並列オブジェクト指向言語 COOL

丸山勝己†

電話交換機のプログラムを始めとする実時間多重処理プログラムは，高効率が要求される大規模・複雑・長寿命かつ頻繁に機能追加が行われるシステムである．また，Intelligent Network やパーソナル移動通信などに見られるように通信網ワイドの分散処理など新しい機能が要求されてきており，プログラムの拡張性・保守性の向上と分散処理能力が求められている．このためには，多重処理・分散処理の効率的な実現が可能なオブジェクト指向モデルと，大規模システム記述に適したモジュール化とコンパイル時エラーチェック能力を持つオブジェクト指向言語が望まれる．本論文では，実時間多重処理システムには，能動オブジェクトと受動オブジェクトからなる並列オブジェクトモデルが適することと，このモデルを簡潔に実現した実時間オブジェクト指向言語 COOL を述べる．COOL は強タイプ付のシンプルな言語であり，能動オブジェクト・受動オブジェクトともデータタイプとして定義される．これは実行効率，読解性，コンパイル時チェック強化に資する．能動・受動の両オブジェクトの統一的設計により容易に適切な活用が可能である．仕様と本体に分けたモジュール機構は大規模システムの構築に効果が高い．グローバルオブジェクト ID により，分散を意識することなく分散処理を記述できる．

Concurrent Object-Oriented Language COOL

KATSUMI MARUYAMA†

Realtime multiprocessing systems, such as switching system programs, are large, complex, long-lived and frequently modified systems. Some realtime systems, such as Intelligent Networks and personal mobile telephone systems, require communication-network-wide distributed processing capabilities. Therefore, they are required efficiency, reliability, program maintainability and extendability, and distributed processing capabilities. To meet these requests, the object-oriented model with efficient realtime multiprocessing capabilities, and an object-oriented language based on the model are desired. This paper explains the concurrent object model for real-time multiprocessing programs and the concurrent object-oriented language COOL. The concurrent object model consists of communicating active objects and passive objects. COOL is a strongly typed simple language. COOL has multiprocessing capabilities and efficiency for realtime applications. Both active objects and passive objects are defined as data type. Strong typing helps run-time efficiency, program-readability and compile-time checking. Active objects and passive objects are designed with good harmonization, thus easy and efficient programming is attained. Module constructs help large system implementations. Active object ID is a global ID and allows a distributed processings.

1. はじめに

実時間多重処理システム，例えば電話交換機の制御プログラムは同時に数千の呼を扱う大規模（数メガ行），複雑，長寿命かつ頻繁に機能追加が行われるシステムである．また，Intelligent Network やパーソナル移動通信などに見られるように通信網ワイドの分散処理など新しい機能が要求がされてきている．従って，プログラムの拡張性・保守性の向上と分散処理能力が求められており，このためにオブジェクト指向プログラミングが効果的である[1)~3)]．

† NTT 交換システム研究所
NTT Communication Switching Laboratories

このような実時間多重処理にオブジェクト指向を適用するには，以下の能力を同時に持つオブジェクト指向言語が望まれる．

(a)数千のオーダの並列処理の記述能力
(b)実時間性能を満足できる効率
(c)大規模システムに適したモジュール化機能
(d)コンパイル時エラーチェックの強化
(e)分散処理の能力

既存オブジェクト指向言語の Smalltalk[8)], C++[11)], Eiffel 等は並列処理機能を提供していない．ABCL/1[9)], Concurrent-Smalltalk[10)]は先駆的な並列処理機能を提供しているが，本稿の分野には効率が十分ではない．

本論文では，並列オブジェクトモデルに基づいた実時間システム記述用のオブジェクト指向言語 COOL について述べる．

2. 並列オブジェクトモデル

Smalltalk, C++ 等のオブジェクトはインスタンス変数とメソッドのカプセルであり，並列処理能力は持たない．並列処理型のオブジェクト指向モデルは，各オブジェクトに並行プロセス機能である“**スレッド**”（本稿では動的静的に軽量な並行プロセスのことをスレッドと呼ぶ）を持たせることで実現できる．このオブジェクトは能動的・並行的に動作できる．これを**能動オブジェクト**と呼ぶ．

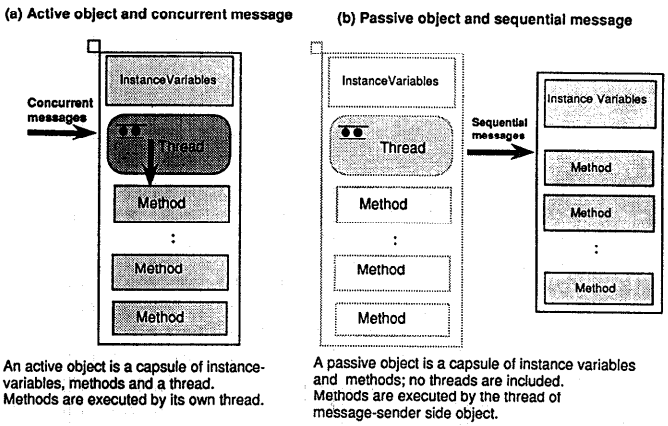


図 1 能動オブジェクトと受動オブジェクト
（a） 能動オブジェクトと並列メッセージ
（b） 受動オブジェクトと直列メッセージ
Fig. 1 Active objects and passive objects.

モデル的には能動オブジェクトだけでシステムを構築できる．しかし，能動オブジェクトは動的にはコンテクスト切替え，静的にはスタック域が必要であり，並列処理が不要なものに対しては動的・静的オーバヘッドとなる．例えば交換プログラムは膨大な数（10^4 オーダ）のリソース（加入者線，中継線，通話パスなど）を有すが，これらすべてを能動オブジェクトとして実現するのは動的効率およびメモリ容量から無理である．またすべてが並列処理能力を必要とするわけではない．従って，スレッドを持たずに受動的に動作するオブジェクトを導入し，これを**受動オブジェクト**と呼ぶ．

つまり実時間超多重処理システムに対しては，受動オブジェクトと能動オブジェクトからなる以下の並列オブジェクトモデル（**図1**）が適切である☆．

（1） 能動オブジェクトと並列メッセージ

能動オブジェクトは，インスタンス変数，メソッドおよび単一のスレッドからなるカプセルである．能動オブジェクトは並列処理単位であり，そのメソッドは自己のスレッドによって実行される．

能動オブジェクトへのメッセージ通信法としては，メッセージが宛先に受理されるまで送信側が待つか否かにより同期型と非同期型とに分けられる．交換プログラムでは，オブジェクト α が β にメッセージを送って，その受理確認が返る前に α は別の入力メッセージを受けなければならない場合が多い．これを同期型通信で実現するにはバッファプロセスの介在等が必要となり構造が複雑化するので，**非同期型通信**を採用する必要がある．非同期型通信では送信側はメッセージ送信後も中断せずに受信側と並列的に動作できるので，これを**並列メッセージ**と呼んでいる．並列メッセージ送信は，内容を書き込んだメッセージバッファを宛先の能動オブジェクトのメッセージ行列に挿入して，自己の処理を続行する．

能動オブジェクトに対するメッセージでも，直ちに返答メッセージを要求する場合もある．この場合，要求メッセージを送ったオブジェクトは“返答待ち状態”に入って返答メッセージが返るまで中断し，その間に到着した返答以外のメッセージは待ち行列に入る．つまり，能動オブジェクトは**メッセージ待ち**，**返答待ち**，**実行待ち**および**実行中**の状態を持つ．

（2） 受動オブジェクトと直列メッセージ

受動オブジェクトは，インスタンス変数とメソッドからなるカプセルである．受動オブジェクトは自己のスレッドを持たないので自らメソッドを実行することができず，メッセージ送信側の能動オブジェクトのスレッドによって間接プロシージャコール形式で実行される．この場合，メソッドを受信した受動オブジェクトは送信側オブジェクトと直列的に実行されるので，このメッセージを**直列メッセージ**と呼ぶ．

本モデルは，能動オブジェクトと受動オブジェクト

☆ Concurrent/Sequential の訳は並行／逐次が一般的であるが，本論文では用語の対称性のために並列／直列という用語を用いている．

の統一的な導入が特徴であり，効率要求の厳しい実時間超多重処理に適している（他言語のモデルとの対応は 11 章を参照）．

3. 言語の基本的な設計方針

本言語の対象は，実行効率と保守性の要求が厳しい大規模実時間処理システムであるので，以下の設計方針を採った．

（1） 並列オブジェクトモデルの実現

並列処理機能を持つ能動オブジェクトの実現法としては，以下が考えられる．

(a)言語が受動オブジェクトのみを提供し，プログラマが OS のスレッド機構を利用して能動オブジェクトを『受動オブジェクト＋スレッド』として作らせる手法（例えば C＋＋ プログラム内で OS の軽量プロセスを用いて能動オブジェクトを実現する方法）．

(b)言語側で能動オブジェクトを提供する手法．

(a)では，スレッド制御を一々記述する必要があり，スレッドを含めて能動オブジェクトとしてカプセル化した意義が無いので(b)を採用する．

また，モデル的には一つの能動オブジェクトが複数のスレッドを含むことも可能である．利点は，同一能動オブジェクト内での多重処理も可能になり，密結合マルチプロセッサでのパラレル計算や多重サーバプログラムの実現が容易なことである．弱点は，モデルが複雑になり，同一オブジェクト内の同期問題が必要となることである．どちらを選択するかは適用問題によるが，実時間多重処理の分野では単一スレッドモデルが簡潔で必要な能力を持っていると言える．

（2） 分散処理記述能力の実現

並列メッセージは機械語レベルではプロセス間通信であり，オブジェクト ID が指定する宛先オブジェクトに配送される．従って，オブジェクト ID にノード情報を含め，ノード間のメッセージ配送を行うことにより，ノード間に跨がった分散処理も可能になる．メッセージ転送の記述はメッセージの宛先がローカルでもリモートでも同一にできる．

（3） 融通性と効率のバランス

Smalltalk のようにメソッドサーチを実行時に行えば融通性は増すが，実行効率が低下する．実行効率と融通性のバランスから，後述（9 章）のように個々のオブジェクトにメソッドテーブルへのリンクを持たせた間接リンク方式を採用する．

（4） 継承の実現

COOL でもマルチプル継承を実現することは可能であるが，大規模システムではマルチプル継承を許すと却って複雑になり保守を難しくする恐れがある．そこで，COOL ではシンプルな単一継承のみ実現する．

（5） コンパイル時エラーチェックの強化

このために強タイプ付言語とする．これによりプログラムの融通性は低下するが，プログラムの解読性とプログラムの実行効率も向上する．システムプログラムではこれらの利点の方が重要である．

（6） 手続き指向とオブジェクト指向の両立

既存システムとの共存性などから，手続き指向も必要である．クラス定義をデータタイプとして扱うことにより，両者は融合できる．従って，クラス定義以外のデータタイプ（整数，ブール，文字，数え上げ，サブレンジ，ポインタ，可変長配列ポインタ，プロシージャ，配列，CASE フィールドを含む構造体等），変数宣言，定数定義，タイプ定義，プロシージャ定義，実行文は Pascal[4]，CHILL[5]をベースに設計して，手続き指向とオブジェクト指向を共存したプログラムを可能とする．

（7） 大規模システムに適したモジュール化機構の実現

プログラムモジュール間インタフェースの管理と分割コンパイルの容易化のために，モジュールを仕様記述部と内部実現部の別ファイルとして記述可能なモジュール機構を持たせる．

（8） 解読性重視

長期に渡って保守されるプログラムのために，プログラムの読みやすさを特に重視する．

4. 受動オブジェクト

受動オブジェクトは，インスタンス変数とメソッドのカプセルである．**図 2** に中心座標と半径で規定される円のオブジェクトの定義例を示す．

オブジェクトの定義は，オブジェクトのインタフェースを記述した**クラス定義**とオブジェクトの内部論理を記述した**メソッド定義**とからなる．

クラス定義には，処理を依頼するのに必要なメッセージ仕様とオブジェクトの生成・クラス継承に必要なインスタンス変数定義とが含まれているので，オブジェクトを生成したりメッセージを送るのに必要な情報はクラス定義を見るだけで得られる．このため，後述のようにクラス定義をモジュール仕様部に，メソッ

```
TYPE circle =
    CLASS
      init(x,y,r:int);
      move(x,y:int)=>(newx,newy:int);
      enlarge(n:int)=> int;
    VAR
      x,y,r: int; /* instance variables */
    END;

METHOD circle<<init(x,y,r:int);
  MY.x := x;  MY.y := y;
  MY.r := r;
END init;

  /* Output-parameter-style method */
METHOD circle<<move(x,y:int)=>(newx,newy:int);
  MY.x += x;  newx := MY.x;
  MY.y += y;  newy := MY.y;
END move;

  /* Function-style method */
METHOD circle<<enlarge(n:int)=>int;
  MY.r *= n;
  RETURN  MY.r;
END enlarge;

VAR  c : circle;       /* Object variable */
VAR  pc: REF circle;   /* Pointer to object */
VAR  i,j,k,r,s,t: int;

CREATE circle TO c << init(0,0,10);
CREATE circle TO pc << init (7,5,3);

(s,t) := c << move(i+j,k); /*Output-parameter-style*/
 r    := c << enlarge(5);  /*Function-style*/
(s,t) := pc << move(i,j);
 r    := pc << enlarge(2);

"var += exp;" is equivalent to "var := var+(exp);"
```

図 2 受動オブジェクト
Fig. 2 Passive object.

ド定義をモジュール本体部に分けて記述することができ，大規模システムの構築も容易となる．

メソッド定義は，自己の属するクラス名，メッセージ名，入出力パラメータを記述した METHOD 文の後に文の並び，END 文を書くことで定義される．メソッドは，一つの値を返す“**関数形式**”（図 2 enlarge）と，複数の値を返せる“**出力パラメータ形式**”（同 move）も可能である．前者はC言語同様に RETURN 式の値が返り，後者はメソッドの実行が終わった時に出力パラメータの値が実パラメータ変数に代入される．

クラスタイプの変数が**オブジェクト変数**であり，そのメモリ域は普通の変数と同様に VAR（変数宣言）文により静的/自動に，あるいは CREATE 文により動的に割り付けることができる．

CREATE クラスタイプ名 TO 変数〔⟨⟨メッセージ〕;
CREATE 文は，（a）“変数”がオブジェクト変数の場合はオブジェクトからメソッドテーブルへのリンク（9章参照）を設定し，（b）“変数”がオブジェクトへのポインタ（REF）変数の場合はオブジェクトのメモリ域を動的に割り付けて，オブジェクトからメソッドテーブルへリンクを設定する．メッセージが記述してあれば，これがオブジェクトに送られる．初期設定メッセージとして便利に使える．

メッセージ交信は，次のように記述する（出力パラメータ形式の場合）．強タイプ付なので，コンパイル時にメッセージインタフェースのチェックが行われる．

（変数の並び）:=オブジェクト指定
⟨⟨メッセージ名(式の並び);

ここに，オブジェクト指定は宛先オブジェクト変数または宛先オブジェクトへのポインタ変数である．ポインタ変数の場合は，自動的にアドレスをたぐって宛先オブジェクトが決定される．代入記号“:=”の左辺は，返答値を代入する変数の並びである．このように出力パラメータ形式の場合は変数の並びを括弧でくくる．インスタンス変数は“MY.”を前置してアクセスする（理由は，プログラム読解性向上と，インスタンス変数とパラメータに同一名を許すためである）．

5. 能動オブジェクト

能動オブジェクトは，インスタンス変数，メソッドおよび単一のスレッドのカプセルである．

能動オブジェクトのクラス定義は，**図 3** に示すようにキイワード“ACTIVE”が付く以外は受動オブジェクトの場合と同様である．メソッド定義もクラス名の後に“⟨⟨”の代りに“⟨:”と書く以外は受動オブジェクトの場合と同様である（この区別はコンパイラのた

```
TYPE worker =
  CLASS ACTIVE
    init(name:string; age:int);
    doWork(what:job_t)=>(hour,pay:int);
    takeBreak(t:int);
    goHome(....);
    ......
    .....
  VAR
     ..../* Instance variables */
     ......
  END ;

METHOD worker<:init(name:string; age:int);
     .....
     .....
END init;

METHOD worker<:doWork(what:job_t)=>(hour,pay:int);
     ....
     ....
END doWork;

METHOD worker<:takeBreak(t:int);
    ....
    ....
END takeBreak;

VAR  Taro :ACTIVE REF worker; /* Global object-ID */
VAR  t,x :int;

CREATE worker TO Taro;  /* Object creation */

Taro <: init("Taro Urashima   ", 999);
(t,x) := Taro <: doWork(Fishing);
Taro <: takeBreak(10);
```

図 3 能動オブジェクト
Fig. 3 Active object.

めではなく，プログラマに受動・能動の違いを意識させるためである)．

能動オブジェクトは自己のメッセージ行列を持っており，送られてきた並列メッセージは本行列に挿入される．能動オブジェクトは，スケジューラから制御を受けるとメッセージ行列からメッセージを取り出し，メッセージ名に対応したメソッドを実行する．

能動オブジェクトも受動オブジェクトと同様にCREATE 文により生成される．違いはタイプが能動クラスタイプになっているだけである．並列メッセージ通信も以下のように転送記号が“<:”になるだけである．

(a) オブジェクト指定〈メッセージ名(式の並び)；

(b) (変数の並び):=オブジェクト指定
〈: メッセージ名(式の並び)；

機械語レベルでは，形式(a)では，実パラメータ値を乗せたメッセージバッファを宛先オブジェクトのメッセージ行列に挿入して自己の処理を続行する．形式(b)では，メッセージを送ったオブジェクトは返答待ち状態になり，返答メッセージが届くとこの変数に返答値が代入される．つまり要求メッセージ送信と返答メッセージ受信のペアーに展開される．

また，例外処理等に備えて優先メッセージ機能もある．各能動オブジェクトは，通常メッセージ行列の他に優先メッセージ行列を持っている．優先メッセージは割り込み実行されるのではなく，優先メッセージ行列に挿入される．能動オブジェクトがスケジューラから制御を受けた時，ならびにプログラマが明記した点で優先メッセージ行列が調べられ，優先メッセージが到着していたら通常メッセージよりも先に実行する．この方式は実現が容易でかつ実用的である．

なお，ABCL/1 同様にメソッドの内部で選択的メッセージ受信をすることも可能であり，利用価値が高い．

6. 継承と実行時拡張

(1) 継　　承

クラス定義に“BASE (クラスタイプ)”を指定することにより，指定タイプの内容が継承される (**図4**)．継承した側を**拡張**[5]，元の側を**ベース**と呼ぶ．受動・能動オブジェクトともにシングル継承が可能である．ここで，図4のクラス定義 person の“AREA (256)”は，オブジェクト生成時にインスタンス変数域として予備域込みで256バイトを割り付けることを意味し[7]，

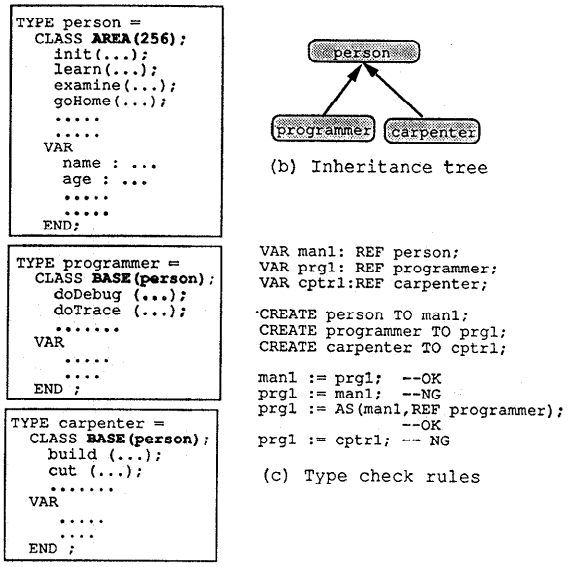


図4 継　　承
(a) クラスの定義
(b) 継 承 木
(c) タイプチェックルール
Fig. 4 Inheritance.

次項の動的拡張を利用する時にのみ必要である．

“REF”タイプの代入では，拡張クラス側からベースクラス側への代入は常に安全なので許すが，その逆は許さない．後者が必要な場合には明示タイプ指定を使う．AS (**式，タイプ**) は式の値を指定タイプに明示変換 (C の casting に相当) することを意味する．9章で述べるように，各オブジェクトはメソッドテーブルへのリンクを持ち，メソッドテーブルは拡張側からベース側へのリンクを持っているので，これを用いて実行時タイプチェックすることも可能であるが，メッセージ転送ごとに動的チェックが入りオーバヘッドとなるので，COOL ではプログラマの責任のもとに明示タイプ変換をさせることにした．

(2) 動 的 拡 張

プログラムを書いてみると，実行中にオブジェクトをその拡張クラスに拡張したい場合がある．例えば，交換機の呼処理プログラムの場合，個々の発呼に対してまずダイアル番号受信の処理を行い，全番号を受信した後でその番号を解析して，実行するサービスを決定するので，この時点で目的サービス機能を持つようにオブジェクトが拡張できると便利である．

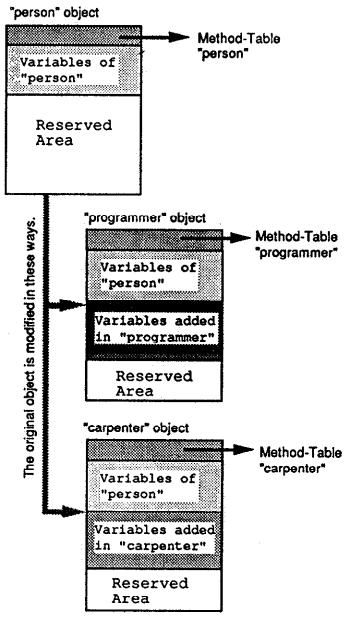
いま，図4で定義されたクラス person とその拡張

```
VAR man1 :REF person;
VAR prg1 :REF programer;
VAR cpt1 :REF carpenter;

CREATE person TO man1;
 /* create a person object */
   ........
man1 << learn(...);
    ........
    ........
IF  .... THEN
  prg1:= EXTEND(man1,programer);
    /* The object pointed by "man1" is
       extended to a programmer.
       "prg1" becomes the object-pointer. */
   ............
  prg1 << doDebug( );
   ............
ELSIF  ....  THEN
  cpt1:= EXTEND(man1,carpenter);
    /* The object pointed by "man1" is
       extended to a carpenter.
       "cpt1" becomes the object-pointer. */
    ........
  cpt1 << build( );
   ........
ELSE
   .........
   .........
FI;
```

[In this program, pointers man1, prg1 and cpt1 point the same memory address.]

(a) Program example



(b) Internal mechanism.

図 5 動的機能拡張
(a) プログラムの例
(b) 内部構造
Fig. 5 Dynamic extension.

クラス programmer, carpenter において，person として生成されたオブジェクト man1 を実行中に条件に応じて programmer あるいは carpenter に拡張したいとする．これを COOL では**図 5** のように実現できる．

EXTEND（**オブジェクト，拡張クラスタイプ**）は動的拡張を行う組み込み関数であり，次のように機能する：①元のオブジェクトの予備域（AREA 属性で確保された領域）を拡張タイプで追加されたインスタンス変数域として使う．②オブジェクトのメソッドテーブルリンクを拡張タイプのメソッドテーブルに切り換える（9章のオブジェクトの内部構造を参照）．③処理が正常ならば“**REF 拡張クラスタイプ**”の値（アドレスは元のオブジェクトのままでタイプのみが拡張されている）を，さもなければ NULL を返す．

7. 大規模システム向きモジュール機構

COOL プログラムはモジュールの集まりである．モジュールはコンパイル単位であり，仕様部（SPEC）と本体部（BODY）に分けて記述する（**図 6**（a）参照）．仕様部には他モジュールから参照可能な各種定義と Seize 文（本モジュールが参照する他モジュール名を指定）を，本体部にはメソッドやプロシージャの内部論理と本モジュール内でのみ参照される定義を記述する．

モジュールが他モジュールに見せる定義や他のどのモジュールを参照するかはすべて仕様部に記述されているので，モジュール間の参照関係は仕様モジュールのみで完全に把握でき，仕様モジュールをプログラム設計時のインタフェース仕様書としても活用できる．

プログラム中で，他の仕様モジュール“m”で定義された名前“x”を参照するには，“m!x”と記述する．この記述のため，どのモジュールのどの名前かが直ぐに分かる．コンパイラがリンカに渡す外部名は“_m_x”となる．

ただし，モジュール文に“BARE”属性を付けておくと，そこで定義された名前“y”はモジュール名を前置せずに裸のままの名前“y”で参照できる．この場合は，コンパイラがリンカに渡す外部名は“_y”となる．この仕組みは，UNIX システムコールのインタフェース定義やモジュールと一体化したクラス定義に効果的に使われる．例えば，UNIX のシステムコールは図 6（b）のように仕様モジュールを定義しておくだけで，普通にプロシージャコールできる（特殊記号“…”は可変パラメータリストを意味し，この場合のみコンパイラはパラメータチェックをしない）．なお，COOL のパラメータ引継ぎは値引継ぎであるが，C言語では配列（文字列を含む）はアドレス引き継ぎなので，COOL の中では配列に対しては明示的にアドレスを渡す必要がある．本例の“&”は“REF char”の値を返す単項演算子である．COOL では定数に“&”を付けることも可能である．

大きいクラスの定義の場合は一つのモジュールに一つのクラス定義を書くことが多く，またクラス定義自体が内部隠蔽の機能を持っていることから，**図 7**（a）のようにモジュールと一体化したクラス定義が可能である．この形式を使えば，Smalltalk と同様にクラス定義をモジュールとして記述することも可能である．これは，内容的には図 7（b）の“BARE”属性モジュールと等価である．

```
MODULE SPEC mm;
 PROC abc(x,y:int)=>(r,s:int);

 TYPE aa = CLASS
          msg1 (x,y:int);
          msg2 (i,j:int);
           ...
          VAR
            ....
         END ;
END mm;
```

```
MODULE BODY mm;
 PROC abc(x,y:int)=>(r,s:int);
    ........
 END abc;
 METHOD aa<<msg1(x,y:int);
   ..........
 END msg1;
 METHOD aa<<msg2(i:int);
   ..........
 END msg2;
   .......
END mm;
```

```
MODULE SPEC nn;
SEIZE mm;
 TYPE bb = CLASS
           BASE(mm!aa);
           msg3(x:int);
         VAR
           .....
        END ;
END nn;
```

```
MODULE BODY nn;
 METHOD bb<<msg3(x:int);
    .........
   (i,j):= mm!abc(a,b);
...
    ........
 END msg3;
  .....
END nn;
```

(a) Module construct

```
MODULE SPEC unix :BARE;
  PROC printf(p:REF char;...)=>int;
  PROC scanf(p:REF char; ...)=>int;
  PROC strcpy(to,from:REF char)=>int;
  PROC malloc(size:int)=>REF char;
  .........
  .........
END unix;

/* This "SPEC" module defines
   unix system call interfaces */
```

```
MODULE SPEC qq;
 SEIZE unix ;
  TYPE cc = CLASS
              msg5(x:int);
              msg6( );
            VAR
              .........
            END;
END qq;
```

```
MODULE BODY qq;
  METHOD cc<<msg5(x:int);
     .........
    printf (& "%s %d\n", &name, x);
      ......
      ......
  END msg5;
  .....
  .....
END qq;
```

(b) Bare module and Unix system calls

図 6 モジュール機構
(a) モジュール機構
(b) Bare 属性モジュールと UNIX のシステムコール
Fig. 6 Module construct.

```
CLASS SPEC bb;
  SEIZE aa;

  BASE (aa);
  gamma (x,y:ss);
  delta (i:int);
VAR
  a,b:int;
  c : bool;
END ;

--------------

CLASS BODY bb;
  TYPE t = ARRAY[0..7]OF int;
  VAR m,n :int;

  PROC f(x,y:int)=>(r,s:int);
    .....
  END f;

  METHOD <<gamma(x,y:ss);
    .....
  END gamma;

  METHOD <<delta(i:int);
    ....
  END delta;
END bb;
```

```
MODULE SPEC bb :BARE;
  SEIZE aa;
  TYPE bb = CLASS
      BASE (aa);
      gamma (x,y:ss);
      delta (i:int);
   VAR
      a,b:int;
      c : bool;
    END ;
END bb;

--------------

MODULE BODY bb;
  TYPE t = ARRAY[0..7]OF int;
  VAR m,n :int;

  PROC f(x,y:int)=>(r,s:int);
    .....
  END f;

  METHOD bb<<gamma(x,y:ss);
    .....
  END gamma;

  METHOD bb<<delta(i:int);
    ....
  END delta;
END bb;
```

(a) Class module (b) Equivalent BARE module

(Bold parts show differences.)

図 7 クラスモジュール
(a) クラスモジュール
(b) 等価な BARE 属性モジュール
(太字部分は違いを示す.)
Fig. 7 Class module.

8. 分散処理

直列メッセージは機械語レベルでは間接プロシージャコールに展開されるので，離れたプロセッサには送れないが，並列メッセージはメッセージバッファを使った非同期メッセージ通信なので，離れたプロセッサ上の能動オブジェクトに送ることができる．つまり，能動オブジェクトと，それに従属して活動する受動オブジェクトのグループが分散配置の単位となる．

能動オブジェクトへのポインタである“ACTIVE REF”タイプには，ノード ID が含まれている．これを**グローバルオブジェクト ID** と呼んでいる．分散処理カーネルは，この情報をみてメッセージを目的ノードの宛先オブジェクトに配送する[12]．メッセージ転送のプログラム記述は，宛先オブジェクトがローカルでもリモートでも同一である．分散 OS が持つネームサーバ等に問い合わせてグローバルオブジェクト ID を得て，それをメッセージの宛先に指定すれば，自動的に目的ノードの目的オブジェクトにメッセージが運ばれる．

9. 内部構造

能動オブジェクトの内部構造例を**図 8** に示す．クラス定義対応に**メソッドテーブル**がある．本テーブルは各メソッドへのリンクと，ベースタイプのメソッドテーブルへのリンク (Ptr_to_base) を持っている．個々の能動オブジェクトは，スレッド制御ブロック，スタック＋インスタンス変数域からできている．インスタンス変数域からは各メソッドのアドレスを記したメソッドテーブルにリンクがある．

並列メッセージはメッセージバッファに乗せて宛先オブジェクトに届けられ，通常のメッセージは最後部 (FIFO 順) に，返答メッセージは先頭に挿入される．メッセージバッファのヘッダ部には，宛先と送

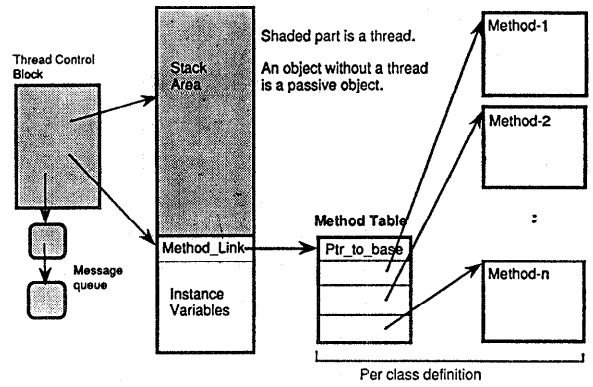


図 8 能動オブジェクトの内部構造
Fig. 8 Internal structure of active objects.

り元のオブジェクト ID，メッセージ番号等が書かれている．メッセージ番号は，各クラス定義対応にメッセージ名の並び順に 1, 2, 3, … という番号が与えられている．能動オブジェクトがスケジューラから制御を受けると，受信したメッセージを取り出し，そこに書かれているメッセージ番号でメソッドテーブルをインデックスして目的メソッドを選んで，それを実行する．

能動オブジェクトからスレッド機構（図 8 の網懸け部）を外すと受動オブジェクトになる．受動オブジェクトのメソッドは，メッセージ送信側のスレッドにより実行される．

このように実行メソッドはメソッドテーブル経由で決定されるので，Smalltalk のような実行時メソッドサーチのオーバヘッドが無く，かつメッセージの多様性（Polymorphism）も実現されている（Cf. C++ の virtual function も同様な機構で多様性を実現している）．また，継承も**図 9** に示すようにメソッドテーブルのリンクにより実現している．継承関係を示すメソッド間リンク（図 9 の Ptr_to_base）は，動的拡張時のチェックに使っている．なお，実行効率よりもプログラムの融通性を重視する分野では，これらの機構を用いて実行時タイプチェックを行うことも可能である．

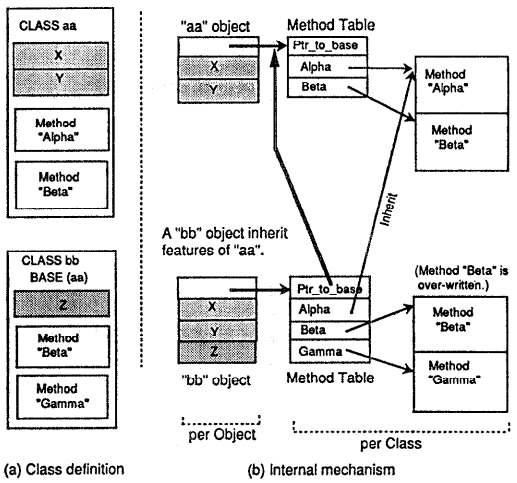


図 9 継承の内部構造
（a） クラスの定義
（b） 内部構造
Fig. 9 Internal mechanism of inheritance.

メソッドを起動する際には，メッセージ受信オブジェクトのインスタンス変数域のアドレスが第 0 番の隠れパラメータとして引き継がれ，続けて明示パラメータが引き継がれる．なお，現用コンパイラでの明示パラメータ引継ぎは，（a）受動オブジェクトの場合には，実パラメータの値をパラメータスタック域に push してメソッド内部に引き継ぐ；（b）能動オブジェクトの場合にはメッセージバッファのアドレスをパラメータスタック域に push して，メソッド内部からはメッセージバッファ上に間接アクセスする手法を採っている．これは，並列メッセージでは大きな構造体を引き継ぐ場合が多いので，パラメータコピーを減らして効率化を図ったためである．

10. コンパイラ

COOL は Pascal 並にコンパイラ作成の容易化も考慮した設計とした．Free Software Inc. の GNU-C コンパイラ（GCC）[14] のフロントエンドを書き換えて COOL コンパイラを開発したが，書換え規模は語彙分析＋構文解析が lex＋yacc＋C で約 1000 行，ネームテーブル作成関連の変更が C で約 4000 行である．COOL は C よりも高い機能と読解性を持つが，構文の簡素化とタイプ解析の容易化により，COOL 用フロントエンドの方が C 用フロントエンドよりも小さい．本コンパイラは GCC のシンボリックデバッガインタフェース，最適化モード，各種マシンへの移植性等の機能を継承している．

SUN 用コンパイラでは，SUN-OS の軽量プロセスを使って能動オブジェクトを実現しているが，研究試作中の通信網用分散処理プラトフォーム “PLATINA” (Platform for telecommunication and information applications)[12),13)] 上では更に効率よく実現される．

11. 他言語との関係

Smalltalk-80, Eiffel, C++ 等は，本稿で述べた能動オブジェクトの機能は持たない．

ABCL/1のオブジェクトはすべて能動型であり受動型は含まない．ABCL/1オブジェクトは，非同期メッセージを交信しながら並列に動作する点で COOL の能動オブジェクトと類似している．ABCL/1 のメッセージの過去型は COOL の並列メッセージに，現在型は返答要求並列メッセージに対応する．ABCL/1 のオブジェクトで“受信メッセージがすべて NOW 型；各メソッドが Reply 実行と同時に終了；複数メッセージの同時到着が無い”場合は，本稿の受動オブジェクトの機能で十分であり，かつ効率化が図れる．

Concurrent-Smalltalk は NonAtomicObject と到着メッセージが FIFO 順にシリアライズされて実行される AtomicObject を持ち，Smalltalk 上位コンパチブルである．非同期メソッドコールにより並列処理ができるが，実時間処理を狙ったものではない．

COOL は，能動オブジェクトと受動オブジェクトの統一的な導入，強タイプ付を活用したコンパイル時処理等により，実時間超多重処理システムに要求される記述能力と効率をシンプルに実現した所に特徴がある．本稿では適用例として通信ソフトウェアを取り上げたが，本手法は各種の並列処理分野に適用可能である．

12. おわりに

能動・受動の両オブジェクトの導入により，交換プログラムのような実時間超多重処理に適用可能な機能と性能が得られた．直列メッセージ通信は間接プロシージャコールに展開されるので，受動オブジェクトのオーバヘッドは十分に小さい．能動オブジェクトの主なオーバヘッドは並列メッセージ転送とスタック域であるが，処理モデルが簡単なので効率よく実現できる．研究中の通信網用の分散処理プラットフォーム PLATINA 上に実現した場合の並列メッセージ通信は約200機械語ステップと軽い．

効率を無視すれば能動オブジェクトのみからなるモデルの方がよりエレガントであるが，能動オブジェクトと受動オブジェクトの対応はよくとれており記述もほとんど同じなので，容易に使い分けが可能であり，実時間システムには本モデルが適切である．能動・受動オブジェクト間の対応は，並列メッセージを同期型とすると更に対応は良くなるが，交換プログラムのような多重処理分野では同期型は使いにくく，非同期型が適切である．

強タイプ付言語なので，実行効率，読解性，エラーチェック能力が強化されている．分散プログラムのインタフェース条件も仕様モジュールで記述されるので，強タイプ付で実現できる．

COOLではスレッド生成・消滅，メッセージ送信・受信，返答送信・受信，メッセージバッファ割付け・解放などのプリミティブルーチンも組み込み関数と同様にコールできるので，メッセージバッファを直接扱った小回りの効く記述などもできる．

COOL は PLATINA の上で動く応用プログラムの記述用言語として設計開発した言語であり，分散処理記述に適している．グローバルオブジェクト ID により，能動オブジェクト間は自在に分散処理ができる．

オブジェクトの内部構造を隠蔽するために，C++ の public に相当する機能は持たない．もしメソッド呼出しのオーバヘッド削減が必須になれば，クラスタイプ内にインラインメソッド定義を組み込むことを考えている．

同一メッセージインタフェースを持つ別クラスのオブジェクトに並列メッセージを転送する Delegation 機能は未実装であるが，クラス定義に従ってメッセージ番号を変換してメッセージバッファを転送する手法により容易に実現できる．

今後，試作中の PLATINA 上の応用プログラムの COOL 記述を通して，クラスライブラリーの蓄積や必要なフィードバックを行っていく．

謝辞 本研究の推進にあたり貴重な御意見やCOOLの使用経験のフィードバックを頂いた NTT 交換システム研究所の諸氏，ならびに論文の質向上のための適切な御指摘を頂いた査読者の方に深謝します．

参考文献

1) Maruyama, K. et al.: A Concurrent Object-Oriented Switching Program in Chill, *IEEE Communication Magazine*, Vol. 29, No. 1, pp. 60-68 (1991).
2) 丸山ほか：オブジェクト指向による交換プログラム構成法，信学論文誌，Vol. J 74-B-I, pp. 757-768 (1991).
3) 丸山ほか：既存並列処理言語によるオブジェクト指向プログラミング，情報処理学会論文誌，Vol. 31, No. 1, pp. 88-97 (1990).
4) Wirth, N.: The Programming Language Pascal, *Acta Informatica*, Vol. 1, pp. 35-63.

(1971).
5) Wirth, N.: The Programming Language Oberon, *Software Practice and Experience*, Vol. 18, No. 7, pp. 671-690 (1988).
6) CCITT: Recommendation Z. 200 CCITT High Level Language (CHILL), International Telecommunication Union (1989).
7) Foxall, G. et al.: PROTEL: A High Level Language for Telephony, *Proc. 3rd Int. Comput. Software Appl. Conf.* (1979).
8) Goldberg, A. et al.: *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley (1983).
9) 米沢ほか：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 (1986).
10) 横手ほか：並列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 (1985).
11) Stroustrup, B.: *The C++ Programming Language*, Addison-Wesley (1986).
12) Maruyama, K. et al.: Platform for Telecommunication and Information Network Applications: PLATINA, *Proc. of TINA Conference* (1992).
13) Kubota, M. et al.: Distributed Processing Platform for Switching Systems: PLATINA, *ISS '92, Proceedings*, pp. 415-419 (1992).
14) Free Software Foundation Inc.: GNU-C compiler のマニュアルとソースコード (1988).

(平成4年8月10日受付)
(平成5年2月12日採録)

丸山 勝己 (正会員)

1944 年生. 1968 年東京大学工学部電子工学科卒業. 1970 年同大学院修士課程修了. 同年日本電信電話公社入社. 現在 NTT 交換システム研究所勤務. 電子交換機の実時間増設方式, 高水準言語と最適化コンパイラ, オブジェクト指向プログラミング, 交換プログラム構造, 通信網用分散処理などの研究実用化に従事. 1985〜1988 年 CCITT 第X研究委員会副議長. 著書『交換用プログラミング言語 CHILL』(電気通信協会). 工学博士. 1990 年度本学会論文賞受賞. 電子情報通信学会会員.