Reflection in an Object-Oriented Concurrent Language

Takuo Watanabe and **Akinori Yonezawa**

Department of Information Science, Tokyo Institute of Technology
Ookayama, Meguro-ku, Tokyo 152, Japan
takuo%is.titech.junet@relay.cs.net,
yonezawa%is.titech.junet@relay.cs.net

Abstract

Our work is along the line of the work of B. Smith and P. Maes. We first discuss our notion of *reflection* in object-oriented *concurrent* computation and then present a reflective object-oriented concurrent language ABCL/R. We give several illustrative examples of reflective programming such as (1) dynamic concurrent acquisition of "methods" from other objects, (2) monitoring the behavior of concurrently running objects, and (3) augmentation of the time warp mechanism to a concurrent system. Also the definition of a meta-circular interpreter of this language is given as the definition of a meta-object. The language ABCL/R has been implemented. All the examples given in this paper are running on our ABCL/R system.

1 Introduction

Reflection is the process of reasoning about and acting upon itself[10][7]. A reflective computational system is a computational system which exhibits reflective behavior. In a conventional system, computation is performed on data that represent (or model) entities which are external to the computational system. In contrast, a reflective computational system must contain some data that represent (or model) the structural and computational aspects of the system itself. And such data must be manipulable within the system itself, and more importantly, changes made to such data must be causally reflected/connected to the actual computation being performed.

B. Smith[11] and other researchers(e.g.,[3][13][1]) investigated the power of computational reflection and emphasized its usefulness. In particular, P. Maes has proposed a reflective system in the framework of object-oriented computing[8] and made good contributions to the fields of object-oriented programming and reflective computation. Her work, however, confined itself to sequential systems, and did not consider systems where more than one object can be active simultaneously.

Our present work proposes a reflective system in the framework of object-oriented *concurrent* computing. This is one of our research results in the paradigm of "Object-Oriented Concurrent Programming"[16][5]. We expect that reflective facilities will become increasingly more important in concurrent computational systems such as (distributed) operating systems, realtime systems, distributed simulation systems, distributed problem solving systems, robot planning/controlling, etc. For reflective capabilities are indispensable when one tries to make the behavior of these systems more powerful and intelligent as well as controllable by the user.

In this paper, we first discuss our notion of reflection in object-oriented concurrent computing and then present a reflective object-oriented concurrent language ABCL/R. We give several illustrative examples of reflective programming such as

- dynamic concurrent acquisition of "methods" from other objects,
- monitoring the behavior of concurrently running objects, and
- augmentation of the time warp mechanism[6] to a concurrent system.

Also the definition of a meta-circular interpreter of this language is given as the definition of a meta-object. This language ABCL/R is an extension of our previously proposed language ABCL/1[14] and has been implemented. All the examples given in this paper are running on our ABCL/R system.

The summary of our present work is given in the final section of this paper.

2 Object-Oriented Concurrent Computation Model

In order to present the framework of our work, we first introduce an object-oriented concurrent computation model. This model is basically a submodel of our existing object-oriented concurrent computation model ABCM/1[14].

2.1 Overview of the Computation Model

In our computation model, a system is a collection of *objects* — autonomous information processing agents. Each object has an individual serial computation power, and may have local persistent memory called a *state memory*. Functions and properties of a conceptual/physical entity in the problem domain are modeled

and represented as such an object. In order to use the functions and properties, a request *message* is sent to the object. When an object receives a message, if the message is acceptable to the object, it starts a sequence of actions which are requested by the message.

Actions performed by an object are combinations of inquiring/updating the object's local state memory, sending messages to other objects (including itself), creating new objects, and other symbolic/numerical operations. Basically, sequences of actions by objects in the system proceed asynchronously. This means that many objects perform their computation in parallel. In our model, the unit of concurrency is an object. Communication among objects and the synchronization of their computation are done only by message passing. Any two objects don't share any data other than the names (addresses, or pointers) of other objects. The state memory of each object cannot be accessed directly by other objects, and only indirect accesses through messages passing are permitted.

Each object is always in one of the two modes: *dormant* and *active*. The mode of an object is dormant at creation time. It becomes active when it accepts a message and starts executing a sequence of actions for the message. When the execution completes, and if no subsequent message has arrived, the object becomes dormant.

From the programming point of view, an object is the basic building block of program. A program is written as the behavior descriptions of objects — what actions to perform when received messages. The description of the behavior of an object is a collection of *scripts* (often called *methods* in other object-oriented languages), which consists of a message pattern and a sequence of actions. A script prescribes the sequence of actions by the object invoked when received a message that matches the message pattern of the script.

2.2 Structure of an Object

Since each object has a single serial processing power, it executes a script one at a time. Although messages can be received by an object which is in active mode, the execution of the scripts for the messages must be postponed until the current script execution completes. Therefore each object has a *message queue* to store incoming messages. (This message queue can be viewed as a part of the receiver memory state.)

The structure of an object consists of a serial evaluator, a set of scripts, a state memory, and a message queue. This structure itself can be regarded as a serial computational system. This structure of an object is a basic one. As will be seen in the subsequent sections, we can build different structures for objects, and such structures can dynamically be changed by using *reflective* language facilities.

2.3 Message Transmission

All message transmission is *asynchronous*. There is no need for any handshaking to send/receive messages. This means that one can send messages whenever it wants, regardless of the current condition (mode) of the target object. When a message is sent to an object, it will be treated by the receiver object in the following way.

1. *Arrival*: First, the message arrives at the receiver object. This event is called the *arrival* of the message. The receiver starts processing of the message. It is assumed that once a message is transmitted, it is guaranteed to arrive at the receiver (as long as the receiver exists).

2. *Receiving*: Next, the receiver object enqueues the arrived message in its message queue — this is the event of *receiving* the message. If the receiver is in dormant mode, it starts trying to accept the messages in the queue (see next).

3. *Acceptance*: If the receiver is in dormant mode, it dequeues the first message in the queue, and checks to see whether the receiver can process it. To be more precise, the receiver tries to find an appropriate script for the message by pattern-matching. *Acceptance* of the message is the event in which the appropriate script for the message is found. If the receiver accepts the message, it starts executing the script for the message. Otherwise, the message is simply ignored (In the language ABCL/R, a warning message is issued).

4. *End of Processing a Message*: When the evaluation finishes, the receiver checks the queue to process subsequent messages. If the queue is empty, the receiver becomes dormant.

2.4 Types of Message Transmissions

Our model has the following two types of message transmission (ABCM/1 also has three types including the *future type*).

- *Past type*: Suppose an object x sends a message to y in the course of computation. Then x does not wait for the message to be received by y, and continues the rest of computation immediately. Using the notation of our reflective language ABCL/R, this type of message transmission is written as:

$$[T \texttt{ <= } M] \qquad \text{or} \qquad [T \texttt{ <= } M \texttt{ @ } R]$$

where T, M, and R are the target object, message, and *reply destination*, respectively. The reply destination is an object to which the receiver can send a reply message. If the reply destination is not specified, the receiver regards `NIL` as the reply destination. (Sending messages to `NIL` causes no effect.)

- *Now type*: When an object x sends a message M to T, x waits for the message to be received by y and further waits for a reply from T to come, blocking the current script execution. A now type message transmission is written in ABCL/R as:

$$[T \texttt{ <== } M]$$

A now type message transmission looks similar to an ordinary remote procedure call, but it is different. In the case of now type, after sending the reply to x, T may continue its computation, and furthermore, T can ask another object y to send a reply to x.

3 Reflection in Our Model

To realize reflection in a system based on our computation model, the causally connected self representation[10][7] of the system must exist within the system. Since reflective computation depends on the way in which self representation is described, choosing the formalism of self representation is the primary concern of building a reflective system.

There are at least two approaches to build the self representation of an object-oriented concurrent computational system. One is to assume the existence of a datum which is the causally connected self representation of the *whole* system, and the other is to introduce the self representation of each object in the system individually. Our approach is the latter one. The remarks on the former approach will be found in the concluding section.

As explained in the previous section, we can regard an object as a serial computational system. Thus we can build a representation of an object as a representation of a serial computational system. The representation of an object contain the representations of the message queue, the state memory, the set of scripts and the evaluator of the object. Besides this structural aspect, the computational aspect of the object — arrival, receiving, and acceptance of a message and the execution of a scripts — must be represented. Our approach is to represent each object as an *object* called a *meta-object*.

For each object x, there exists a meta-object $\uparrow x$, which represents both the structural and computational aspects of x. $\uparrow x$ contains the meta-level information about x. Meta-object $\uparrow x$ represents the object x in a similar way that usual objects represent entities in the problem domain. x is called the *denotation* of $\uparrow x$. The structure of x is represented as the data in the state memory of $\uparrow x$, and the computational aspects of x is described as the scripts (methods) of $\uparrow x$. The following points should be noted.

- An object x and the information about x in $\uparrow x$ are causally connected. Thus the data stored in $\uparrow x$ always represent the current status of x, and operations on the data cause the isomorphic effect on x.

- $\uparrow x$ is an object. So $\uparrow\uparrow x$ also exists. This means that operations on $\uparrow x$ are allowed. Thus, there is an infinite tower of meta-objects $\uparrow x$, $\uparrow\uparrow x$, $\uparrow\uparrow\uparrow x$, ...for each object x. In the actual implementation, meta-objects are created when their access takes place (by lazy creation).

- If one knows the name of x, it can always get the name of $\uparrow x$, and vice versa.

- The correspondence between objects and their meta-objects is one to one. That is, for each object x, y, $x \equiv y \iff \uparrow x \equiv \uparrow y$ holds ($\equiv$ is the identity relation).

The concept of meta-objects in our model is similar to that of meta-objects in 3-KRS[7][8]. In 3-KRS, structural/computational aspects of an object is also represented in its meta-object. But the way of modeling an object as its meta-object is different, because an object is a unit of concurrency in our computation model.

In our model, the causal connection link between an object x and its meta-object $\uparrow x$ is implicit — the changes in x cause the isomorphic changes to the data in $\uparrow x$ not by the message transmission from x to $\uparrow x$ (and vice versa). The reason is that the message transmission takes time, which requires the synchronization of x and $\uparrow x$. As we will see in the later section, $\uparrow x$ is used as the "implementation" of x.

Reflective computation in an object x is performed by x sending messages to its meta-object $\uparrow x$. This enables x to inquire/modify itself through $\uparrow x$, because the structural/computational aspects of x is represented in $\uparrow x$ in a causally connected way. Note that appropriate scripts for operations on x must be prepared in $\uparrow x$. If $\uparrow x$ doesn't have such scripts, it is possible to modify $\uparrow x$ using $\uparrow\uparrow x$ to acquire such scripts. Such examples are found in Section 5.

Of course a meta-object can receive messages from other objects (other than its denotation). In a system that consists of a collection of many objects, we can regard the meta-object of an object as the partial representation of the system. Thus the reflective computation in the system is realized by sending messages to each other's meta-object. Note that such message transmissions may take place concurrently in our computation model.

4 Meta-objects and the Reflective Language ABCL/R

The notion of *meta-objects* is the key concept for the reflection in our computation model. ABCL/R — the description language of our model — is an object-oriented concurrent language with reflective architecture based on the notion of meta-objects. The syntax and basic features of the language are adopted from the language ABCL/1[14][15]. In this section, we describe the definition of meta-object in details in terms of the language ABCL/R.

4.1 Object Definition in ABCL/R

In ABCL/R, an object definition is written in the following form. The value of the form is a newly created object the name of which is *object-name*.

```
[object object-name
  (state variable-declaration...)
  (script
    (=> message-pattern @ reply-destination-variable
                        from sender-variable
        (temporary variable-declaration...)
        behavior description)
    ...)]
```

object-name is optional in the above definition. `(state ...)` is the local state variable declaration. *variable-declaration* is either [*variable* := *initial-value*] or *variable* which is equivalent to [*variable* := nil]. [*variable* := *expression*] is the expression for assignment of the value of *expression* to *variable*. Each `(=>` *message-pattern* ...`)` is the description of a script. The object defined in the above form accepts a message which matches a *message-pattern*. The reply destination and the sender object of an incoming message are bound to the variables *reply-destination-variable* and *sender-variable*, respectively. These two variables are optional. *behavior description* in each script description is a sequence of actions, which are described as expressions of either object creation, message transmission, inquiring/modifying state memory (through state variables), or some other symbolic/numerical calculation (as Lisp expressions). `(temporary ...)` is the declaration of temporary variables used in the script.

4.2 Definition of a Meta-Object in ABCL/R

As explained in Section 3, a meta-object $\uparrow x$ is an object which models the structural and computational aspects of an object x, and x is called the *denotation* of $\uparrow x$.

4.2.1 Modeling the Structure of an Object

Since an object x in our computation model consists of a set of scripts, a state memory, a local serial evaluator, and a message queue, the structural aspect of x is represented as the values of $\uparrow x$'s state variables `scriptset`, `state`, `evaluator`, and `queue`, respectively. Using the ABCL/R notation, this structural aspect is described as the `state`-part of the definition of the meta-object $\uparrow x$ given in Figure 1.

Each element in the value of `scriptset` is a script (represented in a certain data structure, say, character strings), and the values of `state` and `evaluator` are *objects* which represent the state memory and the evaluator. The value of `mode` indicates the current mode of the denotation object, which is either `:dormant` or `:active`.

```
[object  ; a meta-object
  (state [queue := a message queue]
         [state := a state object]
         [scriptset := a list of scripts]
         [evaluator := an evaluator object]
         [mode := either :dormant or :active])
  (script
   (=> [:message Message Reply-Dest Sender] ; message arrival & receiving
       [queue := (enqueue queue [Message Reply-Dest Sender])]
       (if (eq mode :dormant) then
           [mode := ':active]
           [Me <= :begin]))
   (=> :begin ; acceptance & script execution
       (temporary mrs scr newenv [object := Me])
       [mrs := (first queue)]
       [queue := (dequeue queue)]
       [scr := (find-script (first mrs) scriptset)]
       (if scr then ; acceptance
           [newenv := [env-gen <== [:new (script-alist mrs scr) state]]]
           ;; pattern variables, reply & sender variables have been bound in newenv
           [evaluator <= [:do-prg (scr$body scr) newenv [den Me]] @
                         [cont ignore ; the value of the evaluation is ignored
                           [object <= :end]]]
        else ; cannot accept
           (warn "Cannot handle the message ~A" (first mrs))
           [Me <= :end]))
   (=> :end ; termination of the execution
       (if (not (empty? queue)) then
           [Me <= :begin]
        else
           [mode := :dormant]))
   ;; The following scripts are examples of special scripts for meta-level operations
   (=> :queue ; inquiring about the message queue
       !queue) ; returns the value of queue
   (=> [:script Message] ; inquiring about the script whose pattern matches Message
       !(find-script Message scriptset)) ; returns the found script
   ...)]
```

Figure 1: Definition of A Meta-Object

4.2.2 Modeling the Behavior of an Object

Besides the structural aspect, the meta-object $\uparrow x$ models the computational aspect of its denotation — *arrival*, *receiving*, and *acceptance* of messages, and execution of scripts. This aspect is described in the ABCL/R notation as the `script`-part of the definition in Figure 1.

The following is a more precise description of what was explained in Section 2.3 in terms of the ABCL/R notation. Suppose a message M is sent to an object x.

1. *Arrival of a Message*: The arrival of a message M at the object x is represented by acceptance of a message `[:message` M R S`]` by $\uparrow x$. R is the reply destination of M, and S is the sender object of M.

2. *Receiving a Message* (*See the script for* `[:message ...]`): When $\uparrow x$ accepts the message `[:message` M R S`]`, it enqueues the triple `[`M R S`]` to the message queue — the value of the variable `queue`. This represents the situation where the object x receives the message M. If x is in dormant mode — the value of the state variable `mode` in $\uparrow x$ is `:dormant`, then $\uparrow x$ sends a message `:begin` to itself.

3. *Acceptance of a Message* (*See the script for* `:begin`): $\uparrow x$ dequeues one triple `[`M R S`]` from the queue. If there is an appropriate script σ for M in `scriptset` (acceptance), $\uparrow x$ executes the body of σ (see next). If there is no script for M, it just ignores this and sends `:end` to itself after issuing a warning message.

4. *Execution of a Script*: First, $\uparrow x$ creates a new environment — which binds the contents of M to the pattern variables of σ, R to the reply variable of σ, and S to the sender variable of σ —, then evaluates the body of σ under the new environment using the `evaluator` object.

5. *After a Script Execution* (*See the script for* `:end`): When the execution of the script completes, a message `:end` is sent to $\uparrow x$. Then $\uparrow x$ checks the queue, and starts processing of subsequent messages if the queue is not empty (by sending a message `:begin` to itself).

Let us look at the script execution more closely. A new environment is created by the environment generator object `env-gen` from the a-list of pattern variables/values and the state object. The evaluator object `evaluator` is activated by receiving a message `[:do` *Exp Env Me-ptr*`]`, where *Exp* is the expression to be evaluated, *Env* is the environment object, and *Me-ptr* is the object in which the evaluation takes place. The following expression in Figure 1 is executed at the end of the execution of the script for `:begin`.

```
[evaluator <= [:do-prg (scr$body scr) newenv [den Me]]
           @ [cont ignore [object <= :end]]]
```

Message [:do-prg ...] is used instead of [:do ...], and this is used to evaluate the list of expressions (the value of (scr$-body scr)) and the result is the value of the last expression of the list (like **progn** of Lisp). The value of the variable Me is the meta-object itself (Such variables are often named "self" in other languages), and [den Me] (this form is explained in a later section) is the denotation of the meta-object.

Since a *past* type message transmission is used, the execution of the script for :begin immediately completes after the execution of the above expression, and the mode of the meta-object becomes dormant. The result of the evaluation is passed to the reply destination of the message, which is expressed as [cont ...]. Note that the form

```
[cont message-pattern script-description]
```

is syntactically equivalent to the following form.

```
[object
    (script (=> message-pattern script-description))]
```

The notation [cont ...] is intended to be used as the continuation of the evaluation which accepts the evaluation result and does the rest of the task. The result of the script evaluation is bound to a variable ignore and just ignored (the variable ignore is not used in the body), and the rest of the task is to send a message :end to the meta-object which is bound to a variable object.

4.2.3 Inherent Concurrency

Suppose the meta-object $\uparrow x$ has accepted a message :begin and the evaluation of the corresponding script has been started by evaluator. Since this evaluation is triggered by a past type message transmission, now $\uparrow x$ changes to dormant mode and stays in dormant mode until a message :end is sent to $\uparrow x$ from the continuation object ([cont ...]) of the evaluator. (Note that if the evaluator object is executing a script, the mode of x can be active even when $\uparrow x$ is in dormant mode.) Thus $\uparrow x$ can accept the next [:message ...] without waiting for the completion of the current script execution.

This corresponds to the fact that x can receive messages while x is in active mode (*asynchrony* described in 2.3). The fact is called the *inherent concurrency* of the object x.

To model the behavior of an object correctly, it must be guaranteed that the execution of the object's scripts takes place one at a time. We can see that this is guaranteed by the meta-object definition: once a message :begin is sent to the meta-object $\uparrow x$, the next :begin message will not be sent until a message :end is sent to $\uparrow x$. (The value of variable mode becomes :dormant only after the script for :end has been executed.)

The definition in Figure 1 also says that messages arriving at the denotation x are simply enqueued when x is in active mode, and the search of the scriptset by $\uparrow x$ is postponed until the current script execution completes. For more detailed explanation, see [12].

4.3 Meta Circularity of Objects

In ABCL/R, to satisfy the requirement of the causal connection between an object x and its meta-object $\uparrow x$, $\uparrow x$ is used as the actual implementation of x. That is, the contents of the state variables of $\uparrow x$ — a message queue, a set of scripts, a state object, and an evaluator object of x — are used for the actual computation of x. The arrival, receiving and acceptance of messages are performed as we have seen before. Moreover, the evaluation of scripts is carried out by the evaluator object, which is also an ordinary object of ABCL/R. Thus every object of ABCL/R is implemented in a meta-circular way as its meta-object.

The definition of the meta-object in Figure 1 is used in default. In ABCL/R, we can specify other meta-object instead of this in the object definition. An example of a non-default meta-object is described in 6.4.

A message transmission to an object x is defined in terms of its meta-object $\uparrow x$. The form

```
[x <= m @ r]
```

in the script of an object y, which is the sender of the message m, will be reduced to (interpreted as)

```
[↑x <= [:message m r y]]
```

when the above form is evaluated by the evaluator of y. In the definition of the evaluator object, the part for the evaluation of a message transmission expression is actually defined as above. (Of course, it is possible to access the meta-object of the evaluator object.)

Because a meta-object $\uparrow x$ is also an object, there exists an object $\uparrow\uparrow x$ which is a meta-object of $\uparrow x$. This implies that $\uparrow x$ is implemented in $\uparrow\uparrow x$ in the same way as x is implemented in $\uparrow x$. This situation induces an infinite tower of meta-objects for each objects, but in the actual implementation, we can avoid the infinite tower by the lazy creation of meta-objects.

5 Reflective Programming Facilities in ABCL/R

This section explains language facilities for reflection in ABCL/R using simple examples.

5.1 Sending Messages to Meta-Objects

In ABCL/R, $\uparrow x$ can be accessed as the value of the special form [meta x], and the value of [den $\uparrow x$] is x. Thus, for each object x, [meta [den $\uparrow x$]] $\equiv \uparrow x$ and [den [meta x]] $\equiv x$ always holds. Access to $\uparrow x$ enables the inquiry and/or modification of components of x if $\uparrow x$ has scripts appropriate for those operations (e.g., scripts for :queue and [:script Message] in Figure 1).

Let us look at a small program example in which meta-objects are accessed. Suppose that there is a group of objects consisting of the manager object M and some worker objects $W_1, W_2, \ldots$. Each W_i can receive a message of pattern [:job *job-type* :param *parameter*], which is the request for a job of *job-type* with *parameter* (Figure 2).

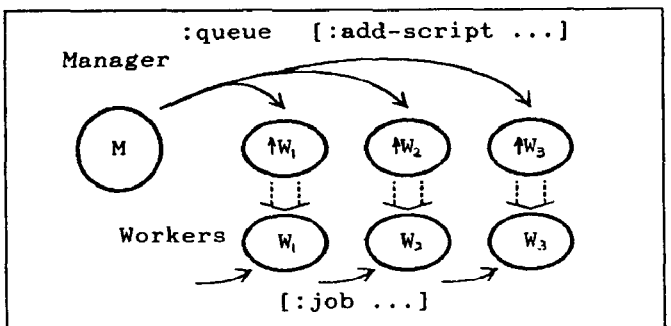

Figure 2: Manager and Workers

M constantly monitors each worker W_i, and if M notices that W_i receives requests of a particular job type (e.g., job 1) very frequently, M gives W_i a new script for [:job 1 :param *parameter*] which is an optimized script for the job type 1. This is realized by accessing $\uparrow W_i$ from M. For example, to know the messages received by W_i, M can simply send a message to $\uparrow W_i$ as

```
[[meta Wi] <== :queue]
```

and also to add the new script for the job type 1, M can send a message to $\uparrow W_i$ as

```
[[meta Wi] <== [:add-script
     '(=> [:job 1 :param parameter-var]
          body of the script)]]
```

Note that this script extension of W_i by M can be done while W_i is executing its jobs — the performance of the whole system is gradually improved while the system is working.

5.2 Reflective Functions

Beside [meta ...] and [den ...], there is another language feature which facilitates reflective programming in ABCL/R. That is *reflective functions*, which are similar to the *reflective procedures* in 3-Lisp[11]. In 3-Lisp, the unevaluated call-time arguments (as in **fexprs** of the old-fashioned Lisp), call-time environment, and call-time continuation can be accessed in arbitrary place/time using reflective procedures. The triple (arguments, environment, and continuation) represents the "snapshot" of a serial computation of 3-Lisp.

The number of the formal parameters of a reflective function in ABCL/R is always five, and they are bound to the list of call-time (unevaluated) arguments, the call-time environment (as an object), the call-time continuation (as an object), the caller object which has invoked the reflective function, and the evaluator object, respectively.

As an example of the use of (user-defined) reflective functions, let us look at the the following definition of a reflective function. This function is actually a definition of a now-type message transmission, namely, the invocation of this function, (now-send T M), is equivalent to the execution of [T <== M].

```
(define (now-send args env cont caller eval) reflect
  [eval <= [:do-seq args env caller] @
        [cont [Target Message]
          [[meta Target]
              <= [:message Message cont caller]]]]])
```

The evaluation of the form (now-send T M) is performed at the level of the evaluator as in 3-Lisp. Let E_x be an evaluator object of an object x. Since E_x is an object, there is a meta-object $\uparrow E_x$. So $\uparrow E_x$ has an evaluator, and it is an evaluator of E_x, namely E_{E_x}. If the above expression is invoked as the part of a script of x, then the formal parameters are bound to the following values: args=(T M), env=*environment object in* $\uparrow$x, cont=*continuation object*, caller=x, and eval=E_x. First, the value of the args is evaluated and the elements of the result is bound to Target and Message. Then a message containing Message, cont and caller is sent to the meta-object of Target. Note that the abbreviation form [cont ...] explained in 4.2.2 is used.

6 Reflective Programming in ABCL/R

In this section, we will present several characteristic examples of reflective programming in ABCL/R. First, we explain the basic methods for dynamically modifying objects. Then we will show that the dynamic acquisition (or dynamic "inheritance") of scripts from other objects are concisely programmed at the user-level by using the means of dynamic modification. Furthermore, we illustrate how an object can monitor other concurrently running object's behavior. In this example, the meta-object of the meta-object of an object is involved. Also we will briefly explain the implementation of the timewarp mechanism[6] using reflective features of the language ABCL/R. The reader should be reminded that all the computations illustrated by these examples are performed in the framework of *concurrent* computation.

6.1 Dynamic Modification of Objects

As we have seen, the internal structure of an object can be manipulated as data in the meta-object of the object. In the default meta-object of an object, some special scripts which manipulate the internal structure of the denotation object (queue, scripts, state, and evaluator) are provided. For example, the following messages can be acceptable by the default meta-object.

- [:add-script s] : Adds a new script s to the denotation object of the target meta-object.
- [:script m] : Returns a script whose message pattern matches m.
- [:delete-script m] : Deletes a script whose message pattern matches m.
- :state : Returns the object which represents the state memory of the denotation of the target object.

Let us look at how these messages are used. First, to add a new script to the object x:

```
[[meta x] <== [:add-script '(=> [:foo X] body-of-script)]]
```

Now x can accept messages that match the pattern [:foo X]. Before adding this script, if x already has a script whose pattern matches [:foo X], this newly added script is used instead of the old one. But the old script still remains and when the new one is deleted, the old one will be used again.

```
[[meta x] <== [:script [:foo 1]]]
    ⟹ (=> [:foo X] body-of-script)
```

(The right hand side of "⟹" is the value of the expression on the left hand side.) The result is the script added before. The execution of the following form deletes it.

```
[[meta x] <== [:delete-script [:foo 1]]]
```

In addition, it is possible to access the object which represents the state memory by:

```
[s := [[meta x] <== :state]]
```

Variable s is bound to the state memory of x represented as the state object of x. To know the value of a variable, a message [:value *variable-name*] is used as follows.

```
[s <== [:value 'X]]  ⟹  1
```

In this example, the value of the state variable X of x is 1. To create a new variable binding in the state memory, the following will do.

```
[s <== [:add-binding 'Y 100]]
```

Then x has a new state variable Y with its value being 100. If the binding of Y already exists before adding, the old one is hidden by the new binding. The old binding remains but cannot be accessed until the new one is deleted.

Using these special scripts of the default meta-objects and state objects, we can write the code to modify the scripts and the state memory of an object dynamically, and such modification can be done while the object being modified is running. The examples described below use these special scripts effectively.

6.2 Dynamic Acquisition (Inheritance) of Scripts

Suppose an object x has received a message M, but x does not have any script for M. If x has the following script:

```
(=> message-pattern-for-M @ reply-var from sender-var
    (inherit msg-pattern-for-M reply-var sender-var y t))
```

x can inherit (acquire) the script for the message M from another object y. (What really happens when the above script is executed is: $\uparrow x$ gets the script dynamically from $\uparrow y$ and then $\uparrow x$ starts execution with the environment (state memory and evaluator) of x as if the script were x's local one.)

inherit is a reflective function whose caller object acquires (inherits) scripts from a specified object. The first, second, and third arguments of inherit are the message, reply destination, and the sender, respectively. The fourth argument is the source of inheritance, which is an object (y) from which the caller (x) inherits a script. If the last argument is a non-nil value, the script inherited is stored in the caller object as its own script. Then the caller object can process the subsequent messages of the same pattern using the newly acquired script, and now it doesn't need to inherit the script for the same message pattern. The following is the definition of the function inherit.

```
(define (inherit args env cont caller eval) reflect
  [eval <= [:do-seq args env caller] @
    [cont [Message Reply Sender Inherit-Source Cache?]
      (let ((scr [[meta Inherit-Source]
                  <== [:script Message]]))
        (if scr then
           [eval <= [:do-prg (scr$body scr)
                     [env-gen
                       <== [:new (script-alist
                                   [Message Reply Sender]
                                   scr) env]]
                     caller]
                 @ [cont Value
                     (if Cache? then
                        [[meta caller]
                          <== [:add-script scr]])
                     [[meta caller] <= :end]]]
         else
           (warn "INHERIT: ~A ~A" Message Inherit-Source)
           [[meta caller] <= :end]))]])
```

First, all the call-time arguments are evaluated by the evaluator object eval, and the values are bound to Message, Reply, Sender, Inherit-Source, and Cache?. The message [:script ...] explained above is used to try to get, from the source of the inheritance (the value of Inherit-Source), a script whose pattern matches the message (the value of Message). If found, the body of the script is evaluated using the environment of the caller object as if it were the caller's local one.

Let us look at a simple example of using inherit. Objects bird and emu are defined as follows:

```
[object bird
  (script
    (=> :has-feather?  !t) ; returns t
    (=> :can-fly?  !t)     ; returns t
    (=> Any @ Reply from Sender
        (inherit Any Reply Sender animal nil)))]

[object emu
  (script
    (=> :can-fly?  !nil)   ; returns nil
    (=> Any @ Reply from Sender
        (inherit Any Reply Sender bird nil)))]
```

These objects model simple knowledge of birds and emus. Since an emu is a bird, the object emu inherits all the scripts from bird except for :can-fly?. In the second script of emu, the single pattern variable Any can match any messages. When a message :can-fly? is sent to emu, it answers using its local script. In the case of :has-feather?, emu inherits the script from bird, and answers using it.

In this example, the fifth argument of inherit is nil. So emu can always answer correctly being consistent with the changes made to the definition of bird changes, because the scripts acquired are not cached in emu.

The function inherit will be used in the examples below. It should be noted that the object-based inheritance scheme in [4] and the proxy-query inheritance in [2] can easily be implemented using our inheritance scheme.

6.3 Monitoring Running Objects

The behavior of an object can be monitored from outside through its meta-object. For example, let us consider how an object can monitor what message have been received by a specified object while the specified object is running. Below we will show simple reflective programming in ABCL/R implements this monitoring facility.

Let Monitor be an object which monitors messages accepted by an object x. To do so, Monitor modifies the behavior of x so that whenever x *accepts* a message m from s with reply destination r, x sends a message [:has-accepted m r s] to Monitor. See Figure 3.

To start monitoring of x, the following will do:

```
[Monitor <= [:monitor x]]
```

Now, whenever x accepts a message m with reply destination r from s, Monitor receives a message [:has-accepted m r s]
To stop this monitoring:

```
[Monitor <= :stop-monitoring]
```

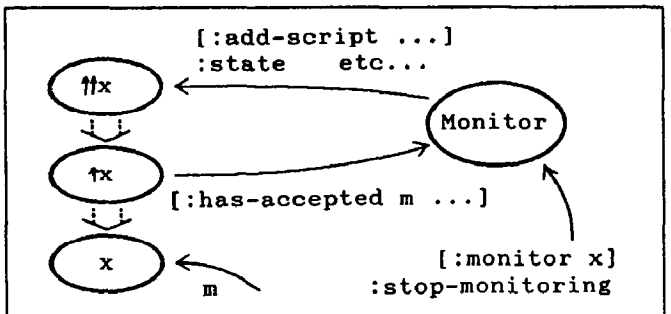

Figure 3: The Monitor Object

We can start/stop monitoring whenever we want — even when the object being monitored, namely the subject of monitoring, is executing its scripts. The inherent concurrency explained in 4.2.3 guarantees that the meta-object can receive and accept messages when its denotation is executing the scripts.

The definition of the monitor object Monitor is as follows:

```
[object Monitor
  (state subject new-name)
  (script
    (=> [:monitor An-object]
        [subject := An-object]
        [new-name := (gensym)]
        [[meta [meta subject]]
           <== [:add-script
                 '(=> :begin
                      ........
                      (if scr then
                          [,new-name
                              <= [:has-accepted . mrs]]
                      .......))]]
        [[[meta [meta subject]] <== :state]
            <== [:add-binding new-name Me]])
    (=> :stop-monitoring
        [[meta [meta subject]]
            <== [:delete-script ':begin]]
        [[[meta [meta subject]] <== :state]
            <== [:remove-binding new-name]]))]
```

When the monitor object receives the message [:monitor x], the monitor object modifies $\uparrow x$ so that $\uparrow x$ may send the monitored information, namely, a message [:has-accepted m r s] when x accepts a message m (from s with reply r). To do this, the monitor adds a new script for a message :begin through $\uparrow\uparrow x$ (see Figure 1). The new script added is almost equal to the default one (in Figure 1) except that the monitored information is sent to x upon acceptance of a message. In order to refer to the monitor object from x, a new state variable is added in x, and the name of the new variable should not conflict the other variables. Thus (gensym) is used to creates the new variable.

Stopping monitoring is simple. The newly added script and variable bindings are simply removed from x. Then the original script for :begin is used again.

The above definition of Monitor specifies just the framework for monitoring. What to do when a message comes [:has-accepted ...] is not specified in its definition. By using this monitor object, actually by acquiring (inheriting) its scripts, the following simple tracer object can be defined.

```
[object tracer
  (state subject new-name)
  (script
    (=> [:monitor An-object] @ R from S
        (inherit [:monitor An-object] R S Monitor t))
    (=> :stop-monitoring @ R from S
        (inherit :stop-monitoring R S Monitor t))
    (=> [:has-accepted Message Reply Sender]
        (format *trace-window* "~&~S accepts ~S from ~S"
                subject Message Sender)))]
```

The object tracer monitors an object and displays the trace of message acceptances on *trace-window*.

6.4 Time Warp Mechanism

A simple *Time Warp* mechanism based on the *Virtual Time* concept[6] has been implemented using the reflective language constructs in ABCL/R.

Object-oriented concurrent programming offers the natural framework for distributed discrete event simulation. Each entity in the simulation domain is modeled as an object, and events among entities are represented as transmission and reception of messages by such objects. The essential problem in this framework is how to manage the *temporal consistency* among events. Our computation model does not assume the existence of the *global clock*.

In [9], this problem is solved with ABCL/1 using a *rollback* mechanism based on the notion of the virtual time[6]. Messages transmitted by objects (which model or represent simulation entities) explicitly contain *timestamps*, and if *time conflict* is detected by an object (i.e., the timestamp τ of a message is older than the time according to the local clock of the object), the object performs undoing of its execution (rollback) to τ. That is, it sends *anti-messages* to objects to which the object has already sent messages since τ, and undoes the execution so far.

As in [9], this roll back mechanism is usually explicitly specified in the scripts of an object mingled with the description of simulation activities. But this explicit specification of roll back severely decreases the modularity of the simulation program and it is very cumbersome an error-proning because the programmer has to write the code for roll back everywhere necessary in the script.

Since the rollback mechanism (of handling anti-messages and undo operations for state variables) is meta-level to the simulation of activities, our implementation explicitly separates the two levels and describes the general roll back mechanism in the definition of the meta-object of an object doing simulation activities.

To define an object which has the Time Warp mechanism, the meta-object specification facility of ABCL/R can be used as in the following object definition.

```
[object a-simulation-object
  (meta TW-meta-gen)
  (script
    (=> message-pattern @ reply from sender
        description of simulation activities)
    ...)]
```

(meta TW-meta-gen) in the above definition specifies the generator (TW-meta-gen) of the meta-object of *a-simulation-object* explicitly. (The definition of TW-meta-gen is described in Appendix.) When the above expression is evaluated, TW-meta-gen, instead of the default meta-object generator, is actually used in creating a new object.

The Time Warp mechanism is fully handled by the meta-object. Thus the programmer of *a-simulation-object* does not need to write the code for rollback. The Time Warp mechanism part and the simulation part are completely separated. Of course it is possible to use TW-meta-gen for defining of other simulation objects. It can be used as library code. Introducing this type of modularity is an important feature of languages with reflective architecture.

7 Concluding Remarks

7.1 Summary

We designed and implemented an object oriented *concurrent* language ABCL/R which has a *reflective* architecture based on the notion of meta-objects. The following is the summary of our present work.

- Each object is represented/implemented by its *meta-object*.

The meta-object incorporates the meta-level representations of structural and computational aspects of the object in a meta circular way. A meta-object is also an object of ABCL/R. This implies the infinite tower of meta-objects. (For its implementation, see below.) An evaluator (interpreter) of the language is also an object. In our computation system, a number of such objects may work in parallel.

- Reflective computation is performed by *message transmissions to meta-objects* and such message transmissions take place concurrently. Reflective computation can be performed in meta-objects of any level because of the infinite tower of meta-objects. Sending messages to a meta-object makes it possible to inquire and alter the structure and behavior of the object. It is possible to send messages to the meta-object of an object while the object is performing its jobs. Thus, a concurrent system can gradually modify itself by means of objects and (their) meta-objects in the system sending messages each other.

- The dynamic modification of running objects in a concurrent system can be described by using reflective language constructs of ABCL/R. We have presented programming examples of dynamic (concurrent) modification such as acquiring (or inheriting) scripts from other objects, and monitoring a running object by modifying its meta-object through the meta-object of the meta-object.

- Enhancement of program modularity can be attained by using meta-objects. The example of a simple Time Warp mechanism has demonstrated this. In a simulation program using this mechanism, the meta-level part is separated from the object-level part by specifying a non-default meta-object for each simulation object.

7.2 Current Status of ABCL/R

So far, we have built a prototype implementation of ABCL/R written in ABCL/1 (written in Kyoto Common Lisp on UNIX and Symbolics Common Lisp on Symbolics Lisp Machines). All the examples described in the preceding sections are actually tested on this implementation.

The primary concern of implementation is how to represent the infinite tower of meta-objects. In our implementation, meta-objects are created in the lazy way. A meta-object $\uparrow x$ is actually created when the access to $\uparrow x$ takes place — when the evaluator first evaluates an expression `[meta x]`.

7.3 Future Work

This work is our first attempt to build concurrent reflective systems. As we mentioned in Section 3, there are at least two approaches to build the causally connected self representation of an object-oriented concurrent system. To completely represent the whole concurrent system as a single datum is difficult because of the causal connection. To do so, we need a good formalism and an appropriate modeling of a concurrent system as a whole, as well as techniques to establish the causal connection.

A possible approach is that the system is divided into some groups of objects in such a way that each group contains objects that are related each other. Then we describe an approximate (or partial) self representation of the computational aspect of each group and make the representation accessible from all the members of the group.

References

[1] J. Batali. *"Computational Introspection"*. Technical Report AIM-701, Laboratory for Artificial Intelligence, Massachusetts Institute of Technology, 1982.

[2] J. Briot and A. Yonezawa. "Inheritance and Synchronization in Concurrent OOP". In *ECOOP '87 Conference Proceedings*, pages 35–43, 1987.

[3] D. P. Friedman and M. Wand. "Reification: Reflection without Metaphysics". In *Conference Proceedings of Lisp and Functional Programming*, pages 348–355, ACM, 1984.

[4] B. Hailpen and V. Nguyen. "A Model for Object-Based Inheritance". In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 147–164, The MIT Press, 1987.

[5] C. Hewitt. "Viewing Control Structures as Patterns of Passing Messages". *Journal of Artificial Intelligence*, 8(3):323–364, 1987.

[6] D. R. Jefferson. "Virtual Time". *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[7] P. Maes. *"Computational Reflection"*. Technical Report 87-2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[8] P. Maes. "Concepts and Experiments in Computational Reflection". In *OOPSLA '87 Conference Proceedings*, pages 147–155, 1987.

[9] E. Shibayama and A. Yonezawa. "Distributed Computing in ABCL/1". In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 91–128, The MIT Press, 1987.

[10] B. C. Smith. *"Reflection and Semantics in a Procedural Language"*. Technical Report TR-272, Laboratory for Computer Science, Massachusetts Institute of Technology, 1982.

[11] B. C. Smith. "Reflection and Semantics in Lisp". In *Conference Record of the Principles of Programming Languages*, pages 23–35, ACM, 1984.

[12] T. Watanabe. *"Reflection in Object-Oriented Concurrent Systems"*. Technical Report, Department of Information Science, Tokyo Institute of Technology, March 1988.

[13] R. Weyrauch. "Prolegomena to a Theory of Mechanized Formal Reasoning". *Artificial Intelligence*, 13(1,2), 1980.

[14] A. Yonezawa, J. Briot, and E. Shibayama. "Object-Oriented Concurrent Programming in ABCL/1". In *OOPSLA '86 Conference Proceedings*, pages 258–268, 1986.

[15] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. "Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1". In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 55–89, The MIT Press, 1987.

[16] A. Yonezawa and M. Tokoro, editors. *"Object-Oriented Concurrent Programming"*. The MIT Press, 1987.

Appendix: Code for Simple Time Warp Mechanism

TW-meta-gen is the generator of meta-objects in which a simple time warp mechanism is implemented. The structure of an object consists of a *local clock*, an *input message queue*, and an *output message queue*, a set of scripts, a state memory, and an evaluator. The local clock, input/output message queues are implemented as values of variables lvt, input-messages and output-history, respectively.

The arrival of a message is represented as the acceptance of the message (in meta-level) which matches [:message Message Reply-Dest Sender Timestamp] where the argument Timestamp is the timestamp of the message. Messages which match the pattern [:anti-message Message Reply-Dest Sender Timestamp] are *antimessages*.

In this program, the rollback works only for the past type message transmissions. The retrieval of the state value is not implemented. In the script description of an object whose meta-object is created by TW-meta-gen, timestamps must be specified explicitly in message sending expressions like following:

[target <= message @ reply-destination :time *receive-time]*

receive-time is the virtual receive time[6] — the time at which the target object receives the message *message*.

The definition of TW-input-queue-gen, TW-output-history-gen, and TW-evaluator-gen is omitted. See [12] for details.

```
;;; Meta-object generator with TimeWarp mechanism
[object TW-object-gen
  (script
   (=> [:new State-Vars Lexical-Env Scripts & Creation-Time]
       ![object TW-object    ; scope of this name is local to TW-object-gen
          (state [input-queue := [TW-input-queue-gen <== :new]]
                 [output-history := [TW-output-history-gen <== :new]]
                 [state := [state-gen <== [:new State-Vars Lexical-Env]]]
                 [scriptset := Scripts]
                 [evaluator := [TW-eval-gen <== :new]]
                 [mode := ':dormant]
                 [lvt := (or Creation-Time 0)])  ; Local Virtual Time
          (script
           (=> [Message-Type Message Reply-Dest Sender Timestamp]
               where (member Message-Type '(:message :anti-message))
               [input-queue <== [:enqueue [Message-Type Message Reply-Dest Sender Timestamp]]]
               (if (eq mode ':dormant) then
                   [mode := ':active]
                   [Me <= :begin]))
            (=> :begin
               (case [input-queue <== :dequeue]
                 ;; positive messages whose timestamp is equal to or newer than LVT
                 (is [:message Message Reply-Dest Sender Timestamp]
                     where (>= Timestamp lvt)
                     (case (find-script Message scriptset)
                       (is [Message-Pattern Script-Body]  ; a script is found
                           [lvt := Timestamp]
                           [evaluator <= [:do-prg Script-Body
                                                  (newenv Message-Pattern
                                                          [Message Reply-Dest Sender]
                                                          state)
                                                  [den Me] lvt output-queue]
                             @ [cont ignore
                                 [TW-object <= :end]]])
                       (is []                                  ; script is not found
                           (warn "Cannot handle the message: ~S" Mesg)
                           [Me <= :end])))
                 ;; Messages whose timestamp is older than LVT
                 (is [Messate-Type Message Reply-Dest Sender Timestamp]
                     where (< Timestamp lvt)
                     [lvt := Timestamp]
                     [input-queue <== [:rollback-to lvt]]
                     ;; Sending anti-messages
                     (case-loop [output-history <== :last]
                       (is [Message Reply-Dest Target Timestamp]
                           where (> Timestamp lvt)
                           [[meta Target]
                               <= [:anti-message Message Reply-Dest [den Me] Timestamp]]
                           [output-history <== :drop]))
                     [Me <= :end])))
            (=> :end
                (if (not [input-queue <== :empty?]) then
                    [Me <= :begin]
                 else
                    [mode := ':dormant]))))]))]
```