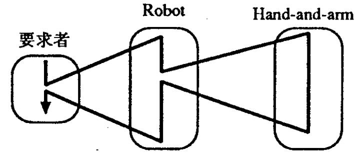


図 11: 終了の検出

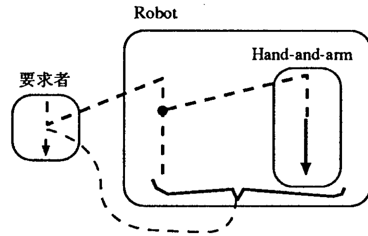


図 12: 要求者の視点

この問題点を克服する方策として次の 2 つが考えられる。

1. Robot オブジェクトが処理の終了を通知するのではなく、作業を引き継いだオブジェクトが終了通知を行う方法

2. 階層構造を構成するオブジェクト群の内部での計算の終了を検出する機構を導入する方法

まず、第 1 の方法について説明する。第 3 節で述べたとおり、ここで用いている並列オブジェクト指向言語では、メソッドの終了を通知するために暗黙の future を引数として渡している。メソッドが終了するとこの future に値が束縛され、メッセージを送信した側は処理の終了を知ることができる。作業を引き継ぐオブジェクトにこの future を渡すことができれば、そのオブジェクトが作業の終了を通知できるようになる。たとえば、future を引き継がせる構文として

[send target message :delegate]

を用意すれば、Robot のプログラムは次のように記述できる。

```
(class Robot (left-arm right-arm)
  (:init
    前と同じ)
  ((:move-from x y :to xx yy)
    (case (which-arm x y xx yy)
      (is left-arm-only
        [send left-arm
              (list :take-from x y :to xx yy)
              :delegate])
```

```
(is right-arm-only
  [send right-arm
        (list :take-from x y :to xx yy)
        :delegate])
(is from-left-arm-to-right-arm
  (send (new Cooperator)
        (list :cooperate left-arm right-arm
              x y xx yy))
        :delegate)
(is from-right-arm-to-left-arm
  (send (new cooperator)
        (list :Cooperate right-arm left-arm
              x y xx yy)
        :delegate)))))
```

Cooperator,Hang-and-arm も同様に:delegate を付加すればよい。この方法の問題点は、あらかじめ処理の最後を担うオブジェクトが決まっている必要がある点である。複数のオブジェクトが最後になる可能性がある場合には、それらを起動したオブジェクトが全体の終了を待って、要求者に処理の終了を通知するプログラムにしなければならない。

第2の方法は、一つの階層を構成するオブジェクト群で行われている計算を管理する仕組みを用意する方法である。reflection 機能 [7][8] を有する言語ではこの機能を容易に実現できる [3]。そして、構文上では、たとえばメッセージ通信によって引き起こされた全計算の終了を待つ send 文として、

{send target message}

を導入すればよい。しかし、この方式は一般的には言語の実行効率を低下させてしまうので、それによって得られる利益とのトレードオフを考える必要がある。処理の最後を担うオブジェクトを特定できる場合が多いアプリケーションでは第1の方法で十分だと考えられる。

6 おわりに

並列オブジェクト指向言語で階層構造を表現する上での問題点を考察し、データの送受信とは独立に処理の実行順序を指定できる機構が必要であることを述べた。とくに、メッセージによって起動した処理が終了したことを認識することの重要性を述べた。この考察に基づいて、3種類の終了待ち形式を持つ並列オブジェクト指向言語を提案した。この言語を用いて2腕ロボット制御問題を記述し、子オブジェクトの制御、子供同士の協調動作を記述できることを示した。しかし、処理終了の意味付けにおいて不十分な点があることを述べ、その解決策として、future を利用する方法と、reflection を利用する方法を述べた。これら2つの解決策には一長一短があり、言語仕様として確定する段階にはまだ至っていない。この部分の検討が今後の課題である。

また、本論文では、オブジェクトはそれが含まれる階層内のオブジェクト、及び、子オブジェクトとだけ通信する例題を扱った。しかし、ロボットどうしが握手をする場合を考えるとロボットのサブ構造である腕どうしが直接インタラクションを行うはずであり (握手問題) 、別個の階層に含まれるオブジェクトどうしの通信も考慮する必要がある。これも今後の課題である。

参考文献

[1] Agha,G.: *ACTORS - A Model of Concurrent Computation in Distributed Systems*, The MIT Press(1986)

[2] 福井: オブジェクト・オリエンテッド並列処理言語の概念構成, 第25回プログラミング・シンポジウム報告集 (1984),pp136-147

[3] 福井: リフレクション機能を備えた actor 言語による階層構造をもつ並列システムの記述, 日本ソフトウェア科学会第7会大会論文集 (1990),pp261-264

[4] Goldberg,A., and Robson,D.: *Smalltalk-80 The Language and its Immplementation*,Addison-Wesley(1983)

[5] Hewitt,C.: Viewing Control Structures as Patterns of Passing Messages, *Artificial Intelligence*, Vol.8, No.3(1977), pp.323-364

[6] Honda,Y., and Yonezawa,A.: Debugging Concurrent Systems Based on Object Groups, In S.Gjessing and K.Nygaard, editors, *European Conference on Object-Oriented Programming(ECOOP)*, LNCS322, Springer-Verlag,1988

[7] Maes,P.: Computational Reflection, TR87-2(Ph.D.Thesis), AI-Lab., Vrije Universiteit Brussel, 1987

[8] 渡部, 米澤: 分散システムのための並列自己反映計算モデルにむけて－アクターモデルによるアプローチ－, 日本ソフトウェア科学会第6会大会論文集 (1989),pp253-256

[9] Yonezawa,A., Brio,J.P., and Shibayama,E.: Object-Oriented Concurrent Programming in ABCL/1,*Proc. of Object-Oriented Prog. Syst. Lang. Appl.(OOPSLA'86)*, pp.258-268(1986)