To appear in Proc. REX School/Workshop on Foundations of Object-Oriented Languages, Noordwijkerhout, the Netherlands, May, 1990, Lecture Notes in Computer Science (forthcoming)

An Actor-Based Metalevel Architecture for Group-Wide Reflection

Takuo Watanabe*
Department of Information Science
Tokyo Institute of Technology
JSPS Fellow (DC)
takuo@is.s.u-tokyo.ac.jp

Akinori Yonezawa
Department of Information Science
The University of Tokyo
yonezawa@is.s.u-tokyo.ac.jp

Abstract

The notion of *group-wide reflection* is presented. Group-wide reflection, a dimension of computational reflection in concurrent systems, allows each computational agent (actor/object/process) to reason about and act upon not only the agent itself, but also a group of agents which may contain the agent itself. Global properties of the group can be dynamically controlled through group-wide reflection. We have developed a simple yet general model for group-wide reflection based on the Actor model[1]. An operational semantics of a group of object-level actors is represented by another group of actors (a group of metalevel actors), which is an implementation of a transition system of the object-level group. We prove that the metalevel group correctly represents the operational semantics of the group in terms of transitions of configurations. Furthermore, migration of an actor from node to node is described as an example of group-wide reflection.

Keywords

reflection, group-wide reflection, actor model, object-oriented concurrent programming, object-group, ABCL/R, ABCL/1

1 Introduction

Our research goal is to construct a solid basis for programming adaptive/self-evolving computational systems. We believe that object-oriented concurrent computation with reflection provides a suitable framework for this goal.

The motivation of group-wide reflection is to cope with the following issues.

*Contact Address: Department of Information Science, University of Tokyo, 7-3-1, Hongo, Bunkyo-ku, Tokyo, JAPAN, 113, TEL/FAX: +81-3-5689-4365

1. *Object Group and Group Communication*:
   The notion of *object (process) group* offers a powerful abstraction mechanism for construction of large systems. Using a metalevel architecture, we like to make experiments on mechanisms such as *group communications*[5], dynamic (run-time) construction/destruction/fusion and migration of groups, and so on.

2. *Dynamic Modification of the Semantics of Message Delivery*:
   It is sometimes useful to have different semantics of message delivery in different groups of objects. If we want to implement the group-communication stated above efficiently, group-wide adaptation of message delivery semantics is often required. For example, consider the situation of using Timewarp mechanism[4] in a group. Changing the order of messages delivery by their timestamps makes it possible to have an efficient message scheduling which avoids frequent rollbacks and anti-messages.

3. *Maintaining Constraints among Objects*:
   Mutually constrained objects form a group. To maintain the constraints among them, controlling metalevel properties often offers a better means than ordinary message passing among the objects.

4. *Modeling Implementation-Related Features*:
   In an actual multicomputer concurrent system, there are some limited ways to obtain/modify global information of its subsystems. For example, in a coarse grained multi-processor system (*i.e.*, more than one object/process may run on each processor in a pseudo-parallel manner), we can obtain and/or control various global dynamic characteristics of objects/processes running in a single processor (*e.g.*, execution scheduling, allocation of resources, control of the messages traffics and so on). In other words, the machine-boundary of a processor inherently forms a group. Of course, such operations on the group have been utilized in actual systems (especially, operating systems), but they are usually introduced in either an ad-hoc or quite limited way ([3, 9]). We need a good formalism for uniform treatment of such implementation models.

In our previous approach to reflective computation in an object-oriented concurrent language ABCL/R[8], we adopted *individual-based* reflection: *i.e.*, each agent (object) can reason about and act upon the agent itself. In ABCL/R, a *meta-object*, which serves as the causally-connected metalevel representation of an object, is introduced for each object. A meta-object represents (implements) the structural/computational aspects of an object, and reflective computation is realized by sending messages to meta-objects. Since the computational activity in an object is sequential, technique for constructing a sequential reflective system can be used. This implies, however, each meta-object models only the local sequential aspects of an object. Thus, our previous approach is not sufficiently powerful to deal with the global information of a group of objects.

To realize a full-fledged computational reflection on a group of objects, a formal model of the group is needed. The model must represent the correct semantics of the group and should be accessible from members of the group. In addition, more importantly, the model and the group should be causally-connected: the model always represents (reifies) the current status of the group, and the changes made to the model should correctly reflect in the behavior of the group.

We use the *transition system* given by Gul Agha[1] for describing the operational semantics of a group of actors. In our approach, the transition system of a group is represented as another actor group which forms a concurrent meta-circular interpreter. We prove that it correctly represents the operational semantics of the group in terms of the transition of configurations.

2 Metalevel Architecture

In order to realize the group-wide reflection, we compose a group of actors and its metalevel description. The latter is also a group of actors. Suppose that S is a group of actors. We let $\uparrow S$ denote the group which forms a metalevel representation (meta-circular interpreter) of S. Actors in two groups can communicate with each other (*inter-level communication*). Since actors in $\uparrow S$ maintains the global information of S, we can access/modify the global metalevel information of a group through sending messages to actors in $\uparrow S$, and vice versa. Of course, the meta-circularity of $\uparrow S$ guarantees the causal connection between S and $\uparrow S$[6].

2.1 Configurations

Before presenting our metalevel description for an actor system, we briefly give a formalism for Actor model. It is basically the same as Agha's formalism. See [1] for more details.

An actor α is a pair $\langle m, \beta\rangle$ of its *mail address* m and its *behavior* β. The domain of actors $\mathcal{A}$ is defined as $\mathcal{A} = \mathcal{M} \times \mathcal{B}$, where $\mathcal{M}$ and $\mathcal{B}$ are the domains of mail addresses and behaviors, respectively. Note that in one system, no two actors have the same mail address.

A *task* is a triple $\langle t, m, k\rangle$ of a tag t, a mail address m and a message value k. A task represents a message which has been sent to an actor (a target actor) whose mail address is m, but which has not been received. The tag t is needed to distinguish two tasks having the same target address and the same message value The domain of tasks is defined as $\mathcal{T} = \mathcal{I} \times \mathcal{M} \times \mathcal{V}$, where $\mathcal{I}$ and $\mathcal{V}$ are the domains of tags and values used in messages. $\mathcal{V}$ is the disjoint sum of domains of mail addresses and other first class primitive values (such as numbers, lists, etc.).

The computation carried out by α in response to a message is described as the result of function application $\beta(t, k) = \langle T, A, \beta'\rangle$, where T is the set of tasks created by α, A is the set of actors created by α, and β' is the replacement behavior. The domain of behaviors $\mathcal{B}$ is defined as:

$$\mathcal{B} = \mathcal{I} \times \mathcal{V} \to F_s(\mathcal{T}) \times F_s(\mathcal{A}) \times \mathcal{B}$$

where $F_s(X)$ is the set of all finite subdomains (subsets) of X.

Let S be a system composed of actors. A *configuration* C represents a computational state of S at a certain frame of reference. This is represented by a pair $\langle \mathcal{A}(C), \mathcal{T}(C)\rangle$, where $\mathcal{A}(C) \in F_s(\mathcal{A})$ is a finite set of actors in S (called the *population* of C) and

$\mathcal{T}(C) \in F_s(\mathcal{T})$ is a finite set of *tasks* in S. Note that $\mathcal{A}$ and $\mathcal{T}$ are the domains of actors and tasks respectively. We let Γ_S denote the set of all configurations of S.

Computation in S is modeled as transitions between configurations in Γ_S. When a task $\langle t, m, k\rangle \in \mathcal{T}(C_1)$ is processed in $C_1 \in \Gamma_S$, the transition which results in a new configuration C_2 is denoted by:

$$C_1 \xrightarrow{\langle t,m,k\rangle} C_2$$

The configuration C_2 represents the situation where the actor having mail address m has just finished its computation for the task $\langle t, m, k\rangle$. If the target actor is in S, the following holds.

$$\begin{array}{l}
\langle t,m,k\rangle \in \mathcal{T}(C_1) \wedge \exists\beta \in \mathcal{B}.\ s.t.\ \langle m,\beta\rangle \in \mathcal{A}(C_1) \\
\mathcal{A}(C_2) = (\mathcal{A}(C_1) - \{\langle m,\beta\rangle\}) \cup A' \cup \{\langle m,\beta'\rangle\} \\
\mathcal{T}(C_2) = (\mathcal{T}(C_1) - \{\langle t,m,k\rangle\}) \cup T' \\
\textit{where}\ \ \beta(t,k) = \langle T', A', \beta'\rangle
\end{array}$$

When the target is in another system S', then the configuration C'_1 of S' will change to C'_2.

$$\begin{array}{l}
\mathcal{A}(C_2) = \mathcal{A}(C_1), \mathcal{T}(C_2) = \mathcal{T}(C_1) - \{\langle t,m,k\rangle\} \\
\mathcal{A}(C'_2) = \mathcal{A}(C'_1), \mathcal{T}(C'_2) = \mathcal{T}(C'_1) \cup \{\langle t, f(m), f(k)\rangle\}
\end{array}$$

The function $f : \mathcal{M} \rightarrow \mathcal{M}$ translates a mail address in S to one in S'.

We write $C_1 \xrightarrow{*} C_k$ $(k \geq 1)$ when the following holds:

$$\begin{array}{l}
C_1 \xrightarrow{\tau_1} C_2 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_{k-1}} C_k \\
\text{where } \exists\tau_i \in \mathcal{T}(C_i)\ (i = 1, \cdots, k-1)
\end{array}$$

The sequence above is called a *possible transition sequence.*

2.2 Metaconfiguration

Now we construct a causally connected metalevel representation for an actor system S. The metalevel representation for S, which is denoted by $\uparrow S$, is constructed as another actor system which implements the transition system of S. $\uparrow S$ is an actor system which represents the *concurrent* computational aspects of S, namely, $\uparrow S$ represents an operational semantics for S. In our model, $\uparrow S$ is also used as a meta-circular interpreter for S. This implies that the causal-connection between S and $\uparrow S$ is automatically guaranteed.

We show that $\uparrow S$ correctly models the semantics of S in terms of transitions between configurations. Notice that $\uparrow S$ to be constructed here is just one case of many possible metalevel representations.

As seen above, to define an actor system, it is not enough to pick up actors in the system — we need to define its configuration. We now define a special configuration of $\uparrow S$, called a *metaconfiguration.* A metaconfiguration is a configuration which *models* a particular configuration of S. We let $\uparrow C$ denote a metaconfiguration which models a configuration $C \in \Gamma_S$. We assume that the initial configuration of $\uparrow S$ is always a metaconfiguration. Below, $\Gamma_{\uparrow S}$ denotes the set of all possible configurations of $\uparrow S$.

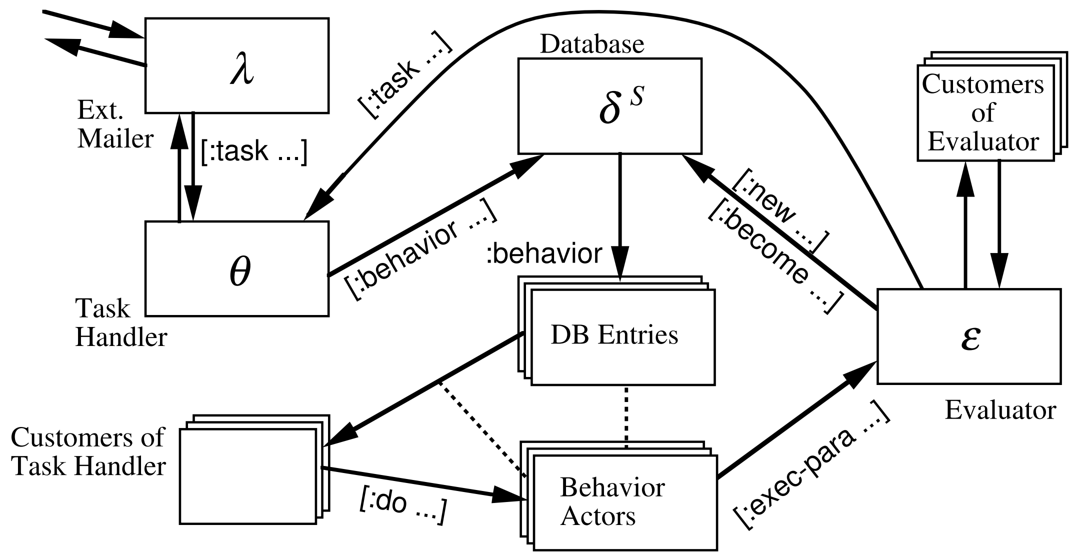

Figure 1: Actors in $\uparrow S$

Definition 1 (Metaconfiguration) *Let C be a configuration of S. A metaconfiguration $\uparrow C \in \Gamma_{\uparrow S}$ of C is defined as a pair $\langle \mathcal{T}(\uparrow C), \mathcal{T}(\uparrow C)\rangle$:*

$$\begin{aligned}
&\mathcal{A}(\uparrow C) = \{\theta, \delta^S, \varepsilon, \lambda\} \cup E^A \cup B_S \\
&\mathcal{T}(\uparrow C) = \{\langle u, m_\theta, \texttt{[:task } \uparrow t \ \uparrow m \ \uparrow k\texttt{]}\rangle \mid \langle t, m, k\rangle \in \mathcal{T}(C)\} \\
&\textit{where} \quad E^A = \{e^\alpha \mid \alpha \in \mathcal{A}(C)\} \\
&\qquad\qquad B_S = \{b^\beta \mid \langle m, \beta\rangle \in \mathcal{A}(C)\}
\end{aligned}$$

A task $\langle u, m_\theta, \texttt{[:task } \uparrow t \ \uparrow m \ \uparrow k\texttt{]}\rangle \in \mathcal{T}(\uparrow C)$ is called a *meta-task* of $\langle t, m, k\rangle \in \mathcal{T}(C)$. It represents a task $\langle t, m, k\rangle$ in the object-level (C). We let $\uparrow\tau$ denote the meta-task of τ. We let u denote the tag of the meta-task. m_θ is the mail address of the *task handler actor* θ described below. $\uparrow t$ and $\uparrow m$, called a *tag handle* and a *mail address handle*[1] (or *handles* in short), are the metalevel representations of the tag t and the mail address m. A handle may denote another handle: $\uparrow\uparrow m$ is also valid. Let $\mathcal{H}$ be the domain of handles. The functionality of $\uparrow$ is:

$$\uparrow: \mathcal{I} + \mathcal{M} + \mathcal{H} \to \mathcal{H}$$

Recall that $\mathcal{I}$ and $\mathcal{M}$ are the domain of tags and the domain of mail addresses. We write $\downarrow$ for the inverse function of $\uparrow$. For any $x \in \mathcal{I} + \mathcal{M} + \mathcal{H}$ and $y \in \mathcal{H}$, $\downarrow\uparrow x = x$ and $\uparrow\downarrow y = y$ holds. Thus $\uparrow$ is a bijection, so $\mathcal{H}$ should be an (recursive) infinite domain.

Handles will be used as keys in the database actor δ^S described below. $\uparrow k$ is the value which has the same structure as k, but every occurrence of mail addresses and handles in k are replaced with their handles. For example, if k is the value `[:do` m_1 `(foo` $\uparrow m_2$`)]` and m_1 and m_2 are mail addresses, then $\uparrow k$ is `[:do` $\uparrow m_1$ `(foo` $\uparrow\uparrow m_2$`)]`.

The metalevel actors in $\mathcal{A}(\uparrow C)$ are categorized as follows (see Figure 1). Their precise definition will be given as actual code definitions in Section 2.4.

- *Task Handler* (θ):
  The task handler actor θ knows other metalevel actors (δ^S, ε and λ) and arranges

[1]The term *handle* is used by B. Smith[7] for an extension of the concept of quotation. We use "handle" as the same way.

metalevel computation representing the task processing in the object-level. When θ receives $\uparrow\tau$, θ spawns a customer actor which continues the subsequent computation (representing the processing of τ).

- *Database* (δ^S) and *Database Entries* (elements of E^A):
  The database actor δ^S has a handle-entry table and manages it. The handle-entry table maps a handle to a database entry actor. Each database entry actor e^α has the information of a specific actor α in the object-level. Suppose that $\alpha = \langle m, \beta \rangle$ is an actor in $\mathcal{A}(S)$. Then the handle-entry table maps the handle $\uparrow m$ to the database entry actor e^α which represents α. e^α contains a message queue and a behavior actor b^β.

- *Behavior Actors* (elements of B_S):
  A behavior actor b^β is the metalevel representation of a behavior β. It contains the code of behavior script and an environment which has the binding information for acquaintance variables. The code of b^β will be evaluated by the evaluator actor ε under the environment (acquaintances).

- *Concurrent Evaluator/Processor* (ε):
  The evaluator actor ε accepts the code (expressions) of a behavior actor and execute concurrently. ε also represents the processor, the resource which offers the processing power of an actor group in the system.

- *External Mailer* (λ):
  The external mailer handles communication between S and outside systems.

Notice that a meta-configuration $\uparrow C$ is a special configuration which represents a configuration C of the object-level. There are many other possible configurations in Γ_S which may contains tasks other than meta-tasks. In sum, a meta-configuration is a configuration of $\uparrow S$ in which all the tasks are meta-tasks.

2.3 An Actor Language

We use a simple reflective actor language ACT/R to describe our model and examples. Before explaining precise definitions of the metalevel actors, we present a brief introduction to non-reflective (normal) features of ACT/R. The syntax of the language is S-expression based, and is very similar to ABCL/1[10]. However, its semantics is the same as other pure actor languages, such as SAL and ACT. See [1] for details on semantics of these languages.

The following is the definition of the *behavior* for a simple value cell actor. The value cell actor accepts two kinds of messages: `:read` and `[:write` *value*`]`.

```
(defBehavior aCell (Value)        ; Value is an acquaintance variable.
  (=> :read @ Customer            ; Returns cell value to the customer
      (become-ready)              ; Equivalent to (become aCell Value).
      [Customer <= Value])
  (=> [:write _NewValue]          ; Update the value.
      (become aCell _NewValue)))
```

The symbol `aCell` is the name of the behavior, and `(Value)` is the declaration of an acquaintance variable[2] `Value`. Note that the `defBehavior` form does not immediately evaluates itself to a cell actor. This is just a declaration for the name of a behavior description. A new instance of cell actor is created when `new` form is evaluated. For example, the expression:

```
(new aCell 100)
```

returns the *mail address*[3] of the new cell actor whose acquaintance variable `Value` is initialized to 100.

A *script* is the prescription of actions in response to incoming messages. It is a collection of *methods* (procedures). The following is the syntax for method description.

`(=>` *message-pattern* `@` *customer-variable expression...*`)`

The *message-pattern* — an S-expression — specifies the format of messages which the method can handle. It may contain pattern variables — a symbol whose name begins with an underscore character "_". Every other symbol in a pattern is regarded as a constant. Upon receiving a message which matches *message-pattern*, expressions in the body of the method (*expression...*) are executed concurrently.

Suppose that x is a cell actor. To express the transmission of a message `:read` to x, the following syntax is used.

`[`x `<= :read @` C`]`

C denotes an actor called a *customer*, which is waiting for the result of the computation performed by x. The value of an expression after `@` should be a customer. In a method description, a variable after "`@`" (customer variable) is bound to the *customers* of incoming messages. In this example, C will be bound to the variable `Customer` in the script of `aCell` and will receive the value of the cell. If no customer is specified, a *null* actor will be used as a customer. This syntax on customer follows that of ABCL/1[10].

To change the values of acquaintance variables, instead of using destructive assignment, we have to specify a new behavior called a *replacement behavior*. When processing the next message, the replacement behavior will be used. In our syntax, to specify a replacement behavior, `become` form is used. Note that at most one `become` form can be executed in a method.

`(become` *behavior-name expression...*`)`

As in `new` form, the values of *expression...* will be bound to the acquaintance variables of the replacement behavior. The form `(become-ready)` specifies the current behavior as the replacement behavior.

[2]A notion similar to an *instance variable* in other object-oriented languages.

[3]Abstraction of the location of an actor. It is a notion similar to *references*.

2.4 Actors in $\uparrow S$

The Task Handler (θ)

Let us look at the actors consisting metaconfiguration more precisely. The following is the definition of the task handler θ.

```
(defBehavior aTaskHandler (DB ExtMailer Evaluator)
  (=> [:task _Tag _Target _Message _Customer]
      (become-ready) ; Immediately after message reception, becomes ready
                     ; for the next message.
      [DB <= [:behavior _Target] @
        [customer _Value ; a customer waiting for the behavior actor
          (case _Value
            (is :external ; the target is outside of S.
                [ExtMailer <= [:task _Tag _Target _Message _Customer]])
            (is _Behavior where (non-null _Behavior)
                [_Behavior <= [:do _Tag _Target _Message _Customer
                                   Self DB Evaluator]]))]]))
                ;; Self denotes the task handler. Not the customer  itself.
```

In Definition 1, θ is the sole instance of `aTaskHandler`. In θ, the values of acquaintance variables `DB`, `ExtMailer` and `Evaluator` should be bound to δ^S, λ and ε, respectively.

When θ receives a message `[:task` $\uparrow t$ $\uparrow m$ $\uparrow k$`]` (where $\tau = \langle t, m, k \rangle$ is a task), it asks the database actor (δ^S) for the behavior actor of the target actor. As soon as θ accepts the message, it will become ready to receive the next request (see `(become-ready)`). To receive the result (a behavior actor) from the database actor, θ spawns a customer c_τ (the value of `[customer _Value...]`. See the next paragraph). c_τ is an actor waiting for the value (a behavior actor) from a database entry. When the customer c_τ receives b^β (which is bound to the pattern variable `_Value`), an execution request message `[:do ...]` is sent to b^β with the message value ($\uparrow k$) and the addresses of other actors (θ, δ^S and ε).

The expression of the form `[customer` *variable expression...*`]` evaluates to a *customer*, which is an actor waiting for the result. This is syntactically equivalent to an actor creation form (`(new ...)`). In the definition of `aTaskHandler`, we can substitute the `customer` form with the following form.

```
(new aTaskHandler_Customer Self DB ExtMailer Evaluator _Target _Message)
```

The definition for the behavior `aTaskHandler_Customer` is:

```
(defBehavior aTaskHandler_Customer
             (Self DB ExtMailer Evaluator _Target Message)
  (=> _Value
      (case _Value  body expressions described above...)))
```

Variables which freely occur in the `customer` form (in this case, `DB`,`ExtMailer`,`Evaluator`, `_Target` and `_Message`) will automatically be introduced as acquaintance variables of the customer actor. Note that the value of the variable `Self` is overridden by the creator actor (a task handler actor θ in this case). Since the customer's mail address is different from that of its creator, specifying the replacement behavior within the `customer` form does not make sense.

The Database Actor (δ^S) and Database Entries (e^α)

The database actor δ^S and database entries e^α are defined as follows. Note that δ^S in Definition 1 is the sole instance of aDB.

```
(defBehavior aDB (Table)
  ;; Table : a set of handle-entry pairs (handle-entry table).

  ;; returns a behavior actor (from the task handler θ)
  (=> [:behavior _Handle] @ Customer
      (become-ready)
      (case (parse-handle _Handle)
        ;; when _Handle denotes the address of an actor in S.
        (is [:local _Addr]
              ;; find-entry searches the handle-entry table and returns
              ;; a database entry actor.
            [(find-entry _Addr Table) <= :behavior @ Customer])
        ;; when _Handle denotes the address of an actor outside of S.
        (otherwise
            [Customer <= :external])))

  ;; specifies a replacement behavior (from the evaluator actor ε)
  (=> [:become _Handle _Behavior]
      (become-ready)
      (case (parse-handle _Handle)
          ;; Here, _Addr denotes an actor in S.
        (is [:local _Addr]
            [(find-entry _Addr Table) <= [:become _Behavior]])
        (otherwise
            (error "no such actor!"))))

  ;; creates a new actor (from the evaluator actor ε)
  (=> [:new _NewEntry _Handle] @ Customer
      (become aDB (add-table _Handle _NewEntry Table))
      [Customer <= _Handle])
  )
```

To obtain the behavior actor which represents the behavior of a target actor (say α), the task handler θ sends a message `[:behavior` $\uparrow m$`]` (where m is the mail address of α) with a customer c_τ to δ^S. The database actor δ^S searches the handle-entry table (the value of the acquaintance variable `Table`) to find the database entry actor e^α. If e^α is found, δ^S sends a message `:behavior` to e^α with the address of the customer c_τ of θ.

While evaluating the script body of an actor, the evaluator ε sends messages `[:new ...]` and `[:become ...]` to δ^S as the evaluation result for `(new ...)` and `(become ...)`, respectively.

The following are the behavior definitions of a database entry.

```
(defBehavior anEntry (Behavior)
  ;; Behavior : a behavior actor
  (=> :behavior @ Customer ; delegated from δ^S
      (become anEntry-Waiting (empty-queue))
      [Customer <= Behavior]))

;;; When some actors have been using the behavior actor.
(defBehavior anEntry-Waiting (WaitingCustomers)
  ;; WaitingCustomers : a queue of customers waiting for the behavior
  (=> :behavior @ Customer ; delegated by δ^S
        ;; When someone has been using the behavior, the customer has to wait
      (become anEntry-Waiting (enqueue WaitingCustomers Customer)))
  ;; a replacement behavior is now provided (by the evaluator actor).
  (=> [:become _Behavior]
      (case WaitingCustomers
        (is ()
            (become anEntry _Behavior))
        ;; If someone is waiting the replacement behavior,  just pass it to him soon.
        (is (_Customer . _Rest) ; The first customer is bound to _Customer.
            (become anEntry-Waiting _Rest)
            [_Customer <= _Behavior]))))
```

A database entry contains a behavior actor (the value of `Behavior`). Once the behavior actor is passed to a customer, the subsequent requests are postponed until a replacement behavior is provided. Actually, the customers of postponed requests are put into the queue (the value of `WaitingCustomers`). In this situation, the database entry has no behavior actor. Instead, it has a queue.

The replacement behavior (actor) will be provided by the evaluator actor ε as a message `[:become ...]`. This will be sent to the database actor as the evaluation result of `(become ...)` expression and will be delegated to the database entry. Then, the first customer (the value of `_Customer`) waiting in the queue (`WaitingCustomers`) will be activated.

Behavior Actors

The following is the definition of behavior actors.

```
(defBehavior aBehavior (Script AcqEnv)
  ;; AcqEnv is an environment contains the bindings for acquaintance variables
  (=> [:do _Tag _Target _Message _TaskHandler _DB _Evaluator]
        ;; A behavior actor does not specify a replacement behavior.
      (case (find-method _Message Script)
        (is NIL
            (error "Cannot handle the message ~S" _Message))
        (is (_MethodBody _Pattern)
              ;; the body of a method will be executed concurrently under the environment
              ;; AcqEnv extended with the bindings of message pattern variables.
            [_Evaluator <= [:exec-para _MethodBody
                                       (extend-env _Pattern _Message AcqEnv)
                                       _Target _Tag _TaskHandler _DB]]))))
```

A behavior actor b^β contains the script description (`Script`) and an environment for acquaintance variable (`AcqEnv`). When b^β receives `[:do ...]`, an appropriate method (an element of `Script`) is searched (`find-method`), a new environment is created, and then a message `[:exec-para ...]` which requests the execution of the body of the script is sent to the evaluator actor ε.

The concurrent evaluator actor ε evaluates the body of the script concurrently. Then, ε sends messages `[:new ...]` and `[:become ...]` to δ^S as the evaluation result for `(new ...)` and `(become ...)`, respectively. For details of concurrent evaluation in ε, see Appendix A.

3 Correctness of $\uparrow S$

Now we will give a brief outline of our proof that $\uparrow S$ correctly represents S in terms of transition relations. A more precise discussion is given in Appendix B.

As described in [1], each transition $C_1 \xrightarrow{\tau} C_2$ in Γ_S is assumed to be atomic. But in $\uparrow S$, τ is represented as a sequence of transitions — the execution (interpretation) of τ in $\uparrow S$ is *not atomic* — which may interleave with other sequences. This implies that an arbitrary configuration in the midst of interleaved transition sequences represents a mixture of intermediate states of task-processing at the object-level (S). So it seems impossible to state that which (object-level) configuration an arbitrary metalevel configuration represents. In contrast, a metaconfiguration $\uparrow C$ clearly represents the configuration $C \in \Gamma_S$.

First, we show that every configuration K of $\uparrow S$ — even if it is not a metaconfiguration — can be regarded as the representation of a configuration C of S. To do so, we introduce a notion of *normalization*. Then each $K \in \Gamma_{\uparrow S}$ can be normalized to a configuration $|K|$ which is a metaconfiguration. So $|\uparrow C| = \uparrow C$ holds for each metaconfiguration $\uparrow C$.

Then a relation $\sim$ is introduced in $\Gamma_{\uparrow S}$: $K_1 \sim K_2$ iff $|K_1|$ and $|K_2|$ are the same metaconfiguration $\uparrow C$. This means that K_1 and K_2 represents the same configuration C at the object-level (S):

$$\begin{tikzcd}
\text{metalevel} & K_1 \arrow[r, phantom, "\sim"] \arrow[dr] & K_2 \arrow[r, phantom, "\sim"] \arrow[d] & \uparrow C \arrow[dl] \\
\text{object-level} & & C &
\end{tikzcd}$$

We show the fact that every configuration is equivalent (by $\sim$) to one and only one metaconfiguration $\uparrow C$. This implies that $\Gamma_{\uparrow S}$ is partitioned by $\sim$. We let $\Gamma[C] \in \Gamma_{\uparrow S}/\sim$ denote the set of all configurations equivalent (by $\sim$) to $\uparrow C$.

Furthermore, we define a transition-like relation $\Longrightarrow$ on $\Gamma_{\uparrow S}/\sim$. The relation $\Gamma[C_1] \Longrightarrow \Gamma[C_2]$ means that for any $K \in \Gamma[C_1]$, there exists $K' \in \Gamma[C_2]$ such that $K \xrightarrow{*} K'$ holds[4]. This represents all possible executions (interpretations) of the transition $C_1 \xrightarrow{\tau} C_2$.

It can be proved that $\Gamma[C_1] \Longrightarrow \Gamma[C_2]$ iff there exists a transition τ such that $C_1 \xrightarrow{\tau} C_2$ (Theorem 5 in Appendix B). This means that every transition in the object-level can be representable in the metalevel (completeness), and any actual transition *sequence* in the metalevel correctly represents a transition in the object-level (soundness). Thus $\uparrow S$ is a correct metalevel representation of S (see below).

$$\begin{tikzcd}
\text{metalevel} & \Gamma[C_1] \arrow[r, Rightarrow] & \Gamma[C_2] \\
\text{object-level} & C_1 \arrow[r, "\exists\tau"] \arrow[ur, phantom, "\Updownarrow"] & C_2
\end{tikzcd}$$

4 Reflective Computation

4.1 Inter-Level Communication

The basic mechanism which realize reflective computation is *inter-level communication* (communication between the object-level and the metalevel), which is based on the com-

[4]This transition sequence should contain one and only one task whose target is a database entry. See Appendix B.

position of S and $\uparrow S$. In our model, $\uparrow S$ is assumed to implement/interpret S. This guarantees the causal connectivity between S and $\uparrow S$. Thus we can affect computation in S by sending messages to actors in $\uparrow S$.

From Object-Level to Metalevel

From the viewpoint of S, $\uparrow S$ is an actor system outside of S. This implies that messages sent from an actor in S to one in $\uparrow S$ are processed by the external mailer actor in $\uparrow S$.

```
(defBehavior anExtMailer (DB OtherMailers)
  (=> [:task _TagHandle _AddrHandle _Message]
      (become-ready)
      (case (parse-handle _AddrHandle)
          ;; messages to an actor in the metalevel ↑S.
        (is [:meta _Addr]  ; _Addr = ↓_AddrHandle
            [_Addr <= ↓_Message])
          ;; messages to an actor outside of S.
        (is [_Mailer _LocalHandle]
            [_Mailer <= [:task _Tag _LocalHandle _Message]]))))
```

$\downarrow$ is the reverse of $\uparrow$. ↓_Message is the value in which that every mail address handle in the value of _Message is converted to the real address it denotes. Note that $\uparrow$ and $\downarrow$ can be used as primitives of our language ACT/R.

From Metalevel to Object-Level

Similarly, sending messages from the metalevel to the object-level is simple. To send a message k to the actor whose mail address is m is performed by creating a meta-task explicitly.

```
[θ <= [:task ↑t ↑m ↑k]]
```

4.2 Language Facilities for Reflection

Good language facilities are needed for accessing metalevel actors from the object-level. There are two approaches to this issue.

- *Fixed*: Only fixed, pre-declared metalevel actors can be accessed from the object-level.
- *Dynamic*: The set of metalevel actors which can be accessed from the object-level may vary in the course of computation.

In our language ACT/R, a `defGroup` special form is used for defining the template of a group. The `defGroup` form includes `:export` declaration to proclaim which metalevel actors can be accessed from the object-level. Note that the set of the metalevel actors declared with `:export` determines how the metalevel is viewed from the object-level. If an empty set is declared with `:export` in a group, no metalevel actors can be accessed from the object-level. In this case, we cannot perform any reflective operation in the group. We can specify arbitrary extra metalevel actors in `:export`. This enables us to obtain an appropriate abstract view of the metalevel. An example is given in the next section.

In ACT/R, the set of metalevel actors accessible from the object-level change in the course of computation. The `:export` declaration specifies only the initial accessible metalevel actors. The following is the definition of a group template named `aGroup` in ACT/R.

```
(defGroup aGroup    ; a name of this group template is aGroup.
    ;; Declaration of instantiation-time parameters.
  (otherMailers)
    ;; Descriptions of the metalevel actors.
  :meta ((evaluator (new anEvaluator))
         (database (new aDB ...))
                      ⋮
         (taskhandler (new aTaskHandler database eval ...))
         (extmailer (new anExtMailer database otherMailers)))
    ;; Declaration of which actor serves as the taskhandler actor.
  :extmailer extmailer
    ;; Declaration of which actor serves as the taskhandler actor.
  :taskhandler taskhandler
    ;; Declaration of accessible metalevel actors.
    ;; In this example, the mail-addresses of eval and database
    ;; can be accessed with these names from the object-level.
  :export ((evaluator evaluator)
           (db database))
  )
```

To create a new actor group, `new` form is used. In this example, a list of the mailers of other groups is given as a parameter value (is bound to `otherMailers`).

`(new aGroup` *a-list-of-mailers*`)`

The value of the above expression is the address of the newly created *group*. The address of the group is actually the address of the external mailer of the group.

To execute a program in a group, the following special form is used.

`(with-group` E_G E_1 $E_2 \ldots$`)`

Where E_G is an expression which evaluates to a group (say G), and E_i ($i = 1, 2, \cdots$) are any expressions. Each E_i is executed within the group G, *i.e.*, Actors created as the result of each E_i belong to G.

5 An Example: Modeling Migration of Actors

Now we give an example of applications of our framework. The example is *inter-group migration of an actor*. By extending the definitions of metalevel actors presented in Section 2.4, we are able to describe a mechanism for inter-group migration of actors.

In this example, a group is an abstraction of a single processor node connected with other nodes via a network. The following is the definition of a group template for a node.

```
(defGroup aNode (aMailer aMigrator)
  :meta ((evaluator (new anEvaluator))
         (database (new aNodeDB (empty-table)))
```

```
        (taskhandler (new aTaskHandler database aMailer evaluator)))
  :taskhandler taskhandler
  :export ((migrator aMigrator))
  )
```

The external mailer will be given as a parameter (`aMailer`) when a new instance of `aNode` is created. In addition, there is another interface actor called *migrator*, which will also be given as a parameter (`aMigrator`). This actor realizes the actual migration mechanism. We can view these interface actors (external mailer and migrator) as an abstraction of "ports" in usual network architectures.

The definition of a migrator is as follows.

```
(defBehavior aMigrator (LocalDB NodeSet)
    ;; Migration request from the object-level.
  (=> [:migrate _AddrHandle _DestinationNodeAddr]
        ;; 1: Obtain the migrator of the destination node.
      (case (find-destination-migrator _DestinationNodeAddr NodeSet)
        (is _Migrator where (non-nil _Migrator)
              ;; 2: Ask for the address of the database of destination.
            [_Migrator <= :database @
              [customer DestinationDB
                   ;; 3: Get the immigrant's new address in the destination node.
                [DestinationDB <= :new-immigrant-addr @
                  [customer NewLocalAddr
                       ;; 4: Perform Migration
                    [LocalDB <= [:migrate-to _AddrHandle DestinationDB
                                             NewLocalAddr]]]]]]])))
  )
```

The value of the acquaintance variable `LocalDB` is the mail address of the database actor of the group to which the migrator belongs. The value of `NodeSet` is a set of (the mail addresses of) other nodes.

Suppose that N_1 and N_2 are (the address of) nodes. When an actor α in N_1 wants to migrate itself to N_2, α sends a message `[:migrate` $\Uparrow m$ N_2`]` (where m is the mail address of α) to the local migrator (the migrator of N_1), say, μ_1. $\Uparrow$ is needed because we have to pass the handle of the immigrant to the migrator. Actually, the form:

`[migrator <= [:migrate` $\Uparrow$`Self` N_2`]]`

is executed in the script[5] of α (μ_1 is assumed to be accessible via the name `migrator` within the node.) μ_1 first tries to obtain the address of μ_2, the migrator of N_2 (using `find-destination-migrator`). Then, μ_1 asks for the address of the database actor of N_2 (δ^{N_2}), and send a message `:new-immigrant-addr` to δ^{N_2}. When δ^{N_2} returns the new local address for α in N_2, μ_1 sends a message `[:migrate-to ...]` to the local database (δ^{N_1}).

The definition of the database actor (`aNodeDB`) is extended to handle new messages such as `:new-immigrant-addr` and `[:migrate-to ...]`. When δ^{N_2} receives the message `:new-immigrantaddr`, δ^{N_2} allocates a new local address for an incoming actor (immigrant actor). Actually, δ^{N_2} creates a new empty database entry actor.

Now, suppose that m is the address of the immigrant (α), and δ^{N_2} returns $\Uparrow m'$ for the newly created address in N_2. When δ^{N_1} receives the message:

[5]Of course, other actors can also make α migrate from N_1.

`[:migrate-to` $\uparrow m$ δ^{N_2} $\uparrow m'$`]`

δ^{N_1} first replaces e^α (the database entry actor which contains the data for α) with an actor called *migration forwarder*. The migration forwarder actually copies the contents of e^α to the newly allocated database entry in N_2. If e^α does not have any behavior actor (*i.e.*, α is processing a message), the migration forwarder waits for the incoming `[:become ...]` from δ^{N_1} and then performs copying. This implies that running actors can consistently migrate to other nodes. The more concrete descriptions of `aNodeDB` and migration forwarder are found in the first author's forthcoming thesis.

Using this migration scheme, and modifying some metalevel actors, we can realize the following mechanisms. Note that these modification can be *dynamically* performed in our framework.

- *Call-by-Move*:
  Actors are usually referred to by their references (mail addresses). By message passing, mail addresses of actors are passed around in a system. In a distributed system, inter-node communication between actors significantly increases the number of *remote* references. To avoid seriously degrading in system performance, a call-by-value like mechanism called *call-by-move* has been proposed[2]. With call-by-move mechanism, when a message is sent to a remote node, some "light weight"[6] actors whose mail addresses are contained in the message also move (migrate) to the destination node of the message. In our framework, this mechanism can be realized by modifying the evaluator.

- *Interaction-by-Move*:
  The mechanism that an actor sends itself (migrates) to its destination node is useful. For example, consider the situation where a "client" actor wants to use a remote "server" actor which requires frequent communication (interaction) while using it. Since the server has much data, it is hard to migrate. To avoid expensive remote communication, the client actor[7] should migrate itself to the server's place (node). After using the server, the client actor returns to its original place. This mechanism can be realized by modifying the task handler and the external mailer to detect communication frequency.

6 Current Status and Future Work

Currently, experiments on group-wide reflection using our prototypical reflective Actor language ACT/R is in progress. The prototype interpreter written in Common-Lisp is implemented in a pseudo-parallel manner and will be extended to actual multiprocessor machines. This implementation is based on the lazy (on-demand) construction of metalevel groups. The mechanism for the lazy construction can also be reified/reflected. The experiments include: modeling of group communication strategies, dynamically changing

[6]Small and immutable objects such as integers or stateless functions are obvious candidates for mobile objects. To determine dynamically which actors can be mobile is another example of application of reflection.

[7]If the client actor has tight communication links with its neighbor actors, the neighbors as well as the client should migrate. This mechanism, group-migration, is an example we are currently investigating.

the message delivery mechanism (using Timewarp algorithm), inter-group migration of actors, adaptive migration of groups, etc.

Acknowledgments

We would like to express our appreciation to Etsuya Shibayama and Satoshi Matsuoka whose comments on the present work were helpful.

References

[1] G. Agha. *Actors : A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1987.

[2] A. Black, N. Hutchinson, E. Jul, H. Levi, and L. Carter. "Distribution and Abstract Types in Emerald". *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[3] T. Chikayama, H. Sato, and T. Miyazaki. "Overview of the Parallel Inference Machine Operating System (PIMOS)". In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS), Tokyo*, pages 230–251. ICOT, December 1988.

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

[5] L. Liang, S. T. Chanson, and G. W. Neufield. "Process Groups and Group Communications: Classifications and Requirements". *IEEE COMPUTER*, 23(2):56–66, February 1990.

[6] P. Maes. "Concepts and Experiments in Computational Reflection". In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 147–155, 1987.

[7] B. C. Smith. "Reflection and Semantics in Lisp". In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, pages 23–35, 1984.

[8] T. Watanabe and A. Yonezawa. "Reflection in an Object-Oriented Concurrent Language". In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), San Diego CA.*, pages 306–315. ACM, September 1988. (Revised version in [10]).

[9] Y. Yokote, F. Teraoka, and M. Tokoro. "A Reflective Architecture for an Object-Oriented Distributed Operating System". In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, July 1989.

[10] A. Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System.* The MIT Press, 1990.

A Definition for the Evaluator Actor ε

The following is the definition of the concurrent evaluator actor ε in Definition 1. ε is the sole instance of `anEvaluator`.

Notice that the form `(become-ready)` appears on the toplevel of each method. This implies that when ε is processing an evaluation request, the subsequent requests can be concurrently processed. The ε can handle a number of evaluation requests concurrently — this is why ε is called the concurrent evaluator.

```
(defBehavior anEvaluator ()
  ;; Concurrent evaluation for a list of commands (_Cmds).
  (=> [:exec-para _Cmds _Env _Addr _Tag _TaskHandler _DB]
        ;; The evaluator soon will be ready to execute the next request.
      (become-ready)
      [Self <= [:exec-para1 _Env _Addr _Tag _TaskHandler _DB 0]])
  ;; In case of empty expression list...
  (=> [:exec-para1 () _ _ _ _ _ _]
        ;; _ matches any value, and is just ignored (void).
      (become-ready))
  (=> [:exec-para1 (_FirstCmd . _RestCmds)
                   _Env _Addr _Tag _TaskHandler _DB _Idx]
      (become-ready)
        ;; The following two expressions will be executed concurrently.
      [Self <= [:exec _FirstCmd _Env _Addr (extend _Tag _Idx)
                      _TaskHandler _DB]]
      [Self <= [:exec-para1 _RestCmds _Env _Addr _Tag _TaskHandler _DB
                            (1+ _Idx)]])

  ;; Command Execution:
  ;; 1: Message transmission : [target <= message]
  (=> [:exec [_Target <= _Message] _Env _Addr _Tag _TaskHandler _DB]
      (become-ready)
      [Self <= [:eval _Target _Env _Addr (extend _Tag 1) _DB] @
        [customer Target*
          [Self <= [:eval _Message _Env _Addr (extend _Tag 2) _DB] @
            [customer Message*
              [_TaskHandler <= [:task (extend _Tag 0) Target* Message*]]]]]])

  ;; 2: become form: (become behavior-description expression...)
  (=> [:exec (become _BehavDesc . _Exps) _Env _Addr _Tag _TaskHandler _DB]
      (become-ready)
      [Self <= [:evlis _Exps _Env _Addr _Tag _DB] @
        [customer Exps*
          [_DB <= [:become _Addr
                           (new aBehavior (script _BehavDesc)
                                (make-env (vars _BehavDesc) _Exps*))]]]]])
  ;; Expression Evaluation:
  ;; 1: Constant
  (=> [:eval _Const _ _ _ _] @ C where (constant? _Const)
      (become-ready)
      [C <= _Const])

  ;; 2: Variable
  (=> [:eval _Var _Env _Addr _Tag _] @ C where (variable? _Var)
      (become-ready)
      [C <= (if (bound? _Var _Env)
                (value-of _Var _Env)
                (if (eq _Var 'Self) _Addr
                    (error "unbound variable: ~S" _Var)))])

  ;; 3: new actor creation: (new behavior-description expression...)
  (=> [:eval (new _BehavDesc . _Exps) _Env _Addr _Tag _DB] @ C
      (become-ready)
      [Self <= [:evlis _Exps _Env _Addr (extend _Tag 1) _DB] @
        [customer Exps*
          [_DB <= [:new (extend _Tag 0)
                        (new anEntry
                             (new aBehavior (script _BehavDesc)
                                  (make-env (vars _BehavDesc) Exps*)))]]]])

  ;; Evaluate a list of expressions in order and returns the list of results.
  (=> [:evlis _Exps _Env _Addr _Tag _DB] @ C
      (become-ready)
      [Self <= [:evlis1 _Exps _Env _Addr _Tag _DB 0] @ C])
  (=> [:evlis1 () _ _ _ _ _ _] @ C
      (become-ready)
      [C <= NIL])
  (=> [:evlis1 (_FirstExp . _RestExps) _Env _Addr _Tag _DB _Idx] @ C
      (become-ready)
      [Self <= [:eval _FirstExp _Env _Addr (extend _Tag _Idx) _DB] @
        [customer FirstExp*
          [Self <= [:evlis1 _RestExps _Env _Addr _Tag _DB (1+ _Idx)] @
            [customer RestExps*
              [C <= (cons FirstExp* RestExps*)]]]]])
  ...
  )
```

Note that the function `extend`, which is used to extend tags, implements the tag-extension technique used in Chapter 5 of [1].

B Correctness of $\uparrow S$

First, we define the notion of *normalization* of a metalevel configuration, and introduce an equivalence relation $\sim$. Then we define a transition-like relation $\Longrightarrow$ on the equivalent classes of the metalevel configurations. Using these relations, we show that a sequence of transitions at the metalevel correctly represents the corresponding transition in the object-level.

Suppose that K is a configuration of $\uparrow S$. We divide the set of tasks $\mathcal{T}(K)$ into two disjoint sets: $\overline{\mathcal{T}}(K)$ and $\underline{\mathcal{T}}(K)$. The set $\overline{\mathcal{T}}(K)$ contains all the tasks of the following forms in K:

$$\langle u_1, m_\theta, \texttt{[:task } \uparrow t\ \uparrow m\ \uparrow k\texttt{]}\rangle$$
$$\langle u_2, m_{\delta^S}, \texttt{[:behavior } \uparrow m\texttt{]}\rangle$$
$$\langle u_3, m_{e^\alpha}, \texttt{:behavior}\rangle$$

m_θ, m_{δ^S} and m_{e^α} are the addresses of task-handler actor (θ), database actor (δ^S) and database entry actors (e^α), respectively. $u_i (i = 1, 2, 3)$ and t are tags. The rest of the tasks are all in $\underline{\mathcal{T}}(K)$.

$$\underline{\mathcal{T}}(K) = \mathcal{T}(K) - \overline{\mathcal{T}}(K)$$

We call $\overline{\mathcal{T}}(K)$ a *pre-search set* and $\underline{\mathcal{T}}(K)$ a *post-search set.*

$\overline{\mathcal{T}}(K)$ and $\underline{\mathcal{T}}(K)$ represent the two phases of interpretation (execution) of a single transition at the object-level. $\overline{\mathcal{T}}(K)$ contains tasks whose corresponding behaviors have not yet been found in database entries. This is why we call $\overline{\mathcal{T}}(K)$ pre-search set. In contrast, $\underline{\mathcal{T}}(K)$ contains tasks after the corresponding behaviors being found — the actual execution of behaviors at the object-level is performed by the tasks in $\underline{\mathcal{T}}(K)$.

We define a finite possible transition sequence in which tasks in $\underline{\mathcal{T}}(K)$ are only processed:

$$\begin{array}{l} K = K_0 \xrightarrow{\tau_0} K_1 \cdots K_{k-1} \xrightarrow{\tau_{k-1}} K_k = K' \\ \text{where } \tau_i \in \underline{\mathcal{T}}(K_i)\ (i = 0, \cdots, k-1) \end{array}$$

When a transition path $\tau_0 \cdots \tau_{k-1}$ is denoted by p, we write $K \overset{(p)}{\longmapsto} K'$ for the above transition sequence. Note that $K \overset{(\epsilon)}{\longmapsto} K$ always holds for an empty transition path ϵ.

The number of tasks in $\underline{\mathcal{T}}(K)$ and the number of customers queued in each database entry are always finite, and it is guaranteed that customers in a database entry e^α are activated one at a time after e^α accepts a `[:become ...]` message. By induction on the length of queues (in e^α) and the depth of expressions evaluated, we have the following.

Lemma 1 *For any $K \in \Gamma_{\upharpoonright S}$, it is always possible to find a finite transition path p and $K' \in \Gamma_{\upharpoonright S}$ such that*

$$K \overset{(p)}{\longmapsto} K' \ \wedge\ \underline{\mathcal{T}}(K') = \phi$$

The customers in the queue of each database entry will be activated one by one from the head of the queue. The other metalevel actors (the evaluator and its customers) except δ^S never change their states. Moreover, the order of reception of `[:new ...]` messages by δ^S does not affect the subsequent computation. Thus, by the induction on the structure[8] of tags and mail addresses, we have:

Lemma 2 *For any $K \in \Gamma_{\upharpoonright S}$, if $K \overset{(p_1)}{\longmapsto} K'$ and $K \overset{(p_2)}{\longmapsto} K''$ and $\underline{\mathcal{T}}(K') = \underline{\mathcal{T}}(K'') = \phi$ hold, then $K' = K''$ holds.*

Lemmas 1 and 2 imply that for $K \in \Gamma_{\upharpoonright S}$ the configuration $K' \in \Gamma_{\upharpoonright S}$ satisfying $K \overset{(p)}{\longmapsto} K'$ and $\underline{\mathcal{T}}(K') = \phi$ is well defined.

Definition 2 *For $K \in \Gamma_{\upharpoonright S}$, K^* denotes a configuration satisfying $K \overset{(p)}{\longmapsto} K^*$ and $\underline{\mathcal{T}}(K^*) = \phi$.*

For $K \in \Gamma_{\upharpoonright S}$, $\mathcal{T}(K)$ can be partitioned into $\overline{\mathcal{T}}(K)$ and $\underline{\mathcal{T}}(K)$. Then K^* can be reached by processing all the tasks in $\underline{\mathcal{T}}(K)$ (together with the ones caused by those tasks) without touching any task in $\overline{\mathcal{T}}(K)$. Note that $\overline{\mathcal{T}}(K^*) \supseteq \overline{\mathcal{T}}(K)$ holds. Furthermore, $\mathcal{T}(K^*) = \overline{\mathcal{T}}(K^*)$ ($\underline{\mathcal{T}}(K^*) = \phi$) holds.

Now we define the normalization of K. Tasks in $\overline{\mathcal{T}}(K^*)$ can again be divided into two groups: the group of tasks whose target is θ (namely, tasks of the form $\langle u, m_\theta,$ `[:task ...]`$\rangle$, which we called meta-tasks in Section 2.2) and the group of the remaining tasks. It is clear that each task τ in the latter group is originated by a meta-task. We denote such a meta-task by $|\tau|$.

[8]For the details of the structure of tags and mail addresses, see Chapter 5 of [1].

Definition 3 *If $\tau \in \overline{\mathcal{T}}(K^*)$ is a task $\langle u, m_{d^S}, \texttt{[:behavior }\uparrow m\texttt{]}\rangle$ or $\langle u', m_{e^\alpha}, \texttt{:behavior}\rangle$, then $|\tau|$ is defined as the following meta-task:*

$$\langle u, m_\theta, \texttt{[:task }\uparrow t\ \uparrow m\ \uparrow k\texttt{]}\rangle$$

where $\alpha = \langle u'', m, \beta\rangle$ and t, u, u', u'' are tags.

When the taskhandler θ receives a message $\texttt{[:task }\uparrow t\ \uparrow m\ \uparrow k\texttt{]}$, θ creates a customer actor which contains the metalevel representation $\uparrow k$ of the message k (as the value of the variable `_Message`). We denote such a customer actor by $c[\tau]$.

Definition 4 (normalization) *For $K \in \Gamma_{\uparrow S}$, the normalization of K, which is denoted by $|K|$, is a configuration $|K| \in \Gamma_{\uparrow S}$ defined as follows.*

$$\begin{aligned}\mathcal{T}(|K|) &= (\overline{\mathcal{T}}(K) - T) \cup \{|\tau| \mid \tau \in T\}\\ \mathcal{A}(|K|) &= \mathcal{A}(K^*) - \{c[\tau] | \tau \in T\}\end{aligned}$$

where T is the set of tasks in $\overline{\mathcal{T}}(K^*)$ whose targets are d^S or e^α for some $\alpha \in \mathcal{A}(S)$.

The set $\mathcal{T}(|K|)$ only consists of the meta-tasks (of the form $\langle u, m_\theta, \texttt{[:task }\ldots\texttt{]}\rangle$) which cause the tasks in $\mathcal{T}(K^*)$. Note again that $|K|$ is a metaconfiguration.

The following lemma holds, which means that every $K \in \Gamma_{\uparrow S}$ represents a configuration of S. The proof is straightforward from the construction of $|K|$.

Lemma 3 *For each configuration $K \in \Gamma_{\uparrow S}$, there exists one and only one metaconfiguration $\uparrow C \in \Gamma_{\uparrow S}$ such that $|K| = \uparrow C$ holds (namely, for each $K \in \Gamma_{\uparrow S}$, there exists a unique normalization).*

Now we can define the relation $\sim$.

Definition 5 *For $K_1, K_2 \in \Gamma_{\uparrow S}$, we write $K_1 \sim K_2$ if there exists a metaconfiguration $\uparrow C \in \Gamma_{\uparrow S}$ such that both $|K_1| = \uparrow C$ and $|K_2| = \uparrow C$ hold.*

It is clear that $\sim$ is an equivalence relation. So we can form the equivalence classes $\Gamma_{\uparrow S}/\sim$. $\Gamma[C] \in \Gamma_{\uparrow S}/\sim$ is a set of configurations of $\uparrow S$ which contains one (and only one) metaconfiguration $\uparrow C$ and all the configurations equal (by $\sim$) to $\uparrow C$. Each configuration in $\Gamma[C]$ represents the configuration $C \in \Gamma_S$.

Definition 6 *We write $\Gamma[C_1] \Longrightarrow \Gamma[C_2]$ if:*

$$\forall K \in \Gamma[C_1] \exists K' \in \Gamma[C_2], K \xrightarrow{*} K'$$

where the transition sequence $\xrightarrow{*}$ contains one and only one task whose target is a database entry.

Lemma 4 *For any $C_1, C_2 \in \Gamma_S$ satisfying $\exists\tau \in \mathcal{T}(C_1), C_1 \xrightarrow{\tau} C_2$, if there exist $K_1 \in \Gamma[C_1]$ and $K_2 \in \Gamma[C_2]$ such that $K_1 \xrightarrow{*} K_2$ (where $\xrightarrow{*}$ contains one and only one task σ whose target is a database entry such that $|\sigma| = \uparrow\tau$)*[9] *holds, then $\Gamma[C_1] \Longrightarrow \Gamma[C_2]$ holds.*

Proof *Suppose that σ is the task whose target is a database entry e^{α}.*

$$K_1 \xrightarrow{*}\xrightarrow{\sigma}\xrightarrow{*} K_2$$

where $\sigma = \langle u, m_{e^{\alpha}}, \texttt{:behavior}\rangle$ such that u is a tag and $m_{e^{\alpha}}$ is the mail address of the database entry e^{α}. By Definition 2, for any $K \in \Gamma[C_1]$, we can construct the following transition sequence:

$$K \xrightarrow{*} K^* \xrightarrow{*}\xrightarrow{\sigma} K'$$

Clearly, $K' \in \Gamma[C_2]$. Thus $\Gamma[C_1] \Longrightarrow \Gamma[C_2]$ holds. ∎

The relation $\Longrightarrow$ represents all the possible metalevel executions for a transition in Γ_S. The next theorem states that $\Gamma_{\uparrow S}/\sim$ and Γ_S are isomorphic.

Theorem 5 *Let $C_1, C_2 \in \Gamma_S$. $\Gamma[C_1] \Longrightarrow \Gamma[C_2]$ iff $\exists\tau \in \mathcal{T}(C_1), C_1 \xrightarrow{\tau} C_2$.*

Proof *Suppose that $\Gamma[C_1] \Longrightarrow \Gamma[C_2]$. Then, from the definition of Γ, $\uparrow C_i \in \Gamma[C_i] (i = 1, 2)$ holds. There is a possible transition sequence:*

$$\uparrow C_1 \xrightarrow{\uparrow\tau}\xrightarrow{*}\uparrow C_2$$

By Definition 1, the above is equivalent to $C_1 \xrightarrow{\tau} C_2$. Conversely, from Lemma 4, if we have $\uparrow C_1 \xrightarrow{*}\uparrow C_2$, $\Gamma[C_1] \Longrightarrow \Gamma[C_2]$. ∎

This theorem means that the execution (interpretation) of an object-level actor system S at its metalevel $\uparrow S$ is complete (every transition path in the object-level is representable in the metalevel) and sound (every transition path in the metalevel correctly represents the transitions in the object-level). So the meta-system $\uparrow S$ is a correct metalevel representation of S.

[9] Note again that $\uparrow\tau$ is the meta-task of τ. See Section 2.2.