An Example for Concurrent Reflective Computations in Rewriting Logic

Hiroshi Ishikawa, Kokichi Futatsugi, and Takuo Watanabe
School of Information Science,
Japan Advanced Institute of Science and Technology (JAIST).
15 Asahidai, Tatsunokuchi, Ishikawa 923-12, Japan.
Telephone : +81-761-51-1255. Fax : +81-761-51-1149.
E-Mail: {h-ishika,kokichi,takuo}@jaist.ac.jp

Abstract

Rewriting logic can represent dynamic behaviours of concurrent and/or reactive systems declaratively. Declarative descriptions in rewriting logic are expected to be amenable to analysis of interesting properties. The group-wide architecture based on the actor model is a specific concurrent reflective computation model based on “group-wide reflection”. It has a potential of modelling cooperative behaviours of several software modules or agents. This paper provides some basic considerations on methods of modelling the group-wide architecture in rewriting logic and a brief proof that the metalevel group of terms correctly represents the operational semantics of the group in terms of transitions of configurations.

Keywords

Actor model, Concurrent Reflective Computation, Group-Wide Architecture (GWA), Group-Wide Reflection, Rewriting Logic

1 INTRODUCTION

The concurrent reflective computation model is recognized to be a useful framework for constructing reliable and easy-to-maintain concurrent system. That framework can basically be constructed in two ways. One is called Individual-Based Architecture [Watanabe and Yonezawa (1990)]. Each object in the system has its own metaobject which governs its computation. Reflective computation [Smith (1984), Maes (1987), Watanabe (1994)] is realized by sending message to metaobjects. The computational activity in an object is sequential. This implies each metaobject models only the local sequential aspects of an object. Therefore, individual-based reflection is not sufficiently powerful to deal with the global information of a group of objects. The other is called Group-Wide Architecture (GWA) [Watanabe and Yonezawa (1991)]. It is a specific concurrent reflective computa-

tion model based on group-wide reflection. It has a potential of modelling cooperative behaviours of several software modules or agents. Our GWA is based on the actor model [Agha (1987)]. All reflective operations are performed solely via message sends, which are interpreted at the metalevel concurrently with interpretations of actors in the baselevel.

Rewriting logic [Meseguer, Futatsugi and Winkler (1992), Meseguer (1993)] can represent dynamic behaviours of concurrent and/or reactive systems declaratively. Declarative descriptions in it are expected to be amenable to analysis of interesting properties.

In this paper, we propose some basic considerations on methods of modelling the group-wide reflection in rewriting logic, and implement it [Ishikawa, Futatsugi, and Watanabe (1994), (1995a), and (1995b)]. We use a functional programming language Gofer, which is a sublanguage of functional programming language Haskell [Hudack, Wadler, et al. (1992)], to describe our model and perform an example of *inter-group* migration of object. Furthermore, we give a brief proof that the metalevel group of terms correctly represents the operational semantics of the group in terms of transitions of configurations.

1.1 Reflection

A system is said to be causally connected to its domains if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect upon the other.

A reflective system is a system which incorporates structures representing (aspects of) itself. We call the sum of these structures the self-representation of the system. This self-representation makes it possible for the system to answer questions about itself and support actions on itself. Because the self-representation is causally-connected to the aspects of the system it represents, we can say that the system always has an accurate representation of itself and that the status and computation of the system are always in compliance with this representation. The latter means that a reflective system can actually bring modifications to itself by virtue of its own computation.

When we try to implement a reflective system, the idea of procedural reflection [Smith (1984)] is useful.

1.2 Group-Wide Architecture

In this architecture, a group of metalevel objects makes up the metalevel. The behaviour of an object is not managed by metalevel objects; rather, the collective behaviour of a group of objects is represented as the coordinated actions of a group of metalevel objects, which make up metagroup. As mentioned above, the GWA we make use of in this paper is based on the actor model. In this case, each object is regarded as actor. Note that the notion of GWA is independent of actor theory.

1.3 Rewriting logic

We introduce the relationship between the logic and the actor model. For the definition of rewriting logic, see [Meseguer, Futatsugi, and Winkler (1992), Meseguer (1993)].

Rewriting logic is understood as a method of correct reasoning about some class of entities. For the logic, the entities above are concurrent systems having *states*, and evolving

by means of *transitions*. Therefore, we can describe concurrent object-oriented computation models in it. In rewriting logic, they are formalized as concurrent ACI (Associative, Commutative and Idempotent)-rewriting. The actor model is a special case of them. In order to make things clearer, we make use of the correspondence the terminology of actor theory such as actor, task, and acquaintances with that of object-oriented computation such as object, message, and attributes respectively [Meseguer (1993)].

2 GROUP-WIDE ARCHITECTURE IN REWRITING LOGIC

Both baselevel and metalevel of the GWA are composed by each group of actors [Watanabe and Yonezawa (1991)]. In order to describe them in rewriting logic, we define objects, messages, and rewrite rules as terms. Furthermore we define the term rewriting system and some functions to construct a causally connection between the baselevel and the metalevel. We make use of rewriting logic, so that terms and rewriting correspond to states and transition of the computational system, respectively. A simple example, modelling migration of objects, is executed on our system.

2.1 System Design

We define the term, termschema data types (used for the definition of rewrite rules), object, message, configuration, and rewrite rule as follows:

```
-- term
data Name = String
data Term = Var Name                -- variable
          | Val Name                -- value
          | Con Int                 -- number
          | App Term Term           -- function application
          | Lis [Term]              -- list
          | Par Term Term           -- pair

-- termschema
data TermSchema = SVar Name         -- variable
                | SVal Name         -- value
                | SCon Int          -- number
                | SApp TermSchema TermSchema -- function application
                | SLis [TermSchema] -- list
                | SPar TermSchema TermSchema -- pair

-- object
Lis [ Val "Oid", Lis [...], ...]

-- message
Lis [(Par Val "Mid" Val "Des"), ...]

-- configuration
Par list-of-objects list-of-messages

-- rewrite rule
(lhs,rhs)
  where lhs = SPar list-of-lhs-objects list-of-lhs-messages
        rhs = SPar list-of-rhs-objects list-of-rhs-messages
```

For example, the message handler named "MH", a message sent to it, and a rewrite rule for it are as follows:

```
-- message handler
mh = Lis [(Val "MH"), mh_acq]
-- message
msg = Lis [(Par Val "mh" Val "MH"),
           Lis [Par (Val "target") (Val "meta(A)"),
                Par (Val "destination") (Val "MH'')]]]
-- message handler's rewrite rule (right hand side is omitted)
rule_of_mh = (lhs, rhs)
-- left hand side
lhs = SPar msgs_of_lhs objs_of_lhs
-- messages of left hand side
msgs_of_lhs = SLis [SLis [(SPar (SVal "TASKM") (SVar "mhname")),
                           (SVar "arg")],
                    SVar "restmsg"]
-- objects of left hand side
objs_of_lhs = SLis [SLis [(SVar "mhname"),
                          (SLis [(SPar (SVal "th") (SVar "thname")),
                                   (SVar "restatt")])],
                    SVar "restobj"]
```

Moreover, we define some functions which can translate data and rewrite rules in the baselevel into data in the metalevel. Inverse functions to those are also needed.

```
-- from base to meta
make_meta_conf :: [(TermSchema, TermSchema)] -> Term -> Term
-- from meta to base
baserule :: Term -> [(TermSchema, TermSchema)]
baseconf :: Term -> Term
```

Function `make_meta_conf` needs two arguments – a list of rewrite rules and a baselevel configuration – and translates them into data in the metalevel. Function `baserule` needs one argument – metalevel representation of rewrite rules of baselevel – and translates it into rewrite rules used in the baselevel. Similarly, the function `baseconf` needs one argument – the metalevel representation of the configuration of the baselevel – and translates it into the configuration representing the baselevel.

When the metalevel representations of configuration and rewrite rules for baselevel are changed by a rewrite rule of metalevel, we call this metalevel execution.

2.2 An Example

Migration of an object from node (a group of metaobjects) to node is described as an example of group-wide reflection. A node consists of some metalevel objects, such as Mail Handler, Task Handler, Database, Evaluator, and Migrator. Migrator is a special object that has rules of doing migration.

When Evaluator accepts the message with the requirement to migrate an object A_1 in node N_1 to another node N_2 and rules of migration, it starts to interpret the rules,

Table 1 Definition of some notations

Notation	Meaning
S	A (baselevel) system composed of terms.
C	The configuration in S . We denote it as $\langle M, O \rangle$.
M	The set of messages in C .
O	The set of objects in C .
R	The set of rewrite rules for S .
Γ_S	The set of all configurations for S .
$\Gamma_{\uparrow S}$	The set of all possible configurations for $\uparrow S$.
$\mathcal{R}$	The set of rewrite rules for $\uparrow S$.
$\langle n \mid atts \rangle$	An object that has its name n , and attributes $atts$.
$[t, v]$	A message which sent to target t with value v .
$(t1 \rightarrow t2)$	An abbreviation of rewrite rule like $[t1, v] \langle t1 \mid atts \rangle \rightarrow [t2, v] \langle t1 \mid atts \rangle \dots$
$\uparrow C$	The metaconfiguration for C .
$\uparrow M$	The set of messages in $\uparrow C$.
$\mathcal{O}(O, R)$	The set of objects in $\uparrow C$. A database in it has the metalevel representations of O and R as its data.

referring to already-known data and rewritten terms. The rules are as follows: obtain the Migrator of N_2 , ask for the address of the database of N_2 , get the immigrant's new address in N_2 , and perform migration.

3 CORRECTNESS OF $\uparrow S$

Now let S be a system consisting of terms and $\uparrow S$ be the metalevel representation for S . We show that $\uparrow S$ correctly represents S in terms of transitions between configurations [Watanabe and Yonezawa (1991)]. Correctness of $\uparrow S$ means that every transition in S can be represented in $\uparrow S$ and any actual transition sequence in $\uparrow S$ correctly represents a transition in S . In order to prove that, we provide some notations and definitions.

3.1 Some notations and definitions

In our formalization the objects in $\uparrow S$ are a message handler (for short **mh**), a task handler (**th**), a database (**db**), an evaluator (**ev**), and customer (**cus**) for **th**. These objects defined in [Watanabe and Yonezawa (1991)] play almost the same role in our work.

- Mail Handler (**mh**): It receives messages from $\uparrow S$ or another system and sends them to the task handler or another mail handler that belongs to another system.
- Task Handler (**th**): It knows other metalevel objects and arranges metalevel computation representing the message processing in the baselevel. When **th** receives a metamessage, it spawns a customer (**cus**) which continues the subsequent computation (representing the processing of the message).
- Database (**db**): It has metalevel representations of objects and rewrite rules of S .

Table 2 Definition of transition relation (sequence)

Notation	Meaning
$C_1 \longrightarrow C_2$	Applying a rewrite rule in R once to $C_1 \in \Gamma_S$, it changes to $C_2 \in \Gamma_S$. $C_1 \xrightarrow{r} C_2$ means that C_1 changes to C_2 with $r \in R$.
$K_1 \xrightarrow{(*)} K_2$	Applying appropriate rewrite rules in $\mathcal{R}$ some times to $K_1 \in \Gamma_{\uparrow S}$, it changes to $K_2 \in \Gamma_{\uparrow S}$, however, the rules do not change metalevel representation for objects and rewrite rules in S .

- **Concurrent Evaluator (ev):** It accepts the metalevel representations of message, object and rewrite rule and carries out the transition specified by the rule.

Let m be a message in S . Then the metalevel representation of m is denoted by $\uparrow m$. The metamessage corresponding to $\uparrow m$ is denoted by $[\text{mh}, \uparrow m]$. All rewrite rules in S are represented as data of db in $\uparrow S$. Thus we can define $\uparrow M$, $\uparrow C$, and $\mathcal{R}$ as follows:

$$\begin{aligned}\uparrow M &= \{[\text{mh}, \uparrow m] \mid m \in M\}, & \uparrow C &= \langle \uparrow M, \mathcal{O}(O, R) \rangle \\ \mathcal{R} &= \{(\text{mh} \rightarrow \text{th}), (\text{mh} \rightarrow \text{omh}), (\text{th} \rightarrow \text{db}), (\text{db} \rightarrow \text{cus}), \\ &\quad (\text{cus} \rightarrow \text{ev}), (\text{cus} \rightarrow \text{mh}), (\text{ev} \rightarrow \text{db})\}\end{aligned}$$

where omh denotes the name of another message handler. The notation for transition relation and transition sequence between configurations is described in Table 2:

3.2 Proof in outline

Before showing our proof, we must note that for any $K \in \Gamma_{\uparrow S}$, there is always a corresponding $C \in \Gamma_S$, that is, K is one of the metalevel representation for C . Thus we define the set $\overline{\uparrow C}$ of K 's equivalent to $\uparrow C$ as follows:

$$\overline{\uparrow C} = \{K \in \Gamma_{\uparrow S} \mid K \xrightarrow{(*)} \uparrow C, \text{ or } \uparrow C \xrightarrow{(*)} K\}.$$

It is important that when $K \in \Gamma_{\uparrow S}$, if $K \in \overline{\uparrow C}$, then the metalevel representations of objects and rewrite rules in S which are data in database of K are the same as those in $\uparrow C$'s. Correctness of $\uparrow S$ can be stated as follows:

Theorem 1 Suppose $C_1, C_2 \in \Gamma_S$. Then $\overline{\uparrow C_1} \implies \overline{\uparrow C_2}$ iff $\exists r \in R$ such that $C_1 \xrightarrow{r} C_2$, where $\implies$ represents a transition-like relation on $\Gamma_{\uparrow S} / \xrightarrow{(*)}$.

Proof of Necessity: We show that $\uparrow C_1 \xrightarrow{*} \uparrow C_2$. Let m be a message in the baselevel which causes a one step transition between C_1 and C_2 using a rewrite rule r in the baselevel. Then the metalevel representation of m appears in $\uparrow C_1$ as the value of a message $[\text{mh}, \uparrow m]$. We pay attention to the value $\uparrow m$, we can make actual transition sequence. This says that the necessity part of our statement is true.

Proof of Sufficiency: To show the sufficiency, we need the following lemma.

Lemma 1 For all $K \in \Gamma_{1S}$, there is a metalevel representation $K' \in \Gamma_{1S}$ and $n \in \omega$ such that $K \xrightarrow{(n)} K'$ and $\mathcal{M}(K') = \{[\text{ev}, \uparrow m] \mid m \in M\}$, where $\mathcal{M}(K')$ denotes the set of messages in K' and M is the set of messages in the C for which K is equivalent to $\uparrow C$.

By above lemma, we can obtain that $K_1 \xrightarrow{(*)} K'_1$ where $K_1, K'_1 \in \overline{\uparrow C_1}$. Now let σ be a rewrite rule in $\mathcal{R}$ that changes metalevel representations of objects and/or rewrite rules of its own baselevel and obtain a transition $K'_1 \xrightarrow{\sigma} K_2$, $K_2 \in \Gamma_{1C_2}$. Then we can get a transition sequence such that $K_1 \xrightarrow{(*)} K'_1 \xrightarrow{\sigma} K_2$. To apply σ to K'_1 , one message in K'_1 is needed, so we denote it $[\text{ev}, \uparrow m]$. The existence of m means that there is the rewrite rule r such that $C_1 \xrightarrow{r} C_2$ with m . Thus we have shown that the sufficiency direction. Hence we have shown that the theorem is true. $\square$

4 CONCLUSION AND FUTURE WORK

This research is a first attempt to formalize group-wide reflection in rewriting logic and lays a foundation for future formalization work.

When S is a system consists of terms and $\uparrow S$ is the metalevel representation for S , we have shown that $\uparrow S$ correctly represents S in terms of transitions between configurations. Currently, we are considering the proof of correctness of $\uparrow S$ using the algebraic specification language CafeOBJ which is an extended version of OBJ3 [Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud (1992)] and based on rewriting logic.

Our GWA is based on the actor model. After this we will describe some other examples for GWA (eg., fault-tolerant system, mechanism for dynamic load sharing, etc). Based on these works, we expect that in future it will be possible to construct a new general reflective architecture based on rewriting logic. Moreover, we expect that it will be possible to design and implement metalevel architectures of concurrent reflective computations more suitable to rewriting logic. This architecture would be more effective in analyzing the properties of the concurrent reflective computation from several aspects.

REFERENCES

- Gul Agha. (1987) *Actors : A Model of Concurrent Computation in Distributed Systems*, The MIT Press.
- Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. (1992) *Introducing OBJ. Technical Report SRI-CSL-92-03*, Computer Science Laboratory, SRI International.
- Paul Hudack, Philip Wadler, et al. (1992) Report on the programming language Haskell, a non-strict purely functional language (Version 1.2). *Technical report*, Yale University / Glasgow University.
- Hiroshi Ishikawa, Kokichi Futatsugi, and Takuo Watanabe. (1994) Concurrent Reflective Computations in Rewriting Logic (in Japanese). *11th Conference Proceedings Japan Society for Software Science and Technology*, 313–6.

- Hiroshi Ishikawa, Kokichi Futatsugi, and Takuo Watanabe. (1995a) Concurrent Reflective Computations in Rewriting Logic. *Theory of Rewriting Systems and Its Applications*, RIMS Kokyuroku **918**, 292–8.
- Hiroshi Ishikawa, Kokichi Futatsugi, and Takuo Watanabe. (1995b) Concurrent Reflective Computations in Rewriting Logic (in Japanese). *12th Conference Proceedings Japan Society for Software Science and Technology*, 49–52.
- Pattie Maes. (1987) Computational reflection. *Ph. D. Thesis 87-2*, Artificial Intelligence Laboratory, Vrije Universiteit Brussel.
- José Meseguer, Kokichi Futatsugi, and Timothy Winkler. (1992) Using Rewriting Logic to Specify, Program, Integrate, and Reuse Open Concurrent Systems of Cooperating Agents. *Technical Report SRI-CSL-92-11*, Computer Science Laboratory, SRI International.
- José Meseguer. (1993) A logical theory of concurrent objects and its realization in the maude language. *Research Directions in Concurrent Object Oriented Programming*, The MIT Press, 314–90.
- Brian Cantwell Smith. (1984) Reflection and Semantics in Lisp. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 23–35.
- Takuo Watanabe, Akinori Yonezawa. (1990) Reflection in an Object-Oriented Concurrent Language. *ABCL: An Object-Oriented Concurrent System*, 45–70, The MIT Press.
- Takuo Watanabe and Akinori Yonezawa. (1991) An Actor-Based Metalevel Architecture for Group-Wide Reflection. *Proceedings of REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL)*, Lecture Notes in Computer Science **489**, 405–25.
- Takuo Watanabe. (1993) Tutorial Introduction to Computational Reflection (in Japanese). *Computer Software*, **11**(3), 5–14.

BIOGRAPHY

Hiroshi ISHIKAWA: Ph.D. student of Japan Advanced Institute of Science and Technology (JAIST). His current research interests are term rewriting system, computational reflection, and algebraic specifications. He is a member of JSSST.

Kokichi FUTATSUGI: Professor of JAIST. He is currently also a Senator of JAIST. In 1975, he entered ETL, MITI, Japanese Government. From 1985 to 1993, he was a section chief at ETL. He was assigned to a Chief Senior Researcher of ETL in April 1992. In April 1993, he was assigned to a professor at JAIST. His current research interests include formal methods for software development, declarative computer language systems, algebraic specifications and their applications to software methodology.

Takuo WATANABE: Associate Professor of JAIST, joined the faculty in 1991. He received his Ph. D. from Tokyo Institute of Technology in 1991. From 1990 to 1992, he worked at Univ. of Tokyo and Univ. of Illinois at Urbana-Champaign as a JSPS junior researcher. His research interests include: metalevel architectures and computational reflection, object-oriented programming, distributed and mobile computing. He currently is investigating a theory of computational reflection based on rewriting. He is a member of ACM, JSSST and JSPS.