抽象状態に基づく並列オブジェクト指向言語 p6

久野 靖† 大木敦雄†

抽象状態とは，データ抽象機能を持つ言語において，内部状態を抽象化した形で外部に公開するものである．本稿では並列オブジェクト指向言語に抽象状態に基づく同期機構を取り入れることを提案し，その一例として開発した並列オブジェクト指向言語 p6 の設計と実装について説明している．抽象状態に基づく同期では，状態情報がオブジェクトのインタフェースの一環をなすため，従来のガードや wait/signal などに基づく同期機構を持つ言語と比較して，選択送信や複数オブジェクトに関わる同期が自然な形で記述でき，記述性も良い．また，p6 の共有メモリ並列マシンにおける実装と評価により，このような言語の効率的な実現が可能であることを示した．

p6: A State Abstraction-based Parallel Object-oriented Language

YASUKI KUNO† and ATSUO OHKI†

"Abstract state" is a programming language facility that makes the internal state information of abstract data types available from outside, in a controlled fashion. In this paper, we propose an abstract state-based synchronization mechanism aimed at parallel object-oriented programming languages. We also describe a design and implementation of a parallel object-oriented programming language "p6" that includes abstract state-based synchronization. Compared with traditional synchronization mechanism (such as guard expressions or wait/signal), abstract state-based synchronization is superior in that (1) it is more natural and comprehensive, (2) multi-object synchronization can be expressed, and (3) straightforward and efficient implementation is possible.

1. はじめに

カプセル化による情報隠蔽はオブジェクト指向プログラミング言語の有用な性質であるが，オブジェクトの内部状態に関する情報を外部に見せることが望ましい場合も存在する．筆者らはこれを系統的に行う言語機構として抽象状態の考え方を提唱している[1]．本稿では並列言語への抽象状態機構の適用とその実装について述べる．以下 2 章では抽象状態の考え方について概略を説明し，3 章でその並列オブジェクト指向言語への適用について検討する．続く 4 章では抽象状態に基づく並列オブジェクト指向言語 p6 の設計について述べ，5 章でその実装について説明する．最後に 6 章で議論とまとめを行う．

2. オブジェクトと抽象状態

2.1 情報隠蔽の得失

オブジェクト指向プログラミング言語では，個々のオブジェクトの状態は実体変数（状態変数）に格納され，外部からはアクセスできないように保護される（カプセル化）．そして，各オブジェクトのインタフェースとしてはそのオブジェクトがどのような操作（メソッド）を持つかという情報だけが公開される．これにより，オブジェクトの内部実現を自由に設計・変更でき（情報隠蔽），同一インタフェースを持つオブジェクトであれば互いの差異を気にせず統一的に操作できる（多相性）．

しかし実際には，つねにすべての操作が任意の時点で呼び出せるわけではない．たとえば「スタック」オブジェクトであれば空のとき pop 操作は実行できないし，「有限バッファ」オブジェクトに対して満杯のとき put 操作は実行できない．すなわち，操作実行の可否はオブジェクトの状態に依存している．従来のオブジェクト指向言語では，プログラマはこのような状況に対し，次のいずれかの方策で対処してきた．

- 各操作を適用できない状態で呼び出すと例外や戻り値によって失敗を通知するようにプログラムする．
- 状態を観測するための操作を別途設ける．

† 筑波大学大学院経営システム科学専攻
Graduate School of Systems Management, The University of Tsukuba, Tokyo

前者の方策は，オブジェクトの状態について「操作が成功するか失敗するか」だけを問題にするときにしか適用できず，「バッファが full のときだけ get する」といった記述は行えない☆．後者の方策はそのような問題はないが，並列言語の場合には観測時点と呼び出し時点で状態が変化する可能性があり，そのままでは適用できない．

2.2 抽象状態

筆者らは，これらの問題をよりスマートに解決するために，オブジェクトの状態を外部に見せるための統一的な機構を導入することが望ましいと考えた．

しかし，すべての内部状態を公開することは，上であげたオブジェクト指向の利点を失わせるものであり，容認できない．そこで，各オブジェクトごとに外部から観測できる状態を（Pascal 言語の列挙型のように）宣言し，内部状態と外部から観測できる状態の対応関係はオブジェクトの各操作によって管理する，という枠組みを考案し「状態抽象」（state abstraction）と名付けた．

ここで「外部から観測できる状態」（抽象状態，abstract state）は，オブジェクトの内部状態そのものではなく，内部状態を抽象化したものとなっている．たとえば有限バッファオブジェクトを考え，その抽象状態は

`{empty, mid, full}`

のいずれかであると定めた場合，これらの状態は「有限バッファ」という抽象データ型の性質に自然に対応するものであり，内部のデータ構造からは独立している．

一方，たとえば抽象状態 `{full}` は，内部表現として環状バッファを使用した場合は出力ポインタが入力ポインタに追い付いた状態に対応するし，内部表現として「頭」つき環状リストを用いた場合は「頭」の次の要素が頭自身である状態に対応するが，このような内部状態から抽象状態への写像はプログラマがオブジェクト操作の一環として制御する．

このように，抽象状態はオブジェクトの内部表現からは独立しており，したがって抽象状態を公開することはデータ抽象の特質である局所性や再利用性を損うものではないと考える．

なお，このような機構の代替として，実行可能な操作の集合（受理集合，enabled sets）を外部に見せることも考えられるが，受理集合では操作の実行可能性の観点からは同一であるような状態は区別できない（たとえば有限バッファの抽象状態として `{empty, midlow, midhigh, full}` を考え，半分以上満ちているかどうかを区別する，といったことはできない）ため，抽象状態の方がより適用範囲が広いといえる．受理集合についてはもともとは並列言語の同期機構として提案[8]されているが，この点については 6.2 節で述べる．直列言語における状態抽象の適用については，文献 1) を参照されたい．

また見方を変えれば，プログラマは受理集合では内部状態から実行可能な操作の集合への写像を直接制御するが，提案方式の場合は内部状態から有限個の宣言された状態（抽象状態）への写像を制御し，各操作の実行可否はこれらの状態に基づき別途指定する．このため，提案方式の方が記述の自由度が高く，より自然な記述が可能になるというのが筆者らの考えである．

なお，従来の言語でも外部から状態を参照する必要がある場合にはプログラマはそのための操作を提供していたと考えられるので，抽象状態の導入により余分に情報が公開されるとは必ずしもいえない．すなわち，抽象状態ではこれまで記述できなかったことを記述するわけではなく，暗黙のうちに記述していたことを明示的に/分かりやすく書けるようにしていると考える．その結果，言語機構が増えることによる繁雑さはあるが，状態について分離して記述させることで汎用性/再利用性の高い設計に到達しやすくなると考える．

3. 並列オブジェクト指向言語と抽象状態

3.1 並列オブジェクト指向言語と同期

並列プログラミングでは通常，複数の実行単位間で同期をとる機構が必要である．同期には無条件同期と条件同期の 2 種類が存在する．

無条件同期（排他同期）では，排他領域（critical region）内に一時に 2 つ以上の実行単位が入らないように，実行中の実行単位が領域から出るまで，領域に入ろうとする 2 番目以降の実行単位の実行を遅延させる．

一方，条件同期とは，排他領域によって保護されるデータ構造の値に依存して，実行単位の実行を遅延させる（たとえばバッファが空のときの get はバッファが空でなくなるまで遅延させる）ための機構である．

並行プログラミングの分野では，これらの同期を実現するために，ロック，セマフォ，モニタ，ガードなど多くの機構が提案されてきた．本稿では並列オブジェクト指向言語の場合を中心に取り上げる．

純粋なアクターモデルは，無条件同期のみをサポートし，排他実行の単位は 1 つのアクターの操作である

☆ 「満杯でないときだけ成功する get 操作」を別に用意すれば可能だが，スマートでない．

(操作の起動要求—メッセージ—は到着順に処理される)．これはモデルとしては簡潔だが，条件同期は複数のアクターを組み合わせて実現しなければならず，記述力の点で不満がある．

このため並列オブジェクト指向言語の多くは，ガード，受理集合，シンクロナイザ，(Hoare のモニタと同様の) wait と signal，明示的な受信，メタメソッドなど，様々な機構を導入してメッセージの選択的処理を可能にしている．これらはいずれもメッセージの受信側において実行を遅延させるもので，遅延の有無は送信側からは観測できず，送信側でメッセージの受理可能いかんに応じて処理を変化させることもできない．

3.2 抽象状態同期

これに対し筆者らは，並列言語に抽象状態に基づいた同期機構を取り入れることで，無条件同期と条件同期を統一的に扱えるのではないかと考え，これを「抽象状態同期」と名付けた．その基本的なアイデアは次のようなものである．

- 操作入口において，各引数ごとに実行可能な抽象状態の集合を指定でき，指定した場合には引数の状態がこれに適合しないときは適合するまで操作の起動が遅延させられる（状態を指定しない引数はこの動作に影響しない)．
- 状態を指定した引数のうち，その操作が所属するクラスの型を持つものについては☆操作起動時に抽象状態が「不定」となり，操作内部で状態を設定するまで「不定」状態にとどまる．

これにより，条件同期の条件と抽象状態（の集合）を対応させることで自然な条件同期記述が可能となり，また抽象状態値が「不定」の間はどの状態ともマッチしないので排他同期も実現できることになる．

以下では具体的なコード例のために p6 言語を使用する（p6 言語とその構文については 4 章で解説し，完全な例もそこに示す)．たとえば，有限バッファを表すクラス `bbuf` の操作 `get` の頭書きは次のようになる (`replies(...)` は返値の型を規定する宣言である)．

```
get = method({full,mid}b:bbuf) replies(int)
```

これを呼び出すコードはたとえば次のようになる．

```
i:int := bbuf!get(b1)
```

この呼び出しは，有限バッファ `b1` が {`empty`} の場合にはそうでなくなるまで待たされることになる．さらに，呼び側で条件をより強くすることも可能であり，

```
i:int := bbuf!get({full}b1)
```

とすると，この呼び出しは `b1` が満杯になるまで待たされる（消費より生産の方が速くてバッファが満杯になったときだけ何らかの動作をさせる場合に役立つ)．

また，Ada の select 構文に類似した構文を導入することで選択的送信（受信オブジェクトの状態によって送り先を選択すること）も行える．たとえば

```
i:int := select bbuf!get(b1) or
                bbuf!get(b2) end
```

により，バッファ `b1`，`b2` のいずれか空でない方から値を取り出す（両方空であればいずれかが空でなくなるまで待つ）ことができる．

さらに，操作の頭書きにおいて 2 つ以上のオブジェクトの状態を指定することで，複数オブジェクトに関わる同期が自然な形で記述できる．たとえば，

```
pick = method({down}f1:fork, {down}f2:fork)
```

とすれば，2 つのフォーク `f1`，`f2` がともに `down` という状態にある場合にのみ操作 `pick` が受理される．

抽象状態の考え方は，Milner のプロセス計算[10]と類似している．プロセス計算においては，複数のエージェントがポートを通じて相互に通信しながら計算が進んでいく．各エージェントは複数の状態を持ち，イベント（=送受信または内部動作）が起きるごとに次の状態へ移行していく．

ここで状態と呼んでいるものは，あくまでも外部から観測可能な（エージェントの挙動の差異と対応づけられる）ものであり，エージェントの内部が（たとえば複数の子エージェントから構成されるなどして）複雑な状態を持つとしても，外部から観測可能でない場合はそれらの内部状態は外には現れない．したがって，プロセス計算における「状態」を本稿の「抽象状態」に対応させて考えるのは自然である．

上の説明だけでは一見,「起動可能な操作の集合」(受理集合) と「状態」が対応するかのように思われるが，そうではない．すなわち，2 つのエージェント式において，生起可能なイベントの集合は同一であるが，式の内部が異なるため，イベントのどれかにより遷移した後では生起可能なイベントが異なってくる場合があり，その場合には 2 つの式は別の状態であるからである．このような遷移列に基づくオブジェクトの等価性や互換性について定式化する試みとして文献 13) などがあげられる．

4. 状態抽象機能を持つ並列言語 p6

4.1 p6 の基本設計

前章に述べた考えに基づき，筆者らは抽象状態同期に基づく並列言語 p6 を設計した．その方針は次のと

☆ この条件がついているのは，あるクラスに所属する操作の中で別のクラスのオブジェクトの状態を設定することは（データ抽象の考えに反するため）許していないからである．

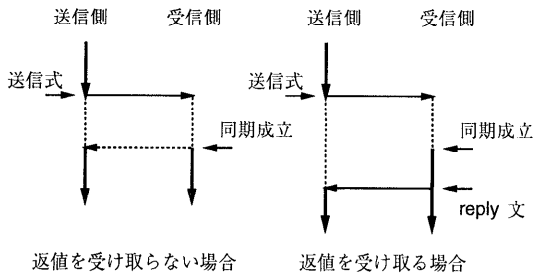


図1 p6のメッセージ送信
Fig. 1 p6's message model.

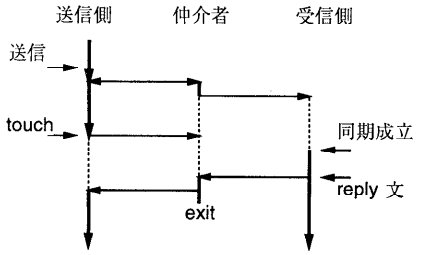


図2 future メッセージの変換
Fig. 2 Translation of future messages.

おり.

a. クラス方式の並行オブジェクト指向言語とした. これはクラス記述に抽象状態の宣言を含めるのが自然であり, また将来継承を導入して状態の分割などを表現したいと考えたことによる.
b. 当面は抽象状態同期の実用性を確認するための最小限の言語仕様とし, 動的配分, 継承, 型パラメータ, 分割翻訳などは省いた.
c. メッセージ送信の意味づけを1種類に統一し, 多様な送信機構は基本的なメッセージの組合せで表せるようにした.

p6ではメッセージ送信はメッセージ送信演算子「!」を使用した次の構文により表現する☆.

$X!method(arg,\cdots)$

var := $X!method(arg,\cdots)$

上は返値を受け取らない場合, 下は受け取る場合を表す. arg は式または式の前に状態集合を付加したもの, X は型(クラス)名または式である. X が式の場合, その型を $typeof[X]$ と表記すると☆☆, 上記の式はそれぞれ次のものと同等となる.

$typeof[X]!method(X,\ arg,\cdots)$

var := $typeof[X]!method(X,\ arg,\cdots)$

p6ではすべての操作はクラスに所属し, 実行中に「カレントオブジェクト」がselfのような擬変数に束縛されているという意味でのインスタンスメソッドはない. ただし, 上の書換え規則により, 第1引数の型が所属するクラスの型であるようなメソッドは呼び側から見ればインスタンスメソッドと同じに扱える.

p6ではプロセス計算と同様, 送信は同期的である. すなわち, 返値を受け取らない送信でも, 送信式の実行が完了した時点で引数群に対する抽象状態条件が1度は成立したことが保証される(**図1** 左). 送信式に続く部分の実行中にその状態が保たれているかどうかは, 他のオブジェクトがそれらのオブジェクトの抽象状態を変更していないかどうかに依存する.

返値を受け取る送信では, 起動された操作内で**reply**文または**return**文が実行され, その値が返送されて初めて送信文が完了する(図1右). **reply**と**return**の違いは, **return**が操作の実行を終結させるのに対し, **reply**は返値を送った後で引き続く文を実行することである(この場合, 操作は本体の最後の文を実行し終わるか, または**exit**文を実行することで終了する).

非同期メッセージやfutureメッセージは間に仲介者オブジェクトを介在させることで実現できる(**図2**). 仲介者は渡された引数をそのまま送信先に中継するものであり, 自身は同期条件を持たないため送信側はただちに実行を継続できる. 仲介者オブジェクトは受信側で同期条件が成立し, コードが実行されて返値が来るまで待つ. 送信側がfutureメッセージの返値を必要とした場合には(touch), 仲介者に返値を要求するが, まだ返値が仲介者まで届いていない場合にはここで同期待ちが発生する(図の場合). すでに返値が仲介者まで届いていれば, 仲介者はただちに返値を転送できる.

4.2 例題1:有限バッファ

基本的な例題として有限バッファのクラスを示す.

```
aint = array[int]                                    %1
bbuf = class { full, empty, mid }                    %2
  slot a:aint, size:int, c:int,
       i:int, o:int                                  %3
  new = method(n:int) replies(bbuf{empty})           %4
    return(bbuf$[a:aint!new(n), size:n,
                 c:0, i:0, o:0]!{empty})             %5
  end new
  put = method({empty,mid}b:bbuf{mid,full}, %6
               i:int)
    b.i := (b.i + 1) // b.size                       %7
    b.c := b.c + 1
    b.a[b.i] := i
    if b.c = b.size then b!{full}
```

☆ なお, 演算子「!」は「$obj!\{$状態名$\}$」という形でオブジェクトの抽象状態を設定するのにも使用する.
☆☆ p6は強い型付けを持つため, すべての式は型が一意に定まる.

```
                    else b!{mid} end
  end put
  get = method({full,mid}b:bbuf{mid,empty}) %8
        replies(int)
    b.o := (b.o + 1) // b.size                %9
    b.c := b.c - 1
    v:int := b.a[b.o]
    if b.c = 0 then b!{empty} else b!{mid} end
    return(v)
  end get
end bbuf
```

1. aint を整数の配列を表す型名として定義する.
2. bbuf をクラス名兼型名として定義する. bbuf 型の値は 3 つの抽象状態のいずれかを持つ.
3. bbuf の内部表現は a, size, c, i, o の 5 つの状態変数を持つ implicit record である（これに加え状態格納や同期のための隠れたスロットも用意される）. 各スロットは配列, 大きさ, 格納数, 入力指標, 出力指標に対応している.
4. new 操作は整数値を 1 個受け取り, bbuf の値を返す. 返される bbuf の状態は empty である.
5. 具体的な処理としては, 内部レコードを生成し状態を empty に設定して返す.
6. put 操作は第 1 引数の bbuf の状態が emtpy か mid のときに実行開始し（そうでないときは待たされる）, 終了時までにその状態を mid か full にすることが期待される☆.
7. 処理としては, 入力指標を進めてから指標の指す場所に値を格納し, 格納数を増やしてそれが size と等しくなったかどうかに応じて状態を full か mid に設定する.
8. get 操作は第 1 引数の bbuf の状態が mid か full のときに実行開始し, 終了時までにその状態を mid か empty にすることが期待される.
9. 処理としては, 出力指標を進めてから指標の指す場所の値を取り出し, 格納数を減らしてそれが 0 になったかどうかに応じて状態を mid か empty に設定する.

次に有限バッファを通じて転送を行う例を示す. 指定された個数のデータを生成/消費する操作を持つクラス worker を示す.

```
worker = class
  produce = method(n:int, b:bbuf)
    i:int := 1
    while i < n do b!put(i); i := i + 1 end
    b!put(0)
  end produce
  consume = method(b:bbuf)
    while b!get() ~= 0 do end
  end consume
end worker
```

実行はクラス main の操作 startup から開始される：

```
main = class
  startup = method()
    b:bbuf := bbuf!new(100)
    worker!produce(10000, b)
    worker!consume(b)
  end startup
end main
```

4.3 例題 2：リーダライタ問題

もう少し複雑な制御を必要とする例題として, リーダライタ問題を取り上げる. 書き手のスタベーションを抑止するため, 書き手が要求を出すと以後新しい読み手は鍵の取得を待たされる.

```
lock = class { free, reading, rtow, writing }
  slot r:int, w:bool
  new = method() replies(lock{free})
    reply(lock$[r:0, w:false]!{free})
  end new
  readlock = method({free,reading}r:{reading})
    r.r := r.r + 1; r!{reading}
  end readlock
  readrelease = method({reading,rtow}r:
                       lock{reading,rtow,free})
    r.r := r.r - 1
    if r.r = 0 then r!{free}
    elif r.w then r!{rtow}
    else r!{reading} end
  end readrelease
  writelock = method({free,reading,rtow}r:
                                lock{writing})
    if r.r = 0 then
      r!{writing}
    else
      r.w := true; r!{rtow}; r!waittowrite()
    end
  end writelock
  waittowrite = method({free}r:lock{writing})
    r!{writing}
  end waittowrite
  writerelease = method({writing}r:lock{free})
    r.w := false; r!{free}
  end writerelease
end lock
```

ロックは free, reading, writing, rtow の 4 状態を持ち, free で始まる. free のときは readlock, writelock がともに呼び出せ, それぞれロックを reading と writing の状態に移行させる. reading のときはさらに readlock を呼べるが, writing のときは writelock は待たされる. writerelease では（書き手は 1 人だから）ただちにロックは free になるが, readrelease では読み手が全部解放したときにはじめて free になる. さらに, reading で writelock が呼ばれると状態 rtow に移行し, その後内部的に

☆ この情報と返値の状態の情報は現在の処理系では参照しておらず, プログラムの読み手のための情報としてのみ役立つ.

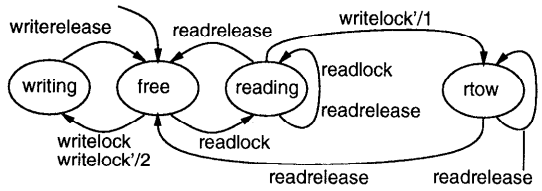


図3 抽象状態遷移図
Fig. 3 An abstract state-transition diagram.

waittowrite を呼ぶことで **free** になるのを待つ（抽象状態の遷移図を**図3** に示した）．

この解では読み手がすべて **readrelease** を実行してロックが **free** になった時点で改めて読み手と書き手がロックを争うので，次が **writing** になる保証はないが，必ず次が書き手の番になるよう直すのも容易である（紙面の都合で省略した）．このように，状態遷移が複雑になった場合にも抽象状態を明示的に記述することで明解なコードが書ける．

4.4 例題3：哲学者の食事

次に哲学者の食事問題を示す．まず，フォークは状態だけが重要なのでスロットを持たないクラスとなる．

```
fork = class { up, down }
  new = method(n:int) replies(fork{down})
    reply(fork$[]!{down})
  end new
  pick = method({down}f1:fork{up},
                {down}f2:fork{up})
    f1!{up}; f2!{up}
  end pick
  release = method({up}f:fork{down})
    f!{down}
  end release
end fork
```

ここで pick は2つのフォークが down の時のみ実行され，両方を同時に up にする．哲学者は次のとおり．

```
phil = class
  life = method(n:int, f1:fork, f2:fork)
    while true do
      % 考える
      fork!pick(f1, f2)
      % 食べる
      f1!release(); f2!release()
    end
  end life
end phil
```

このように，複数オブジェクトに関わる同期を用いれば「同時に取り上げる」という自然な記述が可能になる．

5. p6の実装

5.1 p6/SPARCの基本構成

抽象状態同期を持つ並列オブジェクト指向言語の効率的な実装が可能であることを示すため，共有メモリマルチプロセッサシステムの SparcServer/1000（50 MHz SuperSparc+ × 4 CPU）上で稼働する p6 処理系 p6/SPARC を開発した．その概要を**図4** に示す．すなわち，p6 のソースコードはコンパイラによってSPARCアセンブリコードに翻訳され，これをアセンブルしたものと実行時ライブラリを結合して実行形式ファイルを作成する．

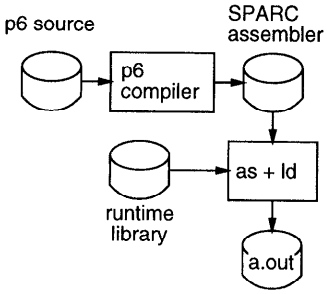


図4 p6/SPARC処理系の構成図
Fig. 4 The structure of p6/SPARC system.

実行時ライブラリには，標準クラスの操作で直接ソース中に展開されないもの，およびスケジューラ等のコードなどが含まれている．実行時ライブラリはCで記述され，Solaris2.3 の標準スレッドライブラリとともに実行形式に組み込まれる．

並列言語の実装については多くの前例があるので，次節以降では抽象状態に固有の事項を中心に説明する．p6/SPARC の設計は共有メモリマルチプロセッサを前提としているが，それ以外の点については特定のシステムアーキテクチャ（命令セット等）とは独立である．

5.2 抽象状態のビット表現と判定

抽象状態の数はプログラマがコード上で列挙するため，さほど多くはないと考えられる．そこで，抽象状態を状態数ぶんのビット列で表し，その中の「現在の状態」に対応するビットのみを1とする☆．これにより，同期条件を各状態の論理和をとったビットマスク（同期マスクと呼ぶ）で表現し，ビット単位の and 演算で現在の状態が集合に含まれているかどうかを効率的に判定できる．1つの呼び出しにつき，この判定を状態指定のある引数の個数ぶん行えばよい．

ただし，この判定は排他的に行う必要がある．排他性を保証する方法として，次の2つが考えられる．

- 複数の実行主体が判定の対象をロックして排他的

☆ この割当てはコンパイラにより一意的に行われる．コンパイラはアセンブラの記号定義を生成しているので，他の箇所からは記号名を用いて状態値を参照することができる．

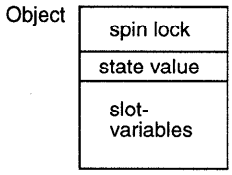


図5 p6/SPARC のオブジェクトの構造
Fig. 5 The object structure of p6/SPARC.

にアクセスする.

- 判定を行う実行主体を限定して競合を避ける.

前者の方法は，オブジェクト間のアクセス競合が激しくない場合には，各コードの呼び出し地点で判定を行って直接呼び出せるので効率が良い．その代わり，アクセス競合がある場合には繰り返しロックを獲得しようとするため効率が悪い．一方，判定を行う実行主体を「スケジューラスレッド」のようなものに限定する場合には，各実行主体は自分の待ち合わせたい条件をいったんデータ構造に格納してスケジューラに渡すのでオーバヘッドは大きくなるが，競合の問題は解消される.

以上の考えから，p6/SPARC ではオブジェクトの先頭部分にスピンロック領域と抽象状態語を起き（図5），次の手順で同期処理を行うこととした.

- 呼び出し地点において各引数を順にロックしながら状態を調べていく．このとき，デッドロックを避けるため，ロックする引数群を論理アドレスの昇順に整列し，アドレスの若いものから順にロックする.
- すべての引数がロックでき，状態も適合した場合は，操作を手続き呼び出し命令で直接呼び出す.
- どれか1つでも状態が適合しないかロックが獲得できなかった場合は，かけたロックをすべて解放し，コンテキストと呼び出し情報をメモリに退避して実行を中断する.

これにより，他のオブジェクトとの競合がなく遅延のともなわない操作起動は，スピンロックと通常の手続き呼び出しにより低オーバヘッドで行える．そして，競合や待ち合わせが発生した場合にはキューに入ってスケジューラにより順次処理されていくため，繰り返しロックを獲得するオーバヘッドが低減できる.

5.3 フレームとスケジューラ

p6/SPARC では，RISC の特徴を活かすため，操作実行時には引数や変数はレジスタ上に置かれている（表1にレジスタ用途を示した）．レジスタの退避/回復は SPARC のレジスタウィンドウ機構に委ねているが，このことは特に本質的ではない.

表1 p6/SPARC のレジスタ用途
Table 1 The register usage of p6/SPARC.

regs.	usage
g0～g6	scratch regs.
g7	resv. for system
o0～o5	call params.
o0	ret. value
o1	ret. status
o6	stack pointer
o7	link register
l0	frame register
l1～l7	local vars./state masks
i0～i5	args. + local vars.
i6	frame pointer
i7	ret. addr.

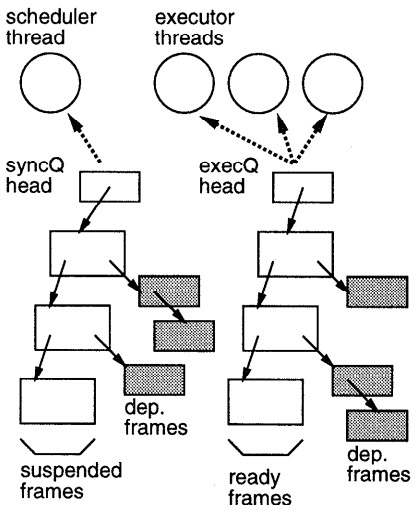


図6 p6/SPARC の実行時フレームワーク
Fig. 6 The runtime frame structure of p6/SPARC.

前節で述べたように，遅延をともなわない呼び出しは手続き呼び出し命令により直接実行するが，遅延が必要な場合はレジスタをヒープ上のメモリブロック（「フレーム」と呼んでいる）に退避し，キューに格納する（図6）．キューには同期キュー（syncQ）と実行キュー（execQ）の2つがある．同期キューは状態同期のため待ち状態にあるフレームを含み，実行キューは実行可能状態のフレームを含む.

実行系には1つのスケジューラスレッドと1つ以上のエグゼキュータスレッドが含まれる．スケジューラスレッドは同期キューを適当なタイミングで走査し☆，同期条件が満たされたフレームを実行キューに移す．エグゼキュータスレッドは実行キューから実行可能なフレームを取り出して状態をレジスタにロードし実行

☆ 現在の実装では同期に関係しているオブジェクトの状態が変化するごとに全キューをスキャンしている.

させることを繰り返す．エグゼキュータスレッドの数はあらかじめ設定されている定数を超えない範囲で，実行可能なフレーム数に応じて増減する．

各フレームには，そのフレームの実行開始を待つフレーム（開始待ちフレーム）や，そのフレームに対応する実行の返値を待つフレーム（返事待ちフレーム）が付属していることがある（開始待ちフレームは抽象状態同期の実行系に固有のものだといえる）．これらを依存フレームと呼ぶ．開始待ちフレームは，元のフレームが同期キューから実行キューに移るとき同時に実行キューに入れられ☆1，返事待ちフレームは元のフレームが値を返したときに実行キューに入れられる．

5.4 送信/中断/返信の実装

抽象状態同期の基本的な設計は前節までで述べたとおりだが，本節では具体的な生成コードの概要について説明する．メッセージ送信を可能な場合は手続き呼び出しで実現する点はStackThreads[7]と同様であるが，抽象状態同期および共有メモリマルチプロセッサのための競合制御が含まれている点が大きな違いである．まず，送信側は次のとおり．

(1) 引数レジスタ（o0～o5）に引数を格納する．
(2) 抽象状態同期をともなう引数の同期マスクを対応するレジスタ l1～l6 に格納する☆2．
(3) 手続き呼び出し命令で操作を直接呼び出す．
(4) 呼び出しから戻ってきたときの状態レジスタ（o1）の値により，次の3通りの処理に分岐する．
(3a) o1 = 0：通常の戻り．返値があれば o0 に入っている．ふつうに実行を続行する．
(3b) o1 = −1：同期条件が満たされないため中断した．o0 には中断したフレームのアドレスが入っている．返値を参照するかどうかに応じて自分のフレームをそのフレームの開始待ちフレームまたは返事待ちフレームとして登録し，自分のフレームをロックしてから中断したフレームを同期キューに入れ☆3，自分も中断する（フレームにレジスタや再開番地等を格納し，o0 にフレームアドレス，o1 = 1 を入れて戻る）．
(3c) o1 = 1：呼んだ操作は実行を開始した後に中断した．o0 には中断したフレームのアドレスが入っていて，なおかつそのフレームにはロックがかかっている．返値を参照しない場合は，そのフレームをアンロックして実行を続行する．返値を参照する場合は，自分のフレームをそのフレームの返事待ちフレームとして登録し，自分のフレームをロック，受け取ったフレームをアンロックし，自分も中断する．

受信側操作の先頭における処理は次のとおり．

(1) 状態同期を必要としない（引数に状態指定のない）操作は特に何もする必要がない．
(2) そうでない場合は，状態指定を持つ引数と対応する（呼び側が渡した）同期マスクをペアにして，ペアを保ったままアドレス昇順に整列する☆4．
(3) 各ペアについて引数に順次ロックをかけながら状態がマッチしているかチェックする．

これ以後は次の2つの場合に分かれる．

(4a) すべての状態がマッチしていた場合には，すべてのロックを解除して操作の実行を続行する．
(4b) マッチしない状態があった場合には，それまでにかけたロックを解除し，o0 に自分のフレームアドレス，o1 に −1 を入れて戻る．

最後に，受信側での返信は reply と return がある．return は o0 に返値，o1 に 0 を入れて戻ればよい．reply は戻る前に自分のフレームを実行キューに投入してから return と同様に動作する（したがって送信側の処理がつねに優先される）．

5.5 性能評価

高速なメッセージ送信という目標の達成度評価のため，p6/SPARC における基本操作の所要時間を測定した．測定は引数が1個で本体が空の操作をループ内で繰り返し呼び出すことで行った．結果を**表2**に示す．これによれば，同期をともなわない送信は手続き

表2 p6/SPARC における基本操作の所要時間
Table 2 Run-time overhead of parallel execution primitives.

操作	時間
同期をともなわない送信	0.2 μsec
同期をともなう送信	0.6 μsec
（抽象状態の変更）	2.2 μsec
中断処理	28.2 μsec
スケジューリング	14.7 μsec
再開処理	21.3 μsec

☆1 開始待ちフレームにさらに開始待ちフレームが付随していることもある．これらもすべて一緒に実行キューに入れられる．
☆2 このため，抽象状態同期をともなう操作呼び出しでは呼び側でレジスタウィンドウを回す save 命令を実行している．これはレジスタウィンドウアーキテクチャでない場合にはローカルレジスタを保存してから同期マスクをロードすることに相当する．
☆3 自分のフレームをロックするのは，中断したフレームをキューに入れると以後はいつでもスケジューラとアクセスが競合する可能性があるため．
☆4 状態指定を持つ引数の数はせいぜい2～3なのでその数を N として $\frac{N(N-1)}{2}$ 回の比較交換を行う命令列を生成している．

呼び出し命令となるため 0.2 μsec と非常に高速に実行できることが分かる．引数に同期条件がある場合にも同期条件のない場合の 3 倍程度のオーバヘッドがかかるが，絶対的な所要時間としては十分高速である．ただし，同期条件がある場合には通常，操作内部で必ず引数の状態を設定することになるが，そのためのオーバヘッドが 2.2 μsec とかなり大きい．これはオブジェクトの状態が変化した場合にスケジューラスレッドを起こすためで，この部分の実装は再考の余地がある．同期条件が不成立で操作の中断が起きる場合には，中断/スケジューラによるスキャン/再開処理とも 10〜30 μsec 程度を要している．この部分のオーバヘッド低減についても今後の課題である．

5.6 その他の機能の実現

本節では p6/SPARC において現在未実装の機能について構想を述べる．3.2 節であげた選択的送信は，次の方法で実装できる．

- 送信側では select の各枝の送信式を順に実行し，返されてきた同期待ちフレーム群を集める．
- 1 つでも実行が開始できれば，それまでに集めたフレームを捨てて開始した操作の実行を選択する．
- すべての操作が同期待ちであれば，集めたフレームのリストをスケジューラに渡して中断する．
- スケジューラはリストになったフレームについては，どれか 1 つが実行可能になった時点で残りのフレームを捨て，送信側フレームにどの枝が選ばれたかの情報を渡して再開させる．

また，メッセージ送信を手続き呼び出し命令によって代替した場合，受信側の実行が長時間にわたる場合，本来は待たなくても済む送信側の処理が待たされるという問題がある☆．これに対処するには，次の方策が考えられる．

- 返事待ちでない送信を手続き呼び出しで代替するときはあわせてタイマーを起動する．
- タイマー割込みが起きた場合には，実行中の操作を強制的に中断する．
- タイマー割込みによる中断フレームはただちに実行キューに入れる．

これにより，タイマー割込み以後は呼び側と呼ばれ側が並行に実行できることになる．

6. 議　　論

6.1 抽象状態とソフトウェア設計

ソフトウェア設計技法や仕様記述などの分野では状態を抽象化して扱うことは広く行われている．しかし，これらは設計上の概念であり，コードのうえでこれに直接対応するものは必ずしも作成されない．これに対し，本稿で提案しているのはコード上に抽象状態を明示的に記述し参照することである．これにより，設計との対応関係が明確になるという利点がある．

また，代数的仕様記述を取り入れたプログラミングシステムでは抽象データ型の仕様を代数的に記述し，実際のコードとの対応関係を証明系により保証する．この場合は本稿の提案よりも厳密な状態が把握できるが，実用規模のシステムに広く使われている状況ではない．

これに対し，本稿で提案している方法では抽象状態の数は有限個であり，実際の状態と抽象状態の対応関係はプログラマによって管理される．その意味ではプログラマの負担は増すが，抽象データ型の利用者は抽象状態のみを考えればよい．一方，実現者は抽象データ型の外部仕様である抽象状態と内部状態の整合性を保つようプログラミングする必要があるが，これはもともと（明示的には意識されなくても）実現者が行っていた作業に含まれていたと考える．

6.2 一般の並列言語との比較

同期機構としての抽象状態について考えると，本稿の方式は実行の遅延が起きるのが操作入口であることから，ガードと類似して見える．しかしリーダライタ問題の例にあるように，ある操作の途中で別の操作を呼ぶことで抽象状態を待ち合わせることが可能なので，ガードに比べて記述力は増している．

また，ガードの場合にはその条件式がオブジェクトの内部状態を直接参照するため，抽象データ型の利用者にとって条件を参照することはあまり役に立たない．これに対し，抽象状態はその抽象データ型の外部仕様の一部であり，利用者にとって有益な情報を含んでいる．

さらに，ガードは真偽を判定するために式の評価が必要であるため，スケジューラからこれを行う場合には環境の切替えなどの問題が生じる．これに対し，抽象状態同期は同期マスクとの and 演算のみで真偽が評価できるため，高速に実行でき（ハードウェアサポートも考えられる），副作用などの心配もない．

選択的送信のような機構は，Ada の select 文でも実現されているが，Ada では呼び出しが受理されるか否かは呼ばれ側の実行が対応する accept 文に到達したかどうかで定まる．これは，呼ばれ側の状態を「実行位置がどこにあるか」で表すことになるため，本稿の状態を明示的に扱う方式に比べて扱いにくいと思わ

☆ StackThreads でも同じ問題がある．

れる．

最後に，本稿の方式では複数の（必要なら異なる種類の）実体に関わる同期を直接記述できる．これにより，(哲学者の食事の例題に示されているように）多くの問題がより簡潔に記述できると考えられる．

6.3 関連研究

並行オブジェクト指向言語の分野では，コードの継承を通じて同期のためのコードを再利用するための機構について多くの研究がある．その中で受理集合（enabled set）を用いる方法[8),9)]は，各集合値を抽象状態に対応するものと考えれば本稿の方式と類似性が認められる．松岡[11)]らの方式は実際それらを状態として扱い，その間の遷移を記述することで同期の制約を記述している．

これらの研究では，オブジェクトの状態について記述するというよりは同期のために操作起動をON/OFFする継承可能な機構としての側面に注意が払われている．また，複数オブジェクトに関わる同期のようなものは考えられていない．

一方，ActorSpace[12)]モデルでは各並行オブジェクトに「属性」を与え，複数オブジェクトについて属性に関わる制約を含んだ同期を指定でき，この「属性」を「抽象状態」に対応させれば，本稿での方式と同様の記述が行える．しかし，この研究においてもオブジェクトの同期に関わる制約を外部的制約として分離記述することに主眼が置かれ，各オブジェクトの状態を抽象化したものというとらえかたはなされていない．

また，Regular Type[13)]は本稿で述べているような状態遷移に基づいて操作のON/OFFを定式化するものである．ただし言語としての実装については述べておらず，複数オブジェクトに関わる同期も扱っていない．

6.4 抽象状態同期と継承

並列オブジェクト指向言語に継承を導入した場合，同期を含むコードをサブクラスで継承し利用することが難しい，という問題（継承異常）が知られている[11)]．p6は継承機構を持たないが，筆者の1人はp6に継承を追加した言語の設計について検討し，抽象状態同期が継承異常に対処する有力な方策となり得ることを示した[2)]．

その基本原理は，サブクラスにおいては親クラスの抽象状態を複数の状態に分割することのみ許し，分割された状態についてはafter daemon☆と同様のメソッドを用いて状態値を修正するというものであり，これによって状態分割，履歴依存，混入という代表的な継承異常の原因をうまく扱うことができる．

これがうまくいくのは，オブジェクトの内部状態を参照することで（少なくとも通常は）あるべき抽象状態が決定でき，これをafter daemonによる微細なコードの追加として実行すれば済むからだといえる．より詳しくは文献2)を参照されたい．

6.5 まとめ

並列実行単位間の同期機構として，抽象状態に基づく同期機構を提案し，またこの同期機構に基づく並行オブジェクト指向言語p6とその上でのプログラミング例と経験について報告した．その結果，抽象状態に基づく同期機構は一般に用いられているガードなどに比べて記述力が大きく，記述も容易であるとの感触を得た．

上記の成果に基づき，現在は抽象状態と継承機構をあわせ持つ言語[2)]，抽象状態を土台とした新しいメッセージ機構に基づく言語[3),4)]，より大規模なシステムへの適用[5)]などについて研究を進めている段階である．

謝辞 東京大学理学部の田浦健次郎氏には，StackThreadsについて有益な情報をいただきました．また，情報処理学会論文誌の査読者（複数）のコメントは，本稿を改善するうえで大きな助けとなりました．ここに感謝します．

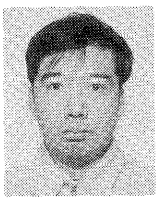
参考文献

1) 久野：状態抽象：モジュール化のもう1つの可能性，情報処理学会研究会報告，93-PRG-14-4 (1993).
2) 久野：抽象状態同期と継承，WOOC'96, http://www.rwcp.or.jp/people/ishikawa/wooc96/wooc96.html (1996).
3) 鵜林，久野，大木：対称型メッセージ送信とその実装，情報処理学会研究会報告，96-PRO-12, pp.67–72 (1996).
4) 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, pp.625–634 (1996).
5) 地引，芦原，山下，上田，大木，久野：分散仮想マシンを用いたオブジェクト指向プログラミング環境，情報処理学会研究会報告，96-PRO-10, pp.37–42 (1996).
6) 久野：多重継承と強い型付けを持つオブジェクト指向言語 Misty，コンピュータソフトウェア，Vol.6, No.3, pp.9–18 (1989).

☆ CLOSなどのオブジェクト指向言語で採用されている言語機構．サブクラスにおいて親クラスのメソッドを置き換えるのではなく，親クラスのメソッドの前や後に付加的に実行されるように指定したメソッドをdaemon method，そのうち「後」に実行されるものをafter daemonと呼ぶ．

7) Yonezawa, A., Matsuoka, S., Yasugi, M. and Taura, K.: Implementing Concurrent Object-Oriented Languages on Multicomputers, *IEEE Parallel and Distributed Technorogy*, Vol.1, No.2, pp.49–61 (1993).
8) Tomlinson, C. and Singh, V.: Inheritance and Synchronization with Enabled-sets, *Proc. OOPSLA'89*, pp.103–112 (1989).
9) Kafura, D.G., Lee, K.H.: Inheritance in Actor based Concurrent Object-oriented Languages, *Proc. ECOOP'89*, pp.131–145 (1989).
10) Milner, R.: *Communication and Concurrency*, p.260, Prentice Hall (1989).
11) S. Matusoka, K. Taura and A. Yonezawa: Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages, *Proc. OOPSLA'93*, pp.109–126 (1993).
12) Agha, G., Frølund, S., Kim, W.-Y., Panwar, R., Patterson, A. and Sturman, D.: Abstraction and Modularity Mechanisims for Concurrent Computing, *IEEE Parallel and Distributed Technology*, Vol.1, No.2, pp.3–14 (1993).
13) Niestraz, O.: Regular Types for Active Objects, *Proc. OOPSLA'93*, pp.1–15 (1993).

(平成7年12月12日受付)
(平成9年 1 月10日採録)



久野 靖（正会員）

1956年生．1981年東京工業大学大学院理工学研究科情報科学専攻修士課程修了．1984年同専攻博士後期課程単位取得退学．同年東京工業大学理学部情報科学科助手．筑波大学大学院経営システム科学専攻講師，同助教授を経て，現在同大学院企業科学専攻助教授．理学博士．プログラミング言語，プログラミング環境，オペレーティングシステム，ユーザインタフェース等に興味を持つ．著書に「言語プロセッサ」（丸善），「UNIXの基礎概念」（アスキー）等がある．日本ソフトウェア科学会，ACM，IEEE Computer Society各会員．

大木 敦雄（正会員）

1957年生．1983年筑波大学大学院理工学研究科修士課程修了．同年静岡大学工学部情報工学科助手．1989年3月筑波大学大学院経営システム科学専攻助手．プログラミング言語，プログラミング環境，オペレーティングシステム等に興味を持つ．著書に「Mule入門」（アスキー）等がある．日本ソフトウエア科学会会員．