Meta-level Architecture in *OCore*

(An Extended Abstract for the OOPSLA '93 Object-Oriented Reflection Workshop)

Takashi Tomokiyo, Hiroki Konaka, Munenori Maeda, Atsushi Hori, Yutaka Ishikawa
e-mail:{tomokiyo,konaka,m-maeda,hori,ishikawa}@trc.rwcp.or.jp

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

1 Introduction

We are designing a parallel object-based language called *OCore*[1] to generate efficient code for massively parallel machines such as RWC-1[3]. *OCore* includes the following features.

- Simple and statically typed semantics makes compiled programs more safe and efficient.
- An object has a single thread, called the *normal thread*, for normal operations. Exceptions are, however, handled by another thread called the *exception handling thread*.
- Communication between objects is based on asynchronous one-way message passing.

OCore has a meta-level architecture for flexible control of handling messages, mapping objects to processors, and profiling executions. The meta-level architecture is carefully introduced in *OCore* without sacrificing efficiency.

In this paper we present an overview of the meta-level architecture and demonstrate usefulness of the meta-level architecture with a program example.

2 Meta-Level Architecture

Purposes of the meta-level architecture are as follows:

- controlling the order of message handling
- load-balancing by mapping objects when created
- handling exceptions
- capturing internal events of objects for profiling and debugging

Our primary research interest is efficiently implimenting meta-level architectures in a massively parallel environment.

The above purposes are fulfilled without meta-circularity[5] in the meta-level architecture of *OCore*; meta-circularity causes extra execution overhead.

In *OCore* an object is created according to a class. Unlike Smalltalk-80, a class is neither an object nor the meta of an object. Rather, a class is just a template as in C++.

Processing of an object is either in the base-level or in the meta-level. Object representation is accessed in the meta level. Each object belongs to one of the finite states of activity and makes state transitions. Meta-level processing is invoked when a state transition event or an exception occurs.

2.1 Events and Handlers

The meta-level architecture in *OCore* handles three kinds of events: state transition events, user-defined events, and exceptions.

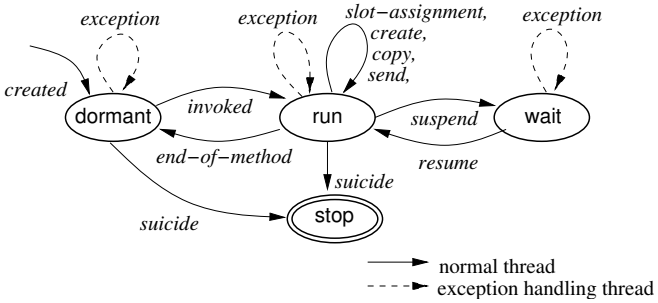

Figure 1: State transition diagram of an object

state transition events An object makes state transitions as shown in Figure 1. We treat each state transition as a *state transition event*. Table 1 shows the state transition events.

An object is always in one of four states of activity: dormant, run, wait, and stop. An object is in the dormant state when created. It changes its state to run when it receives a message. When an object needs to wait for a reply, it proceeds to the wait state; when the reply arrives, it returns to the run state. An object returns to the dormant state after having processed each message. An object goes to the stop state when it commits *suicide*. An object which has received a kill signal will also commit *suicide*. An object in the

Table 1: State transition events

Event Name	description
created	being created
create	creating another object
copy	copying self to another processor
send	sending message to object
suspend	requesting value from Q-structure
resume	receiving value from Q-structure
slot-assignment	assignment to slot
invoked	method invocation
end-of-method	method termination
suicide	committing suicide

`stop` state may be reclaimed by the garbage collector when it ceases to be referred to.

An object may include *event handlers* for events in the meta-level definition. Each handler has the same name as that of the corresponding *event*. An event handler is invoked when the associated event occurs. If there is no handler definition for the event in the class definition, the default handler for the event defined in the system library is invoked.

user-defined events Programmers may define their own events and event handlers. An user-defined event handler is invoked by the following user event trigger function:

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

exceptions The behavior of an object may also be affected by exceptions. They are handled by exception handlers.

Event handlers as well as base-level operations are executed by the normal thread. Exception handlers are executed in the meta-level by the exception handling thread.

Each object may hold meta-level variables, which are accessed only in the meta-level. Some elements of the object internal representation, such as a message queue, are also accessed in the meta-level via meta-level pseudo-variables. Table 2 shows some meta-level pseudo-variables.

Table 2: some meta-level pseudo-variables

pseudo-variable	type	description
message	Message	current message
sender	Object	sender of message
message-queue	(Qstr-of Message)	message queue
environment	Object	environment object

A message queue is represented as a *Q-structure*[4] in the meta-level; we will explain this in more detail in the next section.

An *environment* object is used to manage an exception handling group which is formed when member objects are created. When a member of the environment object cannot handle an exception, it is passed to the environment object. The environment object may also control the members by sending a signal, for example, to all the members.

2.2 Message Handling Mechanism

Message passing in *OCore* is one-way and asynchronous. *Q-structures* are used for receiving reply values.

The Q-structure is an extension of the synchronizing structure memory, I-structure[2]. It accepts multiple read or write requests; each kind of request are put in a queue and processed in FIFO order. Operations to a Q-structure include:

(`qread` *Qstr*) Get the next value from the queue in FIFO order; if the queue is empty, the caller's thread is suspended waiting for a value written by a `qwrite`.

(`qget` *Qstr*) Get the next value from the queue in FIFO order; if the queue is empty, UNDEF is returned.

(`qwrite` *Qstr Val*) Append a value to the queue; if suspended read requests exist in the queue, the value is not enqueued but passed to the first request.

By passing a Q-structure in the message explicitly a sender may receive a reply value. The sender continues processing even after sending a message. When the reply value is really necessary, the sender waits for the reply by `qread`ing. The receiver, meanwhile, writes the return value into the Q-structure by `qwrite`ing when the value is ready.

Figure 2 illustrates the message handling mechanism more precisely. There is a *system queue* in each processor element, which is not accessible to objects. A system queue holds *tasks*. The system on each processor dequeues a task from the system queue and executes it one by one. Sending a message, starting and resuming a method execution, and processing `qwrite` are all treated as tasks.

Each object has a single message queue. A message is formed as `[:`*selector arg0 arg1 ...*`]`. The normal thread in each object dequeues a message from the message queue and executes the method specified by the selector.

Using a Q-structure for a message queue, we represent method invocation in the meta-level as follows:

```
(method-invoke (qread message-queue))      (1)
```

`message-queue` is a pseudo-variable that refers to the queue of messages. (`method-invoke` *Message*) is a meta-level operator that quits meta-level processing and invokes in the base-level the method specified by the message selector.

Expression 1 is executed whenever method execution is finished, since it is the default handler for the *end-of-method* event. It is also executed when an object is created, since it also forms a part of the default handler for the *created* event.

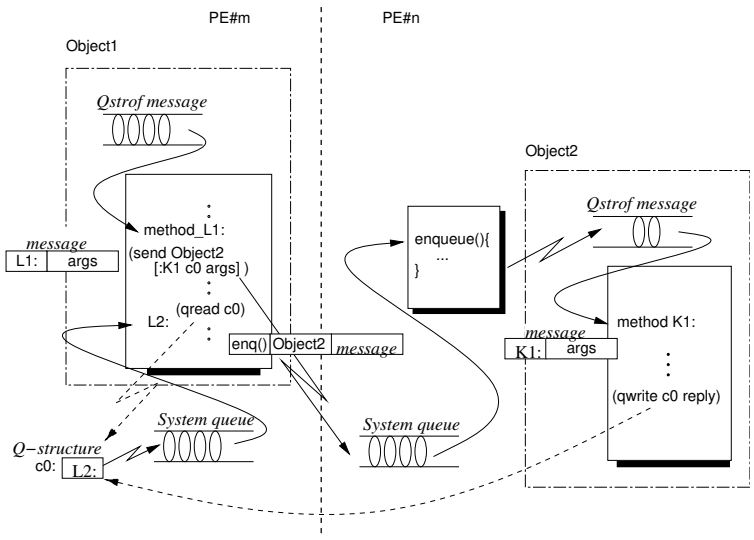

Figure 2: Message handling mechanism

If there is no message, the execution of Expression 1 is suspended at `qread`. Otherwise, the first message is dequeued and processed.

3 Bounded Buffer: An Example of Meta-Level Programming

We present here an example of a bounded-buffer in *OCore*. The program shown in Figure 3 provides a kind of guarded method invocation. Read requests to an empty buffer are suspended until a write request arrives; write requests to a full buffer are also suspended waiting for a read request.

An object of class `Bounded-Buffer` has meta-level variables `write-queue` and `read-queue`, declared in lines 4 and 5. *Q-Structure* variable `write-queue` is used to handle messages `:write`, while *Q-structure* variable `read-queue` is used to handle messages `:read`. The object also has an state transition event handler `end-of-method` shown in line 9, instead of the default handler, and two user-defined event handlers, `buffer-full` and `buffer-empty` shown in lines 7 and 8.

The object has base-level variables `length`, keeping the buffer length; `rp` for a read pointer; `wp` for a write pointer; and `buffer` for keeping values declared in lines 18 and 19.

When the `:write` method declared in lines 21 to 24 is invoked, function `not-full` in line 30 is first called to check whether or not the buffer is full. If the buffer is not full, data is written to the buffer in line 23. Otherwise, the expression in line 24 is executed to invoke the user event handler `buffer-full`. In the `buffer-full` event handler, the message handled by the `:write` method is enqueued in the `write-queue` variable in line 7, since the meta-level pseudo-variable `message` holds the message currently being handled in the base-level. After the execution of the expression in line 24, the execution of the `:write` method is terminated. It should be emphasized that unlike most concurrent object-oriented programming languages, the reply message is not sent to the sender when the execution of a method is finished.

When the `:read` method declared in lines 25 to 28 is invoked, function `not-empty` in line 31 is first called to check whether or not the buffer is empty. If the bounded buffer is not empty, data is extracted from the buffer. Otherwise, the `buffer-empty` user event handler is invoked in line 28. In the `buffer-empty` event handler, the message handled by the `:read` method is enqueued in the `read-queue` in line 8.

When method execution terminates, the `end-of-method` event handler defined in lines 9 to 16 is invoked. If the bounded buffer is not full and there is a suspended write request, the request is extracted from `write-queue` by issuing the `qget` operation in line 12, and then the `:write` method is invoked in line 13. If the bounded buffer is not empty and there is a suspended read request, the request is extracted from `read-queue` by issuing the `qget` operation in line 15, and then the `:read` method is invoked in line 16.

4 Summary

We have presented an overview of the meta-level architecture of *OCore*. It is designed to be implemented efficiently without meta-circularity.

The architecture captures *state transition events* and *exceptions*. An object may include *event handlers* and *exception handlers*. They are executed in the meta-level. If defined, each event handler is invoked when the associated event occurs. When an exception occurs, the appropriate

```
 1 (class Bounded-Buffer
 2 ;;; META LEVEL
 3   (meta-vars
 4    ((Qstr-of Message) write-queue :init (new (Qstr-of Message)))      ; a queue for unsatisfied 'write's
 5    ((Qstr-of Message) read-queue  :init (new (Qstr-of Message))) )    ; a queue for unsatisfied 'read's
 6   (event-handlers
 7    (buffer-full () (qwrite write-queue message))                      ; an user-defined event handler
 8    (buffer-empty () (qwrite read-queue message))                      ; an user-defined event handler
 9    (end-of-method ()                                                  ; a state transition event handler
10     (local ((Message msg))
11       (cond ((not-full)                                              ; if the buffer is not full
12              (set msg (qget write-queue))                            ; and a suspended 'write' exists,
13              (if (not (undefined msg)) (method-invoke msg)) )        ; then process it
14             ((not-empty)                                            ; else if the buffer is not empty
15              (set msg (qget read-queue))                            ; and a suspended 'read' exists,
16              (if (not (undefined msg)) (method-invoke msg)) )))))   ; then process it
17 ;;; BASE LEVEL
18   (vars (Int length :init 10 rp :init 0 wp :init 0)
19         ((Array-of Int) buffer :init (new (Array-of Int) length)) )  ; the buffer
20   (methods
21    ([:write (Int x)]
22      (cond ((not-full) (set (aref buffer (mod wp length)) x)         ; if the buffer is not full,
23             (set wp (1+ wp)) )                                       ; then write normally,
24            (else (trigger buffer-full)) ))                           ; else trigger an user event
25    ([:read ((Qstr-of Int) x)]
26      (cond ((not-empty) (qwrite x (aref buffer (mod rp length)))     ; if the buffer is not empty,
27             (set rp (1+ rp)) )                                       ; then read normally,
28            (else (trigger buffer-empty)) )))                         ; else trigger an user event
29   (functions
30    (not-full () (returns Boolean) (< length (- wp rp)))             ; check if the buffer is not full
31    (not-empty () (returns Boolean) (< rp wp))) )                    ; check if the buffer is not empty
```

Figure 3: a class definition of Bounded-Buffer using meta-level programming

exception handler is searched for and invoked. Some elements of the object internal representation are accessed via pseudo-variables in the meta-level.

We have also demonstrated usefulness of the meta-level architecture with a program example. The example simulates a kind of guarded method invocation without any significant overhead.

References

[1] Hiroki Konaka, et al. An overview of *OCore*: A massively parallel object-based language. In *Proceedings of 1993 Summer United Workshops on Parallel, Distributed, and Cooperative Processing. SIGPRG in Information Processing Society of Japan*, 1993. (in Japanese).

[2] 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.

[3] Shuichi Sakai, Kazuaki Okamoto, Hiroshi Matsuoka, Hideo Hirono, Yuetsu Kodama, Mitsuhisa Sato, and Takashi Yokota. Basic features of a massively parallel computer RWC-1. In *Proceedings of Joint Symposium on Parallel Processing 1993*, pp. 87–94. Information Processing Society of Japan, 1993. (in Japanese).

[4] Mitsuhisa Sato, Yuetsu Kodama, Shuichi Sakai, and Satoshi Sekiguchi. Distributed data structure in thread-based programming for a highly parallel dataflow machine EM-4. In *Proceedings of ISCA 92 Dataflow Workshop*, 1992.

[5] B. C. Smith. Reflection and semantics in lisp. In *Proceedings of 11th ACM Symposium on Principles of Programming Languages*, pp. 22–35, 1984.