オブジェクト指向に基づく並列情報処理モデル ABCM/1 とその記述言語 ABCL/1

米澤 明憲　柴山 悦哉　J.-P. Briot　本田 康晃　高田 敏弘

構成要素が並列に動作するシステムを記述・モデル化するための枠組を与える．この枠組は，ソフトウェアシステムを新たに設計・実現する場合にも有効である．この枠組とともに与えられるプログラミング言語はプロトタイピングに用いることもできる．

1 はじめに

我々のまわりには，構成要素の並列的な情報処理・操作によって合目的な挙動を示しているシステムが，いたるところにある．様々なレベルの通信網で結合された計算機システム，個々の計算機内のオペレーティングシステム，人間の持つ認知・思考システム，種々の工業生産システム，複数の人間により構成される学会・会社等の社会組織等々がそれである．このようなシステムを記述・モデル化し，そのシステム内に見出される並列的な問題解決の諸方式を比喩として用いることは，新たなソフトウェア (システム) を設計・実現する上で極めて興味深い方法である．これはまた，VLSI 技術の登場・発展に伴って安価に提供可能となった計算力の有効利用に寄与するだけでなく，計算機による問題解決能力の飛躍的向上につながるものである．

このような動機に基づいて，システムを記述・モデル化し，新たな設計・実現を行うためには，並列情報処理に対する明確な枠組が必要となる．本論文は，そのような枠組の 1 つとして，オブジェクト指向の考え方 [17] [22] [27] に基づいて考案した並列情報処理モデル ABCM/1 と，このモデルを比較的忠実に反映したプログラミング言語 ABCL/1 [26] の骨子を述べる．ABCM/1 は，先に我々が提案した言語 ABCL の基礎となる計算モデルを発展させたものである．

以下に述べる情報処理 (あるいは計算) モデルは，擬人化した情報処理の実行主体であるオブジェクトと，オブジェクト同士の相互作用により構成される．この相互作用は，合目的な組織 (社会) における人間同士の意志・情報伝達のうち，計算機システム内にモデル化し実現する上で，比較的無理のないものだけを抽出した結果得られたものである．説明は直観的に行うが，そのほとんどの部分は数学的記述が可能である．しかし本モデルの全体を完全に特徴付ける数学的モデルについては，いまだ研究中である．また，ABCM/1 および ABCL/1 の応用分野として，分散問題解決方式 [16]，人間の認知プロセスのモデル化，リアルタイムシステムやオペレーティングシステムの

An Object-Oriented Concurrent Information Processing Model ABCM/1 and its Description Language ABCL/1.

Akinori Yonezawa, Etsuya Shibayama, Yasuaki Honda, Toshihiro Takada, 東京工業大学理学部情報科学科.

J.-P. Briot, パリ第 6 校計算機科学研究所 (LITP) よりの客員研究員.

コンピュータソフトウェア, Vol. 21, No. 5 (2004), pp. 32–48.
［復刻論文］本論文はコンピュータソフトウェア, Vol. 3, No. 3 (1986), pp. 9–23 に掲載された論文を復刻したものである．論文中の人物の所属・肩書き等は原論文執筆時のものである．

モデル化および設計，オフィス・システム [18] の設計実現などを念頭においている．

ABCM/1 の記述説明において，不正確になることを防ぐために，記述言語である ABCL/1 の記法を導入し，ABCM/1 における概念とそれに対応する ABCL/1 の構文要素との関係を明らかにしていく．この結果，ABCL/1 の主要な機能の説明は ABCM/1 のそれと並行することになるが，第 6 節で，ABCL/1 の概要をまとめる．第 7 節でプログラム例として分散型問題解決方式の一例を与え，最後の第 8 節でまとめの議論を行う．

ABCM/1 は，C. Hewitt 等による ACTOR モデル [5] [6] [7] [21] から進化したものであるが，両モデルの比較は第 8 節を参照されたい．

2 オブジェクト

2.1 状態と基本動作

我々のモデル (以下 ABCM/1 と呼ぶ) では,「オブジェクト」と呼ばれる，メッセージを受理することにより一連の動作が起動される抽象的なものの集まりと，そのようなオブジェクト間でのメッセージのやりとりによって情報処理や計算が進行する．メッセージのやりとりは，系の中で複数・並列的に行うことが可能であり，複数のオブジェクトが同時に動作することができる．

各オブジェクトは，それ自身の中に自立した演算能力を持ち，かつそれ自身のみが直接アクセスし，参照・更新できる局所的な記憶を保持することができる．(このような局所記憶を持たないオブジェクトも許される．) 局所記憶を持つオブジェクトは，各時点での局所記憶の内容に対応して，その時点での“ 状態 ”(state) が定義される．

一般にオブジェクトは到着したメッセージを受理することにより，次のような 4 種類の基本動作の組合せが実行される．

(1) 局所記憶に格納可能な値に対する種々の演算．
(2) 局所記憶の内容に対する参照・更新．
(3) 次節で述べるメッセージのやりとり．
(4) オブジェクトの動的生成．

各オブジェクトに対し，それが受理するメッセージは，そのパターン，メッセージ中の値やそれを受理する時のオブジェクトの状態などにより定められている．このため，一般にオブジェクトを定義するには，そのオブジェクトの局所記憶の表現 (representation) と，メッセージを受理する条件，受理した各メッセージに対して，どのような一連の仕事を実行するかを指定してやればよい．

ABCM/1 を記述する言語 ABCL/1 では，オブジェクトの定義のためにおおよそ図 1 のような記法を用いる．到着したメッセージが“ (script…) ”中の〈メッセージ・パターン〉と〈条件〉の対を満足すれば，そのような対のうち一番上にある対が選択され，それに対応する〈動作〉の列が実行される．“〈オブジェクト名〉”,“ (state…) ”,“ where〈条件〉”はなくてもよい．

2.2 3 つのモード

各オブジェクトは，任意の時点で“ 休眠 ”(dormant),“ 活性 ”(active)，および“ 待機 ”(waiting) の 3 つのモードのうち，いずれか 1 つを取る．オブジェクトは，その誕生時には休眠モードにある．オブジェクトごとに，その休眠モードで受理できるメッセージのパターンや条件が決まっており，そのパターンや条件を指定しているのが，図 1 の記法である．このような受理できるメッセージが到着すると，オブジェクトは活性モードに入り，メッセージによって指定される一連の仕事を開始する．一連の仕事の途中で，もし待機モードに入るコマンドを実行しなければ，一連の仕事を終了する．この時，このオブジェクトが休眠モー

```
[object〈オブジェクト名〉
  (state〈局所記憶の表現と初期化〉)
  (script
    (=>〈メッセージ・パターン〉where〈条件〉
      …〈動作〉…
            …
    (=>〈メッセージ・パターン〉where〈条件〉
      …〈動作〉…))]
```

図 1 オブジェクトの定義記法

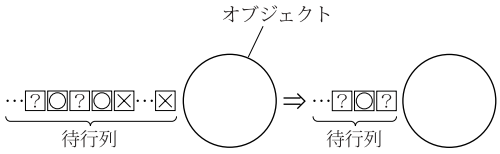

図 2　休眠モードから活性モードへの遷移

ド時に受理できるメッセージが未だ到着していなければ，オブジェクトは休眠モードに入る．既に到着していれば，そのようなメッセージのうちで最初に到着したものが指定する一連の仕事を開始すべく，一旦休眠モードに入った後，直ちに活性モードに入る．(休眠モードに入らず，活性モードに留まると考えてもよい．) 一般に各オブジェクトには，概念的に無限長の待行列 (バッファ) がそれぞれあり，メッセージは到着順にその中に入り，オブジェクトによって処理されるのを待つと仮定する．(しかし，このバッファはあくまでオブジェクトの外側にあり，オブジェクト内の動作とは無関係にメッセージが到着するものと考える．) オブジェクトは，休眠モードにある時，あるいは，一連の仕事を終了した時に，もしこの待行列が空でなければ，休眠モード時に受理可能な最も早く到着したメッセージおよびそれ以前に到着した (受理不能な) すべてのメッセージを待行列から除去し[†1]，その受理可能なメッセージが指定する一連の仕事を開始する．図 2 で☒は現時点で受理不能なメッセージを，また◯は受理可能なメッセージを表示する．(?は任意のメッセージを意味する．)

既に触れたように，オブジェクトは一連の仕事を実行している途中，待機モードに入ることがある．例えば，固定長バッファ(bounded buffer) をモデル化したオブジェクトが，[:get] というバッファの中身を 1 つ取り出すメッセージを処理していたとしよう．このとき，(オブジェクト内の固定長の) バッファが空であればそこで待機モードに入り，以後バッファに〈要素〉を追加する [:put〈要素〉] という形のメッセージの到着を待つ必要がある．逆に [:put…] のメッセージを休眠モード時に受理し，バッファが満杯である場合，待機モードに入り，[:get] メッセージを待つ必要がある．

活性モードにあるオブジェクトが，待機モードに遷移し，その待機モードにおいて，どのようなメッセージが受理可能であるかを指定する必要がある．またメッセージを受理することにより再び活性モードに入り，どのような動作を実行するかを指定する必要がある．ABCL/1 ではこのために図 3 のような記法を用いる．

```
(select
  (=>〈メッセージ・パターン〉 where 〈条件〉
    …〈動作〉…)
    …
  (=>〈メッセージ・パターン〉 where 〈条件〉
    …〈動作〉…))
```

図 3　select 構文

この (select) 構文は，図 1 の記法の中の〈動作〉の 1 つとして書かれるもので，これを実行すると，オブジェクトは待機モードに入り，パターンと条件を満たすメッセージを待つ．

休眠モードと待機モードの相違は，到着したメッセージの待行列の取扱いにある．待機モードでは，受理可能なメッセージの中で待行列の先頭に最も近い，すなわち最も昔に到着したメッセージが選択される点は，休眠モードの場合と同じであるが，待機モードでは，選択されたメッセージだけが待行列から除去され，それ以外の待行列の要素はそのままである．(休眠モードでは，選択されるメッセージ以前に到着したメッセージは廃棄された．) 図 4 は待機モードから活性モードへの遷移を示している．記号の使い方は図 2 と同じである．

休眠，活性，待機の 3 モード間の遷移は図 5 のようになる．次に上で例に掲げたバッファ・オブジェクトの ABCL/1 による定義の概略を与える．

```
[object Buffer
  (state declare–the–storage–for–buffer)
  (script
```

†1 計算モデルとしては，このままでよいが，プログラミング言語としては，このように除去されるメッセージの送り主であるオブジェクトに，何らかのエラー・メッセージが送られることが望ましい．

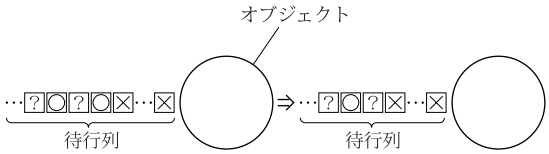

図 4 待機モードから活性モードへの遷移

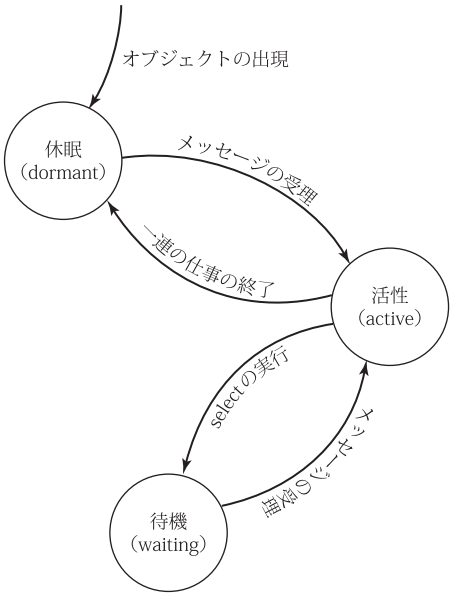

図 5 オブジェクトのモード遷移

```
(=> [:put aProduct]
  (if the-storage-is-full
    then
      (select
        (=> [:get]
          remove-a-product-from-the-storage-
          and-return-it)))
  store-"aProduct")
(=> [:get]
  (if the-storage-is-empty
    then
      (select
        (=> [:put aProduct]
          send-"aProduct"-to-the-object-which-
          sent-the-[:get]-message))
        else remove-a-product-from-the-storage-and-
          return-it)))]
```

3 メッセージのやりとり

3.1 一般的特徴

ABCM/1 におけるオブジェクト同士のメッセージのやりとりの一般的な特徴は次のように要約できる．ただし，A, B は任意のオブジェクトを表わすものとする．

(1) 非放送　　A はその時点で B の存在を“ 知って ”いなければ，B に直接メッセージを送ることはできない．

(2) 動的結合関係　　“ 知っている ”という関係は，動的に変化し得るもので，オブジェクトの誕生時から，ずっと知っている場合もあれば，新たに知ることもある．また忘れてしまうこともある．ただし，自分自身は必ず知っているものとする．

(3) 非同期　　A が B にメッセージを送るとき，A は B の状態やモードに無関係にメッセージを送ることができる．

(4) 到着の保証　　送られたメッセージは宛先に有限時間内に必ず到着する．

(5) 待行列付き　　前節で述べたようにオブジェクトに送られたメッセージは，概念的には，処理されるまでオブジェクトの外側にある 1 本の待行列に到着順に並ぶ．

(6) 送信順序の保存　　A から B に，メッセージ M と M′ をこの順に送ると，M と M′ はこの順序に B に到着する．

ただし，宛先のオブジェクトが存在しない，あるいは消滅していた場合，(4) は保証されないが，記述言語である ABCL/1 では，適当なエラー処理がされる．また，(5) は (3) からの要請であり，待行列の長さについて ABCM/1 では制限がないと仮定する．(6) の送信順序の保存は，一般の計算機ネットワークでは満足されないが，この仮定がないと極めて単純なシステムを記述することも困難となることは明らかであろう．

3.2 3つの形態

第1節でも触れたように，ABCM/1におけるメッセージのやりとりは，人間同士の意志・情報の伝達の形態のうち，並列性が活用でき，かつ計算機システムによって無理なく実現可能なものを選んでモデル化したものである．我々はオブジェクト間のメッセージのやりとりとして，過去 (past) 型，現在 (now) 型，未来 (future) 型の3つの形態を導入する．以下，これらの形態の説明において，メッセージMがオブジェクトOを送り主として，オブジェクトTに送られることによりメッセージのやりとりが開始されることを想定している．

［過去型］ OはMをTに送り，さらに仕事の続きがあればそれを直ちに継続する．OはTがMを受理し，活性モードに入るまで待つということはしない．この型のメッセージのやりとりは，ABCL/1の記法では，

［T <= M］

で表わされる．

過去型のメッセージのやりとりは，人がある人に (メッセージを送って) 仕事を依頼し，その依頼に対する返事や，依頼した仕事の結果が返送されるのを待たずに自分の仕事の続きを行う場合，あるいは単純に情報を伝達する場合に相当する．

［現在型］ OがMをTに送り，Tからの何らかの返答が戻ってくるまで，Oは次の動作に移らない．この型のメッセージのやりとりは，ABCL/1の記法では，

［T <== M］

で表わされる．また，この記法は，メッセージのやりとりを表わすだけでなく，返答の内容自身も表わす．

現在型は，ある人に (メッセージを送って) 仕事を依頼し，その依頼に対する返事や，依頼した仕事の結果が返送されるまで自分の仕事を中断する場合に相当する．また相手との同期を取るためにも使われる．ABCL/1では，値の変数への代入を［〈変数〉:=〈値〉］のように書き表わすので，［〈変数〉:=［T<==M］］のように書くことにより，依頼した仕事の結果や返事を〈変数〉に格納することを指定できる．

現在型のメッセージのやりとりは，いわゆる手続呼出し (procedure call) と同じように見えるが，実際はそれより一般的である．Mの受け手であるオブジェクトTは，依頼された仕事が完了する途中で (あるいは依頼を受けた直後でもよい)，Oに返事を返すことが可能である．手続呼出しのように，返事を返すことが，自分 (すなわちT) の一連の仕事の終了を意味するわけではない．

［未来型］ OはMをTに送って仕事を依頼するとき，その返答の送り先を指定し，返答が戻るのを待たずに自分の仕事を直ちに継続する．ただし，指定した宛先に届けられた返答はO以外のオブジェクトが取り出すことができない．この型のメッセージのやりとりは，ABCL/1の記法では，

［T <= M $ x］

で表わされる．ここで x は，返答の送り先に対応するオブジェクトを値とする変数である．このようなオブジェクトを一般に未来オブジェクト (future object) と呼び，x を未来変数と呼ぶ．

未来型は，人が誰かに仕事を依頼するが，その結果を直ちに必要とはしないので，その人だけが中を調べることのできる特定の場所に仕事の結果を送るように依頼する場合に相当する．仕事の結果が必要になったとき，結果が送付されているかを調べたり，その結果を取り出す手続については，ABCM/1では詳しく規定しないが，ABCL/1では，そのような手続は明確に定義される．直観的には，未来オブジェクトは待行列をモデル化したオブジェクトであるが，詳しい定義は第5節を参照されたい．

3.3 2つのモード

これまでの説明では，オブジェクトが活性モードにある間は，外からのメッセージを受け付けることはできない．メッセージによって指定される一連の仕事が始まるとオブジェクトが自ら待機モードに入らない限り，外部からその仕事を中断させたり，その仕事の流れを制御することはできない．我々人間同士のコミュニケーションでは，オフィスで仕事の最中に電話がかかり，現在の仕事を中断して電話を受け，電話の

内容によっては，現在の仕事を中止したり，電話での依頼の仕事を片付けてから，もとの仕事に戻ったりすることがよくある．このような状況をモデル化し，その比喩を用いて情報処理方式を設計実現するために，これまで説明してきた，通常のメッセージのやりとりのモードの他に，速達 (express) モードを導入する．速達モードでメッセージを送ると，オブジェクトが実行している仕事を外部から中断させることが可能になる．

速達モードの導入は，これによりシステムのモデル化・設計・実現がどれだけ単純・容易なものに，あるいは複雑・困難なものになるかの記述の実験を行うためのものであることを，あらかじめ断っておく．また，数学的モデルを構成する上で大きな障害になることも明らかであるので，以下で説明する速達モードはモデルとしての ABCM/1 の外にあり，言語 ABCL/1 の世界だけに存在するものと考える．さらに，速達モードに対する別のアプローチを第 8 節で示唆する．

ここで導入する速達モードは，いわゆる“ 割込み ”(interrupt) と呼ばれるものの中で最も単純な方式の 1 つで，次の (1)，(2) のような特徴をもつ．ただし以下の記述で割込みを“ 受け付ける ”とそのメッセージによって指定される仕事は直ちに開始されることを意味する．

(1) 速達モードで送られたメッセージによる割込みを受け付ける条件は，受取り側のオブジェクトによって指定される．
(2) 割込みの優先権は 1 レベルのみで，多重の割込みはない．すなわち，速達モードによって依頼された仕事の実行中，あるいはその仕事の途中で待機モードに入っている場合，後から到着した速達モードのメッセージを受け付けるか否かの判断は，現在の一連の仕事が終了するまで延期される．(よって割込みの処理中は割込みは受け付けない．)

割込みを受け付ける条件の指定は，図 1 で与えたオブジェクトを定義する記法の一部を変更することで表わす．すなわち図 1 で，

(=> 〈メッセージ・パターン〉 where 〈条件〉 …〈動作〉…)

の部分は，通常のモードで送られたメッセージに対する一連の仕事の記述と読み換え，新たに，

(=>> 〈メッセージ・パターン〉 where 〈条件〉 …〈動作〉…)

という記法を (script…) の中に書けることとする．これによって〈メッセージ・パターン〉および〈条件〉を満たす，速達モードで送られてきたメッセージを受け付け，…〈動作〉…という緊急の仕事 (割込み処理) を行うことを指定する．

通常のモードのメッセージが指定する仕事を実行している際中，どのようなタイミングで割込みを受け付けるかは，言語 ABCL/1 の詳細な定義の問題である．しかし一般に，オブジェクトの局所記憶にアクセスしている間は，それが終了するまで受け付けは延期される．また，ある動作の列を実行中は割込みがおきないように，その動作の列の記述を“ (atomic ”と“) ”で囲んで指定することができる．さらに，速達モードのメッセージによって割込みを受けた通常モードのメッセージによる仕事は，割り込んだ仕事が終了した時に再開すべきか否か，プログラマが指定できることが望ましい．そのため，ABCL/1 では，(non-resume) というコマンドが速達モードの仕事の途中で実行されると，割り込まれた仕事は放棄されると約束する．このコマンドが実行されないで割り込んだ仕事が終了すると，割り込まれた仕事が直ちに再開される．

速達モードのメッセージを受けるオブジェクトの例として，目覚し時計をモデル化したオブジェクト anAlarmClock の定義を ABCL/1 で与えておく．

```
[object anAlarmClock
  (state person-to-wake count)
  (script
    (=> [:start-and-wake Person :after time]
        [person-to-wake := Person]
        [count := time]
        (while (> count 0)
          do (consume-a-unit-time)
          [count := (sub1 count)])
        [person-to-wake <<= [:time-is-up]])

    (=>> [:wake Person :after time]
```

```
(non-resume)
  [Me <=
    [:start-and-wake Person :after time]])
(=>> [:stop]
  (non-resume)))]
```

この定義において，(state…) の中の persons-to-wake と count はこのオブジェクトの状態を表わす変数の宣言である．このオブジェクトは，[:start-and-wake…]，[:wake…]，[:stop] という 3 つのパターンのメッセージを受理するが，後の 2 つのパターンをもつメッセージが速達モードで送られて来た場合に割込みを受け付けることになる．メッセージのパターンの記述においてコロン (:) で始まる記号はタグを表わし，数字列以外の記号はパターン変数である．また := は左辺の変数への右辺の評価値の代入操作を表わしている．あるオブジェクトが，

[anAlarmClock <= [:start-and-wake A :after12]]

という過去型のメッセージのやりとりを (このオブジェクトと初めて) 行うと，このオブジェクトは活性モードに入り，A というオブジェクトの名前が person-to-wake に，12 が count に代入された後，while ループで一定時間ごとに count の値を 1 減らすことを繰り返す．count の値が 0 になると，[:time-is-up] というメッセージを A に速達モードで送り，anAlarmClock オブジェクトは，休眠モードに入る．

この間 [:stop] というメッセージが速達モードで送られてくると，(non-resume) の実行により，時計の刻みが停止される．また，[:wake…] というメッセージが速達モードで送られてくると，現在の時計の刻みを中断し，[:time-is-up] を送る相手と時刻を変更できる．Me という記号は ABCL/1 では特別な意味を持ち，“ Me ”が現われる動作を記述しているオブジェクトが自身を表わすと約束する．

既に ABCL/1 の説明に少し立ち入ったが，より詳しい説明および，言語の設計方針などは第 6 節で述べる．

4 返答の宛先とオブジェクトの創生

4.1 授受される情報

前節で述べたメッセージのやりとりにおいて，送り手であるオブジェクトから受け手であるオブジェクトに伝えられる情報についてまとめておく．

メッセージの内容は，タグ，あらかじめ指定された値域 (例えば，数，文字列，リスト等) の中の要素，オブジェクト名のどれか，あるいはこれらのものの並びである．メッセージの中に入れて送られるオブジェクトの名前は様々な利用のされかたをする．例えば，オブジェクト O がメッセージ M をオブジェクト T に送り，何か仕事を依頼し，その仕事の結果を指定するオブジェクト C1 に送ってほしい場合，あるいは，依頼した仕事を T 独力で行うのでなく指定するオブジェクト C2 と協力して行ってほしい場合などがある．このような場合，O は C1 や C2 の名前をその目的を指示する適当なタグとともに M に入れて (例えば，[… :reply-to C1…] のようなメッセージとして) 送り，かつ受け手の T をそのようなメッセージを受理し依頼に応じるように定義しておけばよい．

上のようにメッセージの中に明示的に入れて送る情報以外に，ABCM/1 では，メッセージの送り手の名前，および下で説明する“ 返答の宛先 ”(reply destination) を，メッセージの受け手が知ることができる．送り手の名前を，受け手が知ることができるという仮定は，送られてきたメッセージの送り手が誰であるかによってそれを受理するか否かを判断することが可能になり，モデル ABCM/1 の表現力を著しく強化することになる．またこの仮定を実現することは極めて容易である．

4.2 返答の宛先

現在型や未来型によってメッセージを送り仕事を依頼する場合，メッセージの受け手は何らかの返答，すなわち受け取ったという知らせ (acknowledgement)，あるいは依頼された仕事の結果を送らなければならない．このため，どのオブジェクトに返事を返すかという情報は，メッセージの受け手が当然知っていなければならない．未来型の場合，明らかに返答の送り先

は，メッセージと一緒に送られる未来オブジェクトであるが，現在型の場合は，返答先はメッセージを送った送り手のオブジェクトではない．この点については，次節で詳しく論じる．いずれにせよ，このような返答を送る宛先のオブジェクト名を“ 返答の宛先 ”(reply destination) と呼ぶ．

“ 返答の宛先 ”は，仕事を依頼するメッセージとともに送られてくるから，この依頼を受けるオブジェクトは，それを受けるメッセージのパターンの拡張として，“ 返答の宛先 ”専用のパターン変数によってこれを束縛できることにしておくと便利である．ABCL/1 では，オブジェクトの定義において，メッセージを受ける部分の記述として，

(=>〈メッセージ・パターン〉@〈返答先変数〉…)

という記法でこれを表わす．オブジェクト O が，その ABCL/1 による定義の中にこのような部分をもてば，未来型の［O<=M $ x］の場合には〈返答先変数〉に x の値が，現在型［O<==M］の場合は，〈返答先変数〉に第 5 節で定義するあるオブジェクトが束縛される．ただし，M は〈メッセージ・パターン〉と照合するものとする．

さらに過去型のメッセージのやりとりに対しても，メッセージ M のほかに返答の宛先として例えばオブジェクト C を付して送ることを許す．これを ABCL/1 の記法で

［O <= M @ C］

と書くことにする．この結果，C が〈返答先変数〉に束縛されると約束すれば，メッセージの送り手が過去型，現在型，未来型のうちのどの型でメッセージを送りつけても，受け手の方は，それがどの型で送られたかを意識せずに，一様に〈返答先変数〉の値として，それぞれの返答先を指示することができるという大きな利点がある．

この利点は，次のような場合に顕著である．あるオブジェクトが，あるメッセージのパターンに対して，そのパターンと照合するメッセージが現在型で送られてくることを想定して定義されていたとしよう．この場合，そのパターンに照合するメッセージが未来型で送られてきても，また返答先が付加された過去型で送られてきても，どちらの場合にも応じられることになる．この場合の，過去型と未来型の違いは，未来型の場合は，未来オブジェクトという特別の局所的なオブジェクトが返答先になるのに対し，過去型の場合，任意のオブジェクトを返答先として指定できる点にある．

メッセージとともに送られてきた返答の宛先を，他のオブジェクトに送ることもできる．オブジェクト O がオブジェクト T にメッセージ M を，返答の宛先 C を付して仕事を依頼し，メッセージを受けとった T はさらに別のオブジェクト T′ にこの仕事を委託するとしよう．この場合，もし T が T′ に返答の宛先として C を付して仕事を委託すれば，T′ は仕事の結果を T を介することなく O の指定した C に直接送ることができる．

4.3 オブジェクトの創生と返答

既に触れているように，ABCM/1 ではオブジェクトの動作の 1 つとして，動的にオブジェクトを創生することができる．言語 ABCL/1 では一般に，［object…］という文面を実行すると，その文面で挙動が指定されるオブジェクトが創生される．例えば，

［［object A …］<= M］

は，A という名前をもつオブジェクトを創生し，それに M というメッセージを送ることを意味する．同様に，

［B <= ［object C …］］

は，C というオブジェクトを創生し，それをメッセージとして B に送ることを意味する．

様々なシステムをモデル化しシミュレートするような場合，同じような挙動や性質をもつオブジェクトを複数必要とする場合が多い．このようなとき，ABCM/1 では，そのようなオブジェクトを創生するオブジェクトというものを考え，それに創生の引き金となるメッセージや，創生されるオブジェクトの初期化に必要な情報を付加したメッセージを送るという方式をとる．このことを ABCL/1 では，

［object SomeCreator
 (script
  (=>〈初期化情報パターン〉@〈返答先変数〉
   ［〈返答先変数〉 <= ［object …］］))］

のように定義されたあるオブジェクト SomeCreator に，創生の引き金になるメッセージ M を現在型で送り，創生されたオブジェクトを結果として返してもらうように書くことが多い．(創生されるオブジェクトは［object…］によって記述される．) 即ち，

［aNewObject := ［SomeCreator <== M］］

を実行すると，創生されたオブジェクトが変数 aNewObject に代入される．ここで SomeCreator というオブジェクトは，おおむね Simula[13] や Smalltalk80[4] の“ クラス ”に対応する．

上の SomeCreator というオブジェクトの定義記述は,〈返答先変数〉をパターンとして用意し，それに束縛される返答の宛先に何かを送る (この場合は創生されたオブジェクト) というメッセージのやりとりの典型の 1 つを示している．このような典型的な場合の記述を簡潔にするために，ABCL/1 では次のような省略形による記述を許している．

(=>〈メッセージ・パターン〉…!〈式〉…)

これは,

(=>〈メッセージ・パターン〉@〈返答先変数〉
…［〈返答先変数〉］<=〈式〉］…)

の省略形で,“ ！”の直後の〈式〉は，何らかの値を表示する式である．この値が創生されるオブジェクトであるのが上の場合である．よってオブジェクトの創生，返送は,

```
［object SomeCreator
  (script
    (=>〈初期化情報用パターン〉！［object…］))］
```

と書けばよい．

オブジェクトを創生するには，上のように“ クラス ”に相当するオブジェクトに依頼する方法の他に，典型的なオブジェクトをあらかじめ定義しておき，このオブジェクトにメッセージを送ってその copy を生成し返答するよう依頼する方法がある．この方法は[2][11] などに示唆されており，さらにコピーを依頼するとき，現在の状態を変更し目的の状態をもった copy を作ることも可能である[3]．これらのことを可能にするために，ABCM/1 では自分自身のコピー (あるいはクローン) を作るという機能をオブジェクト一般に持たせている．さらに自分自身を殺す機能などもある．これらの機能は ABCL/1 では“ (self-copy) ”,“ (suicide) ”の実行として表現される．

5 既約モデル

本節では，これまでに準備された ABCL/1 の記法を用いながら，次の 2 点を示すとともに未来型のメッセージのより厳密な定義を与える．

(1) 現在型のメッセージのやりとりは，過去型のメッセージのやりとりと待機モードに還元される．
(2) 未来型のメッセージのやりとりは，過去型のメッセージのやりとりと，現在型のメッセージのやりとりに還元される．

この結果，第 3 節で述べた 3 つの型のメッセージのやりとりが，過去型と待機モードの組合せによって表現されることが示される．

5.1 現在型の還元

オブジェクト A がその挙動の 1 つとして，メッセージ M をオブジェクト T に現在型で送ると仮定し，T は M を受理し何らかの返答をするものとしよう．これは ABCL/1 で次のように記述できる．

```
［object A
   …
  (script
     …
    (=>〈メッセージ・パターン〉… ［T <== M］
      …)…)］
［object T
   …
  (script
     …
    (=>〈M に対するメッセージ・パターン〉 @ C
      … ［ C <= 〈返事〉］ …)…)］
```

(ただし，T の定義に 4.3 で述べた “!” を用いて，返答先変数 C によらず，！〈返事〉と書いてもよい．)

ここで，任意のメッセージが到着するとそれを受理し，かつ直ちに A に送るオブジェクト X を導入する．加えて，A はいま導入したオブジェクトを返答の宛先として付けて M を T に過去型で送り，その直後待機モードに入る．そして待機モードから再び活性

モードに戻るために受理するメッセージに対する条件として，その送り主が導入したオブジェクト X の場合に限るとすれば，現在型の意味を損うことなく現在型を過去型と待機モードに還元できる．実際，上の定義を次の定義に書き換えればよい．

```
［object A
   …
  (script
     …
    (=> 〈メッセージ・パターン〉
      (temporary aNewObject …)
        …
      ［aNewObject :=
        ［object
         (script
           (=> any ［A <= any］))］］
      ［T <= M @ aNewObject］
      (select
        (=> value where
              (= &sender aNewObject)
                                  …))…)…)］
```

上の記法で (temporary …) は，直前にある〈メッセージ・パターン〉等の条件を満たすメッセージを受理して活性モードに入ったときに一時的に使う変数の宣言である．メッセージを受け取ると直ちにそれを A に送るという仕事をするオブジェクトが創生され，それが一時変数 aNewObject の値として代入される．where の後の (= &sender aNewObject) は待機モードで受理するメッセージの送り手がこの aNewObject の値でなければならないという条件を記述している．この aNewObject の値となるオブジェクトは，現在型のメッセージのやりとりが行われるたびに創生され，各メッセージのやりとりに対する唯一の識別子の役割を荷っている．また前節で厳密には述べなかった，現在型のメッセージのやりとりの際に送られる返答の宛先となるオブジェクトが，aNewObject の値となっているオブジェクトでもある．

5.2 未来型の意味付けと還元

未来型のメッセージのやりとりは，基本的に，未来オブジェクトという特別なオブジェクトを返答の宛先とする過去型のメッセージのやりとりであるが，未来オブジェクトの中に格納される情報を取り出せるのは，未来型でメッセージを送ったオブジェクト自身のみである．この点を強調するために，ABCL/1 では未来型オブジェクトを値として持つ変数，即ち未来変数は別に宣言し，未来オブジェクトの中の情報を取り出す記法として，メッセージのやりとりを表わす通常の記法を用いず，関数呼出し風の記法 (〈関数名〉〈未来型変数名〉) を用いる．オブジェクト A が，メッセージ M をオブジェクト T に未来型で送り，未来変数 x を指定したとし，その後 A が未来変数 x にアクセスするような状況を ABCL/1 の記法で次のように書いたとしよう．ただし (future …) は未来変数の宣言である．

```
［object A
  (state …)
  (future … x …)
  (script
       …
    (=>〈メッセージ・パターン〉
       … ［T <= M $ x］…(ready? x)…
      (next–value x)…(all–values x)…)…)］
```

第 3 節で触れたように，未来型オブジェクトは待行列をモデル化したもので，(ready?…)，(next–value …)，(all–values …) は，未来オブジェクトに値が届けられているか，即ち，待行列が空でないかをチェックする，待行列の先頭を取り出す，待行列の要素をその順にリストにしてすべて取り出す，ことをそれぞれ意味する．

さて，上で定義したオブジェクト A の挙動は次のように過去型と現在型のメッセージのやりとりによって書き直せる．

```
［object A
  (state ［x := ［CreateFutureObject
                          <== ［:new Me］］］ …)
  (script
```

…
(=>〈メッセージ・パターン〉
… [T <= M @ x] …
… [x <== [:ready?]] …
… [x <== [:next–value]]
… [x <== [:all–values]] …)…)]

ここで，未来変数 x は状態変数として宣言され，次に定義する未来オブジェクトを 1 つ創生してそれを値としていることに注意．さらに［T<=M $$x］は［T<= M @ $x］に，3 つの関数呼出しは対応する現在型のメッセージのやりとりに書き換えられている．

未来オブジェクトの定義は，それを創生するオブジェクト CreateFutureObject の一部として与えられる．このオブジェクトは［:new〈創生者名〉］というメッセージを受け取ると未来オブジェクトを返す．図 6 はその定義である．

未来オブジェクトの定義からわかるように，未来オブジェクトに値 (依頼された仕事の結果等) が届けられる前に，［:next–value］や［:all–values］というメッセージが到着すると未来オブジェクトは創生者以外からのメッセージ，即ち，届けられるべき値を待つ待機モードに入る．このためオブジェクト A は，そのような値が届けられるまで，次の仕事に移れない．(［:next–value］,［:all–values］等は現在型で送られていることに注意．)

6 言語 ABCL/1 の概要と並列・同期の機構

モデル ABCM/1 を説明するために言語 ABCL/1 の記法を援用し説明を加えてきたので，ABCL/1 の輪郭はある程度明らかになったと思うが，本節ではプログラミング言語としての ABCL/1 の設計方針と，これまで述べなかった幾つかの主要な機能を説明する．より詳しい言語仕様は[15] を参照されたい．

6.1 設計方針

ABCL/1 の設計における基本的な方針は次の 2 点である．

(1) メッセージのやりとりの明確な意味付け
ABCL/1 の基礎にあるモデル ABCM/1 にできるだけ忠実であること．特に，メッセージのやりとりに関しては明確な意味付け (semantics) が与えられていること．

(2) 実用性
Smalltalk 等の言語に見られるように，計算・情報処理に関するすべての概念をオブジェクトとして把えるという純粋な立場は取らず，オブジェクトの挙動の定義において，基本的な値やデータ構造 (数やリスト等) をそのまま仮定し，これらへの操作は従来の関数呼出しと考えた．また制御構造もメッセージのやりとりに還元せず，if–then–else, while 等を用いている．

以上のように，ABCL/1 の設計においては，オブジェクト同士の相互作用は完全にメッセージのやりとりとして厳密に定義し，オブジェクト内の挙動の記述，すなわちオブジェクトの定義は，従来の命令型あるいは関数型の計算モデルの考え方に立っている．これは，並列性という本来扱いにくい困難な問題を，できるだけ直観に近い形で記述し，かつ並列性を最大限利用する立場から，メッセージのやりとり以外は，新規性を追求しない方が実用性が高いと考えたからである．

本言語はその構文からも明らかなように，LISP を基礎とし，現在の処理系も LISP で実現されているが，構文が LISP 風であることは本質的ではない．オブジェクト内の挙動の記述に，Pascal，C あるいは Fortran 風の構文を用いることは，何ら妨げるものではない．しかし，実際に，分散的に複数の専門家による問題解決の方式をプログラミングする上で LISP の諸関数がオブジェクト内の動作として使えることは，実用上極めて有利である．

6.2 並列・同期機構

並列性の活用と同期のための機構をまとめておく．

［並列性］

(1) 独立なオブジェクトの並列的な活動
(a) 各オブジェクトに順々にメッセージを送るが，各活動期間は重なり合う．
(b) 各オブジェクトに同時にメッセージを送る．ABCL/1 では，並列構文と呼ばれる次のような記法によってこれを表現する．

```
[object CreateFutureObject
 (script
   ( => [:new creator]
     ![object
       (state [box := [CreateQ <== [:new]]])                    ;待行列オブジェクトを創生して box
       (script                                                   に代入
         ( => [:ready?] where ( = &sender creator)
           !(not [box <== [:empty?]]))

         ( => [:next-value] @ R where ( = &sender creator)      ;創生者からの要求のみ受理
           (if [box <== [:empty?]]
             then (select
                    ( => message where (not (= &sender creator)) ;創生者以外からメッセージが来るま
                      [R <= message]))                          で待機
             else ![box <= [:dequeue]]))
         ( => [:all-values] @ R where (= &sender creator)       ;創生者からの要求のみ受理
           (if [box <== [:empty?]]
             then (select
                    (=> message where (not (= &sender creator)) ;創生者以外からメッセージが来るま
                      [ R <= [message]]))                       で待機
             else ![box <== [:all–elements]]))

         (=> returned–value                                     ;以上の場合以外のメッセージが送ら
           [box <= [:enqueue returned–value]])  )]  ))]         れた場合
```

図 6 未来オブジェクトの定義

{〈メッセージのやりとり〉…〈メッセージのやりとり〉}

並列構文は，各メッセージのやりとりがすべて終了するまで次の動作に移らないという意味付けをしてあるので {…} の中に現在型のメッセージのやりとりが現われると，*cobegin*…*coend* と同様な効果をもち同期の表現にも便利である．

(2) 現在型，未来型のメッセージのやりとりにおける，仕事を依頼する側のオブジェクトの活動と，依頼される側のオブジェクトの活動の並列性．

(3) 多重過去型と多重未来型　　同じメッセージ，同じ返答の宛先，同じ未来型変数を指定して，複数のオブジェクトに同時に，過去型または未来型でメッセージを送る操作を ABCL/1 では，

［〈オブジェクトのリスト〉<=〈メッセージ等〉］

と書くことができる．〈オブジェクトのリスト〉は，これを評価した結果オブジェクトのリストが得られる場合でもよい．

［同期］

(1) 1 つのオブジェクトの活動は，常に 1 つのメッセージの依頼によるもので，複数のメッセージによる依頼を同時に実行することはない．

(2) 待機モード　　(select…) の実行により待機モードに入る．

(3) 現在型と未来型　　現在型では返事が戻ってくるまで仕事を中断する．未来型では，値が届けられる前に未来変数にアクセスすると，値が届くま

で待たされる (ロックされる).

(4) 並列構文　　上で説明のとおり.

7 プログラム例

ABCL/1 によるプログラミングの例題および最も単純な分散問題解決方式の 1 つとして“ プロジェクト・チーム ”による問題解決のモデルを示す. これは, 既に発表した ABCL/1 によるプログラミング例 [26] を改良したものである. ABCL/1 によるその他のプログラミング例は [14] [25] 等を参照されたい.

“ プロジェクトチームに ”よる問題解決とは次のようなものを想定している. マネージャが, ある問題を設定し, 指定した制限時間内にそれを解くようなプロジェクト・チーム, 即ちプロジェクト・リーダとプロジェクト構成員を作ったとする. プロジェクト・リーダはマネージャから問題の仕様と制限時間を与えられると, 目覚時計をその制限時間より少し前に設定してから, プロジェクト構成員に問題の仕様を送る. このとき各構成員は, 異なる戦略を用いて競争で解くものとする. またリーダ自身も, 独自の戦略で問題解決を開始する. 構成員は, 問題が解けるとプロジェクト・リーダが指定した未来オブジェクトに解答を送る. プロジェクト・リーダは構成員から解答が届けられるか, 自分の解答ができるか, あるいはその両方が得られた場合, 最良の解答を選んで, マネージャに報告する. 誰も解答が得られないうちに, 目覚時計が鳴ると, マネージャに期限の延長を願い出る. マネージャは適当な条件によって延長の許諾を決定し, 延長期間 (不許可のときは nil) を告げる. 許可されれば仕事を継続し, 不許可であれば, 構成員の活動を止めて自殺する. 一方, 延期した期限を過ぎても解答が出ない場合, マネージャは, リーダにプロジェクトの中止を告げ, これによりリーダは構成員の活動を止めて自殺する.

この方式を記述するには, マネージャ, リーダ, 目覚時計, および複数のプロジェクト構成員に対応する各オブジェクトを定義する必要がある. 図 7 に各オブジェクトのメッセージの受け渡し関係を矢印で示す. 各構成員に対応するオブジェクト, およびマネージャオブジェクト (Boss) の ABCL/1 による定義は省略するが, マネージャオブジェクトは, 構成員を創生しては［:add–a–team–members…］というメッセージによってその名前をリーダオブジェクト (Leader) に送る作業を何回か繰り返す. その後, 目覚時計オブジェクト (Clock) に自分の名前と目覚し時刻をセットし, リーダに問題の仕様と制限時間を送る. リーダオブジェクトの定義の骨子を図 8 に与える. “ (atomic…) ”を用いて, 解法が見つかってからの処理の最中に割込みが入らないように注意している. [:time–is–up], [:you–are–too–late] の形の速達のメッセージがきたとき, bestSolution が nil でなければ, 即ち最良解を報告した後ならば, これらのメッセージは無視される.

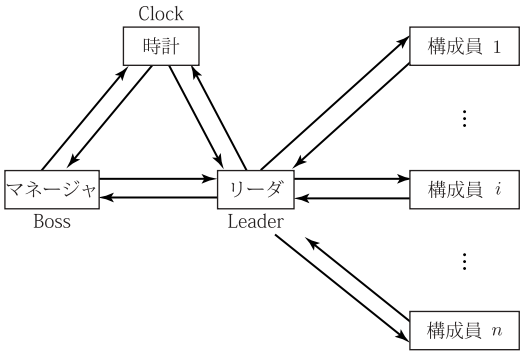

図 7　プロジェクト・チーム方式の問題解決

図 9 は目覚時計オブジェクトの定義である. これは, 複数のオブジェクトに対し, それぞれのオブジェクトごとの起床時刻を設定できるので, 第 3 節で定義した anAlarmClock より多少複雑である. 実際このオブジェクトは［:time–is–up］を送るべきオブジェクト名のリストと, このメッセージを送るべき時刻になるとそれを送る動作をするオブジェクト (これは［CreateBell <== ［:time … :person …］］で創生される) のリストを持ち, 時間刻みごとに, このオブジェクトのリストに多重過去型で［:tick …］を送り続ける.

8 議　論

結論にかえて, 議論し残した部分および特に強調すべき点を最後に述べておく.

```
[object Leader
 (state  [team–members := nil]
         [bestSolution := nil] )
 (future Solutions)    ; 未来変数の宣言
 (script
   (=>  [:add–a–team–member M]    ; チーム員を増すメッセージが来ると,
     [team–members := (cons M team–members)])

   (=>  [:start–solving SPEC   :within TL]    ; プロジェクトの起動を指示するメッセージが来ると,
     (temporary  [mySolution := nil])    ; 一時変数の宣言・初期化

      [Clock <<=  [:wake Me   :after (- TL 20)]]    ; 目覚時刻の設定 (速達モード)
      [team–members <=  [:solve SPEC]  $ Solutions]    ; 多重未来型
      (while (and (not (ready? Solutions)) (null mySolution))
           do ···  〈プロジェクト・リーダ自身も自分の戦略で問題
                   を解き, 解答が得られれば my Solution に代入〉 ··· )
     (atomic
       [bestSolution := (choose–best mySolution (all–values Solutions))]
       [Boss <<=  [:found bestSolution]]    ; 速達モード
       [team–members <<=  [:stop–your–task]]))    ; 多重過去型 (速達モード)

   (=>>  [time–is–up]    ; Clock から目覚ベル信号が来ると,
     (temporary interval)

     (if (null bestSolution)    ; まだ最良解が得られていないなら
       then                     ; 期限の延長を Boss にこう
         [interval :=  [Boss <<==  [:can–extend–deadline?]]]
         (if (null interval)    ; 延長が認められないときは,
          then
            [team–members <<=  [:stop–your–task]]  (suicide)
          else                  ; 延長が OK のときは目覚し時刻を再設定
            [Clock <<=  [:wake Me   :after interval]])))

   (=>>  [:you–are–too–late]    ; Boss から期限切れの通告があると,
     (if (null bestSolution)        ; まだ最良解がなければ,
       then
         [team–members <<=  [:stop–your–task]]  (suicide)   )))]
```

図 8　リーダオブジェクトの定義

```
[object Clock
 (state  [time := 0]   [person–to–wake := nil]
        [wake–up–list := nil])
 (script
   (=>  [:start]              ; 時計をスタートするメッセージが来ると,
     (while t
       do
         (if (not (null wake–up–list))    ; ベルオブジェクトのリストが空でないなら,
           then  [wake–up–list <=  [:tick time]])    ; 時刻シグナルを送る (多重過去型)
         [time := (1 + time)] ))    ; 時刻を進める

   (=>  [:wake Person :after T]    ; 目覚しの対象と時刻を設定するメッセージが来ると,
     (if (not (member Person person–to–wake))
       then             ; もし既にその対象に目覚し時刻が設定されていないなら,
         [person–to–wake := (cons Person person–to–wake)]
         [wake–up–list :=    ; ベルオブジェクトを 1 つ創生してリストにつなぐ
          (cons  [createBell <==  [:time (+ time T) :person Person]]
                 wake–up–list)]
       else             ; 既に設定されているなら, 再設定のためにメッセージを送る (多重過去型)
         [wake–up–list <=  [:wake Person :time (+ time T)]] )))]


[object CreateBell
 (script
   (=>  [:time T :person Person–to–wake]    ; 目覚しの対象と時刻を初期化データとして,

    ![object        ; ベルオブジェクトを 1 つ創生
      (state  [time–to–ring := T] )
      (script
        (=>  [:wake Person :time Time]    ; 再設定のメッセージが来ると,
          (if (= Person Person–to–wake)    ; それが自分に対するものであれば,
            then  [time–to–ring := Time] ))    ; 時刻を再設定する
        (=>  [:tick Time]    ; 時刻シグナルを受けると,
          (if (= Time time–to–ring)    ; それが目覚時刻と一致すると
            then
              [Person–to–wake <=  [:time–is–up]]))))] ))]    ; ベルを鳴らす (速達モード)
```

図 9　目覚時計オブジェクトの定義とベルオブジェクトの定義

8.1 ACTOR モデルとの関係

本モデル ABCM/1 は，MIT において C. Hewitt を中心にして研究されてきた ACTOR モデルの再検討によって得られたものである．ACTOR モデルに基づいて，種々の並列システム，並列問題解決方式の記述を試みると，多くの場合，かなり不自然な記述となる．その原因の第 1 は，第 3 節で述べた“ 送信順序の保存 ”の欠如である．これは ACTOR モデルが極めて一般的に構成されているためで，我々が行ったように“ 送信順序の保存 ”を仮定してしまえば解決がつく．

ACTOR モデルのより本質的欠陥は，すべてのメッセージのやりとりが，(ABCM/1 における) 過去型のものだけである点である．また，ACTOR モデルにおけるオブジェクトに対応するアクター (actor) には，活性か不活性かの 2 つのモードしかない点も大きな欠陥である．この 2 点のために，ある機能を持つ対象物をアクターとしてモデル化しようとすると，本来自然なつながりをもってモデル化されるべきその対象物が行う一連の仕事を，不自然に細かく分断して記述しなければならなくなる．第 2 節の待機モードの説明から明らかと思うが，もし待機モードがなければ，待機モードで待つべきメッセージが到着する以前に，無関係なメッセージが到着すれば，それをオブジェクトの内側に取り込んで保存しておき，望むメッセージの到着・処理後に，取り込んだメッセージの処理にあたらなければならない．この結果，アクターの挙動の記述がきわめて不自然なものとなる．

8.2 速達モード

さらに，第 3 節で導入した速達モードは，ACTOR モデルには，もちろん存在しないものである．我々は，このモードの ABCM/1 への導入は，実験的なものだと断っているが，数学的取扱いがむずかしい点を除けば，モデル化には大変協力な手段であることは事実である．

速達モードは，第 3 節でも触れたようにいわゆる“ 割込み ”の最も単純なものである．しかし，たとえ最も単純な方式であっても，このようなものを高級言語に導入する点には異論が多いかもしれない．この点の判断は，今後の記述実験の進展を待ちたい．

また速達モードの別なアプローチとして，“ 割込み ”を伴わない方式も考えられる．即ち，各オブジェクトに対し，それぞれ速達モードで到着するメッセージを到着順に待たせる専用の待行列を仮定し，たとえ速達モードのメッセージが到着しても割込みは受け付けないものとする．割込みの受付けは，オブジェクトが休眠モードあるいは待機モードにある場合か，オブジェクトが活性モードで特別のコマンドを実行した場合のみとする．例えば，“ (check–express–queue) ”のようなコマンドを実行するとし，もしこのとき速達専用待行列が空でなければ，その先頭のメッセージが指定する緊急の仕事を実行する．そのような待行列が空であれば，そのまま仕事を続行する．

こうすることにより，仕事を実行している際中の，速達モードによる緊急の仕事の受付けのタイミングを，受付け側が完全に制御することができる．この場合，“ (atomic …) ”は不要となり，それ以外は，ABCM/1，ABCL/1 の双方とも変更する必要がなく，数学的に厳密な意味付けも可能となる．しかしモデル化の便利さの点で，多少劣ると考えられる．

8.3 継承機構 (インヘリタンス)

いわゆる“ オブジェクト指向 ”言語の特徴として Simula や Smalltalk などが持つ継承機構が重要だと言われている [17]．我々は，記述量の軽減のためにこの種の機能が有効であることを認めるが，最終的にどのような意味付けによってこの種の機能を並列言語に導入するのがよいか，説得力ある結論に到達していないのが現状である．よって ABCL/1 には継承機構が組み込まれていないが，Lieberman が主張する委託機構 (delegation mechanism) [12] は，第 4 節で説明した“ 返答の宛先 ”機構の活用により自然な形で導入できる．この点の詳しい議論，種々の継承機構の意味付け等について幾つかの着想を得ているが，これについては別の機会に論じる予定である．

8.4 意味記述用モデルとしての ABCM/1

ABCM/1 は並列計算の枠組として十分に一般的であるため，種々の並列計算モデルあるいは並列言語

[8] [9] [10] [20] の意味付けに利用できると考えられる．また，ABCM/1 とこれらのモデルとの比較検討も興味深い研究テーマである．並列論理型言語 GHC [19] の意味付けに ABCM/1 を用いる実験が現在進行中である．

8.5 相対論的観点の問題点

一般に，分散システムの純粋な枠組では，大域的な時間や，大域的な情報の存在を否定する場合が多い．我々の ABCM/1 でも同様の立場を取っている．しかし，これはある意味で相対論的なものの見方を強制することになる．我々がモデル化しようとする現実世界は，それほど極端な世界でない場合が多いので，そのような場合純粋に相対論的な立場をとる必要もない．そこで，例えば，大域的な時間だけは保証する，即ち，各オブジェクトがそれぞれ大域的に同期した時刻を参照することはできるというモデルも考えられる．どの程度まで大域的な情報をモデルの中に許容すべきかは，今後の大きな課題であろう．

謝辞 本研究の初期の段階で熱心な討論に加わってくれた松田裕幸，福井真吾の両氏 (現 NEC) に感謝する．

参 考 文 献

[1] Agha, G. A.: Actors: A Model of Concurrent Computation in Distributed Systems, *Technical Report 844*, Artificial Intelligence Laboratory, MIT, 1985.

[2] Briot, J.-P.: Instanciation et Héritage dans les Langages Objets, (thèse de 3ème cycle), *LITP Research Report*, No. 85–21, LITP-Université Paris-VI, 15 Dec. 1984.

[3] Cointe, P., Briot, J.-P. and Sepette, B.: The Formes System: A Musical Application of Object-Oriented Concurrenr Programming, in *Object-Oriented Concurrenr Programming*, edited by Yonezawa, A. and Tokoro, M., MIT Press ,1986.

[4] Goldberg ,A. and Robson, D.: *Smalltalk 80 —The Language and its Implementation*, Addison Wesley, 1983.

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

[6] Hewitt, C. and Baker, H.: Laws for Parallel Communicating Processes, *Proc. IFIP–77*, Toronto, 1977.

[7] Hewitt, C. *et al.*: A Universal Modular Actor Formalism for Artificial Intelligence, *Proc. Int. Jnt. Conf. on Artificial Intelligence*, 1973.

[8] Hoare, C.A.R.: Monitors: An Operating System Structuring Concept, *Comm. ACM*, Vol. 17, No. 10 (1974), pp. 549–558.

[9] Hoare, C.A.R.: Communicating Sequential Processes, *Comm. ACM*, Vol. 21, No. 8 (1978), pp. 666–677.

[10] Ishikawa, Y. and Tokoro, M.: Orient 84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation, in *Object-Oriented Concurrent Programming*, edited by Yonezawa, A. and Tokoro, M., MIT Press, 1986.

[11] Lieberman, H.: A Preview of Act-1, *AI–Memo 625*, Artificial Intelligence Laboratory, MIT, 1981.

[12] Lieberman, H.: Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object-Oriented Systems, *Proc. of 3rd Workshop on Object-Oriented Languages*, Bigre+Globule, No. 48, Paris, Jan. 1986.

[13] Nygaard, K. *et al.*: *SIMULA Begin*, Auerbach, 1973.

[14] Shibayama, E. and Yonezawa, A.: Distributed Computing in ABCL/1, in *Object-Oriented Concurrent Programming*, edited by Yonezawa, A. and Tokoro, M., MIT Press, 1986.

[15] Shibayama, E.: *et al.*: *ABCL/1 User's Manual*(準備中).

[16] Special Issue on Distributed Problem Solving, *IEEE Trans. on Systems*, Man, and Cybernetics, Vol. SMC-11, No. 1(1981).

[17] Stefik, M. K. and Bobrow, D. G.: Object-Oriented Programming: Themes and Variation, *The AI Magazine*, 1986.

[18] Tsichritzis, D. (ed): *Office Automation*, Springer, 1985.

[19] Ueda, K.: *GHC: Guarded Horn Clauses*, ICOT Technical Report, 103, June, 1985.

[20] Yokote, Y. and Tokoro, M.: Concurrent Programming in ConcurrentSmalltalk, in *Object-Oriented Concurrent Programming*, edited by Yonezawa, A. and Tokoro, M., MIT Press, 1986.

[21] Yonezawa, A. and Hewitt, C.: Modelling Distributed Systems, *Machine Intelligence*, Vol. 9(1979), pp. 41–50.

[22] 米澤明憲: オブジェクト指向型プログラミングについて，コンピュータソフトウェア，Vol. 1, No. 1(1984).

[23] Yonezawa, A.: AI Parallelism and Programming, to appear in *Proc. of IFIP'86*, Dublin, Sept. 1986.

[24] Yonezawa, A. and Matsumoto, Y.: *Object-Oriented Concurrent Programming and Industrial Software Production*, Lecture Notes in Computer Science, No. 186, Springer-Verlag, 1985.

[25] Yonezawa, A., Briot, J.-P. and Shibayama, E.: Object-Oriented Concurrent Programming in ABCL/1, to appear in *Proc. of ACM Conf. on*

Object-Oriented Programming, System, Languages and Applications, Oregon, Sept. 1986(印刷中)

[26] Yonezawa, A., Shibayama, E., Takada, T. and Honda, Y.: Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1, in *Object-Oriented Concurrent Programming*, edited by Yonezawa, A. and Tokoro, M., MIT Press, 1986.

[27] Yonezawa, A. and Tokoro, M.(eds): *Object-Oriented Concurrent Programming*, MIT Press, 1986(in press).