Actor Reflection without Meta-Objects[*]

TANAKA Tomoyuki[†]
IBM Research, Tokyo Research Laboratory
5-19, Sanbancho, Chiyoda-ku, Tokyo 102, Japan

Abstract

We consider how reflection should be introduced into an actor language, where reflection is a mechanism for allowing a program to have access to the data structures of its own processor. Rather than introducing *meta-objects* as in the previous work, we propose to introduce reflection through two kinds of special messages: *reifying* and *reflecting* messages. We show that the full range of reflective programming in an actor language is possible without introducing meta-objects, and argue that our approach provides a more uniform interface for actors than the meta-object approach. All the examples in this paper have been tested on our prototype actor system.

1 Introduction

Reflection is an attempt to allow programs to have knowledge of their text and the context in which they are executed. Reflection was first introduced by Smith in [S82]. Since then there have been various attempts to extend this concept within Lisp [S84] [FW84] [WF86] [B88] [T89], as well as attempts to introduce the concept to different languages and models. Since different languages have different processor structures, working data, and language constructs, a new method of reflection must be devised for each language.

Outside of Lisp, the area where such attempt attracted the widest attention has been that of object-oriented languages, which are either class-based such as Smalltalk-80 [FJ89] or class-less and actor-based [M87b] [IC88] [WY88]. In this paper we consider how reflection should be introduced into an actor language.

2 The actor model and our description language

In this section we examine the key characteristics of the actor model, and then present our description language, the language in which the examples in this paper are presented. All the examples have been tested on our prototype actor system, written in Common Lisp.

[*]This is a condensation of *Actor Reflection without Meta-Objects* (TRL Research Report RT-0047, 1990) which was presented and distributed at Object-Based Concurrent Systems Workshop.

[†]The author's family name is Tanaka.

2.1 Three key characteristics of the actor model

We do not intend to give a detailed introduction of the actor model.[1] For the purpose of the discussion in this paper, the essential aspects of the actor model can be summarized as follows. An actor language is a conventional sequential language augmented with the following characteristics:

A1 Concurrency The model consists of concurrently and asynchronously running *actors* that communicate with each other only by sending messages. This assumes that each actor has a buffer or queue for messages received and waiting to be processed. A data flow network has this aspect.

A2 Actor Creation Unlike a data flow network, nodes or actors can be dynamically created.

A3 Actor Internal States The actors have internal states and therefore are history-sensitive. They can behave non-functionally in that the same input to an actor at two different occasions may cause different behavior.

An actor system (model) consists of two levels: autonomous actors and a general *supervisor*. The supervisor is not necessarily a *global synchronizer* that govern all the actors to execute one cycle at a time, but even in a distributed environment, a supervisor must reside, a major component of which is a *mail system* that keeps track of all the actors that exists in the system, convey messages to the destination actors, and, if this can not be done directly, send the message to the appropriate intermediate site, etc.

2.2 The description language and our prototype system

In order to test our ideas and also to present the examples, we have developed a prototype actor system which interprets actor scripts in a language similar to SAL, a minimal actor language presented in [A86]. Our language will be simply referred to as "the actor language". Like SAL, the messages are positional not keyword-based, and the language has data other than actors, such as numbers, symbols, and lists. Because our description language is simple and is similar to SAL, we believe that no formal

[1]See [A86] or [H77] for a general introduction to the actor model and concept.

description of the language is necessary. Rather we introduce the language by explaining the definitions of two typical actors.

The first example is the factorial actor adopted from [H77] and [A86]. It shows the actor features **A1 Concurrency** and **A2 Actor Creation**. The following is a session with our prototype actors system.

```
----------------------------------------------------
> (def-actor fact
    ()          ; state-vars
    (n cont)    ; main-vars
    ;; main-form
    (if (= n 0)
        (send cont (list 1))
        (let ((c (new Rec-Customer (list n cont))))
             (send self (list (1- n) c)))))
ACTOR DEFINED: fact

> (def-actor Rec-Customer
             (n cont)    ; state-vars
             (v)                ; main-vars
             ;; main-form
             (send cont (list (* n v))))
ACTOR DEFINED: rec-customer

> (defvar f (new fact ()))
VAR   DEFINED: f

> (defvar p (new print-cont ()))
VAR   DEFINED: p

> (send f (list 3 p))
6
----------------------------------------------------
```

First, the two actor definitions are typed into the system. An actor definition consists of four components: the name of the actor; *state-vars*, the variables used to specify the internal state of the actor, which must be initialized when an instance is created; *main-vars*, the positional template for the messages to the actor; and *main-form*, which corresponds to the *script* in traditional actor terminology. Then, the instances[2] of the actors are created with new expressions and bound to global variables. print-cont is an actor whose definition is built into the system. It receives a message of one component and prints the component. Finally, a message of two components, 3 and a printer actor, is sent to the factorial actor. Here, the printer actor is sent to specify what to do after the factorial value is computed. Actors used in such a way are called *continuations* or *customers*.

The *main-form* part of an actor definition is a single command, which is either of

(send *actor-expr message-expr*)
(if *pred-expr then-command else-command*)
(let *var-expr-pairs command*)
(progn *command* ...).

An expression is a simple Lisp expression or a new expression: (new *actor-name expression*).

The second example shows the effect of **A3 Actor Internal States**. A counter actor maintains a count of how many objects it has processed, so that even for the same input (the first and the third messages in the following session) the outputs may be different.

```
----------------------------------------------------
> (def-actor counter
    (count)     ; state-vars
    (obj rec)   ; main-vars
    ;; main-form
    (send rec (list (list (setq count (1+ count))
                          obj))))
ACTOR DEFINED: counter

> (defvar c (new counter (list 0)))
VAR   DEFINED: c

> (send c (list 'aaa p))
(1 aaa)

> (send c (list 'bbb p))
(2 bbb)

> (send c (list 'aaa p))
(3 aaa)
----------------------------------------------------
```

3 A reflective mechanism in the actor language

In this section, we first examine the notion of procedural reflection in Lisp, where reflection was first introduced, then consider how to introduce the concept to an actor language with the three characteristics described in the last section.

3.1 Introducing reflection to a language

It should be stated on the outset that reflection is *not* changing the interpreter. Rather than modifying the interpreter, procedural reflection which Smith introduced in [S82] aims to do two things: (1) To allow programs to have access to the data structures of its own processor, so that code manipulating such data can be seen as running at the level of the interpreter; and (2) If the language at the level above is the same as the level below and the same *reification* mechanism can be used, then the process of shifting-up can be repeated, resulting in a virtual tower of processors.

When introducing a reflective mechanism to a language, three questions must be answered:

1. What are the working data structures of the interpreter? (These should constitute a complete snapshot of the computation.)
2. How should they be made available to the object level program? (This process is called *reification*.)
3. How should they be reinstalled? (This process is called *reflection*.)

The process of reification and reflection may require conversion of data, as in [FW84].

3.2 Reification and reflection in Lisp

In Lisp the data structures of the interpreter chosen to be reified and reflected are the following:

e the expression to be evaluated

r the variable environment

[2] Henceforth we will say *actor instances* or *instances*, instead of *actors*, where distinction need to be emphasized.

k the continuation to which the value of e in the environment r is to be sent

The three data structures are made available to the object level program through an application of a *reifying procedure* (or a *reifier*). For example, when *body* of the reifier in Brown[3] program

```
((abs reify (e r k) <body>) a b c)
```

is evaluated, the variable e is bound to the expression (a b c), r is bound to the environment, and k is bound to the continuation of the entire application form.

There are two ways of reflection in Lisp: one is to invoke the continuation with a value, and the other is to call a function meaning or eval, specifying a set of three values to be installed.[4]

3.3 What are the working data structures of the interpreter?

What is the *interpreter* in an actor system? We stated in Section 2 that an actor system consists of two levels: autonomous actors and a general supervisor. Therefore, an interpreter can be seen to exist at both levels. However, the supervisor is something that programs (scripts) usually do not have access to, and therefore it is not appropriate to see the supervisor as an interpreter, at least as the first step in introducing reflection into an actor language. Pursuing this direction would lead to treating the entire system (the supervisor and all the individual actors), but we will not explore this aspect in the present paper.[5]

Instead, we focus on the processor for just one object as an interpreter, and decide on the the following four data structures of the object interpreter:

main-vars the message template for the actor

main-form the script of one command

state-vars-env the environment for representing the internal state of the actor; must be initialized when an instance is created

queue the queue of messages received by the actor but are still waiting to be processed

The properties **A1 Concurrency** and **A3 Actor Internal States** respectively require that queue and state-vars-env be a part of the object interpreter's data structure. The property **A2 Actor Creation** does not introduce a new aspect to actor reflection over a sequential language because creating an actor only causes an event *outside* the interpreter of the actor (creation of a process that may send messages to the current actor). To the interpreter such an event is no different from creation of a data structure from a heap (e.g., a list).

3.4 Reifying messages: making the processor's data available

The question is, through what mechanism should the data structures of the actor interpreter be made available? Since this is a communication that may take place across different actors, it is natural to model this communication as a kind of message passing. Sending a message is the actor's analogue of Lisp's applying a function. Just as in Lisp special functions (*reifying procedures*) are used, we introduce special messages, which we call *reifying messages* (or *reifiers*). To ask for the data structures of an actor one sends a *reifying message* to the actor.[6]

The exact format is

```
(send <actor> <reifying-message>)
```

where *actor* is the target actor, and *reifying-message* is created by the function make-reify-message.

The function make-reify-message takes two arguments: *freeze?* and *cont*. The *cont* argument is a continuation actor which always takes a message of four components, corresponding to the four data structures of the interpreter. The *freeze?* argument is described below.

When an actor receives a reifying message, and its turn to be processed arrives,[7] it does not go through the regular processing of matching the message to the message template (main-vars), but instead makes a list of the four data structures of its actor processor and sends it to the continuation actor as a regular message.[8] (See Section 4 for how exactly these are specified.)

In this reification mechanism (or in the reflection mechanism to be described below), there is no conversion problem because the four reified components are already data of the object level language.

Freezing and non-freezing reifiers

Should an actor be allowed to continue running while its registers are given to another actor, which may possibly modify and install them into the original actor? This is a question which does not arise in sequential languages, but which must be answered when introducing reflection to an actor language.

We found uses for both types of reifiers: a *non-freezing* reifier (one that lets the target actor continue to run) is good for obtaining a snapshot of the actor (see Section 4.2), whereas a *freezing* reifier, (one that "freezes" the target actor until it receives a reflector) is necessary to define the send command at the user level (see Section 4.3). Therefore we provide both types of reifiers which is specified by the *freeze?* argument to make-reify-message.

An actor in a frozen state does not process any messages in its queue, and it only receives a reflecting message. The arrival of a reflecting message resumes the execution of a frozen actor.

[3]This is in [FW84] notation.

[4]There is also a method of *exiting* which we will see in Section 5.

[5]See [WY89] for work in this direction.

[6]A uniform notation is used for an actor to send reifying messages to other actors as well as to itself, although the implementation may treat this differently when the target is itself, for no inter-actor communication need to be generated in that case.

[7]It is possible also to have it so that all reifying (and reflecting) messages are processed out of order (for example, as soon as they arrive), but we do not see any significant advantage of this.

[8]It is more precise to say that a *copy* of the four data structures of the processor are sent, because sharing of list structures, for example, is not permitted between two different actors.

3.5 Reflecting messages: installing the processor's data

We similarly model reflection as sending messages of a special type, called a *reflecting message.*

```
(send <actor> <reflecting-message>)
```

A *reflecting message* is created by the function make-reflect-message, which takes four arguments corresponding to the four date components of the interpreter.

When an actor receives a reflecting message, it handles it differently from a regular message, and installs the four components of the message into the four "registers" of the interpreter.

The arguments to make-reflect-message may be omitted to specify that the corresponding data structure of the target actor interpretor is to be left untouched.

4 Programming with reifying and reflecting messages

In this section, we show some programming examples using the reifying and reflecting messages, and show the range of programming possible with our approach. In the first two examples, a reifier and a reflector are used by itself, not in pairs. It should be noted that reification and reflection do not have to occur in pairs at all in this model, as there is no global level common to all actors.

4.1 Counter resetter

The first example is an actor which resets the counter actor defined in Section 2. It is an example of an *unmatched reflector*, a reflecting message without the matching reifying message.

```
(def-actor counter-resetter
  ()            ; state-vars
  (counter)     ; main-vars
  ;; main-form
  (send counter
        (make-reflect-message
         :state-vars-env (make-env 'count 0))))
```

When a counter resetter (instance) accepts a counter, it sends to the counter a reflecting message containing the environment to be installed. Here make-env is a function which constructs a valid representation of the environment. This counter resetter can be used on any family of counter actors which stores its counter within the variable named count.

4.2 Load average checker

The second example is a load average checker, which checks the length of the queue of a given actor. It is an example of a use of non-freezing reifier without the matching reflector.

```
(def-actor la
  ;; use: (send lai (list arg-ai p-la))
  ;; sends load average (length of queue) to cont
  ()             ; state-vars
  (arg-ai cont) ; main-vars
  ;; main-form
  (let ((la-sub-i (new la-sub (list cont)))
       (send arg-ai
             (make-reify-message :freeze? nil
                                 :cont la-sub-i))))
(def-actor la-sub
  ;; receives a quadruplet
  ;; sends the length of queue to CONT
  (cont)                  ; state-vars
  (vars form env queue) ; main-vars
  ;; main-form
  (send cont (list (length queue))))
```

An instance of actor la accepts a message consisting of two arguments, the argument actor and a printer continuation, and sends the length of the queue of the argument actor to the printer continuation.

A simple load balancer can also be written using the load average checker which will accept a list of actors, check the load balance of each actor, and re-distribute messages more evenly when possible.

4.3 User-defined send

The next example is a user-defined send. With the following definitions, sending a message to <usi>, an instance of actor user-send, as

```
(send <usi> (list <ai> <message>))
```

produces the same effect as this command:

```
(send <ai> <message>)
```

```
(def-actor user-send
  ;; use: (send usi (list ai message))
  ()            ; state-vars
  (ai m)        ; main-vars
  ;; main-form
  (let ((u-s-sub-i (new user-send-sub (list ai m))))
       (send ai
             (make-reify-message :freeze? t
                                 :cont u-s-sub-i))))
(def-actor user-send-sub
  ;; receives a quadruplet
  ;; push m to ai's queue
  (ai m)                  ; state-vars
  (vars form env queue) ; main-vars
  ;; main-form
  (let ((q (cons m queue)))
       (send ai (make-reflect-message :queue q))))
```

4.4 Trace: monitoring an actor's behavior

The last example is monitoring or tracing the behavior of an actor: whenever a certain actor is about to process a received message, some trace output is printed.

To set the trace on, a message consisting of the argument actor and a printer continuation is sent to an instance of actor trace. After this takes effect, every time the argument actor is about to process a received message, the trace information is sent to the printer continuation.

Sending the traced actor to an instance of untrace actor sets off the trace.

```
(def-actor trace
  ;; use: (send <ti> (list <ai> p-trace))
  ()             ; state-vars
  (ai p-cont)   ; main-vars
  ;; main-form
  (let ((t-sub-i (new trace-sub (list ai p-cont))))
       (send ai
             (make-reify-message :freeze? t
                                            :cont t-sub-i))))
(defvar p-trace
  (new print-message-cont
       (list "traced actor received: ")))
(def-actor trace-sub
  ;; receives a quadruplet
  ;; change ai's main-form to
  ;;    (progn <trace-output> <original>)
  (ai p-cont)           ; state-vars
  (vars form env queue) ; main-vars
  ;; main-form
  (let ((new-main-form
         '(progn
           (send ',p-cont (list (list . ,vars)))
           ,form)))
       (send ai
             (make-reflect-message
              :main-form new-main-form))))

(def-actor untrace
  ;; use: (send <uti> (list <ai>))
  ()    ; state-vars
  (ai)  ; main-vars
  ;; main-form
  (let ((ut-sub-i (new untrace-sub (list ai))))
       (send ai
             (make-reify-message :freeze? t
                                 :cont ut-sub-i))))
(def-actor untrace-sub
  ;; receives a quadruplet
  ;; change ai's main-form from
  ;; (progn <trace-output> <original>) to <original>
  (ai)                  ; state-vars
  (vars form env queue) ; main-vars
  ;; main-form
  (let ((new-main-form (third form)))
       (send ai
             (make-reflect-message
              :main-form new-main-form))))
```

5 Comparison with the meta-object approach

We have presented a method of reflection through reifying and reflecting messages. This is in sharp contrast with the approach in the previous work, all of which involved the notion of *meta-objects*: every object has a meta-object, a meta-meta-object, ... to form an infinite tower [M87b] [WY88] [FB88] [F89].

5.1 Converting from the meta-object approach to our approach

In the meta-object approach, a reflective operation is performed either by sending a message to the meta-object of the target object (actor) [WY88] [F89] or by temporarily changing the meta-object to be a different one [M87b].

A program in ABCL/R [WY88] for sending a message to the meta-object of an object can be translated as follows.

- In case of asking for a snapshot of a data component of the evaluator (with messages such as `:queue`, `:state`, or `[:script` *m*`]`), translate it into a program for sending a non-freezing reifier to the object.
- Other messages that modify some data component(s) of the evaluator (such as `[:add-script` *s*`]` or `[:delete-script` *m*`]`), are translated into a program which first sends a reifying message to the target object to get the data components (the quadruplet), modifies some of the components, and then installs them to the object by a reflector.

Programs in [M87b] and [M87a] which temporarily change the meta-object to a different one can be converted similarly. For the trace example, instead of changing the the meta-object of a program-object, the script of an object can be temporarily modified as our example in Section 4.4.

5.2 Are meta-objects necessary?

It is natural to wonder: Isn't the meta-object tower necessary to implement the reflective tower? The answer is no.

In our model, when

```
(send <target-actor>
      (make-reify-message :freeze? t
                          :cont <cont-actor>))
```

is executed, the quadruplet is sent to the continuation actor, which receives it and have access to the quadruplet. At this point, since the continuation actor's script is manipulating the data structures of the (target actor's) interpreter, the continuation actor's script can be seen as running at the level of the (target actor's) interpreter. Furthermore, if the data structures of this continuation actor is reified and manipulated by another actor, then that actor's script may be seen as running at an yet higher level, the level of the (continuation actor's) interpreter, and so on. This is how we have a reflective tower; no actual meta-object tower is necessary.

Similarly in case of Lisp, no actual reflective tower is created; no separate interpreter is created for each level, but instead the same interpreter is reused with different continuation taken from the meta-continuation, which is a stream-like, virtually infinite list of continuations.

This leads to the question: Isn't the meta-object tower the counterpart of the meta-continuation stream, then? The answer is again no. Each entry in the meta-continuation stream represents "the context in which the interpreter at each level is running". This information is necessary in Lisp because there is a way of *exiting* from within a reifier, which is to return to the toplevel loop of the current level, instead reflecting to the original level below.[9] Having no such *exiting* mechanism, an actor language does not require that kind of information contained in a meta-continuation.

To sum up, in an actor system every object must have an evaluator that specifies the behavior of the object, and this evaluator can be seen as the the meta-object of the object. But it is not necessary that this evaluator exist as a separate, genuine object, nor that these meta-objects also have separate meta-objects (or evaluators) associated with them. Instead of having a meta-object tower, shifting-up levels can be done one step at a time, as described above.

Our approach offers a more uniform interface for actors than the previous meta-object approach. In the

[9]See [WF86] [DM88] or [T90] for a description of this.

meta-object model, there are two kinds of objects: user-defined objects and virtual meta-objects. User-defined objects have interface defined by the user, whereas meta-objects in the infinite tower on top of each user-defined object have a different, pre-defined interface (accepting `:add-script`, etc. in [WY88] or `:handleMsg` in [F89]). In our model, all actors have a uniform interface, in that they all accept reifying and reflecting messages in addition to the user defined messages.

6 Conclusions

We have examined how reflection should be introduced to an actor language, and presented a method using two kinds of special messages: *reifying* and *reflecting* messages. We also introduced the notions of *freezing* and *non-freezing* reifiers. We showed that the full range of reflective programming in an actor language is possible without introducing *meta-objects*, and argued that our approach provides a more uniform interface for actors than the meta-object approach.

The meta-object concept may be useful when modeling conceptual reflection [F88] or when building a reflective architecture for an operating system [YTT89]. From the work in this paper, however, we conclude that reification and reflection can be introduced to an actor language without introducing meta-objects. We are currently investigating further implications of the model based on reifying and reflecting messages, which we hope to report elsewhere.

Acknowledgements

I would like to thank MAEDA Atusi for the discussions in the early stage of this work, as well as Dr. Tsutomu KAMIMURA and Dr. Christian QUEINNEC for the helpful comments. I would also like to thank my managers and colleagues at IBM Tokyo Research Laboratory for support and encouragement.

References

[A86] Agha, Gul A. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[B88] Bawden, A. Reification without Evaluation. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming.* 1988, pp. 342–351.

[DM88] Danvy, O., and Malmkjaer, K. Intensions and Extensions in a Reflective Tower. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming.* 1988, pp. 327–341.

[FB88] Ferber, J., and Briot J-P. Design of a Concurrent Language for Distributed Artificial Intelligence. In *Proceedings of the International Conference on Fifth Generation Computer Systems.* 1988, pp. 755–762.

[F88] Ferber, J. Conceptual reflection and actor languages. In P. Maes, D. Nardi, ed. *Meta-Level Architectures and Reflection.* 1988, pp. 177–193.

[F89] Ferber, J. Computational Reflection in Class based Object Oriented Languages. In *OOPSLA '89 Proceedings.* 1989, pp. 317–326.

[FJ89] Foote, B., and Johnson, R.E. Reflective Facilities in Smalltalk-80. In *OOPSLA '89 Proceedings.* 1989, pp. 327–335.

[FW84] Friedman, D.P., and Wand, M. Reification: Reflection without Metaphysics. In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming.* 1984, pp. 348–355.

[H77] Hewitt, C.E. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence.* Vol. 8, No. 3, June 1977, pp. 323–364.

[IC88] Ibrahim, M.H. and Cummins, F.A. KSL: A Reflective Object-Oriented Programming Language. In *Proceedings of the IEEE International Conference on Computer Languages.* 1988, pp. 186–193.

[M87a] Maes, P. *Computational Reflection.* Technical Report 87-2, Vrije Universiteit, 1987.

[M87b] Maes, P. Concepts and Experiments in Computational Reflection. In *OOPSLA '87 Proceedings.* 1987, pp. 147–155.

[S82] Smith, B.C. *Reflection and Semantics in a Procedural Language.* MIT/LCS/TR-272, MIT, 1982.

[S84] Smith, B.C. Reflection and Semantics in Lisp. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages.* 1984, pp. 23–35.

[T89] Tanaka T. *A Stack Representation of Continuations in a Reflective Lisp Interpreter.* TRL Research Report RT-0024, 1989.

[T90] Tanaka T. "Jumpy" and "Pushy" `call/cc`. *SIGPLAN Lisp Pointers.* ACM, Vol. 3, No. 1, March 1990, pp. 3–4.

[WF86] Wand, M., and Friedman, D.P. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming.* 1986, pp. 298–307.

[WY88] Watanabe, T., and Yonezawa, A. Reflection in a Object-Oriented Concurrent Language. In *OOPSLA '88 Proceedings.* 1988, pp. 306–315.

[WY89] Watanabe, T., and Yonezawa, A. A Concurrent Reflective Computation Model Based on ACTOR Paradigm. In IEICE Technical Report (COMP-89-97). Institute of Electronics, Information and Communication Engineers. 1989, pp. 77–86 (in Japanese).

[YTT89] Yokote, Y., Teraoka, F., and Tokoro, M. A Reflective Architecture for an Object-Oriented Distributed Operating System. In Proc. ECOOP '89. 1989.

First version submitted for publication: 1 March 1990
Revised: 1 August 1990; 8 December 1990