プログラミング 8－12
（1996. 8. 28）

対称型メッセージ送信とその実装

久野　靖†、大木敦雄†、鵜林尚靖‡
†筑波大学大学院経営システム科学専攻、‡(株) 東芝、東大

多くのオブジェクト指向言語に見られるメッセージ通信方式は、メッセージの送り手が相手先や送信順序の制御を持つという点で本質的に非対称である。本稿では動的なメッセージの通信路設定と送信 / 受信メソッドを分離したメッセージ機構 (対称型メッセージ) を提案し、またそれに基づく並行オブジェクト指向言語の設計と実装について論じている。対称型メッセージでは通信路の設定が計算本体と分離するためそのトポロジーを把握しやすく、また各メソッドとオブジェクトの状態の対応関係が明確になる。さらに、メソッド実行途中の待ち合わせが不要なため、効率的な実装が可能である。

"Symmetric" Message Passing and Its Implementation

Yasushi KUNO†、Atsuo OHKI†、Naoyasu UBAYASHI‡
†Graduate School of Systems Management, the Univ. of Tsukuba,
‡Toshiba Corporation and the Univ. of Tokyo

Message passing mecanism as seen on most of current object-oriented programming languages are asymmetric in that both message destinations and orderings are solely controlled by the sender object. In this paper we propose a message-passing mechanism called "symmetric message passing" (SMP), in which dynamic message-link establishment and send/receive methods are clearly separated. We also propose a concurrent object-oriented programming language that incorporates SMP. In SMP, separation of computational method body and message topology connection make it easier to understand the written code, and correspondence between objects' internal states and each method. Moreover, in SMP methods are executed without suspension, enabling efficient and simple implementation.

1 はじめに

並行オブジェクト指向言語は、多数のCPUと局所メモリを持つ並列計算機(MPP)やワークステーションクラスタ(WSC)との親和性が高く、並列性を明示的に記述するプログラミングのパラダイムとして有望である。その反面、並行オブジェクトに基づく計算モデルの原点であるアクターモデルはごく原始的なものであり、実用的な並列オブジェクト指向プログラミングを目指して多くの言語機構が提案されている状況にある。

本稿では旧来のメッセージ送信における非対称性に着目し、これを対称であるように変更したモデルとそれに基づくプログラミングについて論じる。以下第2節ではメッセージの非対称性と本稿で提案する対称型メッセージ機構について述べる。第3節では対称型メッセージ機構に基づく言語p1について説明し、第4節ではその実装について述べる。最後に第5節で議論とまとめを行う。

2 対称型メッセージ送信

2.1 メッセージの非対称性

並行オブジェクト指向プログラミングでは、メッセージ送信はオブジェクト間で情報を受け渡す手段であると同時に、制御の同期を取る手段でもある。1つのオブジェクトについて一時にはたかだか1つのメソッド実行しか許さない言語では、送信されたメッセージのどれを受理するかの選択を通じて条件同期が実現される。複数メソッドの並行実行を許す言語でも、その中で排他実行オブジェクトを通じて同期を実現する場合は同様である。

多くのオブジェクト指向言語では、メッセージの送信側と受信側は次の点で非対称である。[1]

1. 送信側は自分が知っているオブジェクトの中から送信先を選択できるが、受信側は誰からメッセージが送られてくるかを制御できない。
2. 送信側は都合に応じて複数のメッセージを送り分けられるが、受信側は(受信したメッセージの処理順序を調整できることはあっても)メッセージを「送らせない」ようにはできない。
3. 送信側は自分の選んだ順序でメッセージを送ることができるが、受信側はどの順番でメッセージが来るかを制御できない。
4. 送信側はメソッドの実行途中に複数のメッセージを送ることができるが、受信側は1つのメッセージに対して1つのメソッドの先頭から実行を開始することしかできない(図1)。

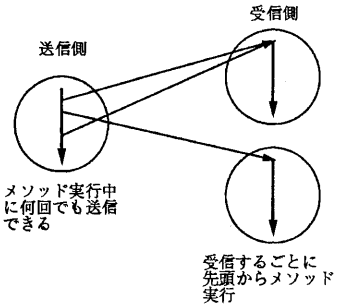

図1: メッセージの非対称性

明示的な受信文を持つ言語の場合はsendとreceiveが対になるので、上記の指摘に該当しない。しかし、このような言語では送信側と受信側が特定の実行位置に来た時にメッセージが交換されるため、双方の状態の把握が難しく、実質は受信側が無限ループの先頭など決まった箇所でだけ受信を行うコード設計が多い。その場合は上記の指摘が当てはまる。

2.2 対称型メッセージ

しかし、プロセス計算[4]などの並列計算モデルを見ると、送信側と受信側が非対称である必然性は特になさそうである。そこで、プログラム言語においても送信側と受信側が対称であるようなメッセージ機構を使用すれば、並列計算モデルとの親和性がよく、プログラミングしやすい言語が構成できる可能性がある。

[1] これは直列言語の構文上で言えば「goto文はあるがcome-from文はない」「call文はあるがcalled文はない」ということに相当している。:-)

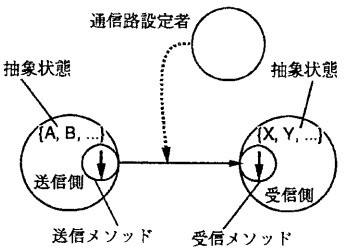

図 2: 対称型メッセージ送信

筆者らが提案してきたプロデューサモデルとそれに基づく言語 Produce/1[3] においては、実際にそのようなメッセージ送信機構 (対称型メッセージ) を採用している。対称型メッセージでは、オブジェクト間の通信路を通信が起きるより前の時点で予め設定する。Produce/1 では通信路を設定できるのはプロデューサと呼ばれる特定のオブジェクトに限られていたが、本稿では任意のオブジェクト (送信者や受信者でも第3者でもよい) が通信路を設定できるものとする (図 2)。

オブジェクト間で明示的に通信路を設定すること自体特に目新しくない。たとえば並列論理型言語に基づくオブジェクト指向プログラミングなどではオブジェクト間をストリームによって接続して通信を行うし、オブジェクトに備わったポート間を接続して通信路とする言語もある。しかし、それらの言語では通信路の記述が静的 (実行時に変更できない) だったり、通信のタイミングが送信者の制御に任されていた。

対称型メッセージでは、抽象状態同期 [1][2] の考えに基づき、送信者と受信者がそれぞれ特定の状態 (抽象状態) になった時に「発火」が起こり、メッセージが送られる。なお、抽象状態とはオブジェクトの内部状態を抽象化したものであり、オブジェクトの種類ごとに宣言され、各メソッドにおいてオブジェクトの内部状態に対応した値に設定される[2]。具体的には、メッセージ送信の手順は次の通りである。

[2]オブジェクトの状態に基づいたプログラムの制御についてはほかに [6]、[7] などの事例がある。

a. 任意のオブジェクトが、送信側、受信側それぞれについてオブジェクトとメソッドを指定して通信路を設定する。

b. 送信側 / 受信側メソッドとも、実行を開始できる抽象状態が規定されている。通信路が設定されても、その両側のメソッドがともに実行可能になるまでは送信は行われない。

c. 両メソッドがともに実行可能になると、まず送信側メソッドが起動されて送信値を組み立て、次に受信側メソッドが起動されて受信値に基づく処理を行う。設定した通信路は、メッセージが 1 度送られるとその時点で消滅する。

対称型メッセージによる計算のイメージは、図 3 にあるように、計算の進行に伴って通信路のネットワークが形成され、その後を追って「発火」のウェーブフロントが追いかけてくる、というものである。

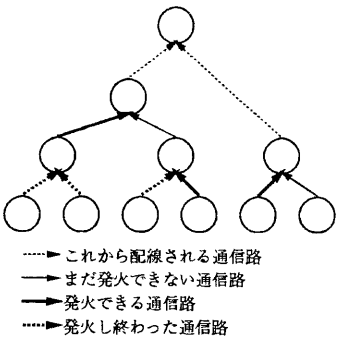

図 3: 通信路の設定と発火

3 p1: 対称型メッセージ送信に基づく言語

3.1 p1 プログラムの基本構造

対称型メッセージ送信の記述性を調べるために、対称型メッセージに基づく並行オブジェクト指向言語を設計し、「p1」と名付けた。p1 は機能的には Produce/1 からプロデューサと俳優の区分をなくして整理し、代りに対称型メッセージに関連

する機能を追加したもので、構文的には筆者らが以前に開発した言語である p6[2] に類似している。

p1 のプログラムはクラス記述の集合である。各オブジェクトはいずれかのクラスに属し、オブジェクトの状態変数やその型、取り得る抽象状態、メソッド等はクラス記述において定義される。クラスの基本的な形は次の通り。

```
クラス名 = class { 抽象状態,... }
  slot 状態変数: 型,...
  メソッド ...
end クラス名
```

メソッドには、手続きメソッドとポートメソッドがある。手続きメソッドは常に同期的に実行され、その実行開始時や実行中に待ち状態となることはない。新しいインスタンスを生成して返すコンストラクタも手続きメソッドの一種である。一方、ポートメソッドは対称型メッセージ送信のために使われる。メソッドの基本的な形は次の通り。

```
名前 = 種別(引数, ...) [returns(型)]
  文 ...
end 名前
```

「種別」は proc(手続き)、port(ポート) のいずれかである。returns は手続きメソッドにのみ書け、返値の型を宣言する。一方、ポートメソッドでは引数指定においてメッセージが発火できるための抽象状態条件を記述できる。通信路の開設は、次の形をした link 文により行う。

```
link ポート呼び; ポート呼び; ... end
```

「ポート呼び」はいずれも「オブジェクト！ポートメソッド名(引数, ...)」の形をとり、メッセージを送受するオブジェクトとメソッドを指定する。引数は入力引数と出力引数に分けられ、ある呼びの出力引数に書かれた名前を別の呼びの入力引数に指定することで値の伝達を指定できる (加えて通信路開設者から各メソッドの入力引数に値を渡すこともできる)。

3.2 例題: エラストテネスのふるい

例題として、エラストテネスのふるいにより素数を求めるプログラムを示す。まず、「ふるい」のクラスを示す。ふるいの各段はデータが入っている状態 full と入っていない状態 empty を持ち、手前の段からデータを受け取ると full になり、次の段にデータを送り出すと empty になる。

```
sieve = class { empty, full }
  slot num:int,data:int,next:sieve,o:stream
  new = proc(n:int, o:stream) returns(sieve)
    s:sieve := sieve$[num:n,data:0,next:s,o:o]
    return(s!{empty})
  end new
  sendfwd = port({full}s:sieve, ^d:int)
    d := s.data; s!{empty}
  end sendfwd
  recvfwd = port({empty}s:sieve, d:int)
    if d // s.num = 0 then
      s!{empty}
    elif s.next == s then
      s.o!putint(d)!newline()
      s.next := sieve!new(d, s.o); s!{empty}
    else
      link s!sendfwd(x);s.next!recvfwd(x) end
      s.data := d; s!{full}
    end
  end recvfwd
end sieve
```

コンストラクタ new はこの段がふるうべき整数と出力ストリームを引数として受け取り、これらを状態変数に格納したインスタンスを生成して返す。最初の状態では状態は empty である。

sendfwd は次の段にデータを送るためのポートメソッドであり、引数 d がそのための出力引数である (^ で表す)。自分が保持している値は、自分までの段ではふるい落とされなかった整数ということになる。

recvfwd は手前の段から値を受け取るためのポートメソッドである。まず受け取った値が自分の保持している整数で割り切れるかどうか調べる (「//」は剰余演算子である)。割り切れればその値は捨ててしまい、自分の状態を empty にする。割り切れず、まだ次の段が存在しなければその数は素数なので出力し、次の段を生成し、これでこの値の処理は済んだので自分の状態を empty にする。最後に次の段がある場合には、次の段が空いた時に自分から次の段にデータを送るべく通信路を開設し、値を記憶した上で自分の状態は full にする。

次に、指定された値までの整数を次々に出力するオブジェクトのクラス ints を示す。

```
ints = class { ready, finish }
  slot cur:int, limit:int, next:sieve
  new = proc(i:int,l:int,n:sieve) returns(ints)
    return(ints$[cur:i,limit:l,next:n]!{ready})
  end new
  sendnext = port({ready}i:ints, ^d:int)
    i.cur := i.cur + 1; d := i.cur
    if i.cur >= i.limit then
      i!{finish}
    else
      link i!sendnext(x);i.next!recvfwd(x) end
      i!{ready}
    end
  end sendnext
end ints
```

このクラスのオブジェクトは「次の値が用意できている」状態と「最後の値を送り終わった」状態を持つ。コンストラクタ new では、はじめの値、終わりの値と、値を送り込む最初の段を受け取り状態変数に格納する。値を送るのは送信メソッド sendnext で、まず現在の値を送った後値を 1 増す。その結果最後の値を越えていれば状態を finish にするが、そうでなければ再度自分とふるいの最初の段との間に通信路を設定する。

実行はクラス main の start(手続きメソッドでなければならない) から開始される。これは単にオブジェクトを生成して最初の通信路を開設するだけである。

```
main = class
  start = proc()
    o:stream := stream!stdout()
    s:sieve := sieve!new(2, o)
    i:ints := ints!new(2, 10000, s)
    link i!sendnext(x); s!recvfwd(x) end
  end start
end main
```

3.3 受信側による制御

先のコードでは、各 sieve が次の段を slot に保持して通信路の設定を行っていたため、比較的旧来の言語によるコードと類似していた (ただし通信が起きるのはある段でデータが用意でき次の段が空いている場合に限られるという点は特徴的)。これに対し、各 sieve が前の段を slot に保持して通信路の設定を行うようにプログラムすることも容易である (コードは紙面の都合から略す)。この両者のスタイルを push-based/pull-based message と呼んで区別することもある。また、送信側と受信側で処理能力のアンバランスがある場合、push-based ではメッセージの取りこぼしと再送が性能低下の原因となるのに対し、pull-based ではこのような問題を回避できるとの報告もある [5]。

4 p1 の実装

p1 の処理系は単一 CPU 版が Unix 上の C 言語へのトランスレータとして実現されている (現在、共有メモリマルチプロセサ版と分散版を設計中である)。この処理系では、p1 のソースコードを C に変換して翻訳し、実行時ライブラリとリンクして実行形式とする。実行時ライブラリには組み込みクラス群のコードとスケジューラが含まれる。

p1 の特徴として、ポートメソッドがその実行途中で休止状態になることはない。このため、各メソッドをそれぞれ C の 1 つの関数に翻訳し、そのまま普通に実行させることができる (手続きメソッドも C 言語の関数に翻訳する)。link 文については、関与するオブジェクトと発火条件等をリンク構造と呼ばれるデータ構造に格納してキューに投入する。

リンク構造 (図 4) は、各ポートメソッドごとにメソッドへのポインタと引数列の情報を保持し、この他に出力変数と配線者が渡す値の領域、および同期指定のあるオブジェクトと同期マスクの領域が用意されている。スケジューラはキューを順次スキャンし、同期条件が満足されているリンクを「発火」させる (つまり各ポートメソッドを引数つきで順次呼び出す)。

現在の実装では単一スレッドのみを使用しているが、p1 ではオブジェクトの状態を参照するのはメッセージ送信条件としてのみなので、並列実装においてもスケジューラスレッドが 1 つであるか、または複数のスケジューラスレッドを用いる場合でもそれぞれが参照するオブジェクト群が排他的

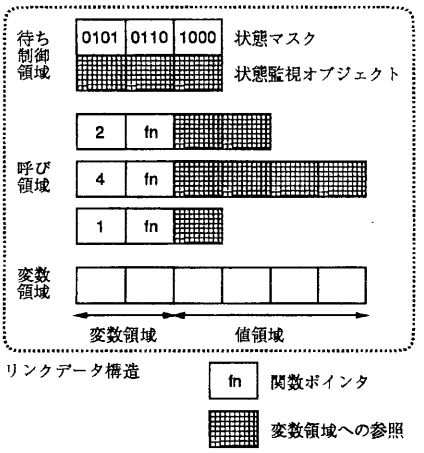

図 4: メッセージデータ構造

であれば、発火できるメソッドを探索する際に毎回オブジェクトをロックする必要はなく、効率よい実現が可能である。このため、1つのスケジューラスレッドを1つのドメインに対応させ、各オブジェクトもそれぞれどれか1つのドメインに所属させ、関係するオブジェクトがすべて同じドメインに所属しているようなメッセージデータ構造はロックを行わずに探索できる実装を計画している。

5 議論とまとめ

以上で対称型メッセージの基本的な考え方とその実装について述べたが、本節ではその拡張の可能性について論じる。実は現在の実装でも link 文に書く呼び指定の個数は2でなくてもよい。これにより、

- ある状態の成立をトリガーとしてメソッドを起動させる (呼びの個数が1の場合)
- 3つ以上のオブジェクトのメソッドを関連づけて起動させる (呼びの個数が3以上の場合)

といった記述が可能となる。また、すぐに考え付く拡張として、すべてのポートメソッドの起動条件の and だけでなく、or による指定 (いずれか条件の成立した配線のみが発火し残りは捨てられる) や、and と or の組合せを可能にすることも考えられる。

本稿では、通常の並行オブジェクト指向言語のメッセージ送信に見られる非対称性に着目し、送信側と受信側の役割りがほぼ対等であるような対称型メッセージ送信とその特徴 / 利点について論じるとともに、対称型メッセージ送信に基づく言語 p1 とその処理系の設計について説明した。今後は並列 / 分散版の p1 処理系を開発しその性能評価を行うとともに、プログラミング経験を積むことによって対称型メッセージを活かすプログラミングスタイルやフレームワークについて調べて行きたい。

参考文献

[1] 久野: 状態抽象: モジュールのもう1つの可能性, 情報処理学会研究会報告, 93-RPG-14-4, 1993.

[2] 久野、大木: 抽象状態に基づく並列オブジェクト指向 p6, 投稿中.

[3] 鵜林、大木、久野: オブジェクト間の協調動作を表現する並列計算モデルと言語, 電子情報通信学会論文誌 D-I, vol. J79-D-I, no. 10 掲載予定.

[4] Robin Milner: Communication and Concurrency, Prentice Hall, 260p, 1989.

[5] Andreaw Chien: ICC++ — A Parallel C++ for Irregular Applications, IWPC++ '96, `http://www-csag.cs.uiuc.edu/talks/IWPC++.ps`, 1996.

[6] Chris Tomlinson, Vineet Singh: Inheritance and Synchronization with Enabled-Sets, Proc. OOPSLA'89, pp. 103-112, 1989.

[7] Oscar Niestraz, Regular Types for Active Objects, Proc. OOPSLA'93, pp. 1-15, 1993.