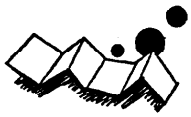
Vol. 20 No. 7　　情　報　処　理　　July 1979

解　説



ACTOR 理論について†*

米澤　明憲††

1. はじめに

本稿は，マサチューセッツ工科大学の計算機科学研究所 (Laboratory for Computer Science, 略して LCS, 旧称 Project MAC) の研究プロジェクトの1つとして，理論・応用の両面で様々な角度から研究されている ACTOR 理論について解説したものである．(プロジェクトは Message Passing Semantics と呼ばれ C. Hewitt が主宰している．)

ACTOR 理論は，C. Hewitt が，彼自身の提案した人工知能研究における問題解決用言語 PLANNER[10] のパターン照合機能のセマンティクスを説明するために用いた比喩，message passing (メッセージのやりとり) がきっかけとなって発展したもので，プログラミング言語のセマンティクス，並列計算，並列処理，分散処理等の様々な情報処理・計算方式に対して，1つの強力な理論的かつ概念的な枠組を提供するものである．また，人工知能研究における“知識”の表現，格納，使用のための一般的形式を与えようとする研究[11),12)] の所産でもある．

以下，**2.**において，ACTORという概念を，具体的な例を用いて直感的に説明し，**3.**で幾分抽象化されたACTOR による計算モデルを与える．**4.** では，ACTOR モデルによる計算を具体的に記述するプログラミング言語の1つ PLASMA[12)]を用いて，ACTOR の階層性とその実現 (implementation) 及び制御構造を論じる．また **5.** においては，多重プロセス／分散処理系のモデルとして ACTOR モデルを把えたときにどのような性質が本質的であるかを議論する．さらに **6.** では ACTOR モデルに基づいた多重プロセス／分散処理系の仕様検証技法についてきわめて簡単に触れ，**7.** において ACTOR モデルの様々な応用について手短に説明する．

† A Tutorial on ACTOR Theory by Akinori YONEZAWA (Department of Information Science, Tokyo Institute of Technology).

†† 東京工業大学理学部情報科学科

* 本稿は，筆者による文献 37) をもとにし，新たな内容を大幅に加えて解説用に書き直したものである．

2. ACTOR とは？

ここで考える計算・情報処理のモデルは，actor と呼ばれる対象 (object) とその対象同志の間で行われる“メッセージのやりとり”とに基づく．actor とは，プログラミング言語の概念を用いて直感的に述べると，(i)手続 (procedure) や関数 (function) などのように計算や操作を行うもの，(ii)記憶場所，データ構造などのように情報を格納するためのもの，及び(iii)数や文字列などのようにデータそれ自身に対応するもの，これら(i)～(iii)の3種類の異なる範疇に属するものを，メッセージを受けとることによって能動化される手続的 (procedural) な対象として統一したものである．

メッセージは，actor に対するオペレーションの要求 (request) や，そのオペレーションを遂行するために必要な情報・データを含む場合と，要求に対する返答 (reply) やオペレーションの結果を含む場合がある．**図-1** は1つの actor がメッセージを受けて返答することを図式的に示したものである．

例えば，階乗を計算して答を返す actor を *FACT* とすると，*FACT* は3というメッセージを受け取って，6を結果として出す(**図-2**)．一般に，計算や操作を行う actor に送られるメッセージには，手続・関数呼び出しに使われる実引数に対応するものが含まれていると考えてもよい．

配列 (array, dimension) に相当する actor を A とすると，その i 番目の要素を参照するには，A に例えば (**図-3** 参照)，(element : i) という要求を含むメッセージを送ればよく，j 番目の要素を *new* で置換えるには，例えば (update : j *new*) という形の要求を含むメッセージを送ればよい．図-3 で，A に対する更新要求の返答はなくても構わない．簡単なデータ構造の1つであるスタックに対応する actor は，(pop :) という要求を含むメッセージを送ることで，現在のス

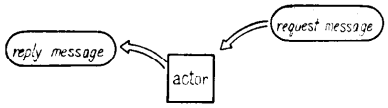


図-1

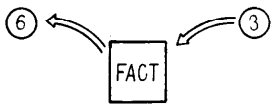


図-2

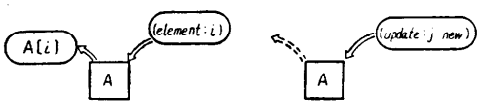


図-3

タックの先頭要素を（もし，存在すれば）取り出し，同様に（push: *new*）で新しい要素 *new* をスタックの先頭要素として載せることができる．一般に，データ構造のように振舞う actor に送られるメッセージには，格納されている情報を参照したり更新したりするためのオペレーションを指定するものと，オペレーションに必要なデータとが含まれる．

数や文字列を1つの actor と考えると，例えば符号を変えたり，先頭文字を取り出すようなオペレーションは，そのオペレーションを指定するものを含むメッセージを送れば良い．また，数や文字列の二項演算（例えば，＋ や append）については，演算の種類を示すものと一方の被演算子である actor を含むメッセージを，もう1方の被演算子である actor に送ることにすればよい．例えば，3+4 を行うには 3 に相当する actor に例えば（plus: 4）という要求を含むメッセージを送ればよい．**図-4** で，7も4も3と同様数字のようにふるまう actor と考えてよい．

このように，上で述べた(i)～(iii)のプログラミング言語における概念はすべて actor とそれに対するメッセージという考え方を用いて統一的に眺めることができる．さらにこの考えを進めて，actor という概念によってプロセス，プロセッサ，ネットワークに接続された個々の計算機システムなどもモデル化できる．また，ハードウェアのメモリ，メモリバンク，ソフトウェア上でのファイル，ファイル・システム，データベース，またそのマネージメントシステム等も，必要に応じて，単一の actor としても，また，何らかの構造をもった actor の集団としても見通しよくモデル化できることが容易に想像されるであろう．（この構造をもった actor の集団や actor の階層性については **4.** でもう少し詳しく論じる．）

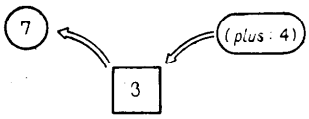


図-4

また，きわめて荒ぼい言い方をすると，一般に“ソフトウェアモジュール”と呼ばれるもの（手続モジュールとデータモジュールの両方とも）は actor という概念の1つの具体例と考えてよいことを注意しておく．

3. ACTOR 計算モデル

本節では，前節で直感的に導入した actor という概念を抽象化した形でもう少し詳しく説明し，この actor に基づいた計算のモデル[2),12),13)]を与える．

3.1 actor

actor は概念的に2つの部分，script と acquaintances とにより構成される．script とは，受けとったメッセージに対して，その actor が反応して，どのように振舞うかを記述したものである．（手続的に記述されていると考えてよく，後に述べるプログラミング言語 PLASMA は，その記述言語の1例でもある．）個々の actor はそれぞれ決ったメッセージの集合をもち，その集合に属するメッセージによって能動化され script に従って振舞う．（その集合に属さないメッセージに対する反応は未定義としておく．）actor の acquaintances とは，その actor が“知っている”（knows-about）actor の集合で有限である．actor A が actor Bを知っている時のみ，AはBへメッセージを直接送ることができる．AがBを知っていても，必ずしもBがAを知っているとは限らず，またAがA自身を常に知っているというわけもない．（この acquaintances の要素がどのように決められるかは **5.** で述べる．）

actor は始めから存在しているものや，計算の過程で他の actor によって生成されるものもある．“計算”（computation）を定義するために，次に説明するイベント（event）という概念が必要となる．

3.2 イベント（event）と継続（continuation）

メッセージMの actor Tへの到着をイベント(event)と定義し，次のような記法で表わすことがある．（数学的にはTとMによる順序対（ordered pair）として定義される．）

$(T \Leftarrow\!= M)$

このとき，Tをこのイベントの標的 (target)，Mをこのイベントのメッセージと呼ぶ．イベントはメッセージの純粋な到着のみを意味し，メッセージによって要求されたオペレーションの遂行やその完了を必ずしも意味しない．

メッセージの中には，要求 (request) するオペレーション及び必要データの他に，そのオペレーションの結果を送る，送り先 (destination) に相当する actor が含まれることがある．このような actor を継続 (continuation) と呼ぶ．継続を含んだメッセージの一般形を次のような記法で表わすことがある．

[request : ⟨a-request⟩ reply-to : ⟨a-continuation⟩]

また操作や計算の結果を送る時には次のような形のメッセージを使うことがある．

[reply : ⟨a-result⟩]

例えば，継続Cを含むメッセージ M=[request : 3 reply-to : C] が階乗を計算する actor，$FACT$ に到着したとすると，その結果6を含むメッセージ M=[reply : 6] が最終的にCに到着する．**図-5** はイベント図 (event diagram) と呼ばれるものの1つで，この階乗の計算の例を示す．一般にイベント図で ⟹ はメッセージが標的に到着することを表わし，⇢ はイベントの時間的前後関係を示す．⟶ は "knows-about" 関係を示す．

3.3 計算 (computation)

ACTOR モデルにおける計算 (computation) は，イベントの半順序集合として定義される．順序づけは狭義で (同一元以外は決して同順位でない)，時間的 "先行" (precedes) を意味し，"⇢" で表わす*．actor の集団の中でメッセージの伝送は同時に2つ以上行われてもよい (**図-6**)．この場合が並列 (parallel) 計算に対応する．3個のイベント E_1, E_2, E_3 に対して，E_1 が E_2 に先行し E_1 が E_3 に先行する．即ち $E_1 \dashrightarrow E_2$ かつ $E_1 \dashrightarrow E_3$ が成立するが，E_2 と E_3 の間に順序づけがないとき，E_2 と E_3 は並列的 (concurrent) におこったと考える．半順序集合の特別な場合として，イベントの全順序集合が考えられるが，これは同時に2つ以上のメッセージ伝送が許されないことに対応し，順次的 (serial) 計算を定義する (図-6)．

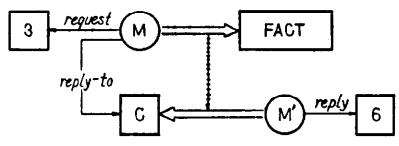


図-5 3! の計算のイベント図

*イベント図では ⇢ で表わされる．

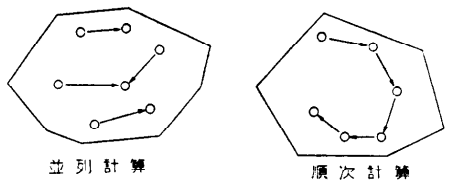


図-6

順序づけの狭義性は，どのイベントも自分自身に先行できないことを保証し，埃 (deadlock) や永久待ち (starvation) のないことを調べる際にも有用な性質である．

さらに，互に順序づけられた任意の2つのイベントの間には，有限個のイベントしか存在しないこと，各イベントに対して，その直後におこる並列的なイベントは常に有限個しかないという仮定を設けることにより，計算の物理的な実現可能性を保証する．

ここで注意したいのは，並列計算のモデルとして，非決定性の計算モデルを使うことがあるが (例えば22))，actor 計算モデルではイベントの半順序集合を用いることである．

3.4 能動化順序関係と到着順序関係

上述の時間的 "先行" に対応する順序づけは，2種類の順序づけ，activation ordering (能動化順序関係) と arrival ordering (到着順序関係) の合併とみなされる．activation ordering は物理的な因果関係を表現するためのもので "$-act\rightarrow$" という記法が用いられる．$E_1 - act \rightarrow E_2$ が成立するとき，E_2 は E_1 によって引き起されたものと考える．1つのイベントによって2つ以上のイベントが互に順序づけなく引き起されてもよい．(fork に対応する．)

arrival ordering は1つの actor A に対して2つ以上のメッセージが送られるとき，その到着順序によって定まる順序づけで，"$-arr\rightarrow_A$" で表わされる．この到着順序に "同時" ということがなく，必ず線形に順序が入ると仮定する．即ち，E_1 と E_2 がともに actor A を標的とするイベントのとき，次のどちらか，$E_1 - arr \rightarrow_A E_2$ か $E_2 - arr \rightarrow_A E_1$ が成立する．

activation ordering も arrival ordering もともに時間的 "先行" の順序の部分順序づけ (subordering) であるから，順序の狭義性と，**3.3** の終りで述べた2つの有限性の条件を満さなければならない．

3.5 純 (pure) actor と不純 (impure) actor

すべての actor はその挙動によって2種類に分類される．第1種に属する actor は時間とともにその挙動を変えない，即ち，同じ要求に対して常に同じ反応を示すもので，この範疇に属するものを純 (pure) actor と呼ぶ．これに対して，時間とともにその振舞が変化する，即ち，同じ要求に対して必ずしも同じ反応を示さない* actor を不純 (impure) actor と呼ぶ．前出の階乗を計算する *FACT* などは，同じ要求（入力）に対して同じ反応（答）を示さなければならないはずであるから，純 actor の例である．一般に情報を格納しておくための actor，例えばメモリセルや配列などは不純 actor の例であるが，この他に乱数発生を行う actor も不純であると考えられる．

純 actor は一般に数学的関数と考えられて，数学的な取扱いが容易であるが，不純 actor は過去に到着したメッセージの歴史によって現在の挙動が決まるので色々な取扱いが面倒になってくる．またこの不純性は一般に副作用と言われるものと深い関係にある．例えば LISP のリストセルにおいて，rplca, rplcd をオペレーションとして許すか許さないかによって，リストセルに対応する actor の純性／不純性が定められる[39]．

4. ACTOR の階層性と制御構造

4.1 階層性と PLASMA による実現

一般に各 actor は階層性を持つことができる．即ち，1つの actor は一般により下位の actor の集団によって構成され，その actor の挙動は下位の actor の間でのメッセージのやりとりによって表現される．階層の中で一番下位にあるのが原始的な (primitive) actor で，他の actor によっては実現されない．どの actor を原始的なものとして選ぶかは，ACTOR モデルの応用分野によって適当に定めればよいが，例えばプログラミング言語のセマンティックモデルとして応用するときは，その言語の基本データ型 (basic data type) をもつ対象 (object) が原始的な actor であると考えてよい．

前にも言及したように，ソフトウェアモジュールを actor と考えてよいから，ソフトウェアの階層的構成は actor の階層性にそのまま対応させることができる．

3.4 で述べた arrival ordering の線形性の仮定は，並列処理，分散処理で用いられるハードウェア及びソフトウェアのモジュールを actor でモデル化する時に注意を払わなければならない．例えば，複数のポート (port) を持ち，本質的にメッセージを並列的に処理するハードウェアは，それ全体を1つの actor としてはモデル化できない．その際は各ポートごとにまとまったそのハードウェア構成要素を1つの actor と考えればよい．

1つの actor の挙動がその actor を構成する actor 間のどのようなメッセージのやりとりによって実現されるかを記述するためのプログラミング言語 PLASMA が開発されている．**4.3** と **4.4** において，PLASMA によって書かれた階乗を計算する actor の2つの異なる実現 (implementation) を与える．PLASMA についての細い説明は 12) を参照されたい．

4.2 制御構造

ACTOR 計算モデルでは actor 間のメッセージのやりとりが唯一の活動であった．それでは，プログラミング言語にある様々な制御構造 (control structure) (例えば, GOTO, 繰返し，再帰呼出し等) は ACTOR モデルではどのように実現されるのであろうか？

このような制御構造は，**3.2** で説明したメッセージ内に含まれる継続 (continuation) のパターンによって実現される[12]．GOTO，繰返し (iteration)，再帰呼出し (recursion)，コルーチン (coroutine) 等は，メッセージ中の継続を適当に選ぶことにより，メッセージのやりとりとして統一的に見ることができる．例えば，GOTO はメッセージに要求 (request) のみあって，継続が含まれていない場合で，GOTO による飛び先は，そのメッセージを受けとる actor に対応するというようにして実現できる．また，通常の関数／手続呼出しは，継続を呼出しのあったテキスト上の直後の部分に対応させることによって実現できる．

次の2つの小節では繰返しと再帰呼出しが，継続の形によってどのように実現されるかを，PLASMA 言語とイベント図 (**3.2** 又は図-5参照) を用いて説明する．コルーチンについては練習問題とする．

4.3 再帰呼出しの actor による実現

この小節では，再帰呼出しの例として階乗を計算する actor の再帰的な実現 (implementation) を PLASMA で記述したものを与えるが，その前に PLASMA の構文*とその解釈を簡単な例で説明する．

actor は一般に有限個の決ったパターンとマッチするメッセージを受理し，そのメッセージに対応して何らかの動作 (action) を行うから，PLASMA による

* ここでは actor の非決定的 (non-deterministic) な挙動は考慮していない．

実現ではメッセージのパターンとそれに対応する動作は次のように書かれる.

(≡≡〉〈pattern〉〈action〉)

ある数を受けとってその数に1を加えた結果を指定された継続 (continuation) に送るという動作をするactor は PLASMA で次のように書ける.

(≡≡〉[request: =n reply-to: =c]
(c ⇐= [reply: n+1]))

[request: =n reply-to: =c] は受理できるメッセージのパターンで [request: N reply-to: C] というメッセージがこの actor に到着するとNは変数nにCは変数cに束縛され，次の行に指定された動作が行われる．即ち，Cに [reply: N+1] というメッセージが送られる．(N+1 は何らかの方法で計算されたと仮定する.)

さて，正整数を受けとってその階乗を指定された継続に送る actor $FACT_r$ の PLASMA による再帰的実現[12]を次に与える.

($FACT_r$≡ ; $FACT_r$ を次のような actor として定義.
(≡≡〉[request: =n reply-to: =c]
; メッセージのパターン
(rules n ; n に対する場合わけ
(≡〉1 ; もしnが1なら
(c ⇐= [reply: 1])) ; c に1を送る.
(≡〉(>1) ; もしnが1より大なら
($FACT_r$ ⇐=
; $FACT_r$ に次行以下のメッセージを送る.
[request: n−1 ; 要求部は n−1.
reply-to: ; 継続部は次の2行の actor.
(≡≡〉[reply: =y]
; その actor の受理するメッセージ・パターン.
(c ⇐= [reply: y∗n]))])))))
; その actor の動作として，y∗n が c に送られる.

コメントから理解されるように，$FACT_r$ で受理されたメッセージの要求部である数Nが1ならばその時に受理された継続Cに1が送られる．nが1より大きいとき，メッセージ [request: n−1 …] が作られ，それが再び $FACT_r$ に送られる．すると今度は N−1がnに束縛されこのメッセージの含まれていた継続 (≡≡〉[reply: =y] (c⇐=[reply: y∗N])) がcに束縛され，今度はN−1と1の大小関係で同じ場合分けが繰り返され，再帰的に進行する.

* ここで示す PLASMA の構文はメッセージのやりとりをきわめて明示的に示すもので，実際のプログラミングでは，省略時解釈，コンベンションを用いた簡略化された構文が用いられている.

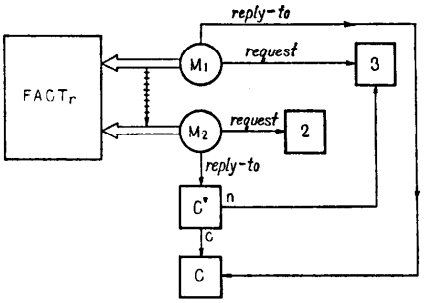


図-7 3! の計算のイベント図（途中）

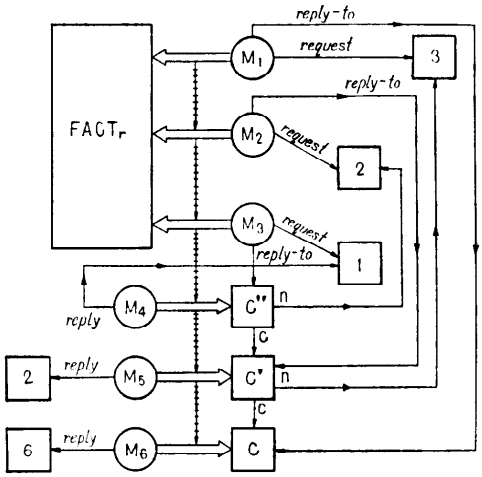


図-8 3! の計算のイベント図（再帰的）

メッセージ M_1=[request: 3 reply-to: c] が $FACT_r$ に送られてから次に$FACT_r$にメッセージM_2=[request: 2 reply-to: C'] が送られる様子を図-7にイベント図で示す．M_2 の中の継続 C' は (≡≡〉[reply: =y] (c⇐=[reply: y∗N]) で，yは C' があとで受理することになっている数である.

$FACT_r$ の中で作り出される継続がどのようにメッセージを受けるかを見るために，$FACT_r$ に到着した最初のメッセージ M_1 中の継続Cが最終的にメッセージ M_6=[reply: 6] を受けとるまでの過程をイベント図で図-8 に示す．C, C', C'' の関係が再帰呼出しを実現するためのスタックのように用いられていることに注意されたい．(矢印⟶ が knows-about 関係を示す.)

4.4 繰返し (iteration) の actor による実現

次に繰返し (iteration) によって正整数の階乗を計算する actor $FACT_i$ の PLASMA による次のような実現[12]を見てみよう．$FACT_i$ はその内部に $LOOP$

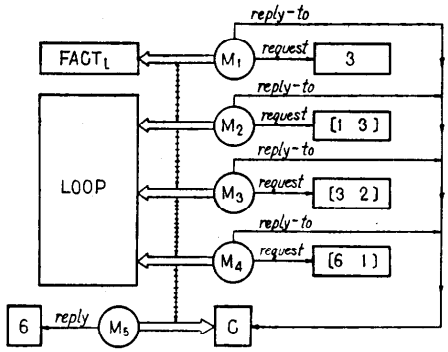


図-9 3! の計算のイベント図（繰返し的）

という actor を持ちこの *LOOP* が繰返しを制御する.

```
(FACT_i≡    ; FACT_i を次のような actor として定義.
  (≡≡⟩ [request: =n  reply-to: =c]
                    ; メッセージのパターン
    ([request: [1 n] reply-to: c] ==⇒
                    ; 要求部が1とnの対
    ; 継続部がcのメッセージが作られ LOOP に送られる.
      (LOOP≡
          ; LOOP は次行以下に定義される actor.
        (≡≡⟩ [request: [=acc =cnt] reply-to:
              =d]
          ; LOOP の受理するメッセージのパターン
          (rules cnt        ; cnt に対する場合わけ
            (≡⟩ 1           ; もし cnt が1なら
              (d ⇐= [reply: acc])
                            ; d に acc を送る.
            (≡⟩ (>1)        ; もし cnt が1より大なら
              (LOOP ⇐==
              ; LOOP に次行以下のメッセージが送られる
                [request: [(acc*cnt) (cnt-1)]
                                ; 要求部
                  reply-to: d]) )))) )))
                                ; 継続部
```

$FACT_i$ に受理されたメッセージ中の数Nはnに，継続Cはcに束縛され，1とNの対 [1 n] を初期値としたメッセージがCとともに *LOOP* に送られる．1は *acc* に，nは *cnt* に，Cはdにそれぞれ束縛され，*acc* はアキュムレータとして，カウンタ *cnt* の値を1つづつ減しながらそれが1になるまで *acc* を用いて累積的に掛け合わせてゆくことを繰返す．この際 *LOOP* はいつも同じ継続Cを受けとる，即ち，再帰的実現の場合のように新たな継続C', C''などは生成されない．

メッセージ M_1=[request: 3 reply-to: C] が $FACT_i$ に到着してからCに M_3=[reply: 6] が送られるまでのイベント図は**図-9** のようになる．

5. 多重プロセス/分散処理系のモデルとして

メッセージを受けることによってある定められた挙動を示す対象 (object) である actor は，本質的に手続的な対象であるプロセスやプロセッサをモデル化する[7),13)]ばかりでなく，情報格納の機能をもつ純粋にデータ的な対象であるメモリ，バッファ，メモリバンク，ファイルなどもモデル化する．また，大量のデータの上にそれを管理使用するためのソフトウェアを被せたものと概念的に見ることのできるデータベースシステムや，その構成要素も actor によってモデル化できる．それ故，こうしたプロセス，プロセッサ，メモリ，データベースなどを，様々な緊密度と位相で結合した計算機システムが，互にメッセージを伝送し合う actor の集団として表現できることは明らかであろう．

5.1 分散型処理系

分散型処理系 (distributed system) の定義は多くの研究者によって提供されている[6),24)]が，ここでは多数のプロセッサと記憶装置が何らかの方式で結合されたもので，システム全体を制御する中枢的部分がハードウェア的にもソフトウェア的にも存在しないものと定義する*．オペレーティングシステム及びデータベースの非集中度や実際のハードウェアの地理的な分散度などにより，ARPA 網等の大規模な計算機網から 10^5 以上のマイクロプロセッサを結合したシステム[31)]や，Hoare[18)]の communicating sequential process まで，広いスペクトラムの多重プロセス／多重コンピュータシステムがこの定義で把えられる．

分散型処理系を原理的に特徴づける最も重要な性質は，システムのいかなる構成要素も各時点でのシステム全体についての正確な情報を持ち得ないということである．メッセージ伝送のための物理的な“遅れ”が無視できないことや，1つの構成要素が他の多数の構成要素の時間的に変ってゆく状態をモニタすることの繁雑さのために，常に局所的な情報のみに基づいてシステム各部が動作しなければならない．

このことは，集中型処理系の計算モデルに対して仮定されているシステム各部に対する一様，絶対的な観

* より簡単な定義としては，“複数プロセッサシステムで相互のコミュニケーションのために共有する中心的記憶場所を用いないもの”[23),40)] などがある．

測系が使えないことを意味し，物理学において絶対的時空間から相対的時空間の使用を余儀なくされた経緯に類似する．それ故，分散型処理系を整合的に記述・モデル化するためには，この言わば“相対論的な観点”がモデルの中に組み込まれるべきである．

5.2 イベントの局所性

3.2 で見たように，ACTOR 計算モデルで最も基本的であるイベント (event) は，メッセージの actor への到着として定義された．イベントの定義として，メッセージの到着でなくて発信を用いなかった理由は，互に物理的交渉のない異なった2つの actor からのメッセージの発信に対して時間的前後関係が決るにはシステム全体に共通して使える大域的時間軸 (global time axis) を用いなければという事実である．またそのような時間的前後関係の比較の必要性はきわめて少い．(全国的な計算機網が存在すると仮定して，北海道にあるホストからのメッセージの発信と九州のホストからの発信の前後関係を論ずるよりも，双方から発信されたメッセージが同一の目的地 (例えば東京のホスト) にどちらが先に到着したかという事の方がより重要であろう．)

3.4 で述べた arrival ordering は各々の actor に到着するメッセージの到着順を定めるものであったが，ここで注意したいのは，この順序づけは個々の actor の間で互に全く独立に定められることである．異なる actor に対する各 arrival ordering は個々に異なった局所的時間軸 (local time axis) を定める．よって，異なる2つの actor A_1 と A_2 がもつ局所的時間軸上の時刻の比較が可能になるためには，A_1 及び A_2 を標的にもつ適当な2つのイベントが存在して，その間に何らかの方法で (例えば activation ordering によって) 順序づけが可能である必要がある．

5.3 情報の流れの局所性

ACTOR 計算モデルにおいて，actor 間の情報の伝達はメッセージによってのみ行われる．3.1 で触れたように，ある actor はその acquaintances に属する actor にだけ直接メッセージを送ることができた．actor A が他の actor B を (正確にはその名前を) 知っている (knows-about) ためには，(i) A がその誕生時から B を知っているか，(ii) 以後何らかのメッセージを受けとった結果として B を知るようになったかのどちらかである．また，A が actor C にメッセージを送って別の actor D の名前を C に知らせるには，A が D を知っていなければならない．

このように，ACTOR モデルでは，actor 間の情報の流れ (それは他の actor の存在に関する知識の伝達に集約されるが，) に強い制限がある．この点をもう少し詳しく説明するために，1つの actor に対して，その局所的時間軸上の各時点で定まるその actor の知り得る actor の集合: acquaintance vector[13] を考える．acquaintance vector は局所的時間軸に沿って変化するが，その変化に対する制約を明らかにするために，我々は以下に定義するイベントの参加者 (participants) という概念を用いる．

あるイベント E の参加者は次のうちのいずれかである．(i) E の標的である actor，(ii) その標的の acquaintances の要素，(iii) E のメッセージ中に含まれる (その名前が言及される) actor，あるいは (iv) E によって生成される actor．すべての actor は，その誕生時に，有限な acquaintance vector を持ち，その要素は誕生の際のイベントの参加者である．この acquaintance vector は，その actor がメッセージを受理するたびに，そのメッセージ中に言及されている actor (それは常に有限個) を新たな要素として付加する．勿論，actor はその acquaintance vector の要素をいつ何時でも忘れることができる．逆に，その時点での acquaintance vector の要素をすべて憶えていてもよい．

こうして，actor はその局所的時間軸上の点において，その acquaintance vector に属さない actor にメッセージを送ることはできないので，情報の流れに局所性が保たれる．特に，あるイベント E がイベント E' を引き起す場合，E' の標的である actor 及び E' のメッセージの中で言及されるすべての actor もまた E の参加者の1人でなければならないことになる．

上述の情報の流れの局所性のために，メッセージが宛名を持たない放送 (broadcast) によって伝達されることは概念的にはあり得ない．

また，ACTOR 計算モデルでは，新しい actor が動的に生成され得るので，ある actor から見て，ある時点で“系の中に存在するすべての actor”という概念がそもそも意味をなさない．しかしながら，この情報の流れの局所性は，分散型処理系におけるプロセッサの結合位置の動的変化を ACTOR モデルで把えるのに何ら支障をきたさないことを注意しておく．

6. 多重プロセス／分散処理系の仕様検証技法

1つの多重プロセス系や分散処理系が，系全体として望みどおりに動作するためには，それを構成する各プロセス，プロセッサの制御プログラム—即ち，OS—の機能が最も重要である．このような制御プログラムの良い設計，正しい実現のためには，系全体に要求される挙動がどのようなものであるかを厳密に記述する仕様が是非とも必要である．また，制御プログラムや支援プログラムが与えられたとき，それらが仕様に定められたとおりに正しく実現されているか検証することが出来ることが強く望まれる．特に多重プロセス／分散処理系の場合，高度の並列処理と微妙なタイミング，同期に系全体の振舞が強く依存するため，設計実現段階での竦 (deadlock) や永久待ち (starvation) の可能性の検出・防止に細心の注意を払わねばならない．それには，多重プロセス／分散処理系に適した仕様・検証技法が必要となる．

ACTORモデルに基づく並列プログラムの形式的仕様及び検証技法が研究されている[34]．この技法においてactorの挙動がlocal state(局所的状態)と呼ばれる数学的に定義されるactorの状態を用いて外部的（即ち，インプリメンテーションと独立）に記述できる．この技法を用いて，多重プロセス系で共有・使用されるモジュールの機能や挙動の仕様が得られる．また，この技法とともに，3.で述べたイベント間の順序づけによる制約条件の適当な形式的言語による記述[7),8)]を併用して，モジュール，プロセス間の相互作用及び系の全体としての振舞に対する仕様を書くことができる．

このような仕様技法及びこの仕様に基づく検証技法についてはここで立ち入らないが，34) において，簡単な郵便局の例を用いて，多数の顧客，係員，集配人を並列に走るプロセスとみなして，それらの相互作用や系（郵便局）全体としての機能の仕様 (task specification) などが与えられ，その実現 (implementation) を検証する例が示されている．また，40) では，多数の旅行代理店から並列的にアクセスできる航空券予約システムのモデルの仕様とその実現の検証が与えてくれている．さらに，多数異種の入出力装置をタイムシェアリングの利用者が効率よく使用するために実際に開発された衛星コンピュータの制御プログラムの仕様・検証のためにこの手法が用いられている[38]．

7. ACTOR モデルの諸応用

本節では，ACTORという概念についてのもう少し基礎的な研究や幾つかの分野への応用について簡単に触れて，本稿におけるACTOR理論の解説を終えることにする．

7.1 Scott-Strachey モデル

D. Scott は lambola calculus[5] の数学的モデル[30]を作り，プログラムの“意味”としてある種の数学的関数を対応づける denotational semantics (表示的プログラム意味論)[32] の分野を開拓した．ACTOR 計算モデルはイベント間の順序づけに関する公理化にその基礎を置くが，3.5 で説明した純 actor の挙動はある連続汎関数の極限として表わせることが示せる[14]．また，actor の local state という概念もある種の連続関数として構成されると考えられる[34]．この他にも，表示的プログラム意味論の枠組の中で actor 理論の研究がある[15]．

7.2 オブジェクト指向型言語のモデル

SIMULA[4], CLU[25], SMALL TALK[19] などの抽象データ型[26]の概念をもったオブジェクト指向型プログラミング言語に対する簡潔で明析なセマンティクスとして ACTOR 計算モデルを用いることができる．これらの言語は並列処理機能を備えていないので，きわめて大雑把な言い方をすれば，ACTOR計算モデルはこれらの言語のセマンティクスとなる計算モデルを並列性を持たせることができるように拡張したものと見ることもできる．

7.3 並列性をもつ抽象データ型

3.1 で述べたように，1つの actor が受理するメッセージのパターンは固定しており，その actor はその受理したメッセージに対する反応によって特徴づけられるので，データ的な振舞をする actor は，ある抽象データ型の生成例 (instance) と考えられる．さらに，このような actor は並列計算に用いられるから，並列性をもつ抽象データ型の生成例と見なすことができる．

抽象データ型に対する形式的仕様技法には様々なアプローチ（例えば 28) の公理的な方法など）があり，27), 33) などによく概観されている．しかし，並列性を許す抽象データ型に対する仕様技法の研究は 35) に見られる程度である．

7.4 同期機構 (Synchronization Mechanism)

並列に走るプロセスが協力して目的を達成するため

には，プロセス間の同期が最も重要である．ACTORモデルでは，arrival ordering (3.4 参照) の線形性を仮定して，メモリ・セルを用いて同期をとることが示せる[9]．さらに Hoare のモニタ[17]の欠陥[36]を改善した同期機構である serializer[1),16)]は，同期をとって利用したい資源をこの機構で囲い込み (encase)，資源へのアクセスを制御する方式をとっている．

7.5 インプリメンテーション

actor という概念を現実の手続的な対象 (object) として実現する1つの手段は，4.で説明に用いたプログラミング言語 PLASMA を用いてコードを書くことである．PLASMA の処理系は MIT の AI. Lab. で稼動中である．その他に 29) において，制御構造に強い制限を加えた μ-actor という概念の実現が報告されている．

個々の actor にマイクロプロセッサを割り当てて，ハードウェアにかなり近い部分から実現してゆこうという考えも話題に上っている．この際にプロセスの廃品回収 (garbage collection) の問題が重要となる[3]．

7.6 ビジネス・オートメーションへの応用

actor のような手続的な対象は，ビジネス・オートメーションの分野に活用できる．紙の書式やドキュメントを部局の間で回して事務を進めるかわりに，TV ディスプレイ上の映像による書式を actor として実現し，この actor を各部局にある TV 端末の間でやりとりする．そのような actor に随伴する手続 (即ち，script, 3.1 参照) によって，TV 端末を使う事務員に対して映像書式の記入法を指示したり，記入された項目のチェック，項目間の整合性の検証などを行うことができる[40]．この方式によって，書式，ドキュメントシステムの柔軟性や保全性及び処理能率の向上が多大であると予想される．

7.7 アニメーションへの応用

2次元平面上で，複数個の比較的単純な物体 (例えば，チャーリ・ブラウン，テニスボール，仔犬等) が同時に運動し，互に何らかの作用を及ぼし合うアニメーションフィルムは，actor の恰好な応用分野である．Kahn[20]は actor に基づいたアニメーション用の言語を開発し，それを用いたインテリジェントなアニメーション作製用システムを，MIT の LOGO Lab. において実験的な児童教育用道具として用いたと 20) に報告されている．

---

* 入手しやすい文献を優先的に示したので，着想が最初に発表されたオリジナルな文献でないことがある。

8. 謝 辞

本稿の原稿を読んで筆者に貴重なコメントを与えた東京工業大学情報科学科の角田博保氏に感謝する．

参 考 文 献*

1) Atkinson, R. and Hewitt, C.: Specification and Proof Techniques for Serializers, IEEE Trans. on Soft. Eng. Vol. SE-5, No. 1, pp. 10-23 (1979).
2) Baker Jr., H.: Actor System for Real-time Computation, (Ph. D. dissertation) Tech. Report TR-197, Lab. for Comp. Sci., MIT, (Mar. 1979).
3) Baker Jr., H. and Hewitt, C.: Incremental Garbage Collection of Processes, Proc. ACM Symp. on Artificial Intelligence and Prog. Lang. SIGPLAN Notice Vol. 12, No. 8 (Aug. 1977).
4) Birtwistle, G., Dalh, O-J., Myhrhaug, B., and Nygaard, K.: SIMULA Begin, Auerbach, New York (1973).
5) Church, A.: The Calculi of Lambda Conversion. Annals of Math. Studies, No. 6, Princeton Univ. Press, Princeton, (1941).
6) Enslow Jr., P.: What is a Distributed System?, IEEE Computer, Vol. 11, No. 1 (Jan. 1978).
7) Greif, I.: Semantics of Communicating Parallel Processes, (Ph. D. dissertation) Tech. Report TR-154, Lab. for Comp. Sci., MIT, (Sept. 1975).
8) Greif, I.: A Language for Formal Problem Specification, CACM Vol. 20, No. 12 (Dec. 1977).
9) Greif, I. and Hewitt, C.: Actor Semantics of PLANNER-73, Proc. ACM SIGPLAN-SIGACT Conf. Palo Alto, CA., (1975).
10) Hewitt, C.: PLANNER: A Language for Manipulating and Proving Theorems in a Robot, Proc. Int. Jnt. Conf. on Artificial Intelligence, Washington D. C., (1969).
11) Hewitt, C.: A Universal Modular Actor Formalism for Artificial Intelligence, Proc. Int. Jnt. Conf. on Artificial Intelligence, Palo Alto, CA., (1973).
12) Hewitt, C.: Viewing Control Structures as Patterns of Passing Messages, J. of Artificial Intelligence, Vol. 8, pp. 323-364 (1977).
13) Hewitt, C., and Baker Jr., H.: Laws for Communicating Parallel Processes, Proc. IFIP Congress, Toronto (1977).
14) Hewitt, C. and Baker Jr., H.: Actors and Continuous Functionals, Proc. IFIP Working Conf. St. Andrews, (1977).

15) Hewitt, C. and Attardi, G.: An Axiomatic Denotation Specification of a Concurrent Programming Language, Working Paper, AI Lab. MIT, (May 1978).
16) Hewitt, C. Attardi, G., and Lieberman, H.: Specifying and Proving Properties of Guardians for Distributed Systems, Working Paper, AI Lab., MIT, (Oct. 1978).
17) Hoare, C. A. R.: Monitors: An Operating System Structuring Concept, CACM Vol. 17, No. 10, (Oct. 1974).
18) Hoare, C. A. R.: Communicating Sequential Processes, CACM Vol. 21, No. 8 (Aug. 1978).
19) Ingalls, D.: The Smalltalk 76 Programming System Design and Implementation, Proc. ACM Symp. on Principles of Prog. Lang. (Jan. 1978).
20) Kahn, K.: An Actor-based Computer Animation Language, Working Paper AI Lab., MIT (Feb. 1976).
21) Kahn, K.: Three Interactions between AI and Education, Machine Intelligence Vol. 8 (1977).
22) Karp, R. and Miller, R.: Parallel Program Schemata, J. of Comp. and Sys. Sci., Vol. 3, pp. 147-195 (1969).
23) Lamport, L.: Time, Clocks and the Ordering of Events in a Distributed System, CACM Vol. 21, No. 7 (July 1978).
24) Le Lann, G.: Distributed Systems—Toward a Formal Approach—, Proc. IFIP Congress, Toronto (1977).
25) Liskov, B., et al.: Abstraction Mechanisms in CLU, CACM Vol. 20, No. 8 (Aug. 1977).
26) Liskov, B. and Zilles, S.: Programming with Abstract Data Types, Proc. ACM Conf. on Very High Level Lang., SIGPLAN Notice Vol. 9, No. 4 (1974).
27) Liskov, B. and Zilles, S.: Specification Techniques for Abstract Data Types, IEEE Trans. on Soft. Eng. Vol. SE-1 pp. 7-19 (Mar. 1975).
28) Nakajima, R., Honda, M., and Nakahara, H.: Describing and Verifying Programs with Abstract Data Types, Proc. IFIP Working Conf. St. Andrews, (1977).
29) 小山 均, 木島裕二, 田中幸吉: μ-actor の実現と知識表現の構造, 情報処理, Vol. 19, No. 9 (1978年9月).
30) Scott, D.: Outline of a Mathematical Theory of Computation, Tech. Mono. PRG-2, Oxford Univ., Comp. Lab. (1970).
31) Sullivan, H., Bashkow, T., and Dlapphoz, D.: A Large Scale Homogeneous Fully Distributed Parallel Machine I, II, IEEE Comp. Soci. and ACM Symp. on Comp. Arch. (1977).
32) Tennent, R.: The Denotational Semantics of Programming Language, CACM Vol. 19, No. 8 (Aug. 1976).
33) 鳥居宏次, 二木厚吉, 真野芳久: プログラミング方法論の展望, 情報処理, Vol. 20, No. 1 (1979).
34) Yonezawa, A.: Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics, (Ph. D. dissertation) Tech. Report TR-191, Lab. for Comp. Sci. MIT, (Dec. 1977).
35) Yonezawa, A.: A Specification Technique for Abstract Data Types with Parallelism, to appear in Lecture Note in Comp. Sci. (E. Blum and T. Takasu Eds), Springer-Verlog (1979).
36) Yonezawa, A.: Comments on Monitors and Path-expressions, J. of Information Precessing, Vol. 1, No. 4 (1979).
37) 米澤明憲: ACTOR 理論と分散型処理系, 情報処理学会ソフトウェア工学研究会資料 7-3, (1978年7月).
38) 米澤明憲: 高度な並列性を内部にもつソフトウェアシステムに対する仕様及び検証技法について, 情報処理学会第 20 回プログラミングシンポジウム予稿集 (1979).
39) Yonezawa, A. and Hewitt, C.: Symbolic Evaluation Using Conceptual Representations for Programs with Side-effects, AI-Memo 399, AI-Lab. MIT (Dec. 1976).
40) Yonezawa, A. and Hewitt, C.: Modelling Distributed Systems, Proc. Int. Jnt. Conf. on Artificial Intelligence, Boston (Aug. 1977). Also in Machine Intelligence Vol. 9 (1979).

(昭和54年3月13日受付)