ICOT Technical Memorandom: TM-1171

TM-1171

Simple Concurrent Constraint Language and Concurrent Reflection

by
H. Sugano (Fujitsu)

April, 1992

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Simple Concurrent Constraint Language and Concurrent Reflection

Hiroyasu SUGANO

International Institute for Advanced Study of Social Information Science,
FUJITSU LABORATORIES LIMITED

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 concurrent 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, intelligent 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, 4, 6, 7, 10, 11]. 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[10, 11].

In concurrent systems, particular attentions are 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 intelligent 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.

[1] For example, [13]. Lots of attempts in this area can be found in [4].

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. We can find a few attempts to incorporate reflective capability in concurrent logic language, for example [12]. 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 concurrent 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 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.

Another problem we should solve is that of granularity of reflection. To realize concurrent reflection, what extent reflective computations have influence should be determined clearly. One extreme is global reflection adopted in RGHC, and another extreme is a local reflection where only the agent which caused reflection is relevant to the reflective computation. In our model, we adopt a grouping facility in the language to give a scope that reflection can have influence.

The organization of this paper is as follows. The next section gives a brief explanation on meta extension to syntax of logic programs proposed in R-Prolog*. In Section 3, we introduce a simple concurrent constraint language without reflection, *CC0* and give it an operational semantics as a transition system. In Section 4, we extend *CC0* by grouping operator, meta-level representation of object-level objects, and reflective capability. Finally, Section 5 concludes this paper.

2 Meta extension in constraint logic programs: preliminary

In this section, we review the basic notions around meta and reflective facilities in logic programming, especially in R-Prolog*. R-Prolog* has some extra syntax comparing to pure Prolog (Horn clause logic), including "quote", "up", and "down" symbols[2]. It does not distinguish between predicate and function symbols, and we deal with them as predicate symbols. We type them as capitalized alphabetical words, e. g. `Append` and `Qsort` are predicate symbols. Terms in R-Prolog* other than variables are quoted and upped atomic formulas. For example, if `P` is a predicate symbols and `X` is a variable, `P(X)` is an atomic formula and `^P(X)` and `#P(X)` are terms, where `^` is an up symbol and `#` is a quote symbol. We sometimes write `p(X)` for `^P(X)`, and we use such terms as usual terms in pure Prolog, which has usual function symbols. Because we also use capitalized words for variables following the conventional notation, we have one more "quote" symbol `$` for variables not to lead to confusion. For example, if `X` is a variable, `$X` is a quoted variable, and it does not behave as a variable, but as a constant. In the sequel, we use a word *forms* to refer to atomic formulas and terms, and use two extra-linguistic symbols u and q to denote up symbol `^` and quote symbol `#` and `$`.

An occurrence of a variable in a form is called *quoted* if the variable occurs in a quoted term. For example, `Y` is quoted in `^P(f(X), #Q(g(Y),a))`, but `X` is not quoted. Variables which are not quoted in a term is called *non-quoted* variables. According to those extension, equality

[2]Down symbol is only used in a restricted manner. Thus, we do not refer to it any more in this paper.

theory characterizing usual Herbrand universe (equality and its negation) have to be extended in a suitable way.

Definition. 2.1 The equality theory ETM, which stands for Equality Theory with Meta-extention, is the set of following formulas;

1. $\forall(X = X)$.
2. $\forall(X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n \leftrightarrow f(X_1, \ldots, X_n) = f(Y_1, \ldots, Y_n))$, for every n-ary predicate symbol F.
3. $\forall(X_1 = Y_1 \wedge \cdots \wedge X_m = Y_m \rightarrow (P(X_1, \ldots, X_m) \rightarrow P(Y_1, \ldots, Y_m)))$, for every m-ary predicate symbol P including $=$ and $\neq$.
4. $\forall(f(X_1, \ldots, X_n) \neq g(Y_1, \ldots, Y_m))$, for every two different predicate symbols F and G.
5. $\forall(t \neq X)$ for every term t properly containing X.
6. $\forall(X = Y \leftrightarrow uX = uY)$,
7. $\forall(ut = qt)$ for every term t without non-quoted variables,
8. $\forall(ut \neq qs)$ for every two different terms t and s such that t does not subsume s,
9. $\forall(qt \neq qs)$ for every two different terms t and s, where it is not the case both t and s are variables,
10. $\forall(qX = qY \rightarrow X = Y)$,

□

1.$\sim$ 5. conditions in the above definition are same as those of Clark's syntactic equality theory (CET), and 6. $\sim$ 10. conditions stipulate the behaviour of up and quote symbols.

Proposition. 2.1 Under the equality theory ETM as an axiom, the following propositions can be proved.

1. For every formula $\phi(X)$ where X is free and non-quoted, $\{X = t\} \vdash_{ETM} \phi(X) \rightarrow \phi(t)$.
2. For every formula $\phi(X)$ where X is free and quoted, $\{X = t\} \not\vdash_{ETM} \phi(X) \rightarrow \phi(t)$.

□

Upped and quoted terms are introduced in R-Prolog* in order to deal with meta-level objects legally in its own language. A quoted term represents a "term" itself as a syntactic object. It is dealt with similar to ground terms because we cannot substitute non-variable terms for it as shown in the above proposition. In other words, they are dealt with as a kind of data. Even for a variable `X`, it is not a data in itself, but a quoted form `$X` is dealt with as a data. Contrasted with quoted terms, upped forms have somewhat dynamic features. Variables in these terms are used as same as those used in ordinary logic languages except for the levels of terms bound to them are shifted up. In other words, they carry an information from object level to meta level, and from meta level to object level. Therefore, we can use upped form, for instance `^P(X)`, similar to compound terms in well-known Prolog.

At the end of this section, we mention models of the extended language. Let T_M be the set of all terms without non-quoted variables. Then, we define an equivalence relation on T_M.

Definition. 2.2 An equivalence relation R_M on T_M is defined as a reflexive, symmetric and transitive closure satisfying the following condition.

1. $ut \sim_{R_M} qt$ for $t \in T_M$.

2. For a pair of terms $t, s \in T_M$, if $t \sim_{R_M} s$ then $ut \sim_{R_M} us$.

3. For an n-ary predicate symbol F, and terms $t_1, \ldots, t_n, s_1, \ldots, s_n$ in T_M, if $t_i \sim_{R_M} s_i$ for all $i (1 \leq i \leq n)$, then $f(t_1, \ldots, t_n) \sim_{R_M} f(s_1, \ldots, s_n)$.

□

We call the quotient set T_M/R_M *extended Herbrand universe*, and models on the extended Herbrand universe are called *extended Herbrand models*.

3 A simple concurrent constraint language *CC0*

In this section, we present a simple concurrent constraint language without reflective capability, and its operational semantics as a transition system. This is similar to the language in [5] which is a very simple example of *cc* [9]. This language has a Flat GHC like syntax, and it has only two constraint (system) predicates, *equality* (=) 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,

$$H :- G_1, \ldots, G_m | B_1, \ldots, B_n.$$

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 the following arguments, we assume that the language (vocabulary) L consisting of predicate symbols, and so on, which will be used is defined previously. Thus, when we use the words programs , goals and so on, it is assumed that they are constructed within the language L implicitly.

We assume that each atom appearing among the computation has a unique identifier. Identifiers are finite sequences of natural numbers, therefore the set of identifiers ID are defined as $N+$, i. e. a free monoid over the set of natural numbers N. 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. 3.1 (Agents, Configurations)

1. Let α be an identifier, A be an atom, and P be a program. A triple $\langle \alpha, A, P \rangle$ is called an *active agent*.

2. Let α be an identifier, V be a finite set of variables, C be a constraint. A triple $\langle \alpha, V, C \rangle$ 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 $\beta \leq \alpha$, there exists a constraint agent $AG' \in c$ with the identifier β, and b) for each active agent $AG \in c$ with the identifier α, there is no agent in c with the identifier $> \alpha$.

□

The set of configurations (on the language L) is denoted by $\mathcal{C}$. The operational semantics of *CC0* is defined as a transition relation on $\mathcal{C}$.

Definition. 3.2 A binary relation $\mathcal{R}$ on the set of configuration $\mathcal{C}$ is defined as follows. Let $c, c' \in \mathcal{C}$.

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

1. For an active agent $AG = \langle \alpha i, A_i, P \rangle \in c$ where A_i is a user-defined atom and a clause $cl \in P$, $\grave{cl} \equiv (H :- G \mid B_1, \ldots, B_m.)$ (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) every extended Herbrand model of C is also an extended Herbrand model of $\exists \bar{z}((uA_i = uH) \wedge G)$, where C is the constraint in the constraint agent $AG_{C'} = \langle \alpha, V, C \rangle$, $c' = (c \backslash \{AG\}) \cup \{\langle \alpha i, V', C' \rangle, \langle \alpha i \cdot 1, B_1, P \rangle, \ldots, \langle \alpha i \cdot m, B_m, P \rangle\}$ where $V' = FV(A_i)$, $z = FV(\grave{cl})$.

2. For an active agent $AG = \langle \alpha i, A_i, P \rangle \in c$ where A_i is a constraint atom $A_i (1 \leq i \leq n)$ such that, if C is the constraint in the constraint agent $AG_{C'} = \langle \alpha, V, C \rangle$, $C' \equiv C \wedge A_i$ is consistent, $c' = (c \backslash \{AG, AG_{C'}\}) \cup \{\langle \alpha, V, C' \rangle\}$

3. For a constraint agent $AG = \langle \alpha i, V, C \rangle \in c$ with its parent constraint agent $AG' = \langle \alpha, V', D \rangle$, if there exists a constraint atom C_a such that $C \vdash_{ETM} C_a$, $D \nvdash_{ETM} C_a$ and $FV(C_a) \cup V \neq \emptyset$, then $c' = (c \backslash \{AG'\}) \cup \{\langle \alpha, V', D \wedge C_a \rangle\}$. Similarly, if there exists a constraint atom C_a such that $C \nvdash_{ETM} C_a$, $D \vdash_{ETM} C_a$ and $FV(C_a) \cup V \neq \emptyset$, then $c' = (c \backslash \{AG\}) \cup \{\langle \alpha i, V, C \wedge C_a \rangle\}$.

□

$\langle \mathcal{C}, \mathcal{R} \rangle$ is a transition system representing the operational semantics of *CC0*. We write $c \longrightarrow c'$ for $c \mathcal{R} c'$.

Let $AG = \langle Id_i, A_i, P \rangle$ be an active agent with user-defined atom in a configuration c. 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. 3.3 For a transition system $\langle \mathcal{C}, \mathcal{R} \rangle$ for *CC0*, a configuration $c \in \mathcal{C}$ is called *terminal node* if there exists no c' such that $c \longrightarrow c'$. Terminal nodes can be devided into following three groups.

1. A terminal node with no active agents is called a *succes 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*.

□

4 Concurrent reflection in *CCR*

4.1 Meta-level representation of object-level objects in *CC0*

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 predicate symbols:

`Id`, `Gid`, `Goal`, `Clause`, `Prog`, `Cnstr`, `Agents`.

For one of the above reserved predicate 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 $cl \equiv H$: - $G_1, \ldots, G_m$ | $B_1, \ldots, B_n$., its representation $\tilde{cl}$ 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 $[\tilde{cl_1}, \ldots, \tilde{cl_n}]$, where $\tilde{cl_i}$ is the meta-level representation of a clause cl_i.

Next, we consider the meta-level representation of constraints (Herbrand theory). 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 following representation. For a constraint agent $AG = \langle \alpha, V, C \rangle$ with its children agents $AG_1, \ldots, AG_n$, the representation of AG, $\tilde{AG} = [\tilde{\alpha}, \tilde{V}, \tilde{C}, [X_1, \ldots, X_n]]$ where $X_i = \tilde{AG_i}$ for all $i (1 \leq i \leq n)$. Based on those definitions, active and constraint agents are represented as follows. An active agent $AG_a = \langle Id, A, P \rangle$ can be represented by $\tilde{AG_a} = [\tilde{Id}, qA, \tilde{P}]$, and a constraint agent $AG_c = \langle Id, V, C \rangle$ can be represented by $\tilde{AG_c} = [Id, \tilde{V}, \tilde{C}]$.

Finally, we briefly describe the behavior of the meta-interpreter of *CC0* , though we do not give its precise definition here. We assume every agent in a configuration has its own meta-interpreter mi. Thus, for an agent AG and its parent agent AG', AG is assumed to have its own meta agent $mi(\tilde{AG}, \tilde{AG'})$. Meta-agents have their own environments, such that programs and constraints, but usually they are extensions of their object-level counterparts. By using this meta-interpreter, we can get meta-configuration of a configuration of *CC0* . 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 *CC0* is faithfulness of its behavior, which means c and $\uparrow c$ behaves exactly same. We will discuss the meta-interpreter matters in other papers.

4.2 Grouping and Reflection

As stated before, in *CCR*, granularity problem of reflection is solved by grouping some agents in some respects. We make a slight syntactic modification on that of *CC0* as follows. 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*. As a result, agents must be extended so that each atom should have a kind of *group id* to group several atoms. Therefore, an *active agent* in *CCR* is defined as a quadruplet $\langle Id, Gid, A, P \rangle$, where Id and Gid are identifiers such that $Gid \leq Id$, A is an atom, and P is a program. Constraint agents in *CCR* are defined by a similar extension.

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 $AG_r = \langle \alpha i, \beta, R(\bar{t}), P \rangle$ with a parent constraint agent $AG_c = \langle \alpha, \beta, V, C \rangle$ 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 some other agents are collected.

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

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

where $Reflect$ is a reserved predicate symbol, $reflect\ template$ is a list whose elements are exactly one reflective atom and some typed term, and $other\_agents$ is a variable at first. For example, a list

$$[r(t_1, \ldots, t_n), id(Id), gid(Gid), prog(P), cnstr(C)]$$

is a reflective template. When a reflective agent $RA = \langle Id, Gid, R(\bar{t}), P \rangle$ in a configuration c is executed, the meta-configuration $\uparrow c$ and the corresponding meta-level agent is composed according to the reflective template. The variable $other\ agents$ mentioned above is used with a special type $agents$ and a special function symbol $fetch$. For instance, following reflect definition clause describes that information about agents with the same group Gid are gathered in Z by throwing a constraint $X = fetch(Gid, agents(Z))$ on variable X;

```
Reflect([r(...),id(Id),gid(Gid),prog(P),cnstr(C)],X) :- | X=fetch(Gid,agents(Z)),
                                                        ...
                                                        ...
```

Then, the meta-level computation proceeds in a meta-configuration $\uparrow c$.

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

4.3 The operational semantics of *CCR*

Based on the transition system as the operational semantics of *CC0* defined in the previous section, we will define the operational semantics of *CCR*.

Configurations of *CCR* are defined similar to those of *CC0* except that agents have group identifiers. 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. 4.1 A binary relation $\mathcal{R}_R$ on the set of configuration $\mathcal{C}_R$ is defined as follows. Let $c, c' \in \mathcal{C}_R$.

$c\ \mathcal{R}_R\ c' \iff$ one of the following conditions holds:

1. For an active agent $AG = \langle \alpha i, \beta, A_i, P \rangle \in c$ where A_i is a user-defined atom and a clause $cl \in P$, $\tilde{cl} \equiv (H :- G \mid B_1, \ldots, B_m.)$ (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) every extended Herbrand model of C is also an extended Herbrand model of $\exists \bar{z}((uA_i = uH) \wedge G)$, where C is the constraint in the constraint agent $AG_C = \langle \alpha, V, C \rangle$, $c' = (c \backslash \{AG\}) \cup \{\langle \alpha i, \beta, V', C' \rangle, \langle \alpha i \cdot 1, \beta, B_1, P \rangle, \ldots, \langle \alpha i \cdot m, \beta, B_m, P \rangle\}$ where $V' = FV(A_i)$, $\bar{z} = FV(\tilde{cl})$.

2. For an active agent $AG = \langle \alpha i, \beta, A_i, P \rangle \in c$ where A_i is a constraint atom $A_i (1 \leq i \leq n)$ such that, if C is the constraint in the constraint agent $AG_C = \langle \alpha, \beta, V, C \rangle$, $C' \equiv C \wedge A_i$ is consistent, $c' = (c \backslash \{AG, AG_C\}) \cup \{\langle \alpha, \beta, V, C' \rangle\}$

3. For an annotated active agent $AG = \langle \alpha i, \beta, \&A_i, P \rangle \in c$ where A_i is a user-defined atom and a clause $cl \in P$, $\tilde{cl} \equiv (H :- G \mid B_1, \ldots, B_m.)$ (each occurrence of variables are ones which does not appear in c), with the same condition as in 1. $c' = (c \backslash \{AG\}) \cup \{\langle \alpha i, \beta', V', C' \rangle, \langle \alpha i \cdot 1, \beta', B_1, P \rangle, \ldots, \langle \alpha i \cdot m, \beta', B_m, P \rangle\}$ where $V' = FV(A_i)$, $\bar{z} = FV(\tilde{cl})$, and $\beta' = \alpha i$.

4. For a constraint agent $AG = \langle \alpha i, \beta, V, C \rangle \in c$ with its parent constraint agent $AG' = \langle \alpha, \beta, V', D \rangle$, if there exists a constraint atom C_a such that $C \vdash_{ETM} C_a$, $D \nvdash_{ETM} C_a$ and $FV(C_a) \cup V \neq \emptyset$, then $c' = (c \backslash \{AG'\}) \cup \{\langle \alpha, \beta, V', D \wedge C_a \rangle\}$. Similarly, if there exists a constraint atom C_a such that $C \nvdash_{ETM} C_a$, $D \vdash_{ETM} C_a$ and $FV(C_a) \cup V \neq \emptyset$, then $c' = (c \backslash \{AG\}) \cup \{\langle \alpha i, \beta, V, C \wedge C_a \rangle\}$.

5. For a reflective agent $AG = \langle \alpha i, \beta, A_i, P_i \rangle$ where A_i is a reflective atom $r(\bar{t})$, c' is constructed as follows. According to the reflective template of its reflect definition clause, reflective form $RA = Reflect(Template, Var)$ is constructed, and meta-level agent $MA = \langle \alpha i^M, \beta^M, RA, P_i^M \rangle$ is composed. Then, c' is defined as $\uparrow (c \backslash AG) \cup \{MA\}$.

6. For a meta-agent $\langle \alpha i, \beta, MA, P \rangle$ with its parent agent $AG_c = \langle \alpha, \beta, V, C \rangle$, if the constraint C has a term $fetch(gid, X)$, $c' = (c \backslash \{AG_c\}) \cup \{\langle \alpha, \beta, V, C \wedge (X = uMA) \rangle\}$.

□

We can define the success, failure and deadlock nodes similar to those of *CC0*.

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),vlist(V),cnstr(C)],X):- |
                                X=fetch(Id,agents([])),
                                mi([Id,Id,A,P],[Id1,Gid,V,C]),
                                parent(Id,Id1).
```

5 Conclusion

In this paper, we proposed a reflective concurrent constraint language with the capability of concurrent reflection. 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 [5, 8] Another interesting direction is exploring useful applications of concurrent reflection. We are now examining several examples of *CCR* in some 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] S. Costantini. Semantics of a metalogic programming language. *Int. J. of Foundations of Computer Science*, Vol. 1, No. 3, pp. 233–247, 1990.

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

[5] 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.

[6] 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.

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

[8] M. Murakami. A declarative semantics of parallel logic programs with perpetual processes. In *Proc. of FGCS '88*, pp. 374–381, 1988.

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

[10] H. Sugano. Meta and Reflective Computation in Logic Programs and Its Semantics. In *Proc. of the Second Workshop on Meta-Programming in Logic*, pp. 19–39, 1990.

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

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

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