Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming

Satoshi Matsuoka Takuo Watanabe
Akinori Yonezawa
Department of Information Science, The University of Tokyo*

Keywords and Phrases:
Actors, Object-Based Concurrency, Object Groups,
Reflection, Resource Management, Virtual Time

Abstract

The benefits of computational reflection are the abilities to reason and alter the dynamic behavior of computation from within the language framework. This is more beneficial in concurrent/distributed computing, where the complexity of the system is much greater compared to sequential computing; we have demonstrated various benefits in our past research of *Object-Oriented Concurrent Reflective (OOCR)* architectures. Unfortunately, attempts to formulate reflective features provided in practical reflective systems, such as resource management, have led to some difficulties in maintaining the linguistic lucidity necessary in computational reflection. The primary reason is that previous OOCR architectures lack the ingredients for group-wide object coordination. We present a new OOCR language with a *hybrid group reflective architecture*, ABCL/R2, whose key features are the notion of heterogeneous object groups and coordinated management of *group shared resources*. We describe and give examples of how such management can be effectively modeled and adaptively modified/controlled with the reflective features of ABCL/R2. We also identify that this architecture embodies *two* kinds of reflective towers, *individual* and *group*.

1 Introduction

Concurrent and distributed computing embodies multitudes of aspects not present in sequential computing. Various system resources such as computational power, communication, storage, I/O, etc. are naturally distributed and limited, and thus must be managed within the system in a coordinated manner; computational power, for example, is limited by the number of CPUs in the system, and thus scheduling and load-balancing become necessary. Such coordinated resource management (in a broad sense) of the system usually become little manifest at the language level; as a result, its control is only available in a fixed, ad-hoc fashion, with little possibility for user extensibility. This is not favorable, since concurrent/distributed architectures are much more complex compared to sequential ones, and the system must be *open* to structured dynamic extensions/modifications for adapting to new problems and environments.

Here, as were pointed out in [24] and [15], *computational reflection* can be beneficial in order to encompass such tasks within the programming language framework for the following reason: A (strict) reflective system embodies the structure and computational process of itself as appropriate abstractions, called the *Causally-Connected Self-Representation(s) (CCSR)*. By introspecting and altering

*Physical mail address: 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan. Phone 03-812-2111 (overseas +81-3-812-2111) ex. 4108. E-mail: {matsu,takuo,yonezawa}@is.s.u-tokyo.ac.jp

the CCSR, the abovementioned objectives can be realized at the language level, while maintaining the flexibility and portability.

Such uses of reflection are already proposed in distributed OS such as Muse[22], and window systems such as Silica[13]; they are important in that they demonstrate the effectiveness of *reflective techniques*. There, the meaning of 'reflection' is broader — their CCSRs are of the elements of the OS or of the window system, that is, their meta-systems are used to describe the implementation of the OS or the window system, and not the programming language. In the traditional linguistic sense, computational reflection is *linguistically lucid* — the view of the CCSR of language entities is intensional from within the language itself. That is to say, CCSR describes the computational process of the language itself. We would like our reflective architecture to be in the latter sense, thereby being able to achieve the necessary level of abstraction for various elements of CCSR at the language level. For an OOCP language, for example, it would be possible to define monitoring of objects within the language in a portable way, without resorting the underlying implementation details.

Our previous research of reflection[20, 21, 11] in *Object-Oriented Concurrent Programming (OOCP)* proposed several different kinds of *Object-Oriented Concurrent Reflective (OOCR)* architectures. For each OOCR architecture, we demonstrated the its applications to various features in concurrent and distributed computing, such as object monitoring and object migration. However, for wider classes of coordinated resource management, we have found some difficulties in maintaining the linguistic lucidity. The primary reason is that the architectures only had partial ingredients for realizing group-wide coordination of objects. By combining the architectures, we could attain more power — but doing so cannot be done without careful design, in order to maintain the lucidity as much as possible.

This paper demonstrates that linguistic lucidity can accompany reflective operations that practical systems require: First, we discuss the limitations of the previous OOCR architectures with respect to group-wide coordination of objects. Second, we present an OOCR language with a *hybrid group reflective architecture*, ABCL/R2, which incorporates heterogeneous object groups with group-wide object coordination and *group shared resources*. It also has other new features such as *non-reifying objects* for efficiency. We describe how these reflective features of ABCL/R2 allow coordinated resource management to be effectively modeled and efficiently controlled — as an example, we study the scheduling problem of the Time Warp algorithm[7] used in parallel discrete event simulations. Third, we contribute feedback to the conceptual side of OOCR architectures and object groups by (1) showing that (heterogeneous) object groups are not ad-hoc concepts but can be defined uniformly and lucidly, and (2) identifying that hybrid group architectures embody *two* kinds of reflective towers, instead of one: the *individual tower* which mainly determine the structure of each object, and the *group tower* which mainly determine computation.

2 Previous OOCR Architectures

Past research works analyzing and classifying metalevel and reflective architectures include those by Maes[10], Harmelen[18], Ferber[6], and Smith[15]. But so far to our knowledge no work has discussed issues in reflection *particular to concurrency.*

The two aspects of CCSR in OO languages are the *structural aspect*, indicating how objects or group of objects in the base-level and the meta-level are constructed and related, and the *computational aspect*, indicating how meta-level objects represent the computation of the base-level objects. The major key in the distinction of the structural aspect is the notion of *metaobjects*, introduced by Pattie Maes for a sequential OO language 3-KRS in [9]. For brevity, we will only briefly state that a *meta-level object* is an object which resides at the meta-level of the object-level computation as an element of the CCSR, and a *metaobject* is an object which reflects the structural, and possibly also the computational aspect of a *single* object. Note that a metaobject is a meta-level object, while the converse is not necessarily so; furthermore, there could be multiple metaobjects representing a single object.

2.1 Individual-based Architecture

In this architecture, each object in the system has its own metaobject(s) which solely govern(s) its computation. The threads of computation among metaobjects become naturally concurrent. By 'individual-based' we mean that an individual object is the unit of base-level computation that has a meaningful CCSR at the meta-level.

An example of this architecture is ABCL/R[20], a reflective version of ABCL/1[23]. Each object has its own unique metaobject, $\Uparrow x$, which can be accessed with a special form `[meta` x`]`. Conversely, given a metaobject $\Uparrow x$, `[den` $\Uparrow x$`]` denotes the object it represents. Correspondence is 1-to-1, i.e., `[meta [den` $\Uparrow x$`]]` $\overset{\text{def}}{=} \Uparrow x$ and `[den [meta` x`]]` $\overset{\text{def}}{=} x$. The structural aspects of x — a set of state variables, a set of scripts, a local evaluator, and a message queue — are objects held in the state variables of $\Uparrow x$ (Figure 1). The *arrival* of a message M at object x is represented as an *acceptance* of the message `[:message` $M\ R\ S$ `]` at $\Uparrow x$, where R and S are the *reply destination* and the *sender*, respectively. Customized metaobjects can be specified on object creation with the optional form `(meta ...)`. A metaobject has its own metaobject $\Uparrow\!\Uparrow x$, naturally forming an infinite tower of metaobjects (Figure 1). Reflective computation in ABCL/R is via message transmissions to its metaobject and other objects in the tower.

Note that the individual-based architecture is independent from the issue of inter-level concurrency. In a sequential OO-reflective architecture, there is only a single computation thread in the tower of metaobjects. This thread performs the interpretation of a certain level, and a reflective operation causes a 'shift' of this level. By contrast, in ABCL/R, the interpretation of x by $\Uparrow x$ is carefully designed so that the concurrency of the (individual) object activity defined by the the computational model of ABCL/1 — the message reception/queueing, and the execution of the user script — is preserved.

Other examples[1] include Tanaka's Actor language[16], Merling III[5], and X0/R[11].

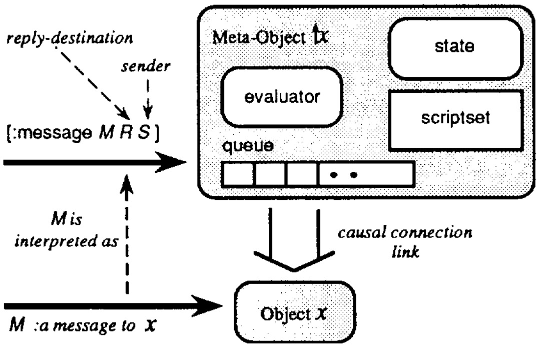

Figure 1: Reflective Architecture of ABCL/R

2.2 Group-wide Reflective Architecture

In this architecture, the behavior of an object is not governed by a single particular metaobject; rather, the collective behavior of a *group* of objects is represented as the coordinated actions of a group of meta-level objects, which comprise the *meta-group*. By 'group-wide' we mean that the entire object group is the unit of base-level computation that has a meaningful CCSR at the meta-level (as

[1]Both Merling III and Tanaka's Actor Language do not employ the term 'metaobject'. However, they do have the notion of some meta-level structure representing the structure and the computation of each object, which we regard as metaobjects in our terminology.

a meta-group); thus, there are *NO* intrinsic meta-relationships between an object and a particular object at the meta-level.

ACT/R[21] is an Actor-based language based on this architecture. The underlying formalism is Gul Agha's Actor model[1]. The meta-architecture of ACT/R is conceptually illustrated in Figure 2. Notice that there are no metaobjects, because the behavior of a single Actor is realized at the meta-level by coordinated action of multiple meta-level Actors. All reflective operations are performed solely via message sends, which are interpreted at the meta-level concurrently with interpretations of Actors at the base level; for various technical details including the faithfulness of the model to the Actor semantics, see [21].

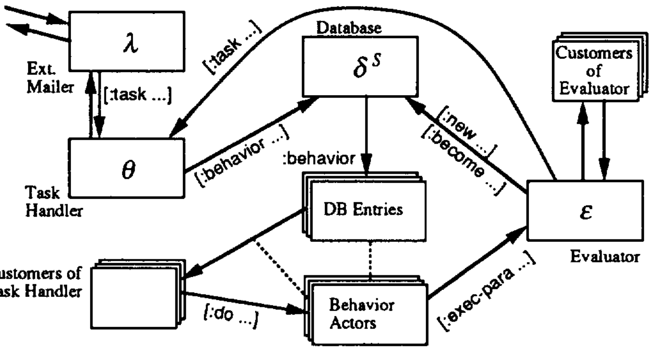

Figure 2: Reflective Architecture of ACT/R

2.3 Limitations of Both Architectures

The limitation of individual-based architectures is that it lacks the 'global view' of computation. Each metaobject is self-contained in a sense that it only controls the computation of its denotation; other objects can only be indirectly introspected or affected through their respective metaobjects. Thus, implicit coordination among the group of objects become difficult.

The limitation of group-wide architectures is that the identity of a base-level object is lost at the meta-level, i.e., identity is not intrinsic to the meta-level, but is implicit. To perform a reflective operation on a particular object, the identity of the object must be constructed explicitly from dispersed objects of the meta-system. As a consequence, what is natural with the individual-based architecture become difficult, for (1) explicit programming is required, and (2) causal connection is expensive to maintain, because it is difficult to obtain the true representation of the current state of the object in the meta-level due to the time delays in the message sends and the concurrent activities of the other parts of the meta-system.

Furthermore, both architectures lack the inherent notion of bounded resources, that is, computation basically proceeds in the presence of an unbounded number of objects representing resources, which would be bounded in real-life. For example, for the individual based, the infinite reflective tower can be constructed for all the objects in the system; for the group-wide, the amount of computation increases by the order of magnitude for the meta-level interpretation, but this is absorbed in the increased parallelism inherent in the basic Actor formalism.

2.4 Hybrid Group Reflective Architecture

In order to overcome the limitations, we propose an amalgamation of both architectures, called the *Hybrid Group (Reflective) Architecture*: In order to preserve the explicit identity and structure of

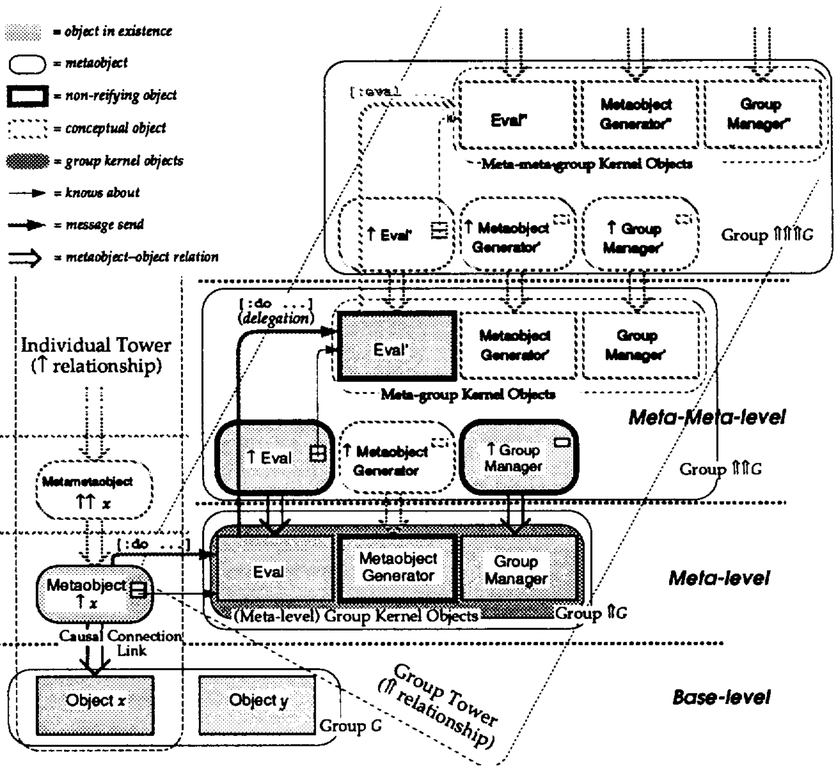

Figure 3: The Individual and Group Reflective Towers in ABCL/R2

objects at the meta-level, we maintain the tower of metaobjects in the same manner as the individual-based architecture. For coordinated management of system resources such as computational resource, we introduce object groups, whose meta-level representation is a group of meta-level objects that are responsible for managing the collective behavior of its member objects. The conceptual illustration of the resulting architecture is given in Figure 3. Note that there are two kinds of reflective towers, the *individual tower* for individual objects and the *group tower* for groups: the details will be described in the ensuing sections.

The hybrid group architecture does not merely combine the benefits of both architectures; the key benefit is that it is possible to model coordinated resource management which were otherwise difficult for previous OOCR architectures.

3 ABCL/R2: A Hybrid Group Architecture Language

ABCL/R2 is our prototype OOCR language with the hybrid group reflective architecture. It is a direct descendent of ABCL/R: each object x has its own meta-object $\uparrow x$, i.e., the unit of CCSR of an object is its metaobject. Also, as in ABCL/R, (1) the message reception and evaluation may proceed concurrently, preserving the ABCL/1 semantics, and (2) conceptually, there is an infinite structural (object-metaobject relationship) reflective tower per each object; the infinite meta-regression is resolved with lazy creation of the metaobject on demand.

3.1 Object Groups in ABCL/R2

The prime new feature of ABCL/R2 is the *heterogeneous object group*, or *group* for short. Members of a group participate in group-wide coordination for the management of system resources allocated to the group. Of special importance is the management of sharing of computational resources, which corresponds to scheduling in concurrent systems.

An object in ABCL/R2 always belongs to some group, with `Default-Group` being the default. A newly created object automatically becomes a member of the *same* group as its creator by default. The restriction is that an object cannot belong to multiple groups simultaneously. At the base-level, the structure of a group is flat in the sense that there are no base-level member objects which perform tasks specific to the group. Rather, analogous to the group-wide reflection, the structure and the computation of a group is explicitly defined at the meta- and higher levels of the group, by the objects called the *group kernel objects*. The group performs management of resources by coordinating among the metaobjects of the members and the group kernel objects.

Groups can be created dynamically with the group creation form as shown in Figure 4. The creation process of a group is not intrinsic, but is given given a concrete metacircular definition with ABCL/R2. As a result, not only that we have the tower of metaobjects, but we also have the tower of *meta-groups* as in ACT/R. We defer the details of group creation until Section 3.2.

```
[group Group-Name   ;;; Group Definition.
  ;; a metaobject generator (required)
  (meta-gen Metaobject-generator)
  ;; a primary evaluator (required)
  (evaluator Evaluator)
  ;; additional resources (optional)
  (resources
     [name := expression]
     ;; example: a scheduler
     [scheduler := [scheduler-gen <= :new]]
                 :
     )
  ;; extra (user definable) scripts definitions (optional)
  (script
    ;; Example: reflective operation to allocate more computational power on request
    (=> [:give-me-more-power Priority]
        ;;; Compute how much computational power can be given to the object.
        ;;;   Assume that the evaluator is extended to have a scheduler.
        [scheduler <= [:give-more-power-to sender computed-amount]])
                :
  )
  ;; initialization expressions
  (initialize
     Initialization-Expressions...
     ;; example: define an initial member of the group
     [object Root ...]
                 :
  )
  ;; initialization for metalevel actors (set-up purpose)
  (initialize-meta
    Initialization-Expressions-for-Metalevel...
    ;; example: notify the initial scheduler
    [[meta evaluator] <= [:set-scheduler scheduler]]
                 :
    )]
```

Figure 4: Group Definition in ABCL/R2

3.1.1 The Group Kernel Objects

The group kernel objects are the CCSR of the group and its management. They are identified with a dark shaded area in Figure 5:

- The *Group Manager* — represents and 'manages' the group. When a group is created, the identity of the group is actually that of the group manager object. It has two state variables holding the mail addresses of the primary evaluator and the primary metaobject generator of the group, but there are no inverse acquaintances. It also embodies the definition of itself for the creation of new groups. Its definition will be described in Section 3.2.

- The *(Primary) Metaobject Generator* — serves as the primary generator of the metaobjects for each member of the group. When a new object is created in the group, its metaobject is always created at the same time. The default metaobject generator of the system, `Metaobject-Generator`, is shown in Figure 6.
- The *(Primary) Evaluator* — represents the shared computational resource of the group. It purpose is to evaluate the methods of member objects. It is no longer a stand-alone, private object as in ABCL/R, but interacts with other group kernel objects and metaobjects for group management. The default definition is given in Figure 7. There could be multiple evaluators per group for parallelism.

Each metaobject of a group member object has state variables in its metaobject containing the mail addresses of group kernel objects for group membership; one is (the address of) the group manager, and the other is the primary evaluator (Figure 5). The group manager object can be accessed from the base-level with the special form `[group-of ...]`. Conversely, the metaobject generator is not directly known to the metaobjects of the group.

3.1.2 The Meta-Group

The group kernel objects are not members of the group they manage, because they reside at the meta-level of the member objects. But since the requirement that all objects belong to a group holds for group kernel objects as well (thus be able to compute in the first place), the *Meta-group* must exist to maintain the linguistic lucidity of reflective architectures. This is similar to ACT/R, where meta-actors comprised the meta-level group. In Figure 3, the metagroup of group G is identified as $\Uparrow G$. The group kernel objects are members of group $\Uparrow G$, while the metaobjects of the group kernel objects are members of $\Uparrow\Uparrow G$, etc. Here, the $\Uparrow$ tower forms a *group tower* distinct from the metaobject towers; we will discussed this in detail in Section 3.4.

In addition to the group kernel objects, there could be other meta-level objects that are members of the meta-group (or higher). In Figure 5, for example, the metaobjects of the evaluator and the group manager, in addition to the *scheduler* object, are members of the meta-meta-group of the base group.

3.1.3 Group Shared Resources

The member objects of a group share *group shared resources*, which are the CCSR of system resources, such as computational resource. The group-wide coordination of resource sharing by the member objects is controlled by the metaobject of each member object, with the aid of the group kernel objects. Coordination of sharing is thus done at the meta-level, and is basically invisible at the base-level. The homogeneity of metaobjects with respect to such coordinated behavior is guaranteed by the metaobjects being generated by the (primary) metaobject generator, which is unique to a group.

By default, the evaluator object is the CCSR of the shared computational resource of the group. By coordinating the evaluation with the metaobjects and the *scheduler* object, we can allocate more computational resources to certain objects in order to achieve higher execution efficiency. Other shared resources could be defined for the purpose of either adding new functionalities to the group, and/or making the implementation details manifest in order to alter some existing behavior. For this purpose, the user specifies a specialized metaobject generator. Examples currently under study include class database objects for distributed class management.

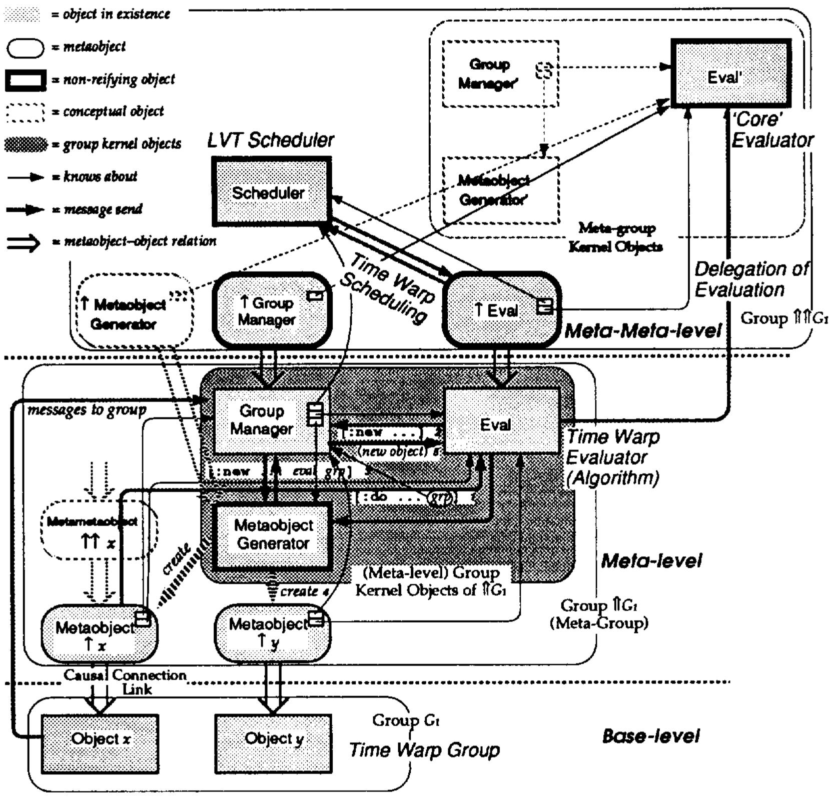

Figure 5: Reflective Architecture of ABCL/R2

3.2 Object and Group Creation in ABCL/R2

In ABCL/R2, the dynamics of object and group creation are manifest in the language; otherwise, the user would not be able to define more sophisticated groups tailored for his particular program and/or hardware architecture. Such extensibility via reflection distinguishes our work from previous works in object or process groups (see [8] for a survey), in which the functionalities of the group, especially its creation, were hard-wired and inalterable.

3.2.1 Object Creation Process

The default group membership of a newly created object is the same as its creator. This can be overridden, however, by explicit designation of a group in the object creation form. Similarly, a specialized metaobject for a particular object can be specified, overriding the primary metaobject generator of that group:

```
[object object-name
    (meta  metaobject generator)      ;; optional, specify alternative metaobject
    (group  group)                    ;; optional, specify alternative group
    ...
```

We show that the behavior results from our reflective architecture. Rather than to force the reader to follow through the details of the code, we give intuitive descriptions of the process of object x creating a new object y in group G_1, as illustrated in the lower part of Figure 5. The labeled message sends in the figure is numbered in correspondence to the explanations below:

```
[object Metaobject-Generator    ;; The 'Vanilla' Primary Metaobject Generator
 (script
  (=> [:new StateVars LexEnv Scripts Evaluator GMgr]
      ![object Metaobject    ;; The name 'Metaobject' is local to this method.
        (state [queue := [queue-gen <= :new]]
               [state := [env-gen <= [:new StateVars LexEnv]]]
               [scriptset := Scripts]
               [evaluator := Evaluator]
               [Group := GMgr]
               [mode := ':dormant])
        (script
         (=> [:message Message Reply Sender]
             [queue <= [:enq [Message Reply Sender]]]
             (when (eq mode ':dormant)
               [mode := ':active]
               [Me <= :begin]))
         (=> :begin
             (match [queue <= :deq]
              (is [Message Reply Sender]
                  (match (find-script Message Reply scriptset)
                    (is [Bindings ScriptBody]
                        [evaluator <=
                          [:do-progn
                              ScriptBody [env-gen <== [:new Bindings state]]
                              [den Me] GMgr] @
                          [cont _
                           [Me <= :end]]]])
                    (otherwise
                     (warn "~S cannot handle the message ~S"
                           [den Me] Message)))))
         (=> :end
             (if (not [queue <== :empty?])
               [Me <= :begin]
               [mode := ':dormant]))
        ;; Methods implementing reflective operations.
        ;; see [20, 23] for details.
        (=> :queue
            !queue)
                 :
        )])
  )]
```

Figure 6: Primary Metaobject Generator in ABCL/R2

1. Object x receives a message, and attempts to evaluate the corresponding script of x which contains an object creation form [object ...]. At the metaobject level, the script, the new environment, and the group (represented by the group manager) are sent to the evaluator object in the [:do ...] message (labeled 1 in Figure 5).

2. When the evaluator encounters the [object ...] form in the script, it sends a [:new ...] message to the target group manager object. If there is an explicit group specification with [group ...], then the target becomes the group manager object of the specified group; otherwise, the target is the group manager that was passed as a parameter in the [:do ...] message in Step 1, causing the group of the new object y to be the same as x (i.e., G_1). In both cases, the evaluator passes the specialized metaobject generator to the target metaobject generator in the message if and only if the object creation form has the (meta ...) option (labeled 2).

3. The group manager sends the [:new ...] message to the metaobject generator. If a specialized metaobject generator is passed in the message from the evaluator, that becomes the target; otherwise, the target is the primary metaobject generator of G_1. In both cases, the group manager passes the evaluator and itself in the message so that the new object would become the member of G_1 (labeled 3).

4. The metaobject generator creates the metaobject of the new object y, and returns it to the group manager of G_1. This metaobject creation would be interpreted as a creation of an object

```
[object Eval   ;; The Primary Evaluator — computational resource for the group.
 (script
  (=> [:do Exp Env Id Gid] @ C    ; Evaluation for a single expression.
      (match (parse-exp Exp)
       ;; Variables
       (is [:variable Var]
           (match Var
            (is 'Me ![den Id])    ; pseudo variable Me
            (is 'Group !Gid)      ; pseudo variable Group
            (otherwise [Env <= [:value-of Var] @ C])))
                 :
       ;; Past-Type Message Transmission
       (is [:send-past Target Message Reply]
           [Me <= [:do-evlis [Target Message Reply] Env Id Gid] @
            [cont [target* message* reply*]
             [C <= nil]
             (if (not (null target*))
               [[meta target*] <= [:message message* reply* [den Id]]])]])
       ;; Now-Type Message Transmission
       (is [:send-now Target Message]  (similar to above, omitted) )
                         :
       ;; Object Creation
       (is [:object-def Name Meta-gen-spec State Script]
           [Me <= [:do Meta-gen-spec Env Id Gid] @
            [cont meta-gen*
             (if (null meta-gen*)
                ;; if a metaobject generator is not explicitly specified, use default.
                [Gid <= [:new State Script] @
                 [cont object
                  (if Name
                    [Env <= [:set Name object] @ C]
                    !object)]]
                [meta-gen <= [:new State Env Script Me Gid] @ C])]])
                :
       )
   ;; composite evaluation messages
   (=> [:do-progn (FirstExp . RestExps) Env Id Gid] @ C
       [Me <= [:do FirstExp Env Id Gid] @
        [cont first* [Me <= [:do-progn RestExps Env Id Gid] @ C]]])
   (=> [:do-progn (LastExp) Env Id Gid] @ C
       [Me <= [:do LastExp Env Id Gid] @ C])
          :
   )]
```

Figure 7: Primary Evaluator in ABCL/R2

at the base-level due to the causal-connection property (labeled 4).

5. The group manager returns the new metaobject $\uparrow y$ to the evaluator, which in turn returns it to $\uparrow x$; this is interpreted at the base-level as x receiving the new y. In the default case as illustrated in Figure 5, y belongs to G_1 — as a result, it shares the computational and other resources with x and other members of G_1 (labeled 5).

3.2.2 Dynamic Group Creation and Bootstrapping

A group is created dynamically at run-time with the evaluation of the group creation form in Figure 4. The name of the group is global; it actually refers to the group manager object. The two required objects are the primary metaobject generator and the primary evaluator of the group. Other shared resources are optional and are defined in the (resources ...) form. Next are the optional user-definable scripts of the group; they become the scripts of the group manager upon group creation. In Figure 4, the user defines a reflective method :give-me-more-power which allocates more computational resource to an object upon request. Finally, there are two initialization forms of the group: the former is the object-level initializer, whose prime purpose is the creation of the initial *fixed members* of the group; the latter is the meta-level initializer for initializing the meta-level objects of the group. Due to the dependency between the meta-level and the base-level objects, the latter is evaluated prior to the former.

The initial bootstrapping of a group is achieved in a manner similar to object creation: when the group creation form is detected, the evaluator object sends the group creation message to the group manager object. The group manager object then evaluates the object creation form (outlined in Figure 8) for the group manager of the new group. It roughly proceeds as follows: The mail addresses of the group shared resources are stored in the state variables, and the user-defined scripts of the group is merged with the default scripts of the new grqup manager. The generated initialization script includes the base-level and the meta-level initialization forms; the former is to be sent to the evaluator of the new group, while the latter is to be executed directly by the new group manager (and is thus evaluated by the meta-group evaluator). The newly created group manager object is then sent the `:initialize` message to start the above initialization sequence.

When an object of group G_1 creates a group G_2, the $\Uparrow G_1$ is identical to $\Uparrow G_2$; in other words, the group kernel objects of G_2 becomes a member of the same group as those of G_1.

```
[[object Group-Manager-Name
  (state [meta-gen := Metaobject-Generator]
         [evaluator := Evaluator]
         [name := expression]                 ;; additional resources
         [scheduler := [scheduler-gen <== :new]]  ;; the scheduler example
                  :
         )
  (script
   ;; initialization (bootstrap) method (automatically generated)
   (=> :initialize
       ;; initialization for metalevel can be directly executed by itself
       Initialization-Expressions-for-Metalevel...
       [[meta evaluator] <= [:set-scheduler scheduler]] ;; example
                  :
       ;; initialization codes must be sent to the new primary evaluator
       ;; Id is unspecified, (for there are no members)
       [evaluator <= [:do-progn '(Initialization-Expressions) global-env NIL Me]]
       )
   ;; default group manager methods (automatically generated)
   (=> :meta-gen
       !meta-gen)
   (=> :evaluator
       !evaluator)
   (=> [:new StateObj Env Script] @ C
       [meta-gen <= [:new StateObj Env Script evaluator Me] @ C])
   (=> [:new StateObj Env Script SpecialEvaluator] @ C
       [meta-gen <= [:new StateObj Env Script SpecialEvaluator Me] @ C])
                  :
   ;; user-defined methods: (the :give-me-more-power example)
   (=> [:give-me-more-power Priority]
       ;;; Compute how much computational power can be given to the object.
       [scheduler <= [:give-more-power-to sender computed-amount]])
                  :
   )]
 <= :initialize]
```

Figure 8: Dynamic Group Creation and Bootstrapping

3.3 Non-reifying Objects

Another new feature of ABCL/R2 is the *non-reifying* object, whose purpose is to attain higher efficiency at the sacrifice for the loss of reflective capabilities. It is created with the following form:

```
[object object-name
    (meta non-reifying-meta)
    ...
```

The behavior of a non-reifying object is almost the same as that of a standard object. The difference is that reflective operations are disallowed — an attempt would result in an error.

The non-reifying object does not have a metaobject, i.e., the metaobject is only of 'hypothetical' existence, prohibiting actual reference to it within the script. (It is possible to have references to metaobjects of other standard objects.) Extensibility, as a consequence, is lost; however, non-reifying objects execute much more efficiently compared to the standard ones. In Figures 3 and 5, non-reifying objects are illustrated with thick borders; notice that they do not have metaobjects that actually exist. In our prototype implementation, a non-reifying object is actually an ABCL/1 object that mimics the interface of metaobjects in ABCL/R2. It runs faster compared to native ABCL/R2 objects due to optimized message handling.

As we see in the figures, some of the meta-level objects are not fully instantiated. This is because the members of the meta-group usually assume default behaviors identical to the ABCL/R objects, and the system thus can avoid the infinite meta-regression using standard techniques. However, the metaobject of the evaluator, for example, is instantiated; this requires that the evaluator of the meta-group be instantiated in order to process *its* evaluation. Other meta-level objects, such as the metaobject generator of $\Uparrow\Uparrow G$, is instantiated lazily at run-time when a new group is created.

3.4 The Two Kinds of Reflective Towers

Before proceeding, let us discuss the relationship between the individual tower and the group tower. The term 'tower' implies that there are some kind of structural relationships between the computations at each level. For example, the reflective tower in LISP is the tower of evaluation[14]. The state of the computation at level n can be given as a triplet data structure $<$ *expression*, *environment*, *continuation* $>$ at level $(n-1)$, which can be reified/reflected at each level.

The reflective tower of an ABCL/R object is an individual tower of object-metaobject relationship $\uparrow$, where each metaobject solely determines the structure of the object it denotes. This is also the case for ABCL/R2as we have seen.

In addition, in ABCL/R2, the meta-group relationship $\Uparrow$, which is structural, forms the group tower as illustrated in Figure 3. This tower parallels the meta-evaluation relationship, which is computational, in the following way: Let G_x be the group of a given object x, and E_x (labeled Eval) be the primary evaluator of G_x, which is a member of $\Uparrow G_x$. Since E_x is an object itself, it needs some computational resource, provided by the primary evaluator E_{E_x} (labeled Eval') of the group G_{E_x} to which it belongs. This group is the meta-group of $\Uparrow G_x$, i.e. $\Uparrow\Uparrow G_x$. Now, E_x has a metaobject, $\uparrow E_x$ (labeled $\uparrow$*Eval*). In our current architecture, we define this object to be also a member of $\Uparrow\Uparrow G_x$, so that the evaluations of both E_{E_x} and $\uparrow E_x$ are performed by the evaluator of the group $\Uparrow\Uparrow G_x$ (labeled Eval"). This forms a homogeneous tower-like structure of meta-groups as seen in the figure.

The above indicates that the reflective towers might not be solely in the direction of the $\uparrow$ relationship, but also in the direction of the $\Uparrow$ relationship. Since this was not manifest in the previous OOCR architectures, we attempt to place some distinctions between them:

- The individual tower mainly determines the structure of the object, including its script. Thus, reflective operations to alter the script is in the domain of the individual tower.

- The group tower mainly determines the group behavior, including the computation (evaluation) of the script of the group members. Thus, changes to have different *interpretations* of the same script are in the domain of the group tower.

The above distinctions correspond to the issue whether a reflective operations would affect the program itself, or affect the *interpretation* of the program. However, we cannot merely say that the individual tower only represents structure, and the group tower only represents computation; for example, we could modify the metaobject via the meta-metaobject so that it would suddenly deadlock after receiving k messages. We need more work to establish more sound conceptual distinctions, and develop the model into a formal one, as has been done for single reflective towers for LISP in the works by Friedman and Wand[19] and by Danvy and Malmkjær ([4], etc.).

Now, there is a choice in the construction of the individual tower of member object x: it can be made either distinct or parallel to the group tower. This corresponds to the issue of the *group membership of metaobjects.* We have deliberately avoided the discussion up to this point, because the membership is dependent on the scheme whereby the meta-circularity is broken. In the current implementation of ABCL/R2, the lazy creation of an metaobject is achieved with *self-reifying metaobject.* An regular ABCL/R2 object x (i.e., not non-reifying), upon evaluation of the form [meta x], does not cause the creation of a new meta-metaobject $\Uparrow x$. Rather, the self-reifying metaobject is essentially 'raised' to become the meta-metaobject, and it is properly initialized so that it would serve as the metaobject of $\uparrow x$. This behavior is unfortunately not currently manifest as CCSR in the current implementation. As a result, the entire tower becomes a member of $\Uparrow G_x$, except for the base level x, as shown in Figure 5. Here, the individual tower is distinct from the group tower, in the sense that there are no correspondences between the meta-levels.

In order for the two towers to be parallel, the metaobject of level n needs to be created by the metaobject generator of the corresponding meta-group of level n. This requires a lazy creation scheme for the group kernel objects upon metaobject creation, which again is not manifest as CCSR in the current implementation. As a future work, we plan to extend the architecture so that the lazy creation of both objects and groups to be manifest, so that the user can have a choice on how to relate the two towers.

4 Reflective Programming in ABCL/R2

Reflective programming in ABCL/R2 is performed in two ways. One is to utilize its metaobject in the same way as ABCL/R, which were described in [20]. Another is to introspect and affect the group-wide coordinated behavior of the group the object is a member of. This is performed with a message to its group, [group-of x] (delivered to its group manager object). The two schemes are not contradictory; in practice, a combination of both schemes is effectively used.

For the remainder of the section, we present an example of computational resource management using reflective programming in ABCL/R2. Management of other resources can be performed analogously.

4.1 Time Warp Scheduling in ABCL/R2

In our previous work with ABCL/R, we have presented an example of how an OOCR architecture can be cleanly implement the *Time Warp* algorithm[7] (also known as the Virtual Time scheme) employed in parallel discrete event simulation. When objects model the entities and the message transmission/reception model the events in the simulation, the Time Warp algorithm serves to maintain the temporal consistency among the events. Consistency management is distributed and optimistic; each object has its own *Local Virtual Time (LVT)* (i.e., there is no global clock), and the messages are timestamped to be compared with the LVT of the recipient. When a conflict is detected, the object performs automatic *rollback* by sending *anti-messages* until it reaches the time just prior to the conflict occurrence.

In our previous implementation, the entire Time Warp algorithm was successfully encapsulated in the within the metaobject of each object, since Time Warp algorithm was meta-level to the execution of the simulation itself. One thing we did not address was the performance issues affected by different scheduling policies. A recent work by Burdorf and Marti[2], however, compared ten non-preemptive scheduling algorithms for the Time Warp algorithm, and discovered that there were orders of magnitude difference in their execution speed for some problems. Thus, we cannot ignore scheduling issues in practice when we implement the Time Warp algorithms with OOCP languages.

Burdorf and Marti made some simple assumptions in their performance measurement; for example, they did not allow interprocessor communication between the schedulers, which is necessary for inter-group load balancing. This is too restrictive for OOCP languages, where inter-scheduler communication would be simple. Also, they did not attempt any adaptive scheduling, that is, to alter the scheduling algorithm dynamically to adapt to better algorithms when excessive rollbacks occur during the simulation. For efficiency, we would like our language to be able to model these

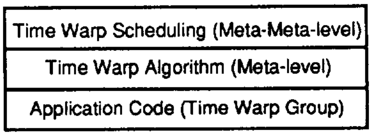
Time Warp Scheduling (Meta-Meta-level)
Time Warp Algorithm (Meta-level)
Application Code (Time Warp Group)

Figure 9: Meta-level Encapsulation of the Time Warp Algorithm

in a clean way; unfortunately, it was not easy with previous OOCP languages for the reasons we discussed in Section 2.3.

With ABCL/R2 we can obtain a clean solution: We define a *Time Warp group*, whose members are specialized with their individual metaobjects so that they coordinate in running the Time Warp algorithm. This is similar to the ABCL/R example, except that group membership automatically dictates Time Warp behavior, not requiring explicit metaobject specification. The actual definition of the Time Warp group are given in the Appendix. Messages sent within the group or to destinations within other Time Warp groups must be of the form:

[*target* <= *message* @ *reply-destination* :vrt *virtual-send-time*]

Since scheduling is meta-level to the execution of the Time Warp algorithm, we would want to encapsulate it in the *meta-level* of the Time Warp algorithm (i.e., meta-meta-level of the actual simulation algorithm), in the same manner that the algorithm itself was encapsulated in the meta-level of the simulation. The conceptual illustration of the encapsulation is given in Figure 9. For implementation, we utilize the group-reflective features of our architecture. We introduce the *Time Warp scheduler* object labeled Scheduler in Figure 5. It is responsible for controlling the allocation of the computational resource within a Time Warp group. For meta-meta-level encapsulation of scheduling, the scheduler does not interact with the evaluator of the Time Warp group; rather, it interacts with the *metaobject* of the evaluator. The metaobject of the evaluator is specialized so that the evaluation request to the evaluator sent from an object in the group is not directly executed, but instead sent to the scheduler. The metaobject then asks the scheduler for the next evaluation job as determined by the algorithm of the scheduler. This behavior is outlined in the abridged code of the evaluator below:

```
[object TW-Eval-meta  ;;; metaobject of the evaluator of the Time Warp group
   (meta non-reifying-meta)
   (state [scheduler := Scheduler])
   (script
    ;; meta-level reception of message to the evaluator
    (=> [:message [KeyWord Expr Env Id Time] R S]
          where (member KeyWord '(:do :do-evlis :do-progn))
        [scheduler <= [:schedule [KeyWord Expr Env Id Time] :with Id Time]]
        (if (eql mode 'dormant)
          [Me <= :begin]))
    (=> :begin
        (match [scheduler <== :next]
              :
        ))]
```

Aside from the behavior specific to Virtual Time, the behavior of the evaluator is almost identical to that of a standard evaluator. The TW-Eval-meta (labeled $\uparrow$Eval in Figure 5) delegates most of the scheduled evaluation requests directly to the evaluator of the meta-group (labeled Eval'). Since Eval' is a non-reifying object, the delegation would terminate there. In effect, Eval' is the sole computational resource for all the members of the group as well as the objects that comprise the group, including the group kernel objects[2]. So, in a sense, Eval' is the native CPU hardware in an operating system; this is a generalization of the conceptual model of reflective operating systems such as Muse[22].

With this framework, dynamic change of the scheduler can be accommodated as given in the Appendix. Furthermore, it would be easy to extend the Time Warp group to add inter-scheduler communication, and/or to have scheduler controlling multiple meta-group evaluators to adapt to growth of computational resource in hardware.

Our plan is to measure the performance of more elaborate versions of Time Warp scheduling algorithms on ABCL/R2. We are not planning real-time benchmarks, however; instead, we plan to simulate the execution of the Time Warp algorithm using parallel discrete event simulation, employing the Time Warp algorithm itself. Treatment of such grossly intricate circularity would be significantly difficult for conventional non-reflective systems, but should be possible with our framework which permit meta-level encapsulation.

5. Discussions and Future Work

5.1 Relationship to Other Works

Here, we discuss the relationships with other works, those in object groups and reflective systems. The usefulness of the concept of an object group has been been widely recognized. But unfortunately, most work on object groups from the language aspect of OOCP has been for homogeneous groups[3, 12]; as for heterogeneous groups, its definition or construction has been mostly vague (for example, [8]). In this work, we have shown that heterogeneous object groups are not ad-hoc concepts, but can be defined constructively and lucidly in an OOCR language, and how cooperative actions of objects in a group with respect to resource management at the base-level can be described in the meta-level architecture. In a sense, our proposal would serve as one reference model for (heterogeneous) object groups.

As for reflective systems, the Muse distributed operating system[22] could be classified as a group-wide architecture, although it has some features of the hybrid architecture. The meta-level objects in the 'meta-space' 'support' the activity of the objects. There are specific meta-level objects responsible for message delivery, scheduling, memory management, etc. The reflective operation is performed by communicating with the meta-space via ports called *reflectors*. Although Muse uses the term 'reflection' in a more loose sense, it nevertheless incorporates many of the ideas from reflection, and thus enjoys their benefits not previously available in the traditional operating systems.

5.2 Current Status of ABCL/R2

The implementation of ABCL/R2 is underway. A subset is almost completely running on the parallel version of ABCL/1 on the IBM TOP-1, a shared-memory multiprocessor machine with ten 80386 CPUs. A program in ABCL/1 is compiled into a program in the parallel version of Common LISP running on TOP-1[17]. This version does not fully implement architecture in this paper, however. We are starting a complete version on Omron LUNA-88k, a multicomputer with four 88000 CPUs running Mach. We are also planning a distributed implementation on an Intel iPSC/2 Hypercube computer, based on the new distributed implementation of ABCL/1 in progress.

5.3 Future Work

Our work is by no means complete or our proposal ultimate. There are still some limitations with ABCL/R2 which we must strive to solve. For example, there is a difficulty in the management of two distinct resources exhibiting collaborative behaviors; this is necessary for realization of features proposed in advanced operating systems, where the virtual memory management coordinates with

[2]The situation would be analogous for multiple evaluators.

the thread scheduler. We could extend our architecture further, and/or make an approach from the group-wide reflection to the individual-based. In either cases, we feel that works in OOCR languages cannot avoid having strong emphasis on the architectural issues, for it is not the language but the language architecture that would contribute the most in solving the problems in practice. The search must go on for even more effective OOCR architectures.

6 Acknowledgements

We would like to thank Daniel Bobrow, Shigeru Chiba, Brian Foote, Mamdouh Ibrahim, Yutaka Ishikawa, Gregor Kiczales, Brian Smith, Jiro Tanaka, Tomoyuki Tanaka, Mario Tokoro, and numerous other people, the discussions with whom truly inspired us. We would also like to thank the comments from the anonymous referees which helped us to clarify and organize our work. Finally, we thank the members of the ABCL project group for their assistance with our everyday research.

References

[1] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[2] Christopher Burdorf and Jed Marti. Non-preemptive time warp scheduling algorithm. *Operating Systems Review*, 24(2):7–18, April 1990.

[3] Andrew Chien and William J. Dally. Concurrent aggregates. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 187–196. SIGPLAN Notices, March 1990.

[4] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in the reflective tower. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 327–341. ACM Press, 1988.

[5] Jacques Ferber. Conceptual reflection and Actor languages. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 177–193. North-Holland, 1988.

[6] Jacques Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings of OOPSLA'89*, volume 24, pages 317–326. SIGPLAN Notices, ACM Press, October 1989.

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

[8] Luping Liang, Samuel T. Chanson, and Gerald W. Newfeld. Process groups and group communications: Classifications and requirements. *IEEE Computer*, pages 56–66, February 1990.

[9] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA'87*, volume 22, pages 147–155. SIGPLAN Notices, ACM Press, October 1987.

[10] Pattie Maes. Issues in computational reflection. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 21–35. North-Holland, 1988.

[11] Satoshi Matsuoka and Akinori Yonezawa. Metalevel solution to inheritance anomaly in concurrent object-oriented languages. In *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[12] Flavio De Paoli and Mehdi Jazayeri. FLAME: a language for distributed programming. In *Proceedings of the 1990 IEEE International Conference on Programming Languages*, pages 69–78, 1990.

[13] Ramana Rao. Implementational reflection in Silica. In *Proceedings of ECOOP'91*. Springer-Verlag, July 1991.

[14] Brian C. Smith. Reflection and semantics in Lisp. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 23–35. ACM Press, 1984.

[15] Brian C. Smith. What do you mean, meta? In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[16] Tomoyuki Tanaka. Actor-based reflection without metà-objects. Technical Report RT-0047, IBM Research, Tokyo Reserach Laboratory, August 1990.

[17] Tomoyuki Tanaka and Shigeru Uzuhara. Multiprocessor Common Lisp on TOP-1. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, 1990. (to appear).

[18] Frank van Harmlen. A classification of meta-level architectures. In Abramson and Rogers, editors, *Meta-Programming in Logic Programming*, chapter 5, pages 103–122. The MIT Press, 1989.

[19] Mitchell Wand and Danel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 111–134. North-Holland, 1988.

[20] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of OOPSLA '88*, volume 23, pages 306–315. SIGPLAN Notices, ACM Press, September 1988.

[21] Takuo Watanabe and Akinori Yonezawa. An actor-based metalevel architecture for group-wide reflection. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL), Noordwijkerhout, the Netherlands*, Lecture Notes in Computer Science. Springer-Verlag, May 1990.

[22] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In Stephen Cook, editor, *Proceedings of ECOOP'89*, pages 89–106. Cambridge University Press, 1989.

[23] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. Computer Systems Series. The MIT Press, 1990.

[24] Akinori Yonezawa and Takuo Watanabe. An introduction to object-based reflective concurrent computations. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, volume 24, pages 50–54. SIGPLAN Notices, ACM Press, April 1989.

A Appendix: Definition of the Time Warp Group

Figures 10, 11, and 12 are the skeletal definition of the Time Warp group. The scheduling algorithm is the Lowest LVT (Local Virtual Time) First scheduler[2], but it can be interchanged dynamically with any valid Time Warp scheduler.

```
;;; The Time Warp group
[group TW-group
 (meta-gen TW-meta-gen)
 (evaluator [TW-eval-gen <== [:new Lowest-LVT-First-Scheduler]])
 ;; ... additional scripts & initialization expressions here ...
 ]

;;; The Lowest Local Virtual Time First Scheduler
[object Lowest-LVT-First-Scheduler
 (state [queue := [priority-queue-gen <== [:new :test-fun #'<]]])
 (script
  (=> [:schedule Message :with Id LVT]
      (match Message
       (is [:anti-message . ARGS]
           ;; Anihiration of the positive of this anti-message.
           (if [queue <== [:have? [:message . ARGS]]]
               [queue <= [:remove [:message . ARGS]]]
               [queue <= [:enq Message :with LVT]]))
       (otherwise
        [queue <= [:enq Message :with LVT]])))
  (=> :next @ C
      [queue <= :deq @ C])
  (=> :contents @ C
      [queue <= :listify @ C])
  (=> [:copy List]
      [queue <= [:enq-list List]])
  )]
```

Figure 10: The Time Warp Group

```
[object TW-Meta-Gen   ;;; The Metaobject Generator of the Time Warp group
 (script
   (=> [:new StateVars LexEnv Script Evaluator GMgr]
       ![object TW-meta
         (state [queue := [queue-gen <== :new]]
                [pqueue := [priority-queue-gen <== :new]]
                [state := [undoable-env-gen <== [:new StateVars LexEnv]]]
                [output-history := [output-history-gen <== :new]]
                [scriptset := Script]
                [evaluator := Evaluator]
                [group := GMgr]
                [mode := ':dormant]
                [LVT := 0])
         (script
           ;; Ordinary messages (omitted)
                :
           ;; TimeWarp messages
           (=> [Type Message Reply Sender VST VRT]
                  where (member Type '(:message :anti-message))
               [pqueue <= [:enq [Message Reply Sender VST VRT] :with VRT]]
               (when (eq mode :dormant)
                 [mode := ':active]
                 [Me <= :begin-tw]))
           (=> :begin-tw
               (if [queue <== :empty?] ; check the ordinary message queue first
                   (match [pqueue <== :next]
                     (is [Message Reply Sender VST VRT] where (>= VRT LVT)
                         [state <= [:push LVT]]
                         [LVT := VRT]
                         (match (find-script Message scriptset)
                           (is [Vars Body]
                               [evaluator <= [:do-progn Body
                                                        [env-gen <== [:new Bindings state]]
                                                        [den Me] GMgr output-history LVT] @
                                              [cont ignore
                                                    [Me <= :end]]]])
                           (is NIL
                               (warn "~A cannot handle the message: ~S"
                                     [den Me] Message)
                               [Me <= :end])))
                     (is [Message Reply Sender VST VRT] where (< VRT LVT)
                         ;; State rolls back itself to the most recent time
                         ;; before VRT and returns the value of the time.
                         [LVT := [state <== [:rollback-to VRT]]]
                         ;; Input queue for TimeWarp messages shold also be rewinded
                         [pqueue <= [:rollback-to VRT]]
                         ;; Send anti-messages.
                         (dolist (h [output-history <== [:history-since VRT]])
                           (match h
                             (is [Target Message Reply VST VRT]
                                 [Target <= [:anti-message Message
                                                   Reply [den Me] VST VRT]]))))))
                   [Me <= :begin]))
           (=> :begin
               ... same as 'vanilla' meta objects ...)
           (=> :end
               (if (not [queue <== :empty?])
                   [Me <= :begin]
                   (if (not [pqueue <== :empty?])
                       [Me <= :begin-tw]
                       [mode := ':dormant])))
         )])
  )]
```

Figure 11: Group Kernel Objects of the Time Warp Group (1)

```
[object TW-Eval-gen       ;;; The evaluator of the Time Warp group
 (script
   (=> [:new Scheduler]
       [object TW-eval
        (meta TW-eval-meta-gen)
        (script
         (=> [:do Exp Env Id Gid Outputs LVT] @ C
             (match (parse-exp Exp)
                      :
             (is [:variable Var]   ; variables and pseudo-variables
                 (match Var
                  (is 'Me ![den Id])
                  (is 'Group !Gid)
                  (is 'LVT !LVT)
                  (otherwise [Env <= [:value-of Var] @ C])))
                  :
             (is [:send-tw-mesg Target Message Reply VRT]
                 [Me <= [:do-evlis [Target Message Reply VRT] Env Id Gid Outputs LVT] @
                  [cont [target* message* reply* vrt*]
                   [C <= ()]
                   (if (not (null target*))
                     (progn
                       [[meta target*] <= [:message message* reply* LVT vrt*]]
                       [Outputs <= [:push [target* message* reply* LVT vrt*]]]
                       ))]])
                      :
             (is [:object-def Name Meta-gen-spec State Script]
                 [Me <= [:do Meta-gen-spec Env Id Gid Outputs LVT] @
                  [cont meta-gen*
                   (if (null meta-gen)
                     [Gid <= [:new State Script] @
                      [cont object
                       (if Name
                         [Env <= [:set-global Name object] @ C]
                         !C)]])]])
             )))]
       [[meta TW-eval] <= [:set-scheduler Scheduler]]
       !TW-eval
   ))]
[object TW-Eval-meta-gen    ;;; The special metaobject for TW-Eval
 (script
  (=> [:new StateVars LexEnv Scripts Evaluator GMgr]
      ![object Eval-Meta
        (meta non-reifying-meta)
        (state [scheduler := nil]
               [evaluator := Evaluator]
               [scriptset := Scripts]
               [mode := ':dormant])
        (script
         (=> [:message Message Reply Sender]
             (match Message   ; = [:do Exp Env Id Gid VRT]
              (is [_ _ _ Id _ _]
                  [scheduler <= [:schedule Message :with Id Time]]))
             (if (eql mode ':dormant)
               (progn [mode := ':active]
                      [Me <= :begin])))
         (=> :begin
             ... almost same as 'vanilla' meta objects ...)
         (=> :end
             ... same as 'vanilla' meta objects ...)
         ;; Additional methods
         (=> [:set-scheduler NewScheduler]
             [scheduler := NewScheduler])
         (=> [:change-scheduler NewScheduler]
             [NewScheduler <= [:copy [scheduler <== :contents]]]
             [scheduler := NewScheduler])
         )]))]
```

Figure 12: Group Kernel Objects of the Time Warp Group (2)