プログラミング言語 21－4
（１９８９．６．１３）

ＯＺ：オブジェクト指向開放型分散システムアーキテクチャ

－　オブジェクト指向型分散プログラミング言語とその実装　－

塚本享治　　四反田秀樹
電子技術総合研究所　松下電器産業

田中伸明　近藤貴士　吉江信夫
松下電器産業　シャープ　住友電気工業

高度な分散型応用には高度なデータ構造が必要である。しかし、これまでの分散プログラミング言語の多くは、分散システム上のデータ表現やデータ交換に重きをおいていなかった。本論文では、分散システム上におけるデータ表現の単位を継承機能を有するオブジェクトとし、分散システムにおけるアプリケーションをオブジェクトがネットワーク状に連結されたデータ構造と考える。前半では、効率の良い分散処理のために、オブジェクト間でネットワークの一部を複写する処理モデルを提案し、それに適した言語を設計する。後半では、このモデルにしたがって実現した『オブジェクト指向開放型分散システムＯＺ』の実装上の工夫について述べる。

Design and Implementation of an Object-Oriented Distributed Programming Language for ＯＺ : Object-Oriented Open Distributed System

Michiharu TSUKAMOTO　Electrotechnical Laboratory
Hideki SHITANDA　Matsusita Electric Industrial Co, Ltd.
Nobuaki TANAKA　Matsusita Electric Industrial Co, Ltd.
Takashi KONDO　Sharp Corporation
Nobuo YOSHIE　Sumitomo Electric Industries, Ltd.

Advanced distributed applications require complex data structures. However, distributed programming languages have payed little attention to data structures nor interchange formats. In our model, every entity has a form of object, and application programs consist of objects which are connected with each other. First, we propose a new model for efficient processing: objects are copied from the caller to the callee keeping the relationships among objects. Secondly, we design a programming language based on this model. And, we spend many efforts to implementaion of OZ: the compiler, the interpreter, and the integration of the distributed interpreters.

1．まえがき

ネットワークの普及によって、計算機間の自由な通信が可能になり、メールやファイルの転送、ファイルや各種機器への遠隔アクセスなどが実用になっている。しかし、それらはベンダの提供する情報転送と資源共有のサービスに限られており、ユーザがそれらのサービスを高機能化したり、分散処理プログラムを新たに開発しようとすると、機種、ＯＳ、言語などが違っていたり、各計算機ごとに別々のプログラムを書かざるをえなかったり、分散システム用のプログラミング環境が整備されていなかったり、といった様々な困難に直面する。この問題を解決するには、機種、ＯＳ、言語の違いを意識することなく、必要な場合には１つのソースプログラムによって、プログラミングできるような分散システム用のプログラミング言語（分散プログラミング言語）とその開発環境が必要である。

分散プログラミング言語に最小限度必要とされる機能としては、計算機の数や機種に無関係にプログラミングできる機能、実行の流れ（スレッド）が複数に分岐したり合併する機能、分散されたプログラム間でデータを交換できる機能、などが上げられる。＊ＭＯＤ[1]、ＳＲ[2]、ＮＩＬ[3]などの初期の分散プログラミング言語の多くは、最初の２つの機能に焦点をあてたものであり、分散システム上で交換できるデータは、整数、実数、文字列、配列などの基本データ型またはそれらを要素とする簡単な複合データ型に限られる。

分散システムに限らず一般に、アプリケーションが高度になるほど、複雑なデータ構造が必要になることが多い。しかるに、分散システム上で交換できるデータ型が制限されると、プログラムの設計や記述、あるいはできたプログラムの配置法などが制約を受ける。ユーザが定義する任意のデータ型のデータが転送できるようにした最初の言語は、抽象データ型言語ｃｌｕを拡張した Argus[4]である。 Argus では、分散システム上で交換するデータ型の場合には、内部形式を転送形式に変換する符号化と転送形式を内部形式に戻す復号化のアルゴリズムを抽象データ型の手続きとしてユーザに記述させる方式を採用している[5]。一方、分散プログラミング言語ではないが、ＸＤＲ、Matchmaker[6]、ＡＳＮ．１[7]などのような、プログラム間の符号化・復号化を含むインタフェースだけを記述することを目的とした言語もあるが、ユーザがデータ型ごとに符号化と復号化の方法を記述しなければならないことにかわりはない。

交換するデータ型ごとに複雑な符号化と復号化の方法を記述しなければならないのでは、そのための記述が膨大になったり、簡単なデータ型のデータしか交換しないように分散プログラムを設計することを余儀なくされる。筆者らは、『アルゴリズム＋データ＝プログラム』の原点に立返って、分散システム上では、分散システムであることに関係なく自由にデータ構造を定義し操作できることがもっとも基本であると判断し、分散システム全体に配置された相互に関係を持ち通信しあうプログラム全体を複雑に連結された１つのデータ構造と見なすという立場をとった。この視点にたてば、分散システム上のすべてのデータ要素は何らかの方法で関連付けられたネットワークを形づくっており、分散処理とは、隣接するデータ要素間で送信側の部分ネットワークを受信側に複写・移動することにほかならない。送信側で定義・作成したデータ要素群を受信側が正しく解釈するには、データ要素にそれを解釈する手続きを付随させることが不可欠であり、データ要素としては抽象化機能を有するオブジェクトを採用するのが正しい選択といえる。筆者らは、以上のような考え方にもとづいて、『オブジェクト指向開放型分散システムＯＺ[8]』のプロトタイプを開発した。ＯＺでは、オブジェクトの形式をとる複雑なデータ構造の定義と操作の記述を助けるために簡単なオブジェクト指向型分散プログラミング言語を提供している。本稿では、ＯＺの分散処理モデル、そのプログラミング言語化、実装などについて述べる。なお、ここで述べる言語は、分散システム上で稼動する単一ユーザ環境のためのものであり、複数ユーザ環境において相互通信するための拡張については別の機会に述べる。プログラミング言語やシステムとして類似のものとしては、 Argus、Eden[9]、Emerald[10]などがあげられるが、それらとの比較は本文中で行なう。

2．分散処理のモデル[8]

分散システム上に配置される、基本的な数や文字、それらを組み合わせたレコードや配列、ユーザが定義するデータ、および命令コード列やプロセス、などすべてのデータをオブジェクトと呼ぶ基本単位で表現する。オブジェクトは、状態を記憶する内部変数の集合であるインスタンスと振る舞いを記述するタイプで構成される。内部変数はオブジェクトを指し、これによって関連あるオブジェクトは複雑な１つのネットワーク状のデータ構造を形成する。オブジェクトを引数として隣接するオブジェクトを呼び出すと、呼ばれたオブジェクト自身によって内部変数の参照や操作が行なわれ、必要ならばオブジェクトが値として戻される。分散システム上でネットワーク状に連結されたオブジェクトどうしが互いに呼びあうと次々とネットワークの形状が変わっていく。この過程が分散処理である。

2．1　グローバルオブジェクトとローカルオブジェクト

このモデルを具体化するに際しては、オブジェクトどうしは相互に連結されてネットワーク状になっているために、引数や値として指定したオブジェクトをどのようにして交換するかが問題となる。交換する範囲と符号化・復号化の方法をユーザに任せる[5]のでは、負担が大きくなりすぎる。引数や値に指定したオブジェクトへのポインタだけを渡し実体は移動させないのでは、受信側から送信側への呼び返しが頻出する。また、引数や値で直接指定したオブジェクトだけを移動させても[10]、大して効果は上がらない[11]。ＯＺではこの問題を解決するために、オブジェクトをグローバルオブジェクトとローカルオブジェクトの２種類に分ける方法を採用した。

グローバルオブジェクトは分散配置される単位であり、ドメイン内部またはドメイン外部にあるどのオブジェクトからでも参照できるものとする。一方、ローカルオブジェクトはグローバルオブジェクトに従属する構成単位であり、仲間のオブジェクト以外からは参照されないものとする。ここで、仲間のオブジェクト(colleague object)とは、グローバルオブジェクトおよびそれから直接またはローカルオブジェクトを介して間接的に参照できるローカルオブジェクトの集合のことであり、仲間のオブジェクトに属するグローバルオブジェクトは常に存在しただ１つに限られる（図１）。また、仲間のオブジェクトは同一ドメイン上に配置され、複数のドメインにまたがって配置されることはない。ここで、ドメイン(domain)とは、直接参照できるメモリ領域を意味する。

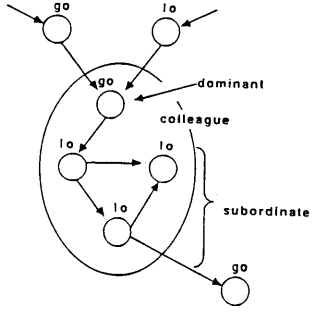


図１　グローバルオブジェクト

2．2　グローバルオブジェクトの参照

ローカルオブジェクトは仲間のオブジェクトからしか参照されないので、その参照には効率の良い直接的なポインタを使用することができる。しかし、グローバルオブジェクトは同一ドメイン内だけでなく他ドメイン内のオブジェクトからも参照される可能性があるので、ポインタは使用することができない。そこで、次のようなプロクシ(proxy)を導入し、グローバルオブジェクトへの参照はプロクシを介した間接参照とする（図2）。プロクシは次の3つのフィールドで構成される。

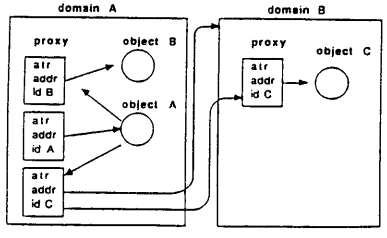


図2　グローバルオブジェクトの参照

(1) 属性フィールド：　プロクシの種類とプロクシが指す目的オブジェクトの属性などを記憶する。目的オブジェクトの属性を含めるのは、他ドメインに存在する可能性のある目的オブジェクトを見ないでもその属性が分かるようにするため。目的オブジェクトが属性を変えてもそれを参照する側に伝播させる必要がないようにするために、この属性はオブジェクト生成時に定まる静的属性に限る。

(2) アドレスフィールド：　属性フィールドの指定によって内容の種類が異なる。目的オブジェクトがドメイン内にあればそのアドレス、なければ目的オブジェクトに至る次のプロクシを保持するドメインのアドレス。

(3) 識別子フィールド：　オブジェクトを分散システム全体で一意に識別するための識別子ＵＩＤ(unique ID) を記憶する。

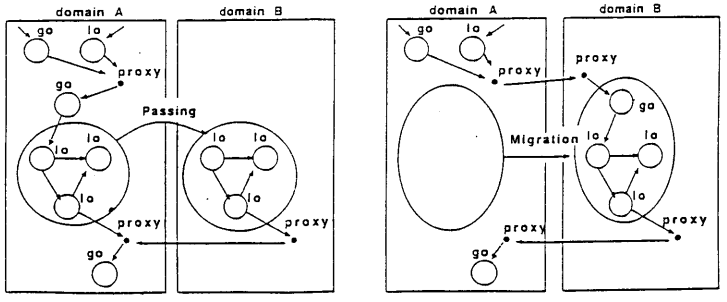
グローバルオブジェクトは同一ドメイン内のプロクシに登録され、その参照は同一ドメイン内であっても、プロクシを介した間接参照となる。グローバルオブジェクトへの参照を他のドメインに渡すときには、アドレスフィールドに目的オブジェクトを登録するプロクシのあるドメインのアドレスを入れ、属性フィールドをそれにあわせて修正したプロクシを渡す。これを受け取った側では、目的オブジェクトへの参照をこのプロクシへのポインタにより代替する。目的オブジェクトが他のドメインに移動するさいには、それまで目的オブジェクトを登録していたプロクシのアドレスフィールドを移動先ドメインのアドレスに変更し、属性フィールドもそれにあわせて修正する。１つのドメイン内では、同一ＵＩＤを持つプロクシを複数設けなければ、プロクシを介して参照する目的オブジェクトが同一であるか否かの検査は、プロクシへのポインタが同一か否かの検査で十分である。

2．3　オブジェクトの転送

分散システム上におけるオブジェクトどうしは、ポインタやプロクシによって関連づけられて複雑で巨大なネットワーク状のデータ構造を成しており、このデータ構造が複数のドメインに分散配置されていると考えることができる。ＯＺでは、分散処理をこのデータ構造の変化としてとらえ、レベルの違う２通りの要因によってデータ構造が変化するものと考えた。第１は、データ構造の要素となっているオブジェクト間のインタラクションに起因するものであり、これをオブジェクトパッシングと呼ぶ。第２は、オブジェクトのドメインへの配置を変更するというメタレベルの操作に起因するものであり、オブジェクトマイグレーションと呼ぶ。このように考える背景には、グローバルオブジェクトは生成後はマイグレーションによらねば所在が変化せず、通常はグローバルオブジェクトを代表とする仲間の間でローカルオブジェクトをコピーして渡すことを想定している。

オブジェクトパッシングは頻繁に発生するので、効率が良く、しかもオブジェクトの配置場所によって結果が変わらない方法でなければならない。仲間のオブジェクトは常に同一ドメイン上に存在する。そこで、その仲間内のオブジェクトパッシングにおいては、オブジェクトへのポインタを渡すだけとする。すなわち、call-by-reference である。一方、異なる仲間の間では、送信側においてポインタで関連付けられた仲間のローカルオブジェクト群をトポロジを保存して受信側の仲間内にコピーする。これにより、ローカルオブジェクトが仲間内でしか参照されないことが保証される。コピーする中にグローバルオブジェクトへの参照が含まれる場合には、グルーバルオブジェクトへのプロクシを受信側にマージする（図3-A）。すなわち、対象がローカルオブジェクトの場合にはcall-by-value 、グローバルオブジェクトの場合にはcall-by-reference 、という２つの方法を併用する。

オブジェクトマイグレーションは、オブジェクトパッシングと共通部分が多い。異なるのは、コピーの対象がグローバルオブジェクトを含む仲間のオブジェクト全体であり、コピー後、オリジナル側のプロクシをコピー先に変え、オリジナルを削除することである。この中に含まれるプロセスの場合には、移動先で処理が再開できるように実行コンテクストの修正が必要である（図3-B）。



(A) パッシング　　(B) マイグレーション

図3　オブジェクトパッシングとマイグレーション

3．プログラミング言語化

以上のモデルを具体化するにあたっては、オブジェクトがグローバルかローカルかの指定方法を工夫しなければならない。オブジェクトの生成時や交換時にオブジェクト単位にいちいち指定するのは煩雑である。ＯＺでは、プログラムの構文によって指定する方法を採用した。

```
class <className>
  super  <className>
  var    <varNameList>
  method <methodName> (<parNameList>)
         { <body> }
     .....................
end <className>
```

```
monitor <monitorName>
  super  <className>
  var    <varNameList>
  que    <queNameList>
  method <methodName> (<parNameList>)
         { <body> }
     .....................
  action <actionName> (<parNameList>)
         { <body> }
     .....................
end <monitorName>
```

図4　クラスとモニタの記述形式

ローカルオブジェクトを記述する基本単位を、SmallTalk-80[12] にならって classと呼ぶ。一方、グローバルオブジェクトを記述する単位を monitorと呼ぶ。両者をあわせてタイプと呼ぶ。ＯＺは分散プログラミング言語の基本的なアイデアを実証することを目的としているため、タイプの継承はとりあえず、単一継承とした。両者の記述形式を図４に示す。

class においては、super によって他のclass を指定すると、指定した上位class の変数 varと手続きmethodが継承できる。これは通常のオブジェクト指向言語と同様である。class のインスタンスは、ローカルなオブジェクトであり、引数をともなってその method を呼ぶと引数に指定したオブジェクトへのポインタがmethodに渡され、直ちに対応するmethodのbodyを実行し、結果のオブジェクトへのポインタを戻す。ＯＺではSmalltalk-80と違って、ブロックコンテクストを採用していない。これは、次に述べるmonitor によってプロセスが生成されるためである。

monitor はグローバルオブジェクトのタイプを定義するものである。monitor においてsuper で指定できるのは、class またはmonitor のいずれでもよい。class の上位タイプはclass に限定されているのに対して、monitor の上位タイプが限定されていないのは、継承木中で classの上位に monitorが表われるようなタイプのインスタンスはグローバルかローカルであるかが判然としなくなるからである。monitor の提供する手続きには、methodとactionの２種類がある。仲間のオブジェクト以外からmethod またはactionが呼ばれると、両者は何ら区別されずにプロセスが作られ、実行がスケジューリングされる。一方、仲間のオブジェクトからmethodが呼ばれると呼んだ側のプロセスによって直ちにbodyが実行されるが、actionが呼ばれるとプロセスが作られて実行スレッドが分岐し、実行がスケジューリングされる。プロセスを生成して実行したmethodまたはactionが終了すると値を戻してプロセスが消滅する。

class とmonitor のmethodまたはactionを呼ぶ形式としては、処理結果を待つ呼出形式と、処理結果をまたないで処理を継続する送出形式の２種類を設けている。前者によって実行スレッドが分岐するが、同時に複数のスレッドはアクティブにならない。同時に複数のスレッドをアクティブにするには、送出形式を用いなければならない。

仲間オブジェクト内では、一般に複数のプロセスが同時に存在する。このときのスケジューリングは、Hoare[13]のモニタと同じように、同時には１つのプロセスしかアクティブにしないように実行が排他制御される。同期を陽に行なうには、que で指定したキューに対して、signalとwaitを発行することで行なう。仲間以外の他のグローバルオブジェクトに対して要求を送って結果を待つ間、仲間の他の処理をブロックすることにすると、いわゆるモニタの入れ子となり、デッドロックになる。そこで、グローバルオブジェクトを呼ぶ場合には、ブロックしないで相互排除を解く方式を採用した。したがって、ブロックする必要のある場合は、que を用いて相互排除するようにプログラミングしなければならない。

ＯＺ言語を用いて、デマンド型のタイプローグ、分散型デバッガ、ランタイムライブラリ、などのサポートプログラムのほか、ロボットの協調作業プログラムなど、数千行のプログラムが開発されている。これらの記述実験を見ると、プログラミングスタイルに２種類の形態があることがわかる。第１のスタイルは、class を用いて基本的なモジュールをコーディングし、それらを並行処理や分散処理させる段階になってはじめて、下位タイプとしてmonitor を定義するスタイルである。これは、逐次的な記述しかできない言語を用いてモジュールごとにプログラミングし、各モジュールをプロセスで実行させるＵＮＩＸ流の伝統的なスタイルであり、分散の単位となる仲間のオブジェクトはかなり大きなものとなる。第２のスタイルは、並行・分散の記述単位を最初から小さくするスタイルである。これは、Smalltalk-80流のオブジェクトをそのまま分散システム上でシミュレートするような形態となる。どちらのスタイルをとるかはユーザに任されている。

オブジェクトに２種類のものを設けた言語としては、Argus、EPLなどがあるが、それらは分散システムを念頭におかない単一ドメイン用の言語を拡張したものである。それらの言語では、タイプの継承機能やローカルオブジェクト群のトポロジを保存してコピーする機能を備えていない。この点が改善されている。

4．　実装

4．1　処理系

タイプは継承関係で関連づけられた半順序集合であり、タイプの集合は木構造（継承木）をなす。この性質を利用すると、上位タイプから下位タイプへとワンパスでコンパイルすることができるし、木構造のルートに近い部分のデバッグの完了したタイプ群と木構造の末端に近いデバッグ中のタイプ群とを分割コンパイルし別のファイルに記憶することも可能になる。しかし、別ファイルに格納された上位タイプの参照方法や、上位タイプ変更時の下位タイプの更新方法などの工夫が必要となる。

ＯＺ言語コンパイラは、ユーザの定義したソースプログラムをタイプ単位にコンパイルし、その結果をパッケージと呼ぶファイルに格納する。コンパイラが更新するパッケージ（アクティブパッケージ）は１つであるが、参照するパッケージ（リファレンスパッケージ）は複数である。コンパイラは、変数やそれ自身を含む上位タイプの手続きをオフセットで参照する形式のバイトコード列を生成し（コンパイル時に定まらないものは名前で参照）、ソースプログラムのファイル名や出現箇所などのデバッグ情報とともに、タイプ名をキーとしてパッケージに格納する。さらに、前述のタイプ間の相互依存関係の管理のために、そのパッケージが記憶するタイプ情報の一覧（タイプ辞書）も記憶する。タイプ辞書には、タイプごとに、タイプ名、上下の継承関係、別パッケージに存在する上位タイプ名、作成日時、下位タイプの再コンパイルが必要になった日時、が記憶される。コンパイラ起動時には、アクティブパッケージとリファレンスパッケージ群のタイプ辞書を読み込んで継承木を組み立てる。タイプをコンパイルするたびにその継承木を保守し、コンパイル終了時に、アクティブパッケージが格納するタイプに関連するタイプ辞書をパッケージに記憶する。パッケージに格納する情報は、異機種計算機で共有利用す

るっことを考え、機種に依存しないＯＳＩの抽象構文規則ＡＳＮ．１"の符号化規則を使って符号化されている。

上位タイプを書き直して再コンパイルしたとき、上位タイプの変更箇所によっては下位タイプを再コンパイル（ＯＺでは変数と手続きのリンクを含む）する必要はない。これを検出する機能は今後の課題である。現在は、下位タイプの再コンパイルの必要になった日時には作成日時を入れており、再コンパイルが必要なタイプの検出はコンパイラとは別のユーティリティで行なっている。

４．２　実行系

実行系は、各ドメインの内部に存在する仮想的な処理装置である。パッケージからタイプをロードしてインスタンスを生成し、２章と３章のモデルのように振る舞う。オブジェクトの識別方法、交換形式、交換手順などが守られている限り、実行系はそれぞれ独自に実装することが可能であるが、現時点では単一の実装方法を採用している。

４．２．１　オブジェクトの内部表現

実行系の処理対象はすべてオブジェクトという形態をとる。その内部形式は、メモリと処理の効率、ガーベッジコレクション、および転送時の形式変換の機械的処理、を考慮して分類し、その区別のためにタグを付けている。オブジェクトの内部形式は、１ワード（４バイト）で表現されるワード形式と複数ワードで表現されるブロック形式に分類される。

ワード形式は、下位２ビットのタグによって、整数、実数、オブジェクトポインタ（ＯＰ）、拡張（ＥＸＴ）に分けられる。拡張形式は、実行系が特殊用途に使用するためのものであり、さらに６ビットのタグによって細分される。ＯＰ以外のワード形式におけるタグはタイプの扱いを受ける。

ブロック形式は、プロクシ部とセル部に分けられる。ローカルオブジェクトはグローバルオブジェクトと違い、論理的には２部に分ける必要はないが、オブジェクトの解釈、ガーベッジコレクション、転送時の形式変換などを統一的に行なうために、グローバルオブジェクトにあわせて２部に分けている。プロクシを集めたものがオブジェクト表である。オブジェクト表には、その実行系内に存在するブロック形式のオブジェクトのプロクシだけでなく、他の実行系内に存在するグローバルオブジェクトを外部参照するためのプロクシも含まれる。

プロクシ部属性フィールドの属性タグにより、セル部の内部表現がさらに次のように細分されている。

(1) シンボル：　　タイプ名、メソッド名、アクション名、リテラルなど。
(2) タイプ：　　パッケージ中のタイプをロード・リンクした実行形式。
(3) ﾓﾆﾀｲﾝｽﾀﾝｽ、ｸﾗｽｲﾝｽﾀﾝｽ、各種配列ｲﾝｽﾀﾝｽ：　それぞれに対応した効率よいメモリ表現をとる。
(4) プロセス、スタック、キュー：　モニタインスタンスを構成するもので、ユーザには見えない。

クラスインスタンス、各種配列インスタンス、プロセス、スタック、キューなどは、実行系内でしか参照されないのでプロクシ部識別フィールドは意味を持たない。しかし、シンボル、タイプ、モニタインスタンス、および外部参照の場合には、分散システム全体で一意に付番されたＵＩＤが入れられる。ある特定のＵＩＤを持つモニタインスタンスはシステム全体で１つに限られるが、シンボルとタイプの場合には、実行系内では１つに限られるが、各実行系にその複製を存在させることができる。

セル部は、それぞれの用途にあった効率よいメモリ表現になっている。その主なものを図５に示す。

```
typedef struct {
  objInt heapHead:
  objPnt type:
  ObjPnt superType:
  objInt typeAtr:
  long   instVQn:
  long   n_metInf:
  long   p_bytCod:
  long   p_litInf:
  long   p_insSymInf:
  long   p_linInf:
  long   p_tmpSymInf:
  metInf metInf[*]:
  bytCod bytCod[*]:
  litInf objPnt[*]:
  insInf objPnt[*]:
  linInf linInf[*]:
  tmpInf tmpSymInf[*]:
  } typeObj;
```

```
typedef struct {
  objInt heapHead:
  objPnt type:
  objInt miAtr:
  objInt msgCnt:
  objPnt entq:
  objPnt recq:
  objPnt rstq:
  obj    instVar[*]
  } monitorInstObj

typedef struct {
  objInt heapHead:
  objPnt type:
  obj    instVar[*]:
  } classInstObj
```

```
typedef struct {
  objInt heapHead:
  objPnt type:
  objPnt prev:
  objPnt next:
  objPnt moninst:
  objExt errObj:
  byte   *pc:
  long   *fp:
  long   *sp:
  long   *ap:
  objPnt stack:
  objInt msgId:
  objInt waitMsg:
  objInt trapWord:
  objInt rstCode:
  } processObj
```

図５　オブジェクトセル部の内部表現

４．２．２　内部構成

オブジェクトを解釈実行する実行系の構成を図６に示す。オブジェクトメモリマネジャＯＭＭは、ヒープ上にオブジェクトを作成・削除し、オブジェクトを構成するインスタンスとタイプを関係づける。バイトコードインタプリタＢＣＩは、タイプが保持するバイトコードを順次フェッチ実行し、インスタンスの内部状態を更新する。バイトコードにしたがって、実行系にあらかじめ組込まれたメソッド群ＰＭＳのコール、クラスインスタンスのメソッドコール、モニタインスタンス間での要求・応答の交換、を行なう。グローバルメッセージマネジャＧＭＭは、モニタインスタンス間における要求・応答の交換時にオブジェクト群のコピーを行なう。相手となるモニタインスタンスが他の実行系に存在するときに

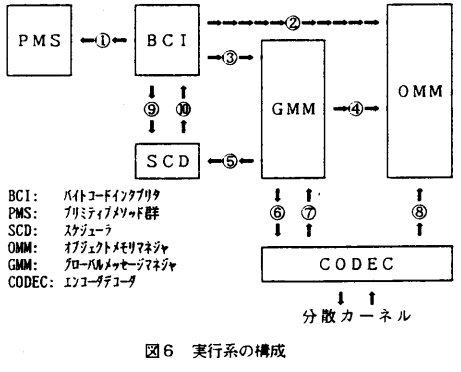


図６　実行系の構成

は、送信側のエンコーダデコーダＣＯＤＥＣで関係を保存するよう符号化してパケットに詰め、受信側のＣＯＤＥＣで復号化してヒープ上にオブジェクトを再構成する。モニタインスタンス内やモニタインスタンス間において、プロセスの切り替えを行なうものがスケジューラＳＣＤである。ヒープ上のオブジェクトは実行系内の全域からアクセスできる。

４．２．３　実行制御

ＢＣＩは、プロセスを切り替える要因が検出されるまで、ＳＣＤが選択したプロセスの動作を連続してシミュレートする。プロセス実行中に制御の流れが大きく変わり、それがユーザに見えるのは、プロセスの切り替えと例外／エラー処理の場合である。

プロセス切替の要因は、他のモニタインスタンスへの要求発生、同一モニタインスタンスのactionに対する要求発生、モニタインスタンスの要求処理の終了、キューに対するwait要求のさいの事象待ち発生、に限られ、その他のときは、プロセスの状態とプロセス実行待ちリストが変更されるだけである。しかし、ミクロに見れば、ＢＣＩは分散カーネルとのインタラクションにおいて次のような複雑な振る舞いをする。パケット送信終了やパケット受信通知の事象をＢＣＩが知るために、イベントフラグが設けられており、ＢＣＩはフェッチ実行サイクルの先頭でこれを検査し、事象が発生していれば、ＧＭＭの呼出しなどの処理に入る。このような方法を採用した理由は、ＵＮＩＸ上に実装する場合に複数のＵＮＩＸプロセスを用いても、プロセス間の通信、同期、相互排除などが複雑になるし、任意の場所でＵＮＩＸプロセスを切り替えても意味がないためである。

オブジェクトの処理中に検出した例外やエラーの処理をユーザがＯＺ言語を用いて記述できるようにするために、ＢＣＩは要因解析に必要な情報をパラメータにして、要因を発生したオブジェクトまたは処理中のオブジェクトの所定のメソッドを陰に入れ子型に呼び出す。例外やエラーは次の３種類に分類されており、それぞれの種類によって呼び出すメソッドが異なる。

(1) トラップ　トラップ命令、端末からの割込など、命令の実行前に検出される例外。
(2) フォルト　タイプやシンボルの未ロードなど、命令の実行中に検出される例外。
(3) エラー　　メソッド未定義、通信エラー、陽なエラー通知など、上記以外の種々の複雑な要因。

トラップとフォルトは、それぞれデバッガとタイプ／シンボルのデマンドローダの実装に使っている。

これらの要因で呼びだされたメソッドからの回復方法には、継続型と放棄型があり、渡されたパラメータによって選択できる。継続型の場合、呼びだされたメソッドの値を例外やエラーの発生した式の値に置き換えて処理を再開する。一方、放棄型の場合は、呼びだされたメソッド／アクションの処理を強制終了させ、呼びだした側にエラーを通知する。

４．２．４　オブジェクトの転送

4.1 に述べた形式のオブジェクトを2.3 のモデルのように実行系間で転送するＣＯＤＥＣについて述べる。符号化と復号化のアルゴリズムの詳細はすでに発表した[4] ので詳細は省略する。

転送されるパケットは図７のような形式をとる。ここで、宛先UID は宛先モニタインスタンスのＵＩＤ、発信UID は発信元モニタインスタンスのＵＩＤ、メッセージIDはRETURNを受信したときにどのCALLに対応するRETURNかを識別するためのＩＤ、セレクタLID は宛先のメソッド／アクションを識別するためのＬＩＤ、引数／結果LID は引数や結果を識別するためのＬＩＤである。ここでＬＩＤ(local ID)とは、オブジェクト列に格納されたオブジェクトと対応をとるためにパケット内だけで一意になるようにつけた通番であり、パケット中のオブジェクトへのポインタに代替するものである。オブジェクト列部は、オブジェクト表のサイズを先頭にもったオブジェクト表とオブジェクトセル列から構成される。オブジェクト表部の各プロクシにはそれを一意に識別するためにＬＩＤがついており、属性とＵＩＤフィールドは実行系内のものと同一であり、意味的には実行系内におけるプロクシと全く同じものである。また、オブジェクトセル列部には、セルが存在する場合には対応するプロクシのＬＩＤと同じＬＩＤを先頭においたオブジェクトセルを詰める。実行系内においてオブジェクトポインタである場所には対応するプロクシのＬＩＤを入れ、アドレスである場所にはオフセット値を入れる。アドレスをオフセットに変換し、逆にオフセットをアドレスに戻せるようにプロセスやスタック中の形式が工夫されている。転送するオブジェクト群はパケット内では、パケット内から外部を参照するプロクシはすべてＵＩＤを持ったグローバルなものとなっており、オブジェクトポインタがＬＩＤに変換されたり、アドレスがオフセットに変換されてはいても、パケット内のオブジェクト列は実行系内のものと意味的には同じものである。具体例を図８に示す。

モニタインスタンスが別の実行系に移動すると、もといた実行系には移動先へのプロクシが残される。この場合、移動前のプロクシを指していたプロクシから目的のオブジェクトのプロクシまでの間に１つ余分にプロクシが挿入され、次々に移動すると中間のプロクシの数が段々と増える。この場合に

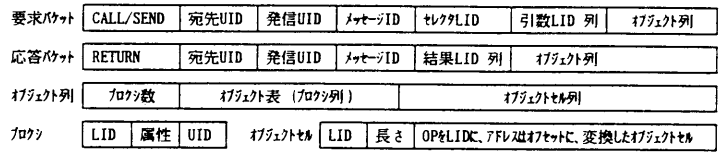


図７　パケットの形式

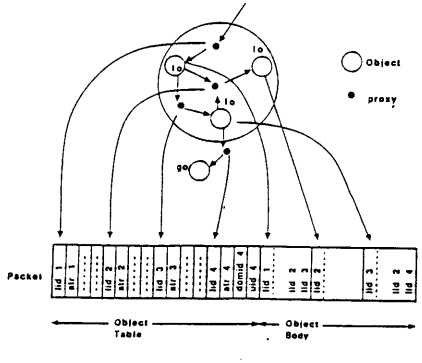
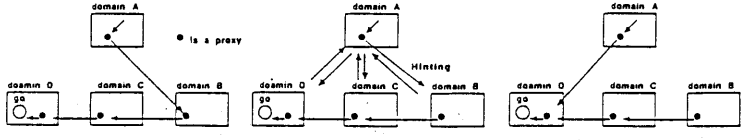


図８　パケットの例

モニタインスタンスが別の実行系に移動すると、もといた実行系には移動先へのプロクシが残される。この場合、移動前のプロクシを指していたプロクシから目的のオブジェクトのプロクシまでの間に１つ余分にプロクシが挿入され、次々に移動すると中間のプロクシの数が段々と増える。この場合には、中間のプロクシを省略して、参照元から目的のプロクシまでを直結する必要がある。ＯＺでは、ヒンティングとよぶ通信方式を考案し、これによって複数段になった参照の短縮を図ることにしている。次のような方式である。他の実行系を指すプロクシが受信したら、移動先を発信元に返し、発信元では自分のプロクシの宛先をそれに変更して、新しい宛先に再送する。オブジェクトは他から参照されている限り消滅することはないので、この過程を繰り返すといずれ最終宛先が得られる（図９）。



a. Before Hinting　　b. Hinting　　c. After Hinting

図９　ヒンティング

４．３　分散型実行系

複数の実行系を組み合わせて実現した、分散して協調動作する単一ユーザ用分散型実行系について述べる。

４．３．１　分散型実行系の構成とユーザ管理系

ＯＺシステムでは、各ステーションにはそのステーション内の管理を行なうステーション管理系が常に稼動しており、ネットワークからの要求を受けて、実行系を作成して起動したり、停止させて削除する。最初に１つの実行系を起動し、その支援を受けながら次々と実行系を起動する形態をとっている。最初に起動された実行系をユーザ管理系、その後に次々と起動されてユーザ管理系の支配化に入る実行系をユーザ実行系と呼ぶ。どちらも実行系としては同じものであるが、ロードされるタイプの違いによって機能が違ってくる。

分散型実行系の中ではユーザ管理系が中核的な役割をはたす。その機能の主なものは、ユーザ実行系の生成起動と停止削除の支援、グローバルオブジェクトに対する一意なＵＩＤの付与、ユーザ実行系へのシンボルとタイプのロード、実行系にまたがった分散ガーベッジコレクション、などの基本的なもののほか、ディレクトリやデータベース的な共有環境の提供である。多くの機能は、モニタsymboltableのインスタンス（以下、シンボルテーブル）が中心となって行なう。

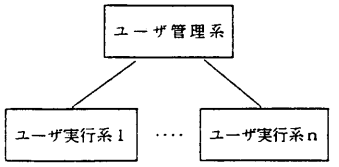


図１０　分散型実行系の構成

４．３．２　環境管理

ユーザ管理系は次のようにして立ち上がる。実行系が起動されると、起動パラメータからユーザ管理系であることが解れば、クラスobject、クラスsymbol、モニタsymboltable 、モニタstartup 、という４つのタイプをパッケージからロードする。次に、startup のインスタンスを生成して初期化メソッドによってシンボルテーブルの生成とデマンドロードに必要なタイプをロードする。最後に、ファイルサービス、端末サービス、ステーション管理系への取次サービス、などを行なうオブジェクトを生成して起動が完了する。

ユーザ実行系の生成起動はシンボルテーブルを介して行なう。シンボルテーブルはドメイン生成エージェントを生成し、ステーション管理系を通して目的のステーション管理系にユーザ実行系の生成起動を指示する。ステーション管理系は、ユーザ管理系のドメインＩＤ、シンボルテーブルのＵＩＤ、ドメイン生成エージェントのＵＩＤなどをパラメータとして実行系を起動する。実行系はシンボルテーブルから必要最小限度のシンボルのＵＩＤを取得したのち、モニタstatutp をロードする。startup のインスタンスを生成して初期化メソッドによってデマンドロードに必要なタイプのロードと各種サービスを行なうオブジェクトへのプロクシをシンボルテーブルから取得する。その後、ドメイン生成エージェントに結果を通知し、シンボルテーブルに実行系のアドレスを登録すると生成起動が完了する。

ユーザ管理系とユーザ実行系は、その生成起動が完了すると、それ以後のタイプとシンボルのロードはデマンドロードが可能である。

ユーザ実行系の停止削除もシンボルテーブルを介して行なう。目的のユーザ実行系のカーネルエージェントに停止削除を要求する。エージェントはステーション管理系に通知し、ステーション管理系が実際の削除を行なう。その結果をシンボルテーブルが受け、シンボルテーブルから実行系のアドレスを削除すると、停止削除が完了する。

４．３．３　ネーミングとローデング

実行系間では、グローバルオブジェクト（シンボル、タイプ、モニタインスタンス）はＵＩＤで識別する。シンボルとタイプの場合には、文字列表現が同じものには同じＵＩＤを与え、異なるものには異なるＵＩＤを与えなければ、言語上の記述どおりには動作しない。これらに対する付番がもっとも多く発生するのは、タイプをパッケージから読み込みリンクしてオブジェクト化するときである。そこで、それを行なうシンボルテーブルで付番することにした。そのため、各ユーザ実行系で名称に対して付番したり、ＵＩＤから対応する実体をえることはできない。シンボルテーブルに処理を依頼しなければならない。モニタインスタンスのＵＩＤは、他の実行系の付番と競合していなければ、勝手に付番してよい。そこで、各ユーザ実行系は、あらかじめ自由に使える番号集合をユーザ管理系から取得しておき、その中から自由に選んで使用する方式をとった。

各実行系がタイプやシンボルが不在なことを検出するとフォルトが発生し、クラスobjectのメソッドを通してシンボルテーブルにロードが要求される。継承木の上位にあるタイプはロードされている可能性が高い。そのため、タイプのロードにあたっては、必要なタイプだけを選んで転送しなければ効率が悪い。各タイプごとに、ロード済の実行系に対応するビットにマークをつけるためのビットベクトルを設けた。シンボルテーブルでは、要求されたタイプから初めて継承木の上位へ順にタイプをたどり、ロード済のマークを検出するまで、未ロードのタイプをバケットにつめ、同時にロード済マークをつける

ことを繰り返す。これにより、必要かつ十分なタイプだけを転送することが可能になった。

4．3．4　分散ガーベッジコレクション

ヒープを消費する大半を占めるローカルオブジェクトは他の実行系から参照されることはないので、実行系独自なローカルＧＣ（ガーベッジコレクション）によって回収できる。しかし、グローバルオブジェクト（モニタインスタンス）と外部を参照するプロクシは複数の実行系に渡って検査しなければゴミかどうか解らない。複数の実行系に渡るＧＣがグローバルＧＣである。

ＯＺでは、ローカルＧＣに重畳させてグローバルＧＣを行なう方法を採用した。両ＧＣともにマーク＆スイープ法をベースにしている。ただし、グローバルＧＣのマーキングは単純な１ビットのマークではなく、mod ３のグローバルＧＣの世代を用いる。プロクシの写しを送信するときには送信側プロクシの世代も伝える。

各ローカルＧＣでは、タイプとシンボルは無条件に保存し、保存が陽に指定されたモニタインスタンスと他実行系から参照されているプロクシをルート（他から参照されてプロクシ一覧をローカルＧＣの開始時にユーザ管理系から取得する）としてグローバルＧＣの世代を伝播させる。伝播が終了したとき他実行系への参照があらたに見つかると、新たに見つかった参照宛先実行系アドレスとＵＩＤの組のリストをユーザ管理系に通知する。その量は次第に減ってくる。このようなローカルＧＣを各実行系が行なう。すべての実行系がローカルＧＣを行ない、しかも新たな外部参照が見つからないときグローバルなマーキングが終了したと見なし、世代を１増やす。ＧＣの新しい世代になって最初のローカルＧＣでは２つ古い世代のプロクシとそれが指すオブジェクトも回収する。その例を図１１に示す。

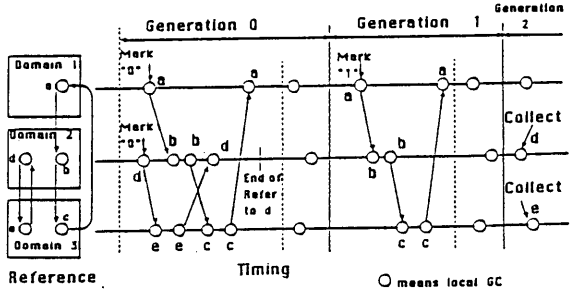


図１１　分散ガーベッジコレクション

5．　むすび

継承機能を有するオブジェクトを基本単位とする『オブジェクト指向開放型分散システムＯＺ』をプログラミング言語システムの側面からとらえて、分散処理のモデル、プログラミング言語、およびその実装について述べた。本稿で述べたシステムはわずかな部分を除き、ほぼそのまま稼動している。本システムの課題の１つは、複雑なデータ構造が交換できるため、Ｃ言語など他言語とインタフェースをとるのがむずかしいことである。実行系と様々なレベルでインタフェースがとるようにしたり、インタフェースジェネレータによってインタフェースを生成する方法などが考えられる。第２の課題は、自由度と性能を両立させるためにバイトコードとネイティブコードを混在させる方法の検討である。機種に依存させないで自由度を求めるときはバイトコード、性能を求めるときはネイティブコードと使いわけられ、両者が混在できるのが望ましい。

ＯＺは、オブジェクト指向型であり、しかも開放型であることを謳い文句にしている。ここで、開放型というのは、ＯＳＩネットワークの世界で用いられている「仕様が公開されていて、だれでもが自由にその仕様に準拠するシステムが作れる」という意味ではない。ＯＳＩの世界では、仕様は規格化されているため、製品開発者や製品利用者がその仕様に不満があっても、拡張や変更は許されない。拡張や変更をすると、他の製品と接続できなくなるからである。筆者が考える開放型とは、規格などの仕様を書き換えることなく、機能的に変更・拡張できることを意味する。このような開放性を備えていなければ、せっかく規格に準拠した製品であっても、機能が不足したり、使い勝手が悪かったりするとだれも使ってはくれない。規格は固いがらこそ、機能的に変更・拡張できることが必要なのである。これを可能にするのが継承機能をもつオブジェクト指向のアプローチである。

仕様を実現したプログラムがネットワークを介して送られてくるというアプローチもＯＳＩと対比できる。ＯＳＩでは、数多くの規格ごとに、それを実装するための仕様書を作り、仕様書に基づいてそれぞれに実装しなければならない。ＯＺでは、オブジェクトを自由に転送し実行する基本システムさえ実装すれば、あとはネットワークを介してプログラムがロードできる。

このアプローチをさらにすすめるには、本稿で述べた単一ユーザ環境を想定したものでは不十分である。複数ユーザ環境を対象として拡張し、様々なバージョンのタイプが混在する場合でも稼動できなければならない。データベース機能の導入も必要と思われる。

謝辞　本研究は通産省大型プロジェクト『電子計算機相互運用データベースシステム』の一環として進めているものであり、その機会を与えられた横上昭男電総研情報アーキテクチャ部長に感謝します。また、ＯＺプロジェクト推進にあたってご協力いただいている、住友電気工業㈱、松下電器産業㈱、シャープ㈱　各社の関係各位にも感謝します。

文献

1) Cook.R.P.: *MOD - A Language for Distributed Programming. IEEE Trans SE. SE-6.6 (1980
2) Andrews.G.R.: The Distributed Programming Language SR - mechanism, design and implementation. Soft - Prac & Exper 12.8 (1982)
3) Strom. R.E. etal.: NIL: An Integrated Language and System for Distibuted Programming. ACM SIGPLAN 18.6 (1983)
4) Liskov.B.: On Linguistic Support for Distributed Programs. IEEE Trans SE. SE8-5(1982)
5) Herlihy. M. and Liskov. B.: A Value Transmission Method for Abstract Data Types. ACM TOPLAS 4.4(1982)
6) Jones. M. etal.: Mach and Matchmaker: An Interprocess Specification Language. Proc 12th ACM POPL (1985)
7) ISO 8825: OSI - Specification of Basic Encoding Rules for ASN.1
8) Tsukamoto.M. etal.: The Architecture of OZ: Object-Oriented Open Distributed System. Proc 2nd ISIIS (1988)
9) Almes.G.T. etal.: The Eden System: A Technical Review. IEEE Trans on SE. SE11-1(1985)
10) Black. A. etal.: Object Structure in the Emerald System. Proc OOPSLA'86 (1986)
11) Jul.E. etal: Fine-Grained Mobility in the Emerald System. ACM TOCS 6.1(1988)
12) Goldberg.A. and Robson.D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley(1983)
13) Hoare. C.A.R: Monitors: an Operating System Structuring Concept. CACM 17.10 (1974)
14) 塚本他：ＯＺ：対象指向開放型分散ｼｽﾃﾑｱｰｷﾃｸﾁｬ - ｵﾌﾞｼﾞｪｸﾄの表現形式と形式変換、情処第37回全国大会(1988)