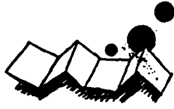
解 説

協同型処理におけるプログラミングパラダイム†



田 畑 孝 一†† 杉 本 重 雄††

1. 協同型処理の概念

計算機のプログラミングにおける協同型処理の必要性を説明するために，社会生活におけるわれわれの協同作業（協同型処理）の例をあげる．鉄道の駅の業務は駅長や駅員の協同作業から成り立っている．駅長は管理者としての係りであり，駅員の係りとしては列車運行係り，乗車券発売係り，改札係り，案内係りなどがある．ほとんど列車が通らない田舎の駅でない限り，駅の業務はこれらの係りを（専任的に）分担する複数の駅員の協同作業から成り立っている．分担による協同作業形態は，駅業務の能率のためのみならず，駅業務の正確さ・安全性のために，欠くことができない．駅に限らず，社会生活は，ありとあらゆるところで，意図的に構成されたグループ・組織や，知らず知らずに成り立っているグループにおける協同作業によって，営まれているといっても過言ではない．それはまた，人間社会の原始時代からの発展の一つの要因となっているに違いない．

計算機が人間の論理的・知的活動を代行するものと考えれば，計算機のプログラムにおける協同型処理の必然性が，上述のことから素直に理解できる．協同型処理に対立する概念を単一型処理ということにすると，単一型処理は，他の処理と協同しないものを指す．通常のプログラム，いわゆるシーケンシャルプログラム (sequential program) は，単一型処理プログラムである．望みのアルゴリズムを単一型処理プログラムで表すことができるということで，協同型処理プログラムの役割りに積極的な理解を示さない人もいる．上の駅の例で，確かに駅の業務は理屈の上では，一人で行う（単一型処理に相当する）ことができるかもしれない．しかし，現実には上に述べたように，よほど列車が通らない田舎の駅でない限り，駅員一人ということはない．言い換えれば，単一型処理のプログラムで表されることは，列車がほとんど通らない田舎の駅の業務の程度のことに過ぎないのかもしれない．単一型処理プログラムで十分と思っている人は，何気なく書いた自分のプログラムが，実行の際，別に用意されている入出力プログラムと協同型処理をしている事実を知らないのかもしれない．

ハードウェアの製作技術の発展にともない，十分余裕をもって協同型処理プログラムを実行させ得る環境が整ってきている．それゆえ協同型処理形態は，それが人間社会の発展の一要因となってきたのと同様，今後のソフトウェアの発展の一つの要因になるに違いない．

ある目的のためにつくられたグループは人々の分担による協同作業の形態で全体としての仕事を遂行している．それと同様に，ある目的のために構築されたシステムはいくつかの構成要素からなり，それらの構成要素が相互に関連し合って協同型処理を行い，全体として機能している．システムをこのような観点からとらえたとき，それを協同型処理システムという[1]．システムの各構成要素の振る舞いをそれぞれプログラムで表現すると，システム全体は相互に関連した複数のプログラムの協同型処理として表現される．システムによっては，構成要素は静的というよりむしろ自律的な性格をもつ動的な実体である．この場合にはそれらの振る舞いを表現したプログラムは互いに同時にあるいは並行に動作していることになる．

協同型処理 (cooperating processing) は，協同の関係にあるいくつかの処理からなっている．それら各処理にそれぞれ一つの処理実体が対応し，各処理はそれら処理実体によってそれぞれ遂行されているとみなされる．協同型処理と関連のある概念に

- 並 列 処 理
- 並 行 処 理
- 分 散 処 理

がある．並列処理 (parallel processing) とは，協同型処理の関係にあるすべての処理実体が同時にそれぞれの処理を遂行していることを指す．並行処理 (con-

† Programming Paradigms for Cooperating Processing by Koichi TABATA and Shigeo SUGIMOTO (University of Library and Information Science).

†† 図書館情報大学

current processing) とは，協同型処理の関係にある処理実体が，形式上はすべて同時にそれぞれの処理をしていることになっているが，実際にはある時点ではいずれか一つの処理実体のみが処理を遂行しており，ある時間間隔をとればその間のどこかでいくつかの，あるいはすべての処理実体が処理を遂行していることを指す．分散処理 (distributed processing) とは，協同型処理の関係にある処理実体が，互いに距離を隔てて所在し，それぞれの処理を遂行していることを指す．距離が近くても，協同型処理の関係にある処理実体が異なる物理装置内に存在して，それぞれの処理を遂行している場合も分散処理に含まれる．

2. 協同型処理システムの構成とプログラミングパラダイム

2.1 協同型処理システムの構成要素

システムは互いに関連したいくつかの構成要素から成り立っている．構成要素となるべき実体の性格は，システムをどのような観点からとらえるかによって，異なってくる．同じシステムを次の二つの観点からとらえることができる．

(1) システムは相互に連関するいくつかの機能を構成要素として構成されている．

(2) システムは相互に連関するいくつかの物（対象）を構成要素として構成されている．

現実の世界はすべて物から構成されているので，(2)の観点は自明である．それに対して，(1)の観点からは物そのものではなく，物の機能，すなわち，物の働き・作用に注目し，それらを要素としてシステムが構成されているとみなす．(2)の場合には，プログラム上でそれらの物を具象的に表現するが，歴史的にはより抽象的な(1)の観点からのプログラミングの方が先行していることは興味深い．

(1)の場合は協同型処理の関係にある処理実体は機能（を果たすもの）を表し，(2)の場合は協同型処理の関係にある処理実体は物そのものを表している．機能を表す処理実体は，普通，プロセス (process) と呼ばれている．この場合システムは複数のプロセスから構成されるので，協同プロセスシステムとも呼ばれる．一方，物を表す処理実体は，普通，オブジェクト (object) と呼ばれ，システムは協同型処理の関係にある複数のオブジェクトから構成されているとみなされる．

1 章であげた鉄道の駅をシステムの例とする．(1)の立場からは，駅システムは管理機能，列車運行機能，乗車券発売機能，改札機能，案内機能のそれぞれを果たすプロセスから構成される．(2)の立場からは，駅システムは駅長，業務管理簿，列車運行駅員，列車運行制御装置，乗車券発売駅員，乗車券発行装置，金銭登録器，改札駅員，改札口，案内駅員，案内表示装置などのオブジェクトから構成される．

オブジェクトは形がある物体はもちろん，思考や行為の対象となるものを表現することができ，それらは観念的なものであってもよい．列車運行駅員，乗車券発売駅員，案内駅員のおのおのがもつ異常運行時対策の知識，旅行経路案内の知識，乗客混雑時対策の知識，また列車運行制御装置のもつ列車運行データなども，（駅員や装置の構成要素となる）オブジェクトとなりうる．

2.2 構成要素の相互関連

協同型処理システムは，相互に関連する構成要素から成っている．協同型処理を行うため，そこには構成要素が互いに関連し合うための何らかの手段が用意されている．いいかえれば，協同型処理の関係にある処理実体には互いに関連し合うための手段が必要である．そのための手段には次の媒体を介したものがある．

(1) 共有情報格納場所

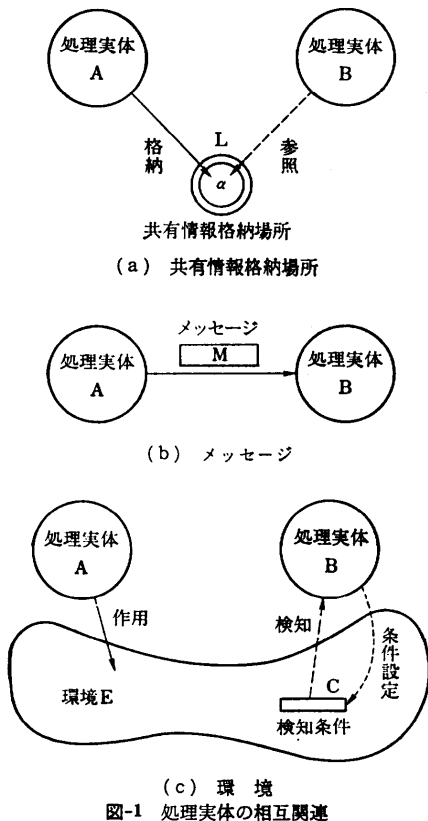
(2) メッセージ

(3) 環 境

(1)の共有情報格納場所はプログラミング言語では共有変数に相当する．二つ以上の処理実体がある情報格納場所を共有し，その場所に処理実体が互いに示し合わせた何らかの情報を格納し，別の処理実体がその情報を参照することによって相互に関連し合う．**図-1 (a)** の例では情報格納場所 L に対して，処理実体 A が情報 α を格納し処理実体 B はそれを参照する．プログラミング言語においては，共有変数に処理実体がある値を代入し，別の処理実体がその値を参照することによって相互に関連し合うことに相当する．

(2)では，処理実体は別の処理実体にメッセージを送り届けることによって相互に関連し合う．各処理実体にはメッセージを送り出す機能，およびメッセージを受け取る機能が備わっている．**図-1 (b)** の例では処理実体Aから送り出されたメッセージMは処理実体Bへ送られる．

(1)と(2)では処理実体は，共有情報格納場所内に置かれた情報，あるいはメッセージを用いて，互いに



（a） 共有情報格納場所

（b） メッセージ

（c） 環 境

図-1 処理実体の相互関連

直接的に関連し合うが，（3）では処理実体はそれらを取り巻く環境を介して間接的に関連し合う．処理実体は環境に作用しそれを変化させる機能をもつ．また処理実体は環境の変化を検知する機能をもつ．処理実体は環境の変化を検知するのに必要な（検知）条件をそれ自身で設定することができる．図-1 (c)の例で処理実体Aは環境Eに作用しそれを変化させる．環境Eが変化し，処理実体Bによって設定された検知条件Cが満たされると，検知結果が処理実体Bにもたらされる．プログラミング言語では環境は，通常，それを表す変数の値で表現される．

2.3 協同型処理システム記述のためのプログラミングパラダイム

パラダイム (paradigm) は範例（模範とすべき例：an example serving as a model) を意味することばである．何かなすべきことがあるときそれにふさわしい良きお手本（範例）があればありがたい．プログラミングパラダイム (programming paradigm) とは，ある問題をプログラミングする際，その問題のプログラム記述にふさわしい範例のことである．

ところで，協同型処理システムは相互に関連するいくつかの構成要素，すなわち処理実体からなっている．処理実体としては，2.1 に述べたように機能を表すもの（プロセス），および物を表すもの（オブジェクト）の二つの立場がある．処理実体の相互関連の手段としては，2.2 に述べたように，共有情報格納場所，メッセージ，および環境がある．このような処理実体の立場，あるいは相互関連の手段の組み合わせによって，協同型処理システム記述のためのプログラミングパラダイムがいくつか導かれるが，ここではそれらを次のように区分する．すなわち，処理実体が互いに直接的に関連し合うか，環境を介して間接的に関連し合うかでパラダイム全体を2分する．前者はさらに2分しプロセス指向型とオブジェクト指向型のパラダイムに分ける．後者はひとまとめにして環境指向型のパラダイムと呼ぶことにする．

プロセス指向型プログラミングパラダイムでは，機能を表すもの，すなわちプロセスが処理実体とみなされ，それらの相互関連の手段として共有情報格納場所（共有変数），あるいはメッセージが用いられている．オブジェクト指向型プログラミングパラダイムでは，物を表すもの，すなわちオブジェクトが処理実体とみなされ，それらの相互関連の手段としてメッセージが用いられている．環境指向型プログラミングパラダイムでは，機能やオブジェクトを表すものが処理実体とみなされ，それらの相互関連の手段として環境（それを表す変数の値）が利用されている．

プログラミングパラダイムは，一般には，それに適したプログラミング言語を選んで，あるいは新たに設計して，その上で表現される．そのような記述言語によるものとは趣を異にしているが，計算機のオペレーティングシステムも協同型処理システムのための一つのプログラミングパラダイムを提供している．そこでは一つのプロセスは一つのタスクとみなされ，プロセスの生成やプロセス間の相互作用はオペレーティングシステムの助力を得て遂行される．助力の依頼はオペレーティングシステムへのマクロ命令によって行われる．

プログラミングパラダイムをいくつか用意しておくと，与えられた問題を素直に表現しうるものをその中から選択できる．対象問題を表現するのに一つのプログラミングパラダイムで十分であることも多い．しかしながら，問題によっては単一のプログラミングパラダイムでは表現しにくいような異なる性質の部分問題

から構成されているものもある．その場合には複合プログラミングパラダイムが適している．複合プログラミングパラダイムでは，それら部分問題をそれぞれ素直に表現しうる各種パラダイムが利用でき，それらが全体として一つの問題を表現するように統合できる．LOOPS[2] では，手続き指向，オブジェクト指向，データ指向，およびルール指向のパラダイムを統合した複合プログラミングパラダイムを提供している．

3. 協同型処理の記述

3.1 プロセス指向型パラダイム

プロセス指向型プログラミングパラダイムでは，機能を表すもの，すなわちプロセスが処理実体とみなされ，それらの相互関連の手段として共有情報格納場所(共有変数)，あるいはメッセージが用いられている．

各プロセスが処理すべき仕事は，普通，手続きとして，すなわち逐次的な動作手順として，それぞれ表現される．手続きの中には自プロセスの動作はもちろんのこと，他プロセスとの相互関連に関する動作が記述される．このパラダイムでは，プロセスの体系の形態，またプロセスの相互関連の立場からプロセス間での変数の共有，メッセージ授受の方法が考慮の対象となる．

(1) プロセスの体系

プロセスの体系には動的なものと静的なものがある．動的なプロセス体系では実行の進行とともにいくつものプロセスが生成しあるいは消滅する．あるプロセスが与えられた手続きに従って実行する動作の一環として，新たなプロセスを生成したりあるいは自らを消滅させている．静的プロセス体系では，あらかじめ定められたすべてのプロセスが実行開始時の初期設定時に生成され，実行中に新たに生成されることはない．

静的なプロセス体系の例は Concurrent Pascal のそれである[3),4)]．Concurrent Pascal ではあらかじめプロセス型を定義しその振る舞いを手続き的に記述しておく．プロセス型からその具現実体としてのプロセスを生成する．プロセスはそのプロセス型をもつ変数(固定変数：permanent variable) で表され，その型をもつ変数を宣言することがプロセスの生成に相当する（生成は宣言部においてのみ可能)．プロセスはプログラム実行開始直後に初期設定され，以後永続して生き続ける．プロセス型はいくつも定義でき，また同一プロセス型から複数のプロセスを生成できる．

動的なプロセス体系の例をいくつかあげる．その最初の例として “fork/quit/join” 方式をあげる[5)]．あるプロセス（すなわち親プロセス）が “fork” を実行すると新たなプロセス（すなわち子プロセス）が生成され，親プロセスと並行に実行される．子プロセスは実行の完了に際し “quit” を実行して自分自身を消滅させる．親プロセスは，自分の子プロセスが “quit” を実行するのを待つ必要があるときは，“join” を実行することによって自分自身の進行を一時停止することができる．PL/I は，その多重タスク機能としてこの方式を取り入れているプログラミング言語である．手続きを TASK として CALL するのが “fork” に相当し，その手続きを本体とする子プロセスが生成される．その手続き中の RETURN が “quit” に相当する．そして，親プロセスの実行する WAIT が “join” に相当する．

Ada[6)] ではプロセスのことをタスク（より正規にはタスクオブジェクト）という．宣言的にタスクを生成しうるのみならず実行時に動的にタスクを生成しうる．あらかじめタスク型を定義し，その振る舞いを手続き的に記述しておく．プログラム単位において，その宣言部でタスク型から（その具現実体としての）タスクオブジェクトを生成しうる．また実行本体部でも実行文中の割り当て子 (new) を用いてタスク型からタスクオブジェクトを生成しうる．タスク型は複数定義でき，また同一タスク型から複数のタスクオブジェクトが生成できる．上述のプログラム単位はタスク型のもの，すなわちタスク単位（並列に実行しうるプログラム単位）でもよいので，タスク（親タスク）は子タスクをもつことができる．子タスクが終了しないと親タスクは終了できない．

Modula-2[7)] ではプロセスを動的に生成する．すなわちプログラム実行中にプロセスを（あらかじめ用意された）手続きの一つの具現実体として生成する．この言語ではプロセス体系をコルーチン方式で実現しており，各プロセスは一つのコルーチンとして表される．

コルーチン方式では，制御（プログラムの実行権）はどの時点においてもいずれか一つのコルーチンにあり，コルーチン間の制御の移動はプログラム中に書かれている制御移動文によって行われる．コルーチン p の実行中に制御移動文に到達すると，p の実行は一時中断され，その文で指定されたコルーチン q に制御が移動する．コルーチン p のそのときの状態は後に実

行が再開するまで保存される．コルーチン q は以前に一時中断されたときに保存された状態において実行が再開される（実行再開は制御移動文の次から行われる）制御の移動の管理をプログラム自身が行わねばならないので，プログラムの記述は必ずしも容易ではない．

LISP に基づく並行プログラミング言語 Concurrent LISP[8),9)] では，プロセス起動関数の実行によって子プロセスを生成起動する．プロセス起動関数は，子プロセス（必要に応じて複数）とその子プロセス（のおのおの）が評価すべき形式を指定する．この言語ではプロセスの再帰的な生成起動ができる．すなわちある関数の定義に際し，その関数を含む形式を評価する子プロセスを生成起動するプロセス起動関数を，その関数の定義（の一部）に用いることができる．プロセスの再帰生成起動を許すプロセス体系では，将来にわたって次々に生まれうるプロセスに前もって名前を与えることが難しいので，プロセス間の相対的な世代間関係を指定する方式が導入されている．たとえば，あるプロセスから“親のすぐ下の弟の第1子に相当するプロセス”が指定でき，そのプロセスの名前を知らなくても，それとの間で協同型処理ができる．

（2） 共有変数による相互関連

プロセス1とプロセス2がある対象Cを共有し，互いに他が使用していないときにそれを利用するものとする．その値が0のときCの空き（利用されていないこと）を示す変数 s を導入し，プロセスはCの利用に際し s に1を代入する．

s=0 の状態でプロセス 1,2 がほとんど同時にCの利用を思い立ったとする．プロセス1が，

（i） s の値を読み取る，そして

（ii） その値を0と判断し s に1を代入する．

もしプロセス2が（i）の直後，（ii）の直前の間に s の値を読み取ると，同様にその値を0と判断し s に1を代入する．その結果，Cはプロセス1と2の両方によって（相手が利用中とも知らずに）同時に利用されることになる．

このような不合理は，一般に，共有変数をあるプロセスが読み取り，あるいは更新している最中に，他のプロセスがそれを更新すると起こりうることで，共有変数へのアクセスの相互排除の問題と呼ばれている．この問題に対処するには Brinch Hansen の critical region[10)] の概念に依るのがよい．

上の例で，動作手順（i）と（ii）からなる領域Rは共有変数 s にとって危機的な（critical な）状況にある．プロセス1が領域R実行中にプロセス2が共有変数 s にアクセスすると不合理が起こりうる．危機を回避するにはプロセス1が領域Rを実行中にプロセス2には共有変数 s の値の変更はもちろんその読み取りもさせないことである．

ある共有変数（一つまたは相互に関連した複数の変数）についての critical region とは，その共有変数へのアクセス（読み取りあるいは更新）を含んだ一連の動作手順からなる領域で，あるプロセスがその領域を実行中にもし他のプロセスがその共有変数にアクセスすると，その共有変数に危機的な状況が起こりうるものをいう（領域の大きさはできるだけ小さいものがよい）．上の例のRは共有変数 s についての critical region である．同一共有変数についてのすべての critical region に関して，そのいずれか一つにあるプロセスが存在しているときには，その他のプロセスはいずれの critical region にも入れないようにして危機を回避する．つまりそのプロセスが領域を出るまで，他のプロセスを待たせる．

Concurrent Pascal では同一共有変数の critical region がいくつものプロセス（型）手順中に分散して表れないように，それらを一つの monitor（型）と呼ばれるシステム構成要素に集めて記述する．プロセスはその monitor 内の手続きを呼ぶことによって相互に関連する．待ちの処理は monitor で行われる．別の共有変数には別の monitor（型）が対応する．

PL/I については，その言語系自身で critical region を記述することができない．その必要があるときは ENQ (enter queue), DEQ (depart queue) と呼ばれるシステムコールを利用する．ある共有変数にアクセスするための待ち行列を q とするとき，ENQ(q) は q に並んで順番がきたらサービスを受けることを示し，DEQ（q）は q から出ることを示す．

Ada ではタスク間の相互関連はメッセージを利用するのが普通である．あえて共有変数を利用したい場合にはそれにアクセスする手順をもつ二つのタスクをランデブ（後述）させる．critical region はランデブの始めと終わりで囲まれた領域で実現する．

Modula-2 はコルーチン方式なので，共有変数について危機的な状況は起こりえない．

Concurrent LISP では，critical region を表すために，LISP 言語本来の表記になじむものとして，CR (critical region) 関数，CCR (conditional critical region) 関数を導入している．

(3) メッセージによる相互関連

共有変数を介したプロセスの相互関連においてはその変数に対する読み書きに際し危機的状況が起こりうるが，メッセージによる相互関連においてはその心配がない．プロセス間におけるメッセージの授受の方法について Hoare[11] の考え方を紹介する．

プロセスはメッセージの送信意思表示を行う（出力コマンドを実行する）．送信意思表示の一環としてメッセージの受信先のプロセス名の指定を行う．一方，別のプロセスにはメッセージの受信窓口（入力コマンド）が用意されている．受信窓口はメッセージの送信元のプロセスごとの専用になっており，窓口にはそのことを示すプロセス名が付けられている．

メッセージの授受は，送信プロセスが送信意思表示を行いそれに対応する（受信先と送信元の名が一致する）受信窓口で待っているとき受信プロセスがその窓口で能動的な受信意思表示を行った場合，あるいは受信プロセスが受信意思表示を行い自身の受信窓口で待っているとき送信プロセスがそれに対応する送信意思表示を行った場合に行われる．その際，送信プロセスが用意したメッセージの内容が受信プロセスに写しとられる（コピーされる）．

一つのプロセスに複数の受信窓口があって，かつそのいくつかに受信意思表示が行われているとき，対応する送信意思表示が同時にいくつかある場合には，いずれか一つが選択され，メッセージの授受が行われる．

Ada ではメッセージに関し以上の考えに類似した方式のタスク（プロセス）間の相互関連を導入している．Ada では送信意思表示に際し受信先タスク名のみならずその受信窓口名を指定する．受信タスクにはその名が異なる複数の受信窓口が用意されうるが，受信窓口は特定の送信タスクに専用のものではなく，いずれの送信タスクに対しても開放されている．同一受信窓口に対する送信意思表示は到着順サービス方式の待ち行列で応対される．

タスクにおける受信窓口を Ada ではエントリと呼ぶ．タスクの手順中に各エントリに対応したアクセプト文が存在する．手順実行中にアクセプト文に達することが対応するエントリ名の示す受信窓口での受信意思表示と解される．タスクからの送信意思表示は Ada ではそのタスクからの目的タスクのエントリ呼び出しに相当し，その形式はタスク内の手続き呼び出しと同様である．その際，タスク名とエントリ名を指定する．エントリが呼び出され，対応するアクセプト文に達すると，アクセプト文が実行される（呼び出しタスクは待機中となる）．この間の二つのタスクの相互作用をランデブと呼ぶ．ランデブの始めと終わりはアクセプト文の実行開始と終了に対応する．ランデブ中に二つのタスクの間で双方向にメッセージが伝達しうる．いずれの方向においてもメッセージの内容が送信側から受信側に写しとられる．

Concurrent LISP ではメッセージはメイリング関数を用いて伝達される．宛先プロセスを指定してメイリング関数を実行すれば，メッセージはそのプロセスに用意されているメイルボックスに届けられる．

3.2 オブジェクト指向型パラダイム

このパラダイムでは対象世界がすべて物を表すオブジェクトからなると考える．とはいってもあまりにも原始的な要素（たとえば数字）をもとにそれらから直接全体を組み立てるのは実際的でない．原始的要素を用いて作られた構成要素，あるいは構成要素をいくつか用いて作られた構成要素をもとに全体が組み立てられていると考えるのが普通である．このパラダイムでは原始的要素，構成要素，および対象世界（システム）全体をすべてオブジェクトと考える．オブジェクト相互はメッセージで関連すると考える．

(1) アクタ

アクタ理論では[12]原始的要素，構成要素（および対象システム全体）はもちろん，それらの間のメッセージを運ぶ使者もアクタ（actor）と考える．アクタは使者が到着すると，それがもたらす情報を参照して行動をする．アクタAはその行動の一環としてアクタBに使者Mを送る（図-2 参照）．使者Mには，Bへのメッセージ，およびメッセージ受信時Bがその行動の一環として使者M′を送るべきアクタ C の名前が託される（CとしてA自身を指定してもよい）．なおA自身がBとなりうる，つまり自分自身に使者Mを送りうる．アクタはその行動の一環として新しいアクタを創成することができ，それに使者を送ることもできる．

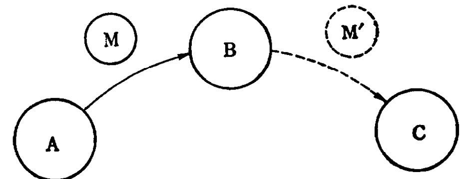


図-2 アクタ

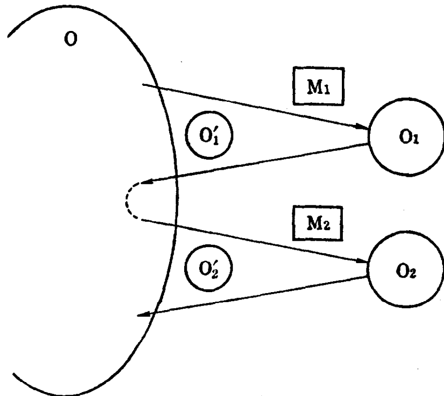


図-3 オブジェクトとメッセージ

（2） オブジェクト

Smalltalk-80[13] では，原始的要素，構成要素（および対象システム全体）をすべてオブジェクトと考える．それらオブジェクトにメッセージが働きかける．オブジェクト O_1 はメッセージ M_1 を受け取るとそれに要請されている行動を行い，その結果としての返答オブジェクト O'_1 をメッセージの送信元 O へ返す（図-3 参照）．一方オブジェクト O の行動は M_1 を O_1 に送り返答 O'_1 を受け取り，次に M_2 を O_2 へ送り返答 O'_2 を受け取り，という形で進行する．メッセージは受信オブジェクト名とそのオブジェクトにいくつか用意された行動のどれを要請するかの選択子（および必要に応じて引数）からなっている．なおオブジェクトは自分自身にメッセージを送りうる．

オブジェクトの行動はそれを定義するクラスにあらかじめ記述されている．クラス（クラスもオブジェクト）にメッセージを送ると，返答オブジェクトとしてそのクラスのインスタンス（具現実体）が生成され実際の活動に利用される．クラスは複数定義でき，また同一クラスから複数のインスタンスが生成できる．クラス(A)に対してサブクラス(B)を定義できる．サブクラス(B)のインスタンスはクラス(A)のインスタンスでかつサブクラス(B)で定義された付加的な行動が可能なものである．

3.3 環境指向型パラダイム

このパラダイムでは処理実体はそれを取り巻く環境を介して互いに関連し合う．環境を介した相互関連の実現方法を 2.2 で示した図-1(c)の例を用いて説明する．図において環境Eの要素の一つを変数 x とし，A あるいはその他の処理実体が x の値を変化させるとする．処理実体Bについての検知条件を C：$x \geqq 1$ とする．x の値の変化は B に関係なく起こるので，条件Cが成立するかを絶えずテストしなければならない．プログラム実行上の難点からこのようなテストを絶えず行うわけにはいかないので，それに代わる次の方法に頼る．すなわち，x の値を変化させた処理実体がその時点にそのことを処理実体Bに通知する．x の値が変化しない限り条件Cの成否は変動しないので，Bはその時点においてのみCをテストすればよい．

（1） 黒 板

エキスパートシステム構築システム AGE[14] では KS (knowledge source：知識源）と呼ばれるいくつかの処理実体が黒板と呼ばれる環境を介して相互に関連する．知識要素全体を互いにより関係の深い知識要素（規則と呼ばれるもの）からなるいくつかの部分に分け，そのおのおのを KS と呼ぶ．黒板には，事実やこれまでにいくつかの KS が書き出した推論仮説や推論帰結が書かれている．それらの情報に適用可能な規則を持っている KS は，それら情報に基づき推論を行い得られた結果を黒板に書き出す．

各 KS が黒板に書かれた情報に対して自身の規則が適用可能かを絶えず調べるわけにはいかないので次のような方法を採り入れている．各 KS はそれを起動するのに必要ないくつかの条件を持つことにする．その条件の一つが成り立つとき KS 中のいずれかの規則が適用可能であるとしておく．行動中の KS は結果を黒板に書き出すとともに特定の事象の発生をそれを待っている KS に通知する．特定の事象の発生とはある起動条件が成立したということを指す．通知を受けた KS が起動される．

（2） アクティブバリュ

LOOPS[2] のもつデータ指向パラダイムにおいては環境を表す変数をアクティブバリュ (active value) と呼ぶ．アクティブバリュ v は特別な形式

#(v, putFn)

をもつ．v に値を書き込むとき（値を変更するとき）行うべき手順 putFn を呼ぶ．

あるプロセスが v を変更するときには v に係わりのあるプロセスを呼ぶ手順 putFn を用意しておけばよい．あるオブジェクトにおける行動で v を変更するときには，v に係わりのあるオブジェクトにそれを通知するメッセージを送る手順 putFn を用意しておけばよい．

4. おわりに

協同型処理プログラミングパラダイムを分類し，そのおのおのの記述を典型的なプログラミング言語での実現法を引用して説明した．

オブジェクト指向型パラダイムの説明がプロセス指向型パラダイムのそれに比べて短くて済むのは，パラダイム自身が簡明であるからである．すなわち，(1)オブジェクト指向型ではその相互関連に共有変数を用いないので，それについて説明の要がない．一方，プロセス指向型ではその相互関連に共有変数を用いる場合には critical region の概念についての理解を欠くことができないので，その説明を要する．(2)オブジェクト指向型におけるメッセージの授受の方式はプロセス指向型のそれに比べて単純である．プロセス指向型におけるプロセスはいずれも能動的な実体である．能動的に行動中の送り手プロセスと受け手プロセスのそれぞれの送信意思と受信意思が時間的に一致した時点においてはじめてメッセージが受け渡される（そのために互いに他を待ち合わせる）．一方，オブジェクト指向型ではオブジェクトは受動的な実体である．いずれのオブジェクトもメッセージの到来をきっかけとしてその行動を開始することになっているので（待ち合わせもなく）単純なものになっている．

オブジェクト指向型はそのパラダイムが簡明なので，より優れているといった見解はありえない．対象問題に合ったパラダイムを選ぶことが肝要である．いうまでもなく現実に使われているプログラムの大部分はプロセス指向型のパラダイムで書かれている．プロセス指向型の典型的なプログラミング言語として，静的なプロセス体系の Concurrent Pascal, 広く利用されている PL/I, これまでの知見を集大成した Ada, コルーチンの Modula-2 はぜひとも引用したいものであった．オブジェクト指向型については，アクタ理論，Smalltalk-80 をその典型としてあげたが，それらを発展させたパラダイムは活発な研究対象となっている[15),16)]．

環境指向型パラダイムにおいては処理実体はどちらかといえば環境の変化そのものに関心があり，変化の原因となる作用が別の処理実体のいずれ（のいくつか）によるものかには必ずしも関心がない．そして処理実体は関心のある変化に対応して自身の行動を決定する．その限りにおいてはこのパラダイムは単純である．けれどもその処理実体の行動の一環として環境への作用が含まれている場合には，対象問題の記述に十分な注意が必要である．そのような処理実体が（いくつか）存在する場合には，ある作用がもととなって，環境に対する連鎖的な作用や循環的な作用が起こりうることを承知しておかねばならない．

環境指向型パラダイムは他のパラダイムと排他的なものではない．対象問題の記述に際し，処理実体の間に環境を介した間接的な関連がある一方で，それらのいくつかあるいは別の処理実体の間に直接的な相互関連が存在しても不自然ではない．LOOPS では環境指向型を複合パラダイムの一環として位置付けている．

以上，協同型処理プログラミングパラダイムについて解説してきた．協同型処理パラダイムは，最近では単にプログラミング言語としての位置付けにとどまらず，エキスパートシステムなど知識情報処理システムの構築の面からも再認識されており，それに対する関心は高いものとなっている．

参考文献

1) 田畑孝一，伊藤　潔，杉本重雄：ソフトウェア工学ハンドブック，第9章，榎本　肇編，オーム社 (1986).
2) Bobrow, D.G. and Stefik, M.: The LOOPS Manual, Xerox Corp. (1983).
3) Brinch Hansen, P.: The Programming Language Concurrent Pascal, IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, pp. 199-207 (1975).
4) Brinch Hansen, P.: The Architecture of Concurrent Programs, Prentice-Hall (1977)
(訳書) 田中英彦訳：並行動作プログラムの構造，日本コンピュータ協会 (1980).
5) Holt, R.C.: Concurrent Euclid, The UNIX System, and TUNIS, Addison-Wesley (1983)
(訳書) 大野　豊監訳，伊藤　潔，広田豊彦訳：並行処理と UNIX, 啓学出版 (1985).
6) Reference Manual for the Ada Programming Language, United States Department of Defense (1983)
(訳書) 情報処理振興事業協会編：最新 Ada 基準文法書　米国規格全訳，bit 別冊，共立出版 (1984).
7) Wirth, N.: Programming in Modula-2, Springer-Verlag (1982).
8) Tabata, K., Sugimoto, S. and Ohno, Y.: Concurrent LISP and its Interpreter, Journal of Information Processing, Vol. 4 No. 4, pp. 195-202 (1981).
9) 田畑孝一，伊藤　潔，杉本重雄：Concurrent

LISP, bit, 5月号臨時増刊, 共立出版, pp. 189-204 (1984).

10) Brinch Hansen, P.: Operating System Principles, Prentice-Hall (1973)
(訳書) 田中穂積, 真子ユリ子, 有澤 誠訳：オペレーティング・システムの原理, 近代科学社 (1976).

11) Hoare, C. A. R.: Communicating Sequential Processes, CACM, Vol. 21, No. 8, pp. 666-677 (1978).

12) Hewitt, C.: Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence, Vol. 8, pp. 323-364 (1977).

13) Goldberg, A. and Robson, D.: SMALLTALK-80 The Language and its Implementation, Addison-Wesley (1983).

14) Hayes-Roth, F., Waterman, D. A. and Lenat, D. B.: Building Expert Systems, Addison-Wesley (1983).
(訳書) AIUEO 訳：エキスパート・システム, 産業図書 (1985).

15) 米澤明憲, 柴山悦哉, J.-P. Broit, 本田康晃, 高田敏弘：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, コンピュータソフトウェア, Vol. 3, No. 3, pp. 9-23, 日本ソフトウェア科学会編集, 岩波書店 (1986).

16) 横手靖彦, 所真理雄：並行オブジェクト指向言語 Concurrent Smalltalk, コンピュータソフトウェア, Vol. 2, No. 4, pp. 2-18, 日本ソフトウェア科学会編集, 岩波書店 (1985).

(昭和61年6月20日受付)