数理解析研究所講究録
918 巻 1995 年 292-298

Concurrent Reflective Computations in Rewriting Logic

Hiroshi Ishikawa Kokichi Futatsugi Takuo Watanabe

{h-ishika,kokichi,takuo}@jaist.ac.jp
School of Information Science,
Japan Advanced Institute of Science and Technology †
(JAIST)

July 26, 1995

Abstract

Rewriting logic can represent dynamic behaviors 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 modeling cooperative behaviors of several software modules (or agents). This paper provides some basic considerations on methods of modeling the group-wide architecture in 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(IBA)[11]. Each object in the system has its own metaobject which governs its computation. Reflective computation[5, 10, 13] is realized by sending message to metaobjects. The computational activity in an object is sequential. This implies each metaobject model 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)[12]. It is a specific concurrent reflective computation model based on Group-Wide Reflection. It has a potential of modeling cooperative behaviors of several software modules (or agents). Our group-wide architecture is based on the Actor model. All reflective operations are performed solely via

†15 Asahidai, Tatsunokuchi, Ishikawa 923-12, Japan.

message sends, which are interpreted at the metalevel concurrently with interpretations of actors in the baselevel.

Rewriting logic[4, 6, 7] can represent dynamic behaviors of concurrent and/or reactive systems declaratively. Declarative descriptions in rewriting logic are expected to be amenable to analysis of interesting properties.

In this paper, we propose some basic considerations on methods of modeling the group-wide reflection in rewriting logic, and implement it[2, 3]. We use a functional programming language Gofer, which is a sublanguage of functional programming language Haskell[8], to describe our model and perform an example of *inter-group* migration of object.

1.1 Reflection

A system is said to be causally connected to its domains(Fig.1 (a)) if the internal structures and the domain they represent are linked in such a way that if one of then 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:

(i) The system always has an accurate representation of itself.

(ii) The status and computation of the system are always in compliance with this representation. This means that a reflective system can actually bring modifications to itself by virtue of its own computation.

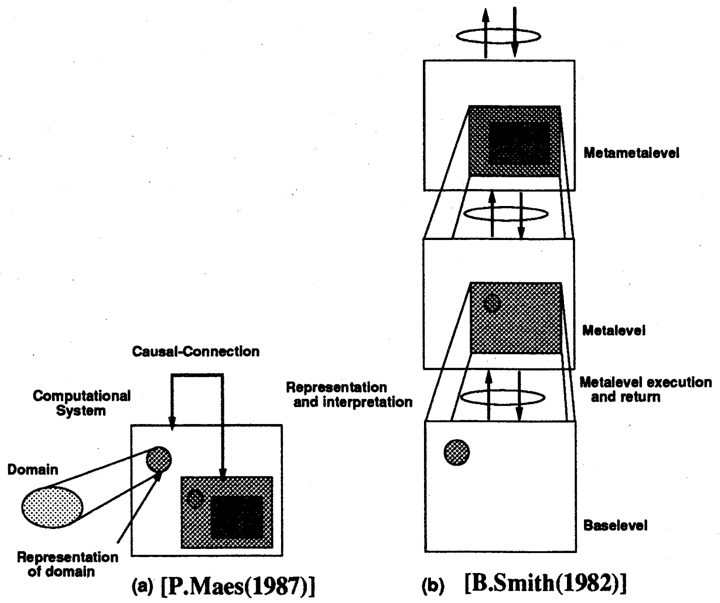

Figure 1: The notion of reflective system

When we implement a reflective system like Fig.1(a), the idea of procedural reflection(Fig.1(b)) proposed by B.Smith[9] is useful to do so.

1.2 Group-Wide Architecture

In this architecture, a group of metalevel objects constructs the metalevel. The behavior of an object is not managed by metaobjects; rather, the collective behavior of a group of objects is represented as the coordinated actions of a group of metalevel objects, which construct meta-group. As mentioned above, the Group-Wide Architecture we make use of in this study is based on the Actor model. Each object and a group of objects are regarded as Actors. Of course, metalevel objects and a group of metalevel objects are also regarded as Actors.

1.3 The Actor model

The Actor model[1] is one of the concurrent computation models. In the Actor model, a system consists of a set of autonomous computation agent called Actors; each Actor represents a physical or conceptual entity in the system's problem domain.

The features of the Actor model are as follows.

Message passing: Actors can send messages to their acquaintance asynchronously.

Dynamic topology: A communication topology can be changed by sending an Actor's address with a message.

Concurrency: Any two Actors may run simultaneously.

Let S be a system composed by actors. A configuration C represents a computational state of S at a certain frame of reference. This is represented by a pair which is composed by two sets. One is a finite set of actors in S and the other is a finite set of tasks in S. Computation in S is modeled as transitions between configurations in the set of all configurations of S.

Now we construct a causally connected metalevel representation for an actor system S. Let $\uparrow S$ be a metalevel representation for S. It 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. In our model, $\uparrow S$ is also used as a meta-circular interpreter for S. Therefore, the causal-connection between S and $\uparrow S$ is automatically guaranteed.

In this way, it becomes clear that $\uparrow S$ is also an actor system, then a special configuration of $\uparrow S$ is called a meta-configuration.

1.4 Rewriting logic

First, we introduce the definition of rewriting logic [4, 6, 7].

Definition 1 A (*labelled*) *rewriting theory* $\mathcal{R}$ is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where Σ is a ranked alphabet of function symbols, E is a set of Σ-equations, L is a set of *labels*, and R is a set of pairs $R \subset L \times (T_{\Sigma,E}(X)^2$ whose first component is a label and whose second component is a pair of E-equivalence classes of times, with $X = \{x_1, \cdots, x_n, \cdots\}$ a countably infinite set of variables. Elements of R are called *rewrite rules*. We understand a rule $(r, ([t], [t']))$ as a labelled sequent and use for it the notation $r : [t] \rightarrow [t']$. To indicate that $\{x_1, \cdots, x_n\}$ is the set of variable occurring in either t or t', we write $r : [t(x_1, \cdots, x_n)] \rightarrow [t'(x_1, \cdots, x_n)]$, or in abbreviated notation $r : [t(\overline{x^n})] \rightarrow [t'(\overline{x^n})]$.■

Given a rewrite theory $\mathcal{R}$, we say that $\mathcal{R}$ *entails* a sequent $[t] \rightarrow [t']$ and write $\mathcal{R} \vdash [t] \rightarrow [t']$ if and only if $[t] \rightarrow [t']$ can be obtained by finite application of the following *rules of deduction:*

1.**Reflexivity.** For each $[t] \in T_{\Sigma,E}(X)$,

$$\frac{}{[t] \rightarrow [t]}$$

2.**Congruence.** For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{[t_1] \rightarrow [t'_1] \quad \cdots \quad [t_n] \rightarrow [t'_n]}{[f(t_1, \cdots, t_n)] \rightarrow [f(t'_1, \cdots, t'_n)]}$$

3.**Replacement.** For each rewrite rule $r : [t(x_1, \cdots, x_n)] \rightarrow [t'(x_1, \cdots, x_n)]$ in R,

$$\frac{[w_1] \rightarrow [w'_1] \quad \cdots \quad [w_n] \rightarrow [w'_n]}{[t(\overline{w}/\overline{x})] \rightarrow [t(\overline{w'}/\overline{x})]}$$

4.**Transitivity.**

$$\frac{[t_1] \rightarrow [t_2] \qquad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$$

■

Rewriting logic is understood as a method of correct reasoning about some class of entities. For rewriting logic, the entities above are concurrent systems having *states*, and evolving by means of *transitions.* Therefore, we can describe concurrent object-oriented models with rewriting logic. In this logic, concurrent object-oriented computation is formalized as concurrent ACI-rewriting. The Actor model is a special case of those models.

2 Group-Wide Architecture in Rewriting Logic

In [12], both baselevel and metalevel of the group-wide architecture are composed by an each group of actors. In order to describe them in rewriting logic, we define them, namely actors(objects), messages, and rewrite rules as terms. After that, we compose the Term Rewriting System and some functions to construct a causally connected between baselevel and metalevel.

Because of using in rewriting logic to describe our system, terms and rewriting correspond to states and transition of the computational system, respectively. In our group-wide architecture, the state of the system consists of finite set of messages and finite set of objects. In addition, the transition of the system means a change of the states. In rewriting logic, we can represent terms as the states and a pair of terms as a change of the states. Furthermore a simple example, modeling migration of objects, is executed on our system.

2.1 System Design

In the first place, we define some terms and termschemas (using for definition of rewrite rules) as follows:

```
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

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

In the second place, we define actor(object), message, configuration, and rewrite rule as follows:

```
-- actor(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
```

Finally, we define some functions which can translate data and rewrite rules in baselevel into data in metalevel and those inverse one.

```
-- 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 – list of rewrite rules and configuration of baselevel – and translates them into data of metalevel. Function `baserule` needs one argument – metalevel representation of rewrite rules of baselevel – and translates it into rewrite rules used in baselevel. In same way, function `baseconf` needs one argument – metalevel representation of configuration of baselevel – and translates it into configuration representing baselevel.

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 is constructed some metalevel objects, such as

External Mailer, Task Handler, Database, Evaluator, and Migrator. Migrator is a special object that has a method 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, referring already-known data and rewritten terms. The migration rules are as follows.

- Obtain the Migrator of the destination N_2.
- Ask for the address of the database of destination N_2.
- Get the immigrant's new address in the destination N_2.
- Perform Migration.

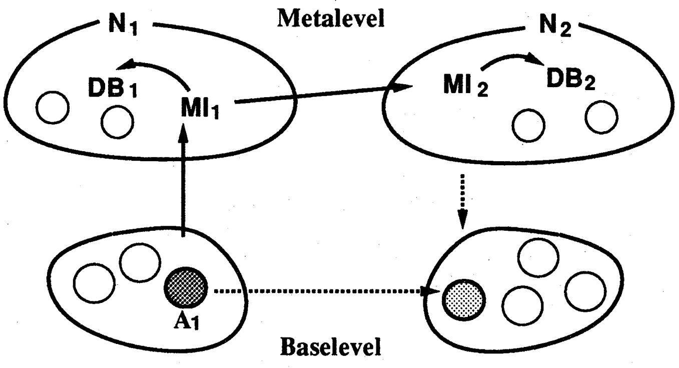

Figure 2: Migration for object

3 Discussion

Constructing the example in a way mentioned above, we faced the following problems:

- When we describe a mechanism for inter-group migration of objects, we must consider the direct sum of baselevel and metalevel of those groups. In our case, there are two groups N_1 and N_2. Then there will be a possibility of conflict of rewrite rules, that is, a rewrite rule application to a configuration can not be decided.
- Whether do the direct sum of the metalevel both N_1 and N_2 represent the direct sum of their baselevel or not.

We will be able to clear the former in the following way. When we define the rewrite rules of such groups, we do the rules without same identifiers beforehand.

The latter one is as follows. In [12], it is proved that $\uparrow S$ correctly represents S in terms of transition relations, where S and $\uparrow S$ mean a group of actors and the group which forms a metalevel representation of S respectively. In our case, we will be able to prove such a property in the same way.

4 Conclusion and Future Work

We described group-wide reflection based on actor model in rewriting logic. By constructing direct sum of configurations and rewriting rules in two groups, we were able to describe a mechanism for inter-group migration of objects. We can expect that we describe it more than three groups in the same way.

The research reported in this paper is their first attempt of formalizing group-wide reflective architecture in rewriting logic. This research would become important.

Our group-wide architecture is based on the Actor model. After this we will describe some other examples for GWA. Based on this work, we expect that it will be possible to construct a new general reflective architecture based on the rewriting logic in future. Moreover we expect that it will be possible to design and implement metalevel architecture of concurrent reflective computations more sutable to Rewriting Logic. The architecture would be more effective in analyzing the properties of the concurrent reflective computation from several aspects.

References

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

[2] Hiroshi Ishikawa, Kokichi Futatsugi, and Takuo Watanabe. Concurrent Reflective Computations in Rewriting Logic (in Japanese). In *11th Conference Proceedings Japan Society for Software Science and Technology*, pp. 313–316. Japan Society for Software Science and Technology(JSSST), 1994.

[3] Hiroshi Ishikawa. Concurrent Reflective Computations in Rewriting Logic (in Japanese). Master Thesis, Japan Advanced Institute of Science and Technology, February 1995.

[4] José Meseguer, Kokichi Futatsugi, and Timothy Winkler. 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, 1992.

[5] Pattie Maes. Computational reflection. Ph. D. Thesis 87-2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[6] José Meseguer. Conditional rewriting logic as a unified model of concurrency. Technical Report SRI-CSL-92-08, Computer Science Laboratory, SRI International, 1992.

[7] José Meseguer. A logical theory of concurrent objects and its realization in the maude language. In Peter Wegner Gul Agha and Akinori Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming.* The MIT Press, 1993.

[8] Paul Hudack, Philip Wadler, et al. Report on the programming language Haskell, a non-strict purely functional language (Version 1.2). Technical report, Yale University / Glasgow University, 1992.

[9] Brian Cantwell Smith. Reflection and Semantics in a Procedural Language(Ph D.Thesis). Technical Report TR-272, Laboratory for Computer Science, MIT, 1982.

[10] Brian Cantwell Smith. Reflection and Semantics in Lisp. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 23–35, 1984.

[11] Takuo Watanabe, Akinori Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *ABCL:An Object-Oriented Concurrent System*, pp. 45–70. The MIT Press, 1990.

[12] Takuo Watanabe, Akinori Yonezawa. An Actor-Based Metalevel Architecture for Group-Wide Reflection. In *Proceedings of REX School/Workshop on Foundations of Object-Oriented Languages(REX/FOOL)*, Lecture Notes in Computer Science 489, pp. 405–425, New York, 1991. Springer-Verlag.

[13] Takuo Watanabe. A Tutorial Introduction to Computational Reflection (in Japanese). *Computer Software*, 11(3):5–14, May 1994.