プログラミング－言語・基礎・実践－ 13－13
（１９９３． ８． ２０）

超並列オブジェクトベース言語 *OCore* における メタレベルアーキテクチャ

友清 孝志, 石川 裕, 小中 裕喜, 前田 宗則, 堀 敦史
技術研究組合 新情報処理開発機構 (RWC) つくば研究センタ
e-mail:{tomokiyo,ishikawa,konaka,m-maeda,hori}@trc.rwcp.or.jp

本稿では，現在我々が設計を進めている超並列オブジェクトベース言語 *OCore* のメタアーキテクチャについて述べる．*OCore* は超並列計算モデル研究者の Research Vehicle となることを目標に，柔軟かつ効率的な並列プログラムの記述を狙った言語であり，各種イベントや例外処理を柔軟に処理するためのメタレベルアーキテクチャを持つ．これらについて *OCore* のオブジェクトの内部動作とともに述べる．

Meta-level Architecture in *OCore*

Takashi Tomokiyo, Yutaka Ishikawa,
Hiroki Konaka, Munenori Maeda, Atsushi Hori

Tsukuba Research Center, Real World Computing Partnership
Tsukuba Mitsui Building 16F, 1-6-1 Takezono, Tsukuba,
Ibaraki, 305, JAPAN

In this paper, we present an overview of the meta-level architecture of OCore, a massively parallel object-based language. We are designing OCore as a research vehicle for massively parallel programming models. OCore has introduced a meta-level architecture without any significant overhead. It handles events caused by both exceptions and state transition of objects in a flexible manner.

1　はじめに

我々は並列計算モデルを効率良く研究していくために，並列オブジェクト指向モデルのもつ処理の柔軟性と抽象化能力に着目した．言語の開発にはまず並列性を陽に記述しその柔軟な制御を可能とする核言語を考える．そしてその上に抽象度の高い言語を階層的に実装し，これら異なる抽象度の言語が協調できる枠組を提供するというアプローチをとる．

超並列オブジェクトベース言語 *OCore* はこのためのベースとなるべく設計されている言語であり，超並列計算のための基本的な機能を提供することを目的としている [4]．*OCore* ではさまざまな計算モデルをプロトタイピングするための基本機能を提供するとともに，ユーザによるデフォルトの言語機能の拡張を可能とするための機能を提供することをめざす．

本稿で提案する *OCore* のメタレベルアーキテクチャは，オブジェクトの実行制御，例外処理，デバッグや性能評価のための処理に目的をしぼり，これらを効率良く扱う機能を提供する．メタ機能の導入にあたり，我々はつぎのような設計方針をとる．

- メタの導入によるオブジェクトの基本動作の処理速度低下はできる限り抑える．
- 強い型づけとコンパイル指向という *OCore* の性質にあった枠組とする．例えばコード領域の動的な変更などは考えない．

2　プログラミング言語 *OCore*

OCore は超並列プログラミングモデル研究者の research vehicle として使用されることを目的とした，超並列オブジェクトベース言語である．以下にその特徴を挙げる．

- MIMD 型超並列計算機を主なターゲットとする．
- 強い型づけと簡素な言語セマンティクスにより，実行時オーバヘッドの少ない効率的なコード生成を可能とする．
- オブジェクトの集合体の構造化と処理の分散を目的として「共同体」という概念を導入する．
- オブジェクト内のスレッドは1本であり，オブジェクト内の並列性は扱わない．ただし例外処理スレッドは別に存在する．
- メッセージは非同期送信を基本とし，返答値が必要な場合は I-structure[1]，および Q-structure[3] などの同期構造体を用いて明示的に同期を行う．

これら *OCore* の基本機能の詳細については文献 [4] を参照されたい．

オブジェクトはクラスのインスタンスとして動的に生成される計算主体である．生成されるオブジェクトの挙動はクラス定義によって記述される (図 1)．

```
1 (class Foo
2   (vars ...)
3   (meta-vars ...)
4   (event-handlers ...)
5   (exception-handlers ...)
6   (methods ...)
7   (functions ...) )
```

図 1: クラス記述のテンプレート

クラス定義はクラス名をはじめとして，インスタンス変数，メタ変数，イベントハンドラ，例外ハンドラ，メソッド，局所関数定義などから構成される．これらのうちメタ変数，イベントハンドラ，例外ハンドラはメタレベルに関する記述である．*OCore* によるプログラム例を図 2 に示す．

図 2において，クラス Bounded-Buffer のオブジェクトは整数型の変数として length,rp,wp を，整数配列型の変数として buffer をもつ．またメソッドとして:write と:read をもつ．ここではリード，ライトの要求を正しく扱うための同期は行なっていない．メタを用いて正しく扱うように記述した例は図 6を参照されたい．

3　メタレベルアーキテクチャ

3.1　メタレベル制御の目的

ここで提案するメタレベルでは以下の事項を扱うことを目的とする．

- スケジューリングの制御
- プロセッサへのオブジェクトのマッピングによる負荷分散の制御
- 例外処理
- デバッグ，性能評価

```
 1 (class Bounded-Buffer
 2   (vars (Int length :init 10          ; buffer length is 10.
 3             rp :init 0                ; 'rp' stands for read pointer.
 4             wp :init 0)               ; 'wp' stands for write pointer.
 5        ((Array-of Int) buffer         ; 'buffer' is an integer array.
 6         :init (new (Array-of Int) length)))
 7   (methods
 8    ([:write (Int x)]
 9       (local ()                       ; put a integer into buffer, and
10              (set (aref buffer (mod wp length)) x)
11              (set wp (1+ wp))))       ; increment write pointer.
12    ([:read ((Qstr-of Int) x)]
13       (local ()
14              (qwrite x                ; return value should be wrote into
15                      (aref buffer (mod rp length))) ; Q-structure.
16              (set rp (1+ rp))))))     ; increment read pointer.
```

図 2: クラス Bounded-Buffer の記述例

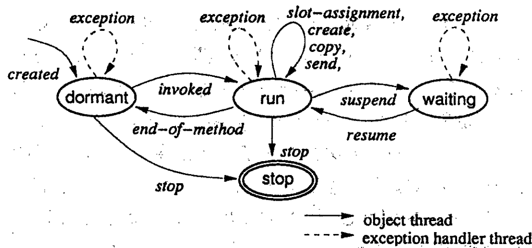

図 3: オブジェクトの状態遷移

3.2 メタレベルの概要

オブジェクトの振舞いは, メタレベルにおけるオブジェクト内部状態遷移およびそれら状態遷移に対応する振舞いによって定義される. 状態遷移を引き起こす原因は状態遷移イベントと呼び, イベントに対応した手続き (イベントハンドラ) がメタレベルで定義される. 状態遷移イベントの種類を表 1に示す.

オブジェクトで生じる例外イベントに対する振舞いは, メタレベルで定義される例外イベントに対応した手続き (例外イベントハンドラ) によって規定される. 例外イベントの種類を表 2に示す.

各オブジェクトにはメタレベルに上がった時のみアクセスすることのできる変数を宣言することができる. これらの変数をメタレベル変数と呼ぶ. また, メタレベルではオブジェクトの内部表現 (例えばメッセージキュー) をメタレベル擬変数として扱うことができる. メタレベル擬変数の一覧を表 3に示す.

メタレベル擬変数において環境オブジェクトとは, オブジェクトの生成時に形成される例外処理のためのグループを管理するために使用されるオブジェクトである. 環境オブジェクトはメンバのリストを内部に持ち, メンバオブジェクトへの例外発生等を請け負う.

表 1: 状態遷移イベントの種類

イベント名	説明
created	自オブジェクトが生成された
create	他オブジェクトを生成しようとした
copy	自分のコピーを生成しようとした
send	メッセージを送ろうとした
slot-assignment	スロットに値を設定しようとした
suspend	Q-structure の値を読もうとした
resume	Q-structure から値が返ってきた
invoked	メソッドが invoke された
end-of-method	メソッドの処理が終了した
stop	自オブジェクトが死んだ

メタレベルで定義されるオブジェクトの振舞いについて具体的に説明する (図 3). オブジェクトは,dormant(休眠) 状態として生成され, created イベントハンドラによって定義された動作を行なう. オブジェクトは単一のメッセージキューを持つ. メッセージが到着するとオブジェクトは run 状態となり invoked イベントハンドラを呼び出す. invoked イベントハンドラでは, メッセージによって指定されているメソッドを実行する. 1 つのメッセージの処理を終えると end-of-method イベントハン

表 2: 例外イベントの種類

例外	説明
:floating-exception	浮動小数点演算例外の発生
:integer-exception	整数演算例外の発生
:timeout	デッドラインミス
:signal	signal の受け付け
:operation-to-undef	UNDEF オブジェクトへの操作
:unused-Use-Once	Use-Once 型参照の破棄

表 3: メタレベル擬変数

擬変数名	型	説明
environment	Object	環境オブジェクト
message	Message	当該メッセージ
message-queue	(Qstr-of Message)	メッセージキュー
sender	Object	メッセージセンダ
killed	Boolean	消去状態フラグ

ドラによって,dormant 状態に戻る. メソッド実行中に Q-structure 等の同期構造体からの読み出しなどが行われると waiting 状態に移行し, その同期構造体への他からの書き込みなどで run 状態に復帰する.

オブジェクトの実行は, 例外により非同期に発生する別スレッドからも影響を受ける (図 3の点線部).

オブジェクトは自殺するかあるいは例外処理スレッドにより stop 状態に遷移する.stop 状態になったオブジェクトは, その後, 参照がなくなった時点で GC により回収される.

つぎに, オブジェクトの動作を図 4に示す. 各オブジェクトは1本のメッセージキューと1つのスレッドとをそれぞれ持っている. オブジェクトに送信されるメッセージは [:*selector,arg0,arg1,...*] の形をしており, オブジェクトはメッセージキューよりメッセージを取り出し, 指定されたセレクタに対応するメソッドを実行する. メッセージキューは同期つき構造体の一種である Q-structure を用いて表現することができ, 上記操作は以下のように表現できる.

`(method-invoke (qread message-queue))` (3.1)

ここで,`message-queue` はメッセージキューを指す擬変数,`(qread` *qstr*`)` は Q-structure に対する読みだし要求を行う関数,`(method-invoke` *message*`)` はメッセージ中に指定されたセレクタを呼び出すシステム関数である.(3.1) により関数 `(method-invoke)` へのコンティニュエーションが Q-structure に渡される.

一方, メッセージの送信処理は, 相手先オブジェクトのメッセージキューに対する enqueue 操作である.

`(qwrite (message-queue-of` *obj*`))` (3.2)

enqueue 操作は (3.2) をコンティニュエーションとするタスクを生成することに相当する. このタスクは各プロセッサエレメントに 1 つずつ存在する System Queue に enqueue される. システムは System Queue よりタスクを 1 つずつ取り出し実行する. (3.2) のタスクが実行されると,Q-structure の同期機構により受信側オブジェクトの `(method-invoke)` へのコンティニュエーションをもつタスクが System Queue に enqueue される.

なお,*OCore* におけるメッセージ送信は一方的な非同期送信であり, 返答値を受けとらない. リプライが必要な場合は Q-structure のエントリを生成しておきメッセージに含めて陽に渡す. 受信側は受けとった Q-structure エントリに対して `qwrite` により返答値を書き込む. 一方, 送信側は送信後もそのまま処理を続け, 必要に応じ `qread` によって返答値に対する同期をとる. この返答値に対するコンティニュエーションもメッセージ送信の場合と同様に各 PE の System Queue に enqueue される.

1 つのメソッドの実行が終了すると, 再び (3.1) を実行する. これにより, もしキューにメッセージがあればそのメソッドを実行し, なければサスペンド状態となる. また, 新たに生成されたオブジェクトも, メッセージ待ちとなるよう同様に (3.1) を最初に実行しておく.

3.3 ハンドラ

3.3.1 ハンドラ定義例

イベントハンドラはイベント名と同名のハンドラ, 例外ハンドラは例外名と同名のハンドラを記述することにより宣言される (図 5). イベントおよび例外ハンドラはクラス内 (4〜10行目,11〜14行目) および catch 文内 (17〜24行目) で定義される.

catch 文内で定義されるハンドラは,catch 文内の本体が実行されている時にのみ有効である. catch 文の外で実行されている時には, クラス内で定義されるハンドラが有効となる. クラス内にハンドラが宣言されていない場合, イベントハンドラと例外ハンドラでハンドラ探索の手順が異なる.

3.3.2 イベントハンドラの探索

状態遷移イベントに関しては, クラス内にイベントハンドラが宣言されていない場合, システム関数として定義されたデフォルトハンドラが呼ばれる.

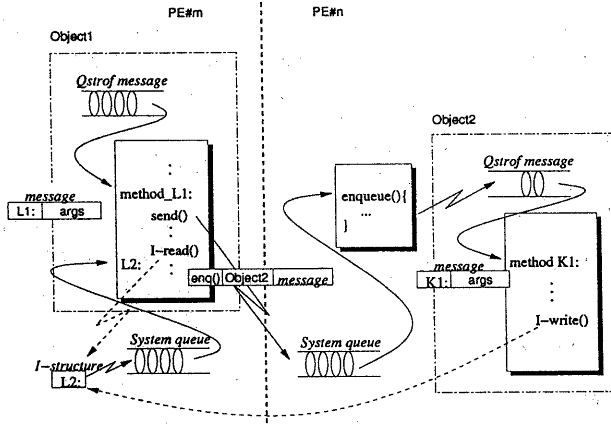

図 4: メッセージハンドリング

```
 1 (class Foo
 2   (vars ...)
 3   (meta-vars ...)
 4   (event-handlers
 5    (create (args) hook-body)
 6    (created (args) hook-body)
 7    (send (args) hook-body)
 8    ...
 9    (user-hook1
10     (args) hook-body))
11   (exception-handlers
12    ([:floating-exception] (...))
13    ([:operation-to-undef] (...))
14    ([others ...] (...) ))
15   (methods
16    ([:m1 args]
17     (catch
18       (
19        ; main body
20       )
21       (event-handlers
22         (send (args) hook-body))
23       (exception-handlers
24         ([:overflow ...] ...)))))
25   (functions ...) )
26
27
```

図 5: イベントハンドラ定義例

ユーザは, ユーザ定義のイベントおよびイベントハンドラを記述することが出来る. 例えば,9行目では,**user-hook1**イベントハンドラを定義している. このイベントハンドラは以下のようなユーザイベント発生機能により, 呼び出される.

```
(trigger 'user-hook1 args)
```

3.3.3 例外ハンドラの探索

例外ハンドラにおいては,catch 文がネストしていた場合, 最も内側で当該ハンドラが定義されていれば, そのハンドラが起動される. catch 文の外側あるいは,catch 文内でハンドラが定義されていない場合, クラス内で当該ハンドラが定義されていれば, そのハンドラが起動される.

クラス内ハンドラも存在しない場合は送信オブジェクト側のハンドラを探しにいく. 送信オブジェクトでも受け付けられない場合は, 例外が発生したオブジェクトが属する環境オブジェクトのハンドラが起動される. 環境オブジェクトはネスト可能であり, 内側の環境オブジェクトで受理されない場合は順に外側の環境オブジェクトを探索していくことになる.

以下に環境オブジェクトにおける例外ハンドラ定義例を示す.

```
1 (class Env1
2   (vars ((List-of Object) members))
3   (exception-handlers
4     ([:overflow ...] ...)
```

```
5      (others ...) )
6     (methods
7      ([:add (Object obj)]
8       (set members (cons obj members)))
9      (...) ))
```

オブジェクトのメソッドおよび例外ハンドラ内で,raise文を用いて例外を発生することが可能である.また,以下のようにsignal文を用いて,あるオブジェクトに対して例外を発生させることも可能である.

(signal *an-object exception-name*)

4 メタレベル制御の例

4.1 Bounded Buffer

Bounded-Buffer(図2)を改良したプログラム例を図6に示す.図2のプログラムがバッファfullの状態での書き込みにおいてデータを上書きしたのに対して,本プログラムではガードつきメソッド呼びだしと同様のセマンティクスを与える.すなわちバッファempty状態におけるreadメッセージの処理は,後の書き込みがあるまでサスペンドされる.バッファfull状態におけるwriteメッセージ処理も同様である.

メッセージ処理のサスペンドを実現するために,オブジェクトはメタレベル変数として読みだしと書き込みのそれぞれに対してメッセージを蓄えるFIFOキューを用意する(4-7行目).

メソッドが呼び出されるとメッセージが実行可能かどうかを調べ,(30,37行目)満たされない場合にはユーザ定義のイベントハンドラを介して各要求毎の保存用キューにenqueueする(10,12行目).保存されたメッセージは,メソッド終了イベントハンドラ(end-of-method)において条件が成立した場合に処理が再開される(16-23行目).再開条件が成立しない場合にはベースレベルに戻り,通常の処理として受けとったメッセージの処理をおこなう.

ここで(qget *Qstr*)はQ-structureに対するノンブロッキング型の呼びだしオペレータであり,キューが空の場合にUNDEFオブジェクトを返す.また(method-invoke *msg*)はメタレベルの処理を終了してベースレベルに復帰し,引数メッセージ*msg*の処理をおこなうオペレータである.

4.2 優先度スケジューリング

優先度スケジューリングの例を図7に示す.本プログラムでは,メソッド終了のイベントであるend-of-methodに対するハンドラにおいて,現在のメッセージキューの内容をメタ変数に用意したpriority-queueに振り分けてenqueueし,その中で最も優先度の高いメッセージをとり出して処理する.

5 おわりに

超並列オブジェクトベース言語*OCore*におけるメタレベルアーキテクチャについて述べた. *OCore*はオブジェクトの状態遷移に伴って発生するイベントと例外イベントをメタレベルで処理するための機構を備え，これらのハンドラの記述により処理の柔軟な制御を可能にすると同時に，処理効率の低下を最小限に抑えている.

今後，アプリケーション記述を通して必要な機能をフィードバックし，超並列計算機RWC-1[2]上に実装する予定である.

謝辞

本研究を進めるにあたって，有意義な議論をして頂いたRWC超並列ソフトウェアワークショップならびにRWC超並列言語ワーキンググループのみなさまに感謝いたします.

参考文献

[1] R.S. Nikhil and K.K. Pingali. I–Structure: Data Structures for Parallel Computing. *ACM Trans. on Programming Languages and Systems*, Vol. 11, No. 4, pp. 598–639, 1989.

[2] 坂井修一, 岡本一晃, 松岡浩司, 広野英雄, 児玉祐悦, 佐藤三久, 横田隆史. 超並列計算機RWC-1の基本構想. In *JSPP '93*, 1993.

[3] 佐藤三久, 児玉祐悦, 坂井修一, 山口喜教. 高並列計算機EM-4における分散データ構造を用いたマルチスレッドプログラミング. 情報処理学会計算機アーキテクチャ研究会, 1992.

[4] 小中裕喜, 石川裕, 前田宗則, 友清孝志, 堀敦史. 超並列オブジェクトベース言語*OCore*の概要. In *Proceedings of the SWoPP '93*, 1993.

```
 1 (class Bounded-Buffer
 2 ;;; META LEVEL
 3   (meta-vars
 4    ((Qstr-of Message) write-queue        ; queue for suspended 'write'
 5     :init (new (Qstr-of Message)))
 6    ((Qstr-of Message) read-queue         ; queue for suspended 'read'
 7     :init (new (Qstr-of Message))))
 8   (event-handlers
 9    (buffer-full ()                       ; try to write into full buffer !
10     (qwrite write-queue message))        ; put current message into queue.
11    (buffer-empty ()                      ; try to get from empty buffer !
12     (qwrite read-queue message))         ; put current message into queue.
13    (end-of-method ()                     ; a system defined event
14     (local ((Message msg))
15       (cond
16        ((not-full)
17         (set msg (qget write-queue))     ; if buffer is not full and
18         (cond ((not (undefined msg))     ; suspended 'write' exists,
19                (method-invoke msg))))    ; then invoke write method.
20        ((not-empty)
21         (set msg (qget read-queue))      ; if buffer is not empty and
22         (cond ((not (undefined msg))     ; suspended 'read' exists,
23                (method-invoke msg)))))))); then invoke read method.
24 ;;; BASE LEVEL
25   (vars (Int length :init 10 rp :init 0 wp :init 0)
26         ((Array-of Int) buffer :init (new (Array-of Int) length)))
27   (methods
28    ([:write (Int x)]                     ; write method
29      (local ()
30             (cond ((not-full)           ; if buffer is not full,
31                    (set (aref buffer (mod wp length)) x)
32                    (set wp (1+ wp)))    ; then write normally,
33                   (else                 ; else trigger user event
34                    (trigger buffer-full))))) ; to enter META level.
35    ([:read ((Qstr-of Int) x)]           ; read method
36      (local ()
37             (cond ((not-empty)          ; if buffer is not empty,
38                    (qwrite x (aref buffer (mod rp length)))
39                    (set rp (1+ rp)))    ; then read normally,
40                   (else                 ; else trigger user event
41                    (trigger buffer-empty)))))) ; to enter META level.
42 ;;; LOCAL FUNCTIONS
43   (functions
44    (not-full () (retunrs Boolean)       ; check if buffer is not full.
45             (< length (- wp rp)))
46    (not-empty () (returns Boolean)      ; check if buffer is not empty.
47              (< rp wp))))
```

図 6: クラス Bounded-Buffer のメタによる再定義例

```
 1 (globals (Int n_priorities :single-assign :init 5)) ; The number of priorities
 2
 3 (typedef Mq (Qstr-of Message))          ; Message queue type
 4 (typedef MqArray (Array-of Mq))         ; Array of message queue type
 5
 6 (class Foo
 7 ;;; META LEVEL
 8   (meta-vars
 9    (MqArray priority-queue :init (new MqArray n_priorities)))
10   (event-handlers
11    (end-of-method                       ; an event handler
12     (local ((Message msg)
13             (Int priority))
14       (do ((msg (qget message-queue)
15                 (qget message-queue) )) ; repeat reading the message queue
16           ((undefined msg))             ; until empty
17           (match [:read priority _] msg); get priority value
18           (qwrite (aref priority-queue  ; put message into priority queues
19                         priority) msg) )
20       (do ((priority 0 (1+ priority)))  ; find highest priority message
21           ((>= priority n_priorities))  ; in my internal queue
22           (set msg (qget (aref priority-queue priority)))
23           (if (not (undefined msg))     ;
24               (method-invoke msg))))))  ;
25 ;;; BASE LEVEL
26   (methods
27    ([:read  (Int priority) ((Qstr-of Int) ans)]
28             ( <method-body> ) )))
```

図 7: 優先度スケジューリングのメタによる記述例