Modeling Group Reflection in a Simple Concurrent Constraint Language

Hiroyasu SUGANO

FUJITSU LABORATORIES, IIAS

1-17-25 Shin-Kamata, Ota-ku, Tokyo 144, Japan

E-mail: suga@iias.flab.fujitsu.co.jp

Abstract

We propose a reflective computation model *CCR* based on a simple concurrent constraint programming framework. Primitive communication in *CCR* is carried out by means of publishing primitive constraints, and a group reflection which allows concurrent execution of both object level and meta level computations is formalized. *CCR* does not depend on any hardware architectures and implementation issues to keep its generality and abstractness, and we give its operational semantics in terms of a transition system. Thus, in our framework, complex behaviors of concurrent systems could be model naturally and transparently.

1 Introduction

Reflection and meta-level programming have received much attention in programming languages researches for recent years [1, 2, 3, 5, 6, 8]. Particularly, in case of logic programming, reflection provides a uniform framework in which pure logic languages can be extended in a flexible and logically transparent manner. In fact, taking advantages of reflection, we developed a reflective, but sequential, logic programming language R-Prolog*, and showed that lots of extra-logical predicates in programming language Prolog can be redefined with a logically clear semantics[8].

In concurrent systems, particular attention is paid to reflection, especially in concurrent object-oriented models[1]. The reason is that concurrent systems with reflective capability have more advantages than sequential systems, because concurrent systems can naturally model complex behaviors including co-operative works in real world activities, operating systems, programming systems with debugging facilities, and so on. Concurrency can also be incorporated in logic programming languages naturally. As its famous examples, we can see several committed-choice logic programming languages, being one useful embodiment of parallel execution of logic programs. One defect of committed-choice languages is that logical feature of programs cannot clearly be grasped. The notion of constraints give logical view back to concurrent logic programming languages.

In this paper, we propose a reflective computation model in concurrent constraint logic languages (called *CCR*) which allows concurrent executions of object level and meta level computations by means of group reflection. We can find a few attempts to incorporate reflective capability in concurrent logic language, for example [9]. However, their approaches do not make a good use of reflection because they only deal with reflection sequentially. This means that, when meta-level computation is executed, object-level computation is paused, and thus possibilities of reflective computation in concurrent systems are restricted unfortunately. By allowing group reflection, concurrent logic programming language will obtain wider application areas.

When we consider reflective computation in concurrent language, however, several problems arise. The level of abstraction is one of the important matters. Reflection can be regarded as an attempt to cross the language level and implementation level, which is unpleasant in a sense of abstractness. In order to keep our argument in a suitable abstraction level in R-Prolog*, we used formal semantics instead of stepping into implementation level. In this work, we also adopt the same way, and we want to avoid to consider

[1] For example, [10]. Lots of attempts in this area can be found in [3].

implementation matters such as number of processors, processor topology, and so on. One of our policies is to build a model which does not depend on those matters, and to enable us to argue formally in some abstract level. We give the operational semantics of *CCR* as a formal model of transition system, and we are working on further researches based on this model. Not only in application areas, theoretical results are also expected because reflection is a kind of programming technique in a current status.

2 Concurrent constraint model *CCR*

2.1 Basic Syntax

Our language is based on a very simple example of *cc* [7], and it is similar to the language in [4]. This language has a Flat GHC like syntax, and it has only two constraint (system) predicates, *equation* (=) and its negation (/=), for simplicity. As R-Prolog*, it has "quote" and "up" symbols and no function symbols, i. e. upped atomic formulas are considered as compound terms with functors in Prolog. Clauses and programs are defined similar to those of GHC. Thus,

$$H :- G_1, \ldots, G_m | B_1, \ldots, B_n. \qquad (m, n > 0)$$

is a clause where H is a user-defined atom, $G_1, \ldots, G_m$ are constraint atoms (with only system predicates), $B_1, \ldots, B_n$ are body part which consists of user-defined atoms and constraints. In addition, we introduce an annotation symbol to incorporate grouping operation for agents. For clauses in *CCR* , user-defined atom in its body can be annotated by the annotation operator &, in order to make the atom be an origin of new group. For an user-defined atom a, $\&a$ is called annotated atom. We redefine clauses of *CCR* so that annotated atoms could appear in its body. For example,

```
P(X,Y) :- X=[a|X1] | Y=[b|Y1], &P(X1,Y1).
```

is a clause of *CCR*.

We next define agents in *CCR*, which are basic constituents of computational states in *CCR*. We assume that each agent appearing among the computation has an identifier and a group identifier. Identifiers are finite sequences of natural numbers, therefore the set of identifiers ID are defined as $N+$, a set of finite sequence of natural numbers. Each identifier is attached to the corresponding atom so that it reflect the tree structure of computational state of logic programming. This means the followings; for an atom with the identifier α in a goal, if it has child goals $G_1, \ldots, G_n$, then they have identifiers $\alpha 1, \ldots, \alpha n$ respectively. An ordering relation $\leq$ on ID is defined as follows; for identifiers α, β, if there exists an identifier δ such that $\alpha\delta = \beta$, $\alpha \leq \beta$.

A finite set of constraint atoms is called *Herbrand theory* or simply *constraint*.

Definition. 2.1 (Agents, Configurations)

1. Let α, β be an identifier such that $\alpha \leq \beta$, and A be an atom.A triple $\langle \alpha, \beta, A \rangle$ is called an *active agent*.

2. Let α, β be an identifier such that $\alpha \leq \beta$. V be a finite set of variables, C be a constraint, and P be a program. A quintuple $\langle \alpha, \beta, P, V, C \rangle$ is called a *constraint agent*.

3. A finite set c of active agent and constraint agent is called a *configuration* if, a) for each agent $AG \in c$ with the identifier α and any identifier $\alpha' \leq \alpha$, there exists a constraint agent $AG' \in c$ with the identifier α', and b) for each active agent $AG \in c$ with the identifier α, there is no agent in c with the identifier $> \alpha$.

□

In 1. and 2. in the above definition, α is called the identifier of the agent, and β is called the group identifier of the agent. Constraint agents correspond to local storages of groups of agents. The set of configurations (on the language L) is denoted by $\mathcal{C}$.

2.2 Meta-level representation of object-level objects

In the computational reflection framework, it is required that the representation system in meta-level can describe the object-level computational state, such that programs and constraints. We assume that some

special predicate (function) symbols are prepared to specify some terms as representations of object-level states.

Sorts (or types) representing semantic notions are following reserved function symbols, i. e. upped predicate symbols;

`id, goal, clause, prog, cnstr, agents, cvar.`

Here, these sorts correspond to identifiers, goals, clauses, programs, constraints, agents, communication variables, respectively. Communication variables are used in meta-interpreters to communicate with its parent agent. For one of the above reserved function symbol f and a term t, $f(t)$ is called a typed term and this means that t is a meta-level representation of type f. For instance, $goal(X)$ is a typed term. However, if we take a term arbitrary, it cannot always compose a intuitively meaningful typed term. Acceptable typed term should be able to unify the meta-level representation of the corresponding type.

Programs are represented as a list of clause representation. Clause representations are given as follows: for a clause $cl \equiv H :- G_1, \ldots, G_m \mid B_1, \ldots, B_n.$ its representation $\bar{cl}$ is a term $clause(qH, [qG_1, \ldots, qG_m], [qB_1, \ldots, qB_n])$, where qA is a quoted form of an atom A. Thus, a program $P = \{cl_1, \ldots, cl_n\}$ is represented by a list $[\bar{cl_1}, \ldots, \bar{cl_n}]$, where $\bar{cl_i}$ is the meta-level representation of a clause cl_i.

Next, we consider the meta-level representation of constraints. Since a constraint is a set of constraint atoms, meta-level representation of constraints can be given as lists of quoted constraint atoms. For instance, a constraint $\{X = f(Y), Y = Z, Z = a\}$ can be represented by a list `[#(X=f(Y)),#(Y=Z),#(Z=a)]` in meta-level. However, we cannot naturally represent constraint agents and the communication using constraints in meta-level in this manner. Thus, we adopt the representation with two components, one is a definite amount of constraints and the other is candidates for definite constraints before consistency check. The former has a form of $[C_\epsilon, \ldots, C_\kappa]$, where $C_i = [\hat{A}_1^i, \ldots, \hat{A}_{n_i}^i | X_i]$ is corresponding representation of constraints in constraint agents of the identifier i. The latter has the form of $[X_{a_1}, \ldots, X_{a_m}]$ where $a_1, \ldots, a_m$ are identifiers of children agents.

Finally, we briefly describe meta-agents and meta-configurations. Here we use the notion of the meta-interpreter of *CCR*, though we do not give its precise definition here. We assume that every constraint agent in a configuration has its own meta-agent with the meta-interpreter mi. Thus, for a constraint agent $AG = \langle a, \beta, P, V, C \rangle$ with its children agents $AG_i (1 \leq i \leq m)$. AG is assumed to have its own meta-goal

$$mi_c(\hat{a}, \hat{P}, \hat{V}, \hat{C}, [X_{a_1}, \ldots, X_{a_m}], O_\beta).$$

where O_β is a variable for constraint publication called *communication variable* to its parent agent. And for an active agent $AG_i = \langle a_i, a, A_i \rangle$ with its parent agent AG, AG_i is assumed to have its own meta-goal

$$mi_a(\hat{a_i}, \hat{P}, \hat{A_i}, \hat{C}, X_{a_i}).$$

where X_{a_i} is a variables common with its parent. Group identifiers do not appear in those meta-interpreters because the parent relation is implicitely represented by communication variables. Meta-agents have their own environments, such that identifiers, group identifiers, programs and constraints, and usually they are determined by their object-level counterparts. By using this meta-interpreter, we can get meta-configuration of a configuration of *CCR*. For a configuration c, the meta-configuration of c, $\uparrow c$, is defined as a new configuration built by replacing all agents by their meta-agents. The important requirement on meta-interpreter of *CCR* is faithfulness of its behavior, which means c and $\uparrow c$ behaves exactly same. We will discuss the meta-interpreter matters in other papers.

3 Reflective computation in *CCR*

3.1 Reflective definition clauses

We now describe what is reflective computation in *CCR*. Reflection begins at the point when a reflective goal (agent) is executed, which is defined and declared by reflect definition clause mentioned below. Active agents with user-defined atom which is defined by reflect definition clause are called *reflective agents*. When a reflective agent $AG_r = \langle a_i, a, R(\bar{t}) \rangle$ with a parent constraint agent $AG_c = \langle a, \beta, P, V, C \rangle$ is executed, the identifier, the group identifier, the program P and the constraint C are reified and treated as data in the reflective computation. At the same time, following the grouping information specified in the reflect definition clause, data of other agents of the same group are collected.

Reflect definition clauses describe a reflective computation in meta level. Its head must have a special form

$Reflect(reflect\_goal, reflect\_template, other\_agents)$.

where *Reflect* is a reserved predicate symbol. *reflect_goal* specifies the form of this reflect goal in object level. *reflect_template* is a list whose elements are typed terms, and *other_agents* is a variable at first. For example, a list $[id(Id), gid(Gid), prog(P), constr(C)]$ is a reflective template. When a reflective agent $RA = \langle Id_i, Gid_i, R(\bar{t}), P_i \rangle$ in a configuration c is executed, the meta-configuration $\uparrow c'$ of the subconfiguration c' beginning at the parent of RA, and the reflecitve meta-level agent is composed according to the reflective template. Then, the meta-level computation proceeds in a meta-configuration $\uparrow c'$, and other agents out of c' are executed in object level.

We give the formal definition of the operational semantics on reflective computation in the next subsection.

3.2 The operational semantics of *CCR*

The set of configurations of *CCR* is written $\mathcal{C}_R$. Meta-level representations defined in the previous section is modified according to this extension.

Now we define the operational semantics of *CCR* as a transition system on $\mathcal{C}_R$.

Definition. 3.1 A binary relation $\mathcal{R}_R$ on the set of configuration $\mathcal{C}_R$ is defined as follows. Let $\gamma, \gamma' \in \mathcal{C}_R$.

$\gamma \mathcal{R}_R \gamma' \iff$ one of the following conditions holds;

1. For an active agent $AG = \langle \alpha, \beta, A \rangle \in c$ whose parent constraint agent is $AG_C = \langle \beta, \beta', P, V, C \rangle$, where A is a user-defined atom and a clause $cl \in P$, $cl \equiv (H :- G \mid B_1, \ldots, B_m.)$ (each occurrence of variables are ones which does not appear in c), such that (a) $C' \equiv C \wedge (uA_i = uH) \wedge G$ is consistent, and (b) $C \vdash_{ETM} \exists^{\bar{z}}((uA_i = uH) \wedge G)$ $\gamma' = (\gamma \backslash \{AG, AG_C\}) \cup \{\langle \beta, \beta', P, V', C' \rangle, \langle \alpha 1, \beta, B_1 \rangle, \ldots, \langle \alpha m, \beta, B_m \rangle\}$ where $\bar{z} = FV(cl)$.

2. For an active agent $AG = \langle \alpha, \beta, A \rangle \in c$ where A is a constraint atom such that, if C is the constraint in the constraint agent $AG_C = \langle \beta, \beta', P, V, C \rangle$, $C' \equiv C \wedge A$ is consistent, $\gamma' = (\gamma \backslash \{AG, AG_C\}) \cup \{\langle \alpha, \beta, P, V, C' \rangle\}$.

3. For an annotated active agent $AG = \langle \alpha, \beta, \&A \rangle \in c$ whose parent constraint agent is $AG_C = \langle \beta, \beta', P, V, C \rangle$, $\gamma' = (\gamma \backslash \{AG\}) \cup \{\langle \alpha, \beta, P, V', C \rangle, \langle \alpha 1, \alpha, A \rangle\}$ where $V' = FV(A)$.

4. For a constraint agent $AG = \langle \alpha, \beta, P, V, C \rangle \in c$ with its parent constraint agent $AG' = \langle \beta, \beta', P', V', D \rangle$, if there exists a constraint atom C_a such that $C \vdash_{ETM} C_a$, $D \not\vdash_{ETM} C_a$ and $FV(C_a) \cup V \neq \emptyset$, then $\gamma' = (\gamma \backslash \{AG'\}) \cup \{\langle \alpha, \beta, V', D \wedge C_a \rangle\}$. Similarly, if there exists a constraint atom C_a such that $C \not\vdash_{ETM} C_a$, $D \vdash_{ETM} C_a$ and $FV(C_a) \cup V \neq \emptyset$, then $\gamma' = (\gamma \backslash \{AG\}) \cup \{\langle \alpha i, \beta, V, C \wedge C_a \rangle\}$.

5. For a reflective agent $AG = \langle \alpha, \beta, R(\bar{t}) \rangle$ with its parent agent $AG' = \langle \beta, \beta', P, V, C \rangle$ and the list of its sibling agents AGs, γ' is constructed as follows. According to the reflective template of its reflect definition clause, reflective form $RA = Reflect(qR(\bar{t}), Template, AGs^M)$ is constructed, and meta-level agent $MA = \langle \beta 1, \beta, RA, \rangle$ and $MA' = \langle \beta, \beta', P^M, V^M, C^M \rangle$ is composed. Then, γ' is defined as $(\gamma \backslash \{AG'\}) \cup \{MA', MA\}$. Here, AGs^M is a list of meta-agents of AGs, (AG') is a set of all agents with identifier greater than AG'.

□

$\langle \mathcal{C}_R, \mathcal{R}_R \rangle$ is a transition system representing the operational semantics of *CCR*.

Let AG be an active agent with user-defined atom in a configuration γ. AG is said to be *in failure* if there is no clause in P satisfying the condition (a) of 1. in the above definition. AG is said to be *suspended* if (a) of 1. in the above definition is satisfied, but (b) of 1. can not be satisfied for any clauses satisfying (a).

Definition. 3.2 For a transition system $\langle \mathcal{C}_R, \mathcal{R}_R \rangle$ for *CCR*, a configuration $\gamma \in \mathcal{C}_R$ is called *terminal node* if there exists no γ' such that $\gamma \longrightarrow \gamma'$. Terminal nodes can be devided into following three groups.

1. A terminal node with no active agents is called a *success node*,
2. A terminal node such that some agents are in failure is called a *failure node*.
3. A terminal node such that all agents are suspended is called a *deadlock node*.

□

Before concluding this paper, we will show one example of reflection. We introduced the grouping operator & in Section 4.2. By using reflection, this operator can be re-defined in our framework. The following definition for the reflective predicate new-group define the same behavior as the grouping operator &.

```
Reflect(new-group(A),[id(Id),gid(Gid),prog(P),cnstr(C),cvar(O)],Z):- |
                                    mi_c(Id, P, V1, C,[Y], O),
                                    mi_a(Id1, P, A, C, Y),
                                    concat(Id, 1, Id1),
                                    varlist(A, V1),
                                    map_mi(Z).
```

Here, the predicate varlist returns the list of variables in A to $V1$, and map_mi applies meta-interpreter *mi* to each elements of Z.

4 Conclusion

In this paper, we proposed a reflective concurrent constraint language with the capability of group reflection. In our framework, grouping is specified by means of grouping operator given in program clause. Thus, it naturally has a clear structure based on the computational tree of logic programming, which makes us to analyze the behavior of reflective computation. It is still insufficient, however, because dynamic grouping is not modeled currently. This also models a kind of non-atomic publication of constraints. This makes us to formalize the truly distributed environment on which group reflection is realized.

It is expected that further research based on this framework can be continued in the various directions. One of them is a direction to several matters on semantics. Although we only give an operational semantics of *CCR* in this paper, other kinds of semantics from different viewpoints are worthy to be considered, for example, reactive behavior semantics and denotational semantics [4]. Another interesting direction is exploring useful applications of group reflection. We are now examining several examples of *CCR* in useful application areas.

Acknowledgments

This research was carried out as a part of Fifth Generation Computer System project of Japan. I wish to thank Jiro Tanaka, D. W. Shin, Youji Kohda and Munenori Maeda for their fruitful discussions and helpful comments.

References

[1] H. Abramson and M. H. Rogers, editors. *Meta-Programming in Logic Programming*. The MIT Press, 1989.

[2] M. Bruynooghe, editor. *Proceedings of the Second Workshop on Meta-programming in Logic (META 90)*. 1990.

[3] ECOOP/OOPSLA. *ECOOP/OOPSLA'90 Workshop. Reflection and Metalevel Architectures in Object-Oriented Programming*. 1990.

[4] H. Gaifman, M. J. Maher, and E. Shapiro. Reactive behavior semantics for concurrent constraint logic programs. In *Logic Programming, Proc. of the North American Conference '89*. pp. 553 - 569. 1989.

[5] P. M. Hill and J. W. Lloyd. Analysis of meta-programs. In *Proceedings of the Workshop on Meta-Programming in Logic Programming (META 88)*. pp. 27–42. 1988.

[6] P. Maes and D. Nardi, editors. *Meta-level Architectures and Reflection*. North-Holland. 1988.

[7] V. A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, 1989.

[8] H. Sugano. Semantic Considerations on Reflective Logic Programs. In *Proc. of the Logic Programming Conference '90*, pp. 95–104. ICOT, 1990.

[9] J. Tanaka. Meta-interpreters and reflective operations in GHC. In *FGCS '88*, pp. 774–783. 1988.

[10] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proc. of ACM Conf. on OOPSLA*, September 1988.