ICOT Technical Report: TR-418

TR-418

A Parallel Problem Solving Language for Concurrent Systems

by
A. Takeuchi, K. Takahashi and
H. Shimizu(Mitsubishi)

September, 1988

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

A Parallel Problem Solving Language for Concurrent Systems

Akikazu TAKEUCHI Kazuko TAKAHASHI Hiroyuki SHIMIZU

Central Research Laboratory, Mitsubishi Electric Corporation
8-1-1 Tsukaguchi-Honmachi, Amagasaki, 661, JAPAN

Abstract A parallel problem solving language *ANDOR-II* which combines and- and or-parallelism is presented. *ANDOR-II* is designed to have the ability of solving such problems as combinatorial problems and cooperative planning on concurrent systems. *ANDOR-II* has language constructs which enable declarative description of concurrent systems consisting of many determinate, indeterminate and nondeterminate components acting interactively. Execution of *ANDOR-II* comprises multiple simulations of all possibilities which are dynamically derived from nondeterminate components. A new and/or parallel computation model based on a concept of a color is presented and a compilation from an *ANDOR-II* program into that of a committed choice language is also shown. Finally, an application to a distributed plan generation is shown.

1. INTRODUCTION

Recent advances in computer systems are accelerating the extensions of methodology and application domains of problem solving techniques. One of the main streams of such extensions is the exploitation of parallelism. Exploiting parallelism in methodology includes design and implementation of parallel programming languages for problem solving and discovery of parallel and/or distributed problem solving algorithms. Exploiting parallelism in application domains orients to solving problems on concurrent systems. Growing interest in these directions is forming a new paradigm called *distributed artificial intelligence* [Huhn 87].

Distributed artificial intelligence can be defined to be cooperative problem solving by decentralized set of agents which can see only parts of the whole world and interconnect information and control with one another. The whole solution is obtained by composing their sub-solutions. The great advantage of this approach is its modularity. Firstly, since a problem is expressed by a collection of modules, readability and maintenance are good. Secondly, even for a large problem which is unable to be treated in a centralized manner, the approach is easily applicable by decomposing the problem into distributed modules, and at the same time efficient execution can be expected because of parallel execution of these modules. Moreover, there are a lot of problems which have distributed characteristics by nature.

However parallelism explored so far in distributed artificial intelligence is limited to so-called and-parallelism, that is, parallel behaviors in one possible world. Yet another parallelism, called or-parallelism, corresponding to simultaneous consideration of several possible worlds is not investigated enough, but is quite powerful in describing some class of problems.

When one describes a problem on a concurrent system whose behavior is not perfectly understood, a concept of a nondeterminate[†] component which behaves indefinitely makes a declarative

[†] In this paper, following automata theory, the term "nondeterminate" means pursuit of all the possible choices and the term "indeterminate" means arbitrary selection.

description of the problem possible. Consider, for example, fault diagnosis of a circuit. A faulty circuit can be naturally modelled by regarding doubtful components as nondeterminate ones which take correct or incorrect actions indefinitely. In distributed artificial intelligence framework, generally each agent has several possible actions which can be taken at some moment depending on its local environment. Again such an agent can be naturally described as a nondeterminate component. The behavior of a concurrent system with nondeterminate components has many possibilities depending on the series of actions taken by them. Furthermore, these possibilities increase dynamically when such components take actions repeatedly.

From the point of view of computation, handling of nondeterminate components comprises simultaneous consideration of many possible worlds which may grow dynamically. Although they can be executed sequentially by backtracking, parallel implementation of this or-parallelism together with and-parallelism is of great interest [Takeuchi 84] [Conery and Kibler 85]. In this paper, we propose a parallel problem solving language *ANDOR-II* that has a language construct for describing a nondeterminate component together with its implementation scheme achieving both and- and or-parallelism.

ANDOR-II is derived from a current study of committed choice languages such as Concurrent Prolog [Shapiro 83], PARLOG [Clark and Gregory 84] and GHC [Ueda 86a]. Thus the syntax, semantics and implementation of *ANDOR-II* is heavily influenced by these languages. The syntax of *ANDOR-II* is similar to one of the committed choice languages, GHC, but in *ANDOR-II* predicates are divided into two types : an AND-predicate for description of a determinate or indeterminate component and an OR-predicate for a nondeterminate one. A program of *ANDOR-II* is compiled into a GHC program and is executed by a GHC processor.

The paper is organized as follows. In section 2, overview of *ANDOR-II* computation model is informally introduced. Specification and computation model of *ANDOR-II* is given in section 3. Compilation into GHC is briefly described in section 4. Section 5 gives an application to distributed plan generation. Finally comparison with related works and the remaining problems are described in section 6.

2. OVERVIEW OF ANDOR-II COMPUTATION

ANDOR-II computation starts with a set of processes, which are then executed in parallel. This corresponds to and-parallelism. When a nondeterminate process which has several possible actions is invoked, conceptually the world, that is, a set of all processes, proliferates into several worlds, each of which corresponds to a world in which one of the actions has been taken. And these worlds are executed in parallel. This corresponds to or-parallelism. In this model, the most crucial part is the proliferation. Its naive implementation is to make a copy of a set of all processes, but it seems unacceptable because of its expected overhead. We introduce an another implementation scheme called *coloring*.

In *coloring* scheme, a world is associated with an identifier called *color*. Every term belonging to a world with a color C is also associated with C. On proliferation of a world, new colors are generated and assigned to new worlds. A color records the history of world proliferation. In *coloring* scheme, a color is used to determine whether a set of data belongs to the same world or not, and to enable to share processes among worlds and to prevent interference among independent worlds. Thus it makes copying of processes unnecessary.

Before introducing the language syntax, we first explain the computation model based on the coloring scheme using a simple example.

Example 2.1.

A process *compute* picks up an arbitrary element from the input list and returns the sum of its squared value and cubed value. Here picking up is assumed to be a nondeterminate operation. If the input is $[1,2,3]$, possible solutions are 2 (1^2+1^3), 12 (2^2+2^3) and 36 (3^2+3^3).

The program is informally expressed in a conventional style as follows (*ANDOR-II* definition will be shown later).

```
def compute(X:list) Z:integer
    type determinate
       Y :=pickup(X),
       Y2:=square(Y),
       Y3:=cube(Y),
       Z :=add(Y2,Y3)
end.

def pickup (X:list) Y:integer
    type nondeterminate
       select an arbitrary element Y from the input list X
end.

def square(X:integer) Y:integer
    type determinate
       Y:=X*X
end.

def cube(X:integer) Y:integer
    type determinate
       Y:=X*X*X.
end.

def add(X,Y:integer)  Z:integer
    type determinate
       Z:=X+Y.
end.
```

Figure 2.1. shows the data flow graph inside *compute*. A node and an edge are regarded as a process and a communication channel, respectively. When *pickup* generates an output value via the channel Y, processes *square* and *cube* receive the value and generate squared value via $Y2$ and cubed value via $Y3$, respectively. Then process *add* adds them.

Suppose that *compute* is invoked with the input $[1,2,3]$. The list is directly sent to *pickup*. *Pickup* is a nondeterminate process and has three possible actions, that is, selection of 1, 2 and 3. All these possible outputs are painted by distinct colors and packed in arbitrary order in a vector form $\{v(1,c1), v(2,c2), v(3,c3)\}$, where $\{.....\}$ denotes a vector and $v(X,C)$ denotes a value X with a color C. Instead of sending each output value, this vector is sent to *square* and *cube* via Y. For each element of the vector, *square* and *cube* perform their operations and create new vectors, $\{v(1,c1), v(4,c2), v(9,c3)\}$ and $\{v(1,c1), v(8,c2), v(27,c3)\}$, respectively. Note that since *square* and *cube* are determinate they never change colors. *Add* is invoked with these two vectors.

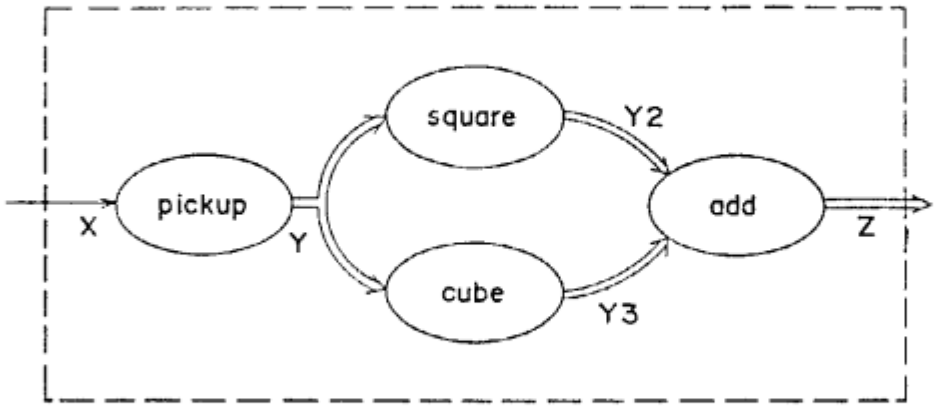

Fig 2.1. Data Flow Graph of Compute

Conceptually *add* is requested to add X^2 and Y^3 if X and Y are identical. Namely, when they are derived from the same *pickup* action. Otherwise, no addition is applied. Information about identity of two elements from two vectors are carried by their colors, that is, the elements with the same color are derived from the same action of *pickup*. Therefore, two vectors are preprocessed to make a set of pairs with the same color, and for each element of the set *add* operation is applied. Finally, a vector of the solutions $\{v(2,c1), v(12,c2), v(36,c3)\}$ is obtained.

3. LANGUAGE ANDOR-II

3.1. Syntax

In *ANDOR-II* , a program is a set of AND-predicate definitions and OR-predicate definitions. An AND-predicate definition consists of a mode declaration and a set of AND-clauses. An OR-predicate definition consists of a mode declaration, OR-relation declaration and a set of OR-clauses. A predicate defined by an AND(OR)-predicate definition is called an *AND(OR)-predicate.*

Definition(AND-clause,OR-clause)
A clause in the form

```
H :- B1,...,Bm.
```

is called *an OR-clause*, and a clause in the form

```
H :- G1,...,Gn | B1,...,Bm.
```

is called *an AND-clause.*

In the above definition, H is called a *head*, G_1,...,G_n are called a *guard part*, and $B_1, ..., B_m$ are called a *body part*. AND-predicates are used for defining determinate or indeterminate processes and OR-predicates are used for defining nondeterminate processes. An OR-predicate definition must be headed by OR-relation declaration of the form:

```
:- or_relation p/n.
```

where *p* is a predicate symbol and n is its arity. In a mode declaration, '+' and '−' denote input

and output mode, respectively. The meanings of input and output are the same as in Edinburgh Prolog except that for an OR-predicate, input mode imposes the restriction that the corresponding argument must be either a ground term or a list whose head is a ground term at the moment when head unification has succeeded. Note that a clause of either type can contain both AND-predicates and OR-predicates in a body part. A goal in a guard part is restricted to a test predicate. This is the same restriction as those of flat committed choice languages (see, for example, [Ueda 86a]).

Two *ANDOR-II* programs are shown below. The syntax of *ANDOR-II* is similar to committed choice languages, especially GHC, and can be read in the same way as GHC except for the OR-predicate.

Example 3.1.
The following is an *ANDOR-II* program of *compute* discussed in section 2.

```
:- mode compute(+,-), pickup(+,-), square(+,-), cube(+,-), add(+,+,-).

compute(X,Z) :- true |
    pickup(X,Y), square(Y,Y2), cube(Y,Y3), add(Y2,Y3,Z).

:- or_relation pickup/2.
pickup([X|L],Y) :- Y=X.
pickup([_|L],Y) :- pickup(L,Y).

square(X,Y) :- true | Y:=X*X.
cube(X,Y)   :- true | Y:=X*X*X.
add(X,Y,Z)  :- true | Z:=X+Y.
```

Example 3.2.
The following is an *ANDOR-II* program of two communicating processes.

```
:- mode cycle, p1(+,-), p2(+,-), multi(+,-),
        square(+,-), cube(+,-), add(+,+,-).

cycle :- true | p1([2|X],Y), p2(Y,X).

p1([stop],Y) :- true | Y=[].
p1([X|X1],Y) :- true | add(X,1,A), Y=[A|Y1], p1(X1,Y1).

p2([X|X1],Y) :- X>20  | Y=[stop].
p2([X|X1],Y) :- X=<20 | multi(X,A), Y=[A|Y1], p2(X1,Y1).

:- or_relation multi/2.
multi(X,Y) :- square(X,Y).
multi(X,Y) :- cube(X,Y).

square(X,Y) :- true | Y:=X*X.
cube(X,Y)   :- true | Y:=X*X*X.
add(X,Y,Z)  :- true | Z:=X+Y.
```

In the clause defining *cycle*, processes $p1$ and $p2$ form a cyclic structure with the communication channels X and Y. $p1$ receives the stream via its first argument, increments the element of the stream by 1, and sends the value to $p2$ via its second argument. $p2$ receives the stream via its first argument, executes the goal *multi* on the received element, and sends the results to $p1$. In this way, the values put onto each cell of the stream X and Y are determined incrementally by affecting each other.

[Remarks]

Some restrictions are imposed on programs in the current *ANDOR-II* .

(1) *unique handler for a stream*

If a variable is expected to be bound to a stream, each element in the stream is instantiated by the same process.

(2) *prohibition of multiple writers*

Only one goal can refer a variable in an output mode in a clause.

(1) is introduced to make the direction of data flow unique. (2) is introduced to reduce execution overheads.

3.2. Computation Model

ANDOR-II supports both and- and or-parallelism, that is, all the conjunctive goals are executed in parallel (and-parallelism). And for an OR-predicate, clauses whose heads are unifiable with the OR-predicate are executed in parallel (or-parallelism). In order to coordinate both and- and or-parallelism, the notion of a color is introduced. The idea is that when an OR-predicate is invoked and possibly returns several answer substitutions, they are attached with distinct colors so that basic computations such as unification and arithmetic operations are applied only to a tuple of data sharing the same color.

The computation rule of AND-predicates is similar to that of GHC [Ueda 86b]. That is, two rules are imposed: *rule of suspension* and *rule of commitment*. And only rule of suspension is imposed on the execution of OR-predicates. Here, by the term, *guard computation of a clause C*, we mean both head unification and execution of the guard part.

[Rule of Suspension]

(1) Unification invoked directly or indirectly in guard computation of an AND-clause C called by a goal G cannot instantiate the goal G.

(2) Unification invoked directly or indirectly in the body of an AND-clause C called by a goal G cannot instantiate the guard and the head of C until C is selected for commitment (see below).

(3) Unification between a goal G and the head of an OR-clause C called by the goal G cannot instantiate the goal G.

(4) Unification invoked directly or indirectly in the body of an OR-clause C called by a goal G cannot instantiate the head of C until the head of C is unified with G.

A piece of unification that can succeed only by violating the rules above is suspended until it can succeed without such violation *(end of the rule of suspension)*.

[Rule of Commitment]

When some AND-clause C called by a goal G succeeds in solving its guard, the clause C tries to be selected for subsequent execution of G. To be selected, C must first confirm that no other clauses in the program have been selected for G. If confirmed, C is selected indivisibly, and the execution of G is said to be committed to the clause C *(end of the rule of commitment)*.

Fig. 3.1 shows the computation model of *compute* in the example 3.1.

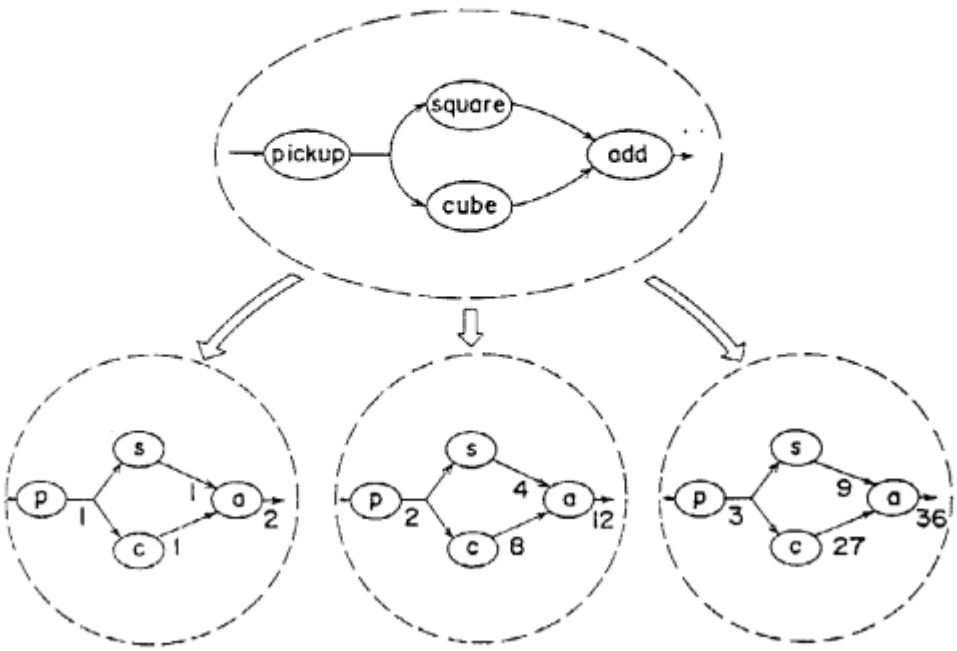

Fig 3.1. Computation model of *compute*

3.3. Colored World

A world is a conceptual entity in which goals are semantically connected by a logical connective *AND* and executed in parallel (and-parallelism) operationally. Worlds are semantically connected by a logical connective *OR* and executed in parallel (or-parallelism). In coloring scheme, there is no concrete object corresponding to a world. Instead of directly representing and handling worlds, coloring scheme achieves the same effect by coloring data objects.

Here, we give a formal definition of *color* and *colored value*.

Definition (primitive-color,color,colored-value)

```
primitive color ::= (clause-number,branching-point)
color           ::= [] |
                    [primitive-color|color]
colored-value   ::= v(value,color)
```

where *branching-point* is a unique identifier of an invocation of an OR-predicate, and *clause-number* is the identifier of a selected clause at the branching point.

When an OR-predicate including a variable, say X, is invoked in a world, conceptually the world splits into several worlds along OR-clauses. In coloring scheme, instead of actually creating new worlds, a new color C_i is generated for each OR-clause. In computation of each OR-clause whose associated color is C_i, the variable X might be instantiated to a value V_i. Such multiple binding to the variable X is realized by instantiating X to a vector of colored values, denoted by $\{v(V_1, C_1), v(V_2, C_2), ..., v(V_n, C_n)\}$. We use a term *colored vector* or simply *vector* to denote the data of this type and distinguish it from the stream, which is a different concept for stream programming. A data without a color (i.e. simple value) is called *scalar*.

Definition (same,orthogonal,productive)

For a pair of colors C_1 and C_2, one of the following three relations holds.

(1) If there exists a branching point bp such that $(n1, bp)$ is included in C_1 and $(n2, bp)$ is included in C_2 where $n1 \neq n2$, then C_1 and C_2 are defined to be *orthogonal*.

(2) Let $bp_i (i = 1, ..., k)$ be a branching point appearing both in C_1 and C_2 and let n_i and m_i be associated clause numbers in C_1 and C_2, respectively. If $n_i = m_i$ for all $i (i = 1, ..., k)$, then C_1 and C_2 are defined to be *the same*.

(3) If C_1 and C_2 share no branching point, C_1 and C_2 are defined to be *productive*.

Intuitively, values with the same color have selected the same clauses at common branching points and values with the orthogonal colors have selected the different branches at the common branching points, and values with productive colors have no common branching point.

Definition(consistency)

For colored values $v(V_1, C_1)$ and $v(V_2, C_2)$, if C_1 and C_2 are either the same or productive, then C_1 and C_2 are said to be *consistent*. For colored values $v(V_1, C_1), v(V_2, C_2), ..., v(V_n, C_n)$, if for any $i, j (i \neq j)$, C_i and C_j are consistent, then $C_1, C_2, ..., C_n$ are said to be *consistent*.

Definition(joint color)

When a goal receives the set of colored-values $v(V_1, C_1), v(V_2, C_2), ..., v(V_n, C_n)$, each of which is received via different input arguments, and $C_1, C_2, ..., C_n$ are consistent, then the goal is applicable to the values $V_1, V_2, ..., V_n$. Let R be the result. Then, the color associated with R is defined as the union of $C_1, C_2, ..., C_n$. It is called *joint color*.

Example. 3.3.

Let $C, C1, C2$ and $C3$ be the following colors, respectively.

```
C  : [(n1,#1),(n2,#2)]
C1 : [(n1,#1),(n1,#3)]
C2 : [(n2,#1),(n1,#3)]
C3 : [(n1,#3)]
```

C and $C1$ are the same and the joint color is `[(n1,#1),(n2,#2),(n1,#3)]`.

C and $C2$ are orthogonal and no joint color is defined.

C and $C3$ are productive and the joint color is `[(n1,#1),(n2,#2),(n1,#3)]`.

4. COMPILATION

An *ANDOR-II* program is compiled into a GHC program. The compiler consists of two main modules : DFA module and TRA module. In DFA, a source program is analyzed and an intermediate code with some information is generated. In TRA, the intermediate code is transformed into a GHC program.

The compiler discussed here is the revised version of that described in [Takeuchi et al. 87]. Since the technique is somewhat complicated, we will here give a brief sketch of the system.

4.1. DFA

DFA(Data Flow Analysis) contains the following subprocedures:

(1) construction of DFG(Data Flow Graph)

(2) type check of channels and shells

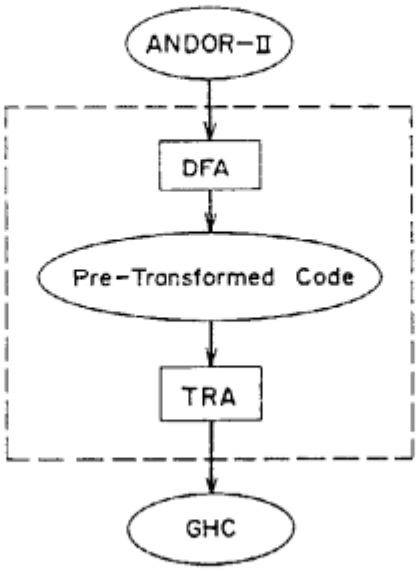

Fig 4.1. Structure of ANDOR-II System

(3) generation of an intermediate code

As an aid for grasping the data flows among goals through shared variables and data types flowing through them, the compiler makes graphs called DFG's for all the clauses in an *ANDOR-II* program, respectively, based on mode declarations. A DFG is a graph in which nodes correspond to goals appearing in the clause and edges to shared variables. Direction of each edge is determined by the following definition.

Definition (input channel, output channel)
For a node N in a DFG, if a variable V appears in the argument of the input mode of the corresponding goal, then V is said to be *an input channel of* N, and if V appears in the output mode, then it is said to be *an output channel of* N.

In order to collect more information about nodes and edges, a data type of each edge is examined. In the *ANDOR-II* execution, three types of data may appear: *scalar*, *flat vector* and *layered vector*.

Definition(flat vector, layered vector, scalar)
(1) *Flat vector* has the form $\{v(V_1, C_1), v(V_2, C_2), ..., v(V_n, C_n)\}$ where each $v(V_i, C_i)$ is a colored-value.
(2) *Layered vector* has the form of a list $[X|Y]$ where X is a ground term of scalar type and Y is either a flat vector or a layered vector.
(3) Any other term is called *Scalar*.

A layered vector will appear as an output stream when a stream is sent to an OR-predicate.
A channel which is instantiated to either a flat vector or a layered vector is called a *vector type channel*. If a node has an input vector type channel, it is necessary to pick up each colored value from the vector. Therefore such a node is marked as a *shell-covered node*.

The main part of DFA module is to determine channel types and node types. It is mainly done by checking whether a node has a direct or indirect data flow from the node corresponding to an OR-predicate or not, and whether a channel is used for stream processing or not. Note that types are undecidable. Thus, we have to analyze as far as possible and adopt a safer information. At last, an intermediate code annotated by these information is generated.

4.2. TRA

TRA(TRAnsformation) transforms the intermediate code into a GHC program. It consists of the following subprocedures:

(1) shell creation
(2) OR-to-AND transformation
(3) predicate transformation

First, for each shell-covered node, *shell_creation* clauses are created. *Shell_creation* clauses decompose a set of input vectors into a tuple of scalar values, pass them to the corresponding core processes, and put the output values together into a set of vectors again. Note that all the output channels of shell-covered nodes are always vectors. Each core process corresponds to computation with one color, which is a joint color of colors attached to input data. Core processes are executed in parallel. Some core processes may succeed and return solutions, while others may fail or deadlock and return no solution. In principle, solutions are put into output channels as soon as they are generated by fair merge operators, so that all the solutions are obtained without being disturbed by failure or deadlock in some worlds.

Example 4.1.

The shell-creation clauses for a shell-covered node *add* in the example 3.1 are shown below.

```
add_Shell_2_1([v(X,Cx)|Xs],Y,Z) :-
    true |
    add_Shell_2_2(v(X,Cx),Y,Z1),
    add_Shell_2_1(Xs,Y,Z2),
    merge_BLT(Z1,Z2,Z).                          % merging solutions
add_Shell_2_1([],_,Z) :- true |  Z=[].

add_Shell_2_2(v(X,Cx),[v(Y,Cy)|Ys],Z) :-
    true |
    add_Check_2_1(v(X,Cx),v(Y,Cy),Z1),
    add_Shell_2_2(v(X,Cx),Ys,Z2),
    merge_BLT(Z1,Z2,Z).                          % merging solutions
add_Shell_2_2(_,[],Z) :- true |  Z=[].

add_Check_2_1(v(X,Cx),v(Y,Cy),Z) :- true |
    consistent_Color([Cx,Cy],R),
    add_Check_2_2(R,X,Y,Z).

add_Check_2_2(success(C),X,Y,Z) :- true |
    add_Core(X,Y,Z0,w(C)),                       % core process
    Z=[v(Z0,C)].                                 % solution Z0 is
                                                 % associated with its
                                                 % color C.
add_Check_2_2(fail,_,_,Z) :- true | Z=[].
```

Special treatment is necessary for a shell-covered node with layered vector channels since stream processing is expected. It means that if a colored value is put onto the stream, the remaining part which will be instantiated to successive outputs must have the same color.

Example 4.2.
The shell for a shell-covered node $p2$ in the example 3.2 is shown below.

```
p2_AShell(Xs,[v(Y,C)|Ys]) :- true |          % C is the color for one stream.
        p2_AGo(Xs,Y,C),                      % The handler for that stream.
        p2_AShell(Xs,Ys).
p2_AShell(Xs,[]) :- true | true.

p2_AGo([v(X,C1)|Xs],Y,C) :- true |
        p2_ACheck_1(v(X,C1),Y1,C),           % Processing of input X with
                                             % color C1 and the output
                                             % stream with color C.
        p2_AGo(Xs,Y2,C),                     % Processing of other
                                             % input and the output
                                             % stream with color C.
        merge_BLT(Y1,Y2,Y).                  % See below.
p2_AGo([],Y,_) :- true | Y=[].

p2_ACheck_1(v(X,C1),Y,C) :- true |
        consistent_Color([C,C1],Res),
        p2_ACheck_2(Res,X,Y).

p2_ACheck_2(success(C),X,Y) :- true | p2_Core(X,Y,w(C)).
p2_ACheck_2(fail,X,Y) :- true | Y=[].
```

A shell-covered node of the form, $Z = [Xs \mid Y]$, where Xs is a vector is translated into a special predicate *makeslot_BLT*. *makeslot_BLT* generates a new stream for each element of Xs. The definition of *makeslot_BLT* is shown below.

```
makeslot_BLT([v(X,C)|Xs],Y,Z) :- true |      % Head is X with C.
    Y=[v(Y1,C)|Ys],                          % Create a new tail
                                             % with C.
    Z1=[X|Y1],                               % Create a new stream.
    Z=[v(Z1,C)|Zs].                          % A new stream is output.
makeslot_BLT([],Y,Z) :- true | Y=[], Z=[].
```

Next, OR-clauses are translated into a determinate AND-clause. In a resultant clause, OR-parallel execution of OR-clauses are realized by AND-parallel execution of goals corresponding to their computations. And their solutions are collected as a colored vector by the fair merge technique again.

Example 4.3.

```
pickup_Core(X,Y,w(C)) :- true |
```

```
get_Branching_Point(BP),
pickup_Core_1(X,Y1,C,BP),          % computation of the
                                   % first clause.
pickup_Core_2(X,Y2,C,BP),          % computation of the
                                   % second clause.
merge_BLT(Y1,Y2,Y).
```

In the above program, *get_Branching_Point* is a system predicate which generates a unique identifier on execution.

Finally, clauses corresponding to core processes are generated. The following example shows the transformed code of two clauses defining $p2$ in the example 3.2.

Example 4.4.

```
p2_Core([X|X1],Y,w(C)) :- X>20  |
    Y1=[stop],
    Y=[v(Y1,C)].
p2_Core([X|X1],Y,w(C)) :- X=<20 |
    multi_Core(X,A,w(C)),
    makeslot_BLT(A,Y1,Y),
    p2_Shell(X1,Y1).
```

In this model, when a shell-covered node has more than two input vectors, *shell_creation* clauses need to check consistency of colors, which causes an overhead. However there are some possibilities to reduce it in some situations. We briefly explain two optimization techniques which are currently implemented below.

(1) Elimination of color consistency check

Let N be a shell-covered node with k input vector channels $I_1, ..., I_k$. If no I_i and I_j $(i \neq j)$ share data flowing directly or indirectly from a common node in a DFG, any pair of colors is productive. Joint color is determined by appending their color lists without checking their consistency.

(2) Colorless block (Special case of the first)

Let p be an OR-predicate and $G_1, G_2, ..., G_n$ be DFG's of OR-clauses defining p. If every $G_i (i = 1, ..., n)$ satisfies the following two conditions, then p is said to be a *colorless block*.

(i) A node in G_i is either p or an AND-predicate which does not directly nor indirectly call an OR-predicate other than p.

(ii) G_i includes no cycle.

(iii) There exists no node such that it is a shell-covered node with more than one input vector channels and two of them share data flowing directly or indirectly from the same node.

Suppose p is a colorless block. During the computation of p with scalar input data, when two colored values would meet at some node, they are always productive. Orthogonally colored values never meet at any node. A set of solutions of p is then a set of values which are colored orthogonally each other. In this case, we *can omit coloring* during computation. Although solutions become colorless, they are known to be orthogonal each other, so they are given orthogonal colors just before they are exported to outside the block.

More general condition for the second optimization, thus stronger optimization, is now being studied.

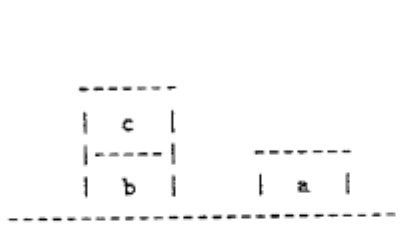

Fig 5.1(a) Initial State

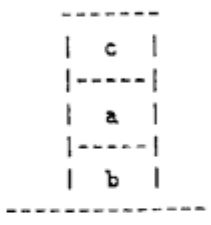

Fig 5.1(b) Goal State

5. APPLICATION

In this section, we will discuss an application of *ANDOR-II* to parallel problem solving using distributed plan generation of block loading as an example.

5.1. Modelling

There are three blocks a,*b* and *c* on a table. The goal is to reload the blocks in the initial state shown in Fig.5.1(a) into the goal state Fig.5.1(b).

A block can move only when there is nothing on it. A table is assumed to be large enough to accept any number of blocks and is used for a dugout for a block being asked to get away from the current position. These are well-known characterizations of "block world." Our purpose is to generate a partially ordered plan which combines sequences of actions of each block. According to this purpose, three assumptions featuring our distributed plan generation are introduced: (1) Each block is assumed to be an active agent knowing only its own goal state and trying to achieve it by exchanging messages with other blocks. It can see only its current state, i.e. blocks on and under it. (2) More than one blocks can move simultaneously if their tops are clear. (3) It takes finite time (non zero) for a block to move from one place to another.

Each block can be modelled as a nondeterminate component which has a variety of actions depending on its current state and received messages. Its actions are classified into two classes: active actions and passive actions. The former are for achieving its own goal, while the latter for fulfilling received messages.

Communication between blocks are modelled as follows. Any pair of blocks are connected by two stream communication channels, one is for sending messages and the other for receiving. The input streams to a block from the other two blocks are merged on arriving (Fig.5.2.)

The top level of the *ANDOR-II* source program is shown below. The eighth argument of the predicate *block* is used for storing the history of actions which corresponds to the plan for the block, and the last argument is for the output of the plan.

```
pf1(Plan) :- true |
        block(a, AB,AC, BA,CA, c(clear,table), f(c,b),     [start], Pa),
        block(b, BC,BA, CB,AB, c(c,table),     f(a,table), [start], Pb),
        block(c, CA,CB, AC,BC, c(clear,b),     f(clear,a), [start], Pc),
        Plan=[Pa,Pb,Pc].

block(Id,S1,S2,R1,R2,Current,Final,H,Plan) :- true |
        or_merge(R1,R2,In),
        action(Id,S1,S2,In,Current,Final,H,Plan).
```

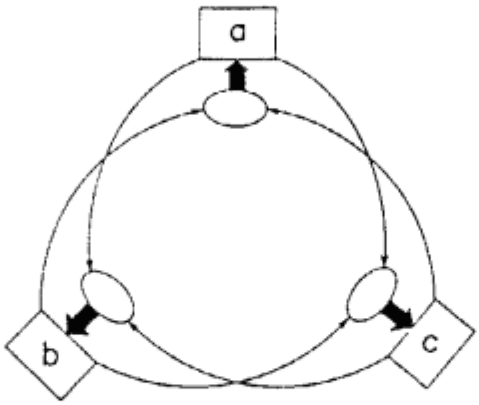

Fig 5.2. Model of Blocks World

In determinate parallel programming paradigm, the solution becomes quite complicated. Since a top of each block can be regarded as a shared resource, putting one block on another block needs complicated locking and unlocking of the target block. It needs more procedures when the top of the target is already occupied by another block. Since the whole map is divided into local maps which are maintained by each block, when one block moves onto another block, their maps must be updated indivisibly. These are the essential problems when one encounters in programming distributed systems. Owing to these problems, the resultant program becomes complicated and almost all of the program are wasted by establishment of synchronizations of blocks.

In our model, when two message streams to a block are merged into one stream, nondeterminate merge, *or_merge*, is used instead of indeterminate merge. A nondeterminate merge is vital to our modelling, since it greatly contributes to the simplification of the description. Using a nondeterminate merge, the following assumption becomes possible.

If a block has once sent a message, then it can believe that the request is fulfilled eventually in its local time.

This is because of a nondeterminate merge and a nondeterminate block enumerates all the possible situations. It is worth noting that what a nondeterminate merge does is to enumerate all the possible serialization (in global time) of events which are known to occur only in local time. If the request can be fulfilled without affecting the sender, then there is at least one situation in the set of possible situations where the request can be fulfilled eventually in global time. If the request is not satisfiable, then the belief is incorrect and the world in which this message sending occurs just deadlocks or fails. Even in this case, alternative computations proceed in other worlds. Owing to the above assumption, synchronization of blocks becomes completely unnecessary and hence the description of a block becomes quite straightforward.

An informal description of a block is shown below (The whole program in *ANDOR-II* is shown in appendix), where $c(On, Under)$ represents that a block is currently on an object (a block or a table) $Under$ and under an object On, $c(clear, Under)$ expresses that nothing is on the block. A goal state of a block is represented by $f(FOn, FUnder)$.

Each block BLK behaves according to the following rules:

(i) active behavior

If $On = FON$ and $Under = FUnder$ (Goal is achieved)

then it terminates.
If $Under \neq FUnder$,
if $On = clear$,
then send the message $move(BLK)$ to $FUnder$,
send the message $clear$ to $Under$,
and also update its current state to $c(clear, FUnder)$.
if $On \neq clear$,
then send the message $remove\_from(BLK)$ to On,
and also update its current state to $c(clear, Under)$.

(ii) passive behavior

On receiving the message $move(B)$,
if $On = clear$,
then update its current state to $c(B, Under)$,
if $On \neq clear$,
then send a message $remove\_from(BLK)$ to On
and also update its current state to $c(B, Under)$.
On receiving the message $remove\_from(B)$,
if $Under \neq B$,
then ignore the message.
if $Under = B$,
if $On = clear$,
then move onto $table$.
and also update its current state to $c(clear, table)$.
if $On \neq clear$,
then send a message $remove\_from(BLK)$ to On,
move onto $table$.
and also update its current state to $c(clear, table)$.
On receiving the message $clear$,
then update its current state to $c(clear, Under)$

Let us see what happens in the initial state according to these rules : Block c wants to move onto the block a in order to achieve its goal. Therefore, c sends a message to a. At the same time, it sends a message to b telling it to depart. On the other hand, the block a wants to move onto the block b in order to achieve its goal. Therefore, two messages are sent to b. It causes two alternatives depending on which message arrives first. If the message from c arrives first, b can accept a immediately. On the other hand, if the message from a arrives first, b have to remove c before accepting a. These two possible situations are automatically pursuited by a nondeterminate merge process in front of the block b.

The source program is transformed into a GHC program by *ANDOR-II* compiler. On execution of the transformed program, several plans are generated. Let BLK and B be the sender and the receiver of a message, respectively. There are three kinds of messages: $move(BLK)$, $remove\_from(BLK)$ and $clear$. If a block BLK sends a message $move(BLK)$ to a block B, then BLK's action is recorded as $move\_to(B)$, and B's action is recorded as $accept(B)$. Similarly, as for the message $clear$, their actions are recorded as $clear$ and $cleared$, respectively. As for the message $remove\_from(BLK)$, the action of BLK is recorded as $remove(B)$, and if B is no longer on BLK, B's action is recorded as $ignore$, otherwise $escape$. It is also noted that these action record is associated with an event identifier which is explained in the next subsection. In the following,

we show some of solutions:

```
a: [end,(accept(c),#1015),(clear,#96),(move_to(b),#96),start]
b: [end,(accept(a),#96),(remove(c),#297),start]
c: [end,(clear,#1015),(move_to(a),#1015),(escape,#297),start]
```

```
a: [end,(accept(c),#346),(clear,#61),(move_to(b),#61),start]
b: [end,(accept(a),#61),(cleared,#346),start]
c: [end,(clear,#346),(move_to(a),#346),start]
```

```
a: [end,(accept(c),#10299),(clear,#3286),(move_to(b),#3286),(remove(c),#3228),
        (accept(c),#346),start]
b: [end,(accept(a),#3286),(cleared,#346),start]
c: [end,(clear,#10299),(move_to(a),#10299),(escape,#3228),(clear,#346),
        (move_to(a),#346),start]
```

:
:
:

5.2. Towards A Total Plan Generation

A plan is a partially ordered set of actions. After getting a local plan of each block, a total plan is constructed. *An event* is defined to be a message sending and is associated with a unique identifier. In a plan, message sending and corresponding receiving are labelled by the event identifier.

Suppose that a block has a local plan in a colored world W

$$E_1, E_2, ..., E_n$$

and that another block has a plan in W

$$F_1, F_2, ..., F_m$$

where $E_1, E_2, ..., E_n$ and $F_1, F_2, ..., F_m$ are actions and they are totally ordered, respectively.

$$E_1 < E_2 < ... < E_n$$
$$F_1 < F_2 < ... < F_m$$

Let the event identifier of $E_i (1 \leq i \leq n)$ be k_i, and that of $F_i (1 \leq j \leq m)$ be l_i, respectively. If there is a pair of i and j such that $k_i = l_j$ and E_i and F_j are message sending and receiving, respectively, then F_j should happen after E_i. Thus, the order $E_i \leq F_j$ is imposed.

A set of local plans is consistent if and only if ordering imposed by events is consistent. A consistent set of local plans is a *total plan* (The program in the appendix does not include consistency check of local plans).

Three plans shown above are all consistent and thus constitute total plans. Here we give concrete interpretations of the plans 1, 2 and 3.

```
% Plan 1 (See Fig.5.3):
a: [end,(accept(c),#1015),(clear,#96),(move_to(b),#96),start]
b: [end,(accept(a),#96),(remove(c),#297),start]
c: [end,(clear,#1015),(move_to(a),#1015),(escape,#297),start]
```

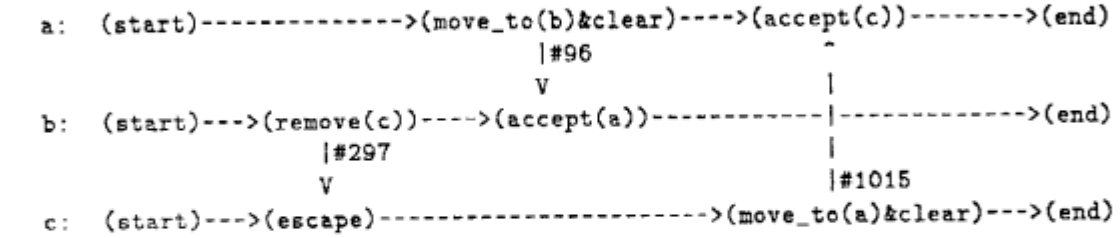

Fig 5.3. Plan 1

One of the possible readings of the plan 1 is as follows:

1. Block b removes block c onto the table and block a takes off to block b.
2. Block a lands on block b and block c takes off to block a.
3. Block c lands on block a.

```
% Plan 2 (See Fig.5.4):
a: [end,(accept(c),#346),(clear,#61),(move_to(b),#61),start]
b: [end,(accept(a),#61),(cleared,#346),start]
c: [end,(clear,#346),(move_to(a),#346),start]
```

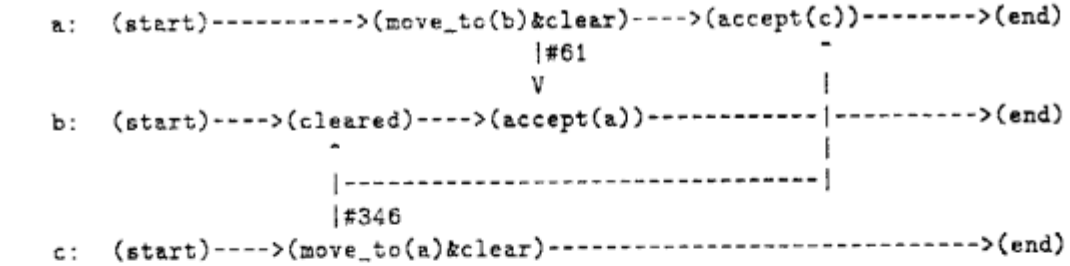

Fig 5.4. Plan 2

One of the possible readings of the plan 2 is as follows:

1. block a takes off to block b and block c takes off to block a
2. block a lands on block b
3. block c lands on block a

```
% Plan 3 (See Fig.5.5):
a: [end,(accept(c),#10299),(clear,#3286),(move_to(b),#3286),(remove(c),#3228),
      (accept(c),#346),start]
b: [end,(accept(a),#3286),(cleared,#346),start]
c: [end,(clear,#10299),(move_to(a),#10299),(escape,#3228),(clear,#346),
      (move_to(a),#346),start]
```

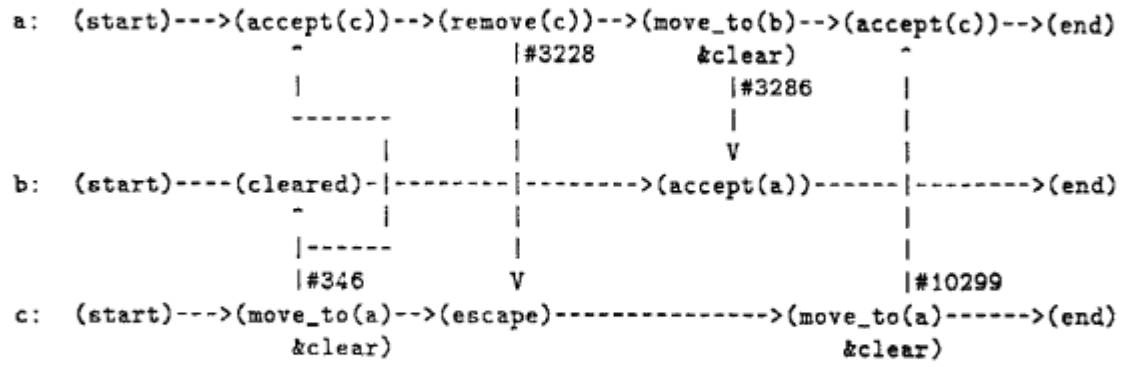

Fig 5.5. Plan 3

One of the possible readings of the plan 3 is as follows:

1. Block c moves onto block a.
2. Block a removes block c onto the table.
3. Block a moves onto block b and block c takes off to block a.
4. block c lands on block a.

6. DISCUSSION

6.1. Comparison with Other Works

Design and implementation of a new language which has both features of and-parallelism with *indeterminacy* and or-parallelism (*nondeterminism*) are studied intensively as an ultimate combination of a logic programming language and a committed choice language by many researchers.

Clark and Gregory pointed out the importance of further research in these areas and they suggested the combination of PARLOG and Prolog [Clark and Gregory 87]. In this combination, although two languages can comfortably call each other, but truly mixed combination of and/or-parallel execution is not considered.

Yang proposed a language P-Prolog which subsumes both and- and or-parallelism [Yang 87] and achieves true mixture of both parallelism. In this respect, P-Prolog is closely related to *AND OR-II*. One of the main differences is synchronization mechanism. In P-Prolog, clauses are divided into single-neck and double-neck clauses and for single-neck clauses exclusiveness plays a central role in synchronization, while in *ANDOR-II* mechanism similar to that of GHC is adopted. Other main difference is implementation. We designed *ANDOR-II* so that it can be compiled into a committed choice language, while P-Prolog seems to be designed together with a new implementation scheme.

Naish proposed a parallel NU-Prolog which is an extension of NU-Prolog [Naish 87]. It can express and-parallelism together with nondeterminism. A nondeterminate code can call and-parallel code which (in restricted way) can call nondeterminate code. However, nondeterminism is only handled sequentially.

Brand [Brand et al. 88] proposed the language Andorra which is aimed at a superset of both OR-parallel Prolog and a committed choice language. Andorra and *ANDOR-II* share many features. One of the main differences is that invocations of nondeterminate goals are lazy in Andorra, while eager in *ANDOR-II*. Also scheduling of a nondeterminate goal is infinitely unfair in Andorra, though this is for compatibility to Prolog. Implementation is also different. *ANDOR-II* adopts a

compiler approach, while they are designing a new machine for Andorra.

Program transformation from a nondeterminate program to a determinate program which collects all the solutions is also intensively studied.

Ueda proposed continuation-based transformation from an exhaustive search program in Prolog into a determinate GHC/Prolog program [Ueda 86c]. Or-parallelism in the original program is realized by and-parallelism in the transformed program, while the intrinsic and-parallelism is not considered. He reports that transformed programs has much more efficiency for a class of programs, and that they do not lose much efficiency for others. In [Ueda 87], Ueda proposed the extension of continuation-based transformation to a nondeterminate program with coroutine. It is realized by statically determining scheduling of coroutines by compile-time analysis of a source program. However, it is difficult to apply continuation-based scheme to nondeterminate processes communicating each other since compile-time of process scheduling is undecidable.

Tamaki presented stream-based transformation from a logic program with and/or parallelism into the one in a committed choice language. In his system, like our system, a set of solutions from a nondeterminate goal are propagated to other goals in a stream form. Owing to a stream communication, dynamic process scheduling becomes straightforward and and/or parallelism in a source program can be naturally realized. In comparison with our language, his language has some restrictions. One is that elements of a stream are processed one by one by a conjunction of goals (called a *block*), that is, only one element can exist in a block at a time, while in our system a conjunction of goals can process any number of elements in parallel. The other is that his language does not allow communication among a conjunction of goals, while our language does allow and such communication is essential to our applications. Conversely, in his system these restrictions make an extra mechanism such as coloring unnecessary.

Okumura and Matsumoto proposed another approach called layered-stream programming [Okumura and Matsumoto 87]. It is a kind of programming paradigm in which recursively defined data structure called a layered-stream is used. All the values in the same layer are alternative solutions to the same problem. If we omit an associated color from a layered vector defined in section 3, a similar data structure would be obtained. Although the program written based on this paradigm provides high degree of parallelism, it is not declarative and it seems burdensome for a novice user to describe a problem using this paradigm.

To sum up, our contribution is as follows:

(1) Design of a logic programming language with and- and or-parallelism. In other words a parallel programming language with nondeterminism.
(2) and/or parallel execution model based on coloring scheme.
(3) Compilation to a committed choice language.

6.2. Future Works

There are some remaining problems for future works.

Suppression of irrelevant computations is an important problem to increase efficiency. If a process fails, then the conjunctive goals need not to be computed any more. However, the current system completes all the computations.

Another problem is to share the result of computation in the different worlds. Logically, computation in different worlds are independent. But from the pragmatic point of view, the knowledge discovered in a world could benefit other worlds. It is desirable to utilize such cross information flow over worlds.

7. SUMMARY

We have proposed a parallel problem solving language *ANDOR-II* which can declaratively

describe behaviors of concurrent systems consisting of many determinate and nondeterminate components acting interactively. It also has an ability of solving the problems such as combinatorial problems and cooperative planning on these systems. We have also shown the computation model based on coloring in which reasoning such as search and simulation over all possibilities can be executed efficiently. Although handling of multiple environment is expensive, it is inevitable to solve a problem on concurrent systems, which is our target.

ACKNOWLEDGMENTS

This research was done as one of the subprojects of the Fifth Generation Computer Systems (FGCS) project. We would like to thank Dr.K.Fuchi, Director of ICOT, for the opportunity of doing this research and Dr. K. Furukawa, Vice Director of ICOT, for his advice and encouragement.

REFERENCES

[Brand et al. 88] Brand,P., S.Haridi and D.H.D.Warren, "Andorra Prolog, The Language and Application in Distributed Simulation," Proc. of FGCS'88, to appear.

[Conery and Kibler 85] Conery,S and F.Kibler, "AND Parallelism and Nondeterminism in Logic Programs," New Generation Computing, Vol.3, No.1, pp.43-70, 1985.

[Clark and Gregory 84] Clark,K.L. and S.Gregory, "PARLOG: Parallel Programming in Logic," Research Report DOC 81/16,Imperial College of Science and Technology,1984.

[Clark and Gregory 87] Clark,K.L. and S.Gregory, "PARLOG and Prolog United," Proc. of 4th Int. Conf. on Logic Programming, pp.927-961,1987.

[Huhn 87] Huhn,M.N., "Distributed Artificial Intelligence," Pitnum/Morgan Kaufmann Publishers, 1987.

[Naish 87] Naish,L., "Parallelizing NU-Prolog," Technical Report of University of Melbourne, 1987.

[Okumura and Matsumoto 83] Okumura,A. and Y.Matsumoto, "Parallel Programming by Layered-Stream Methodology," pp.224-231 Proc.of Symposium on Logic Programming, 1987.

[Shapiro 83] Shapiro,E.Y., "A Subset of Concurrent Prolog and Its Interpreter," ICOT TR-003, 1983.

[Takeuchi 84] Takeuchi,A., "On An Extension of Stream-Based AND-Parallel Logic Programming Languages," Proc.of 1st Conf. of Japan Society of Software Science and Technology, pp.291-294, 1984 (in Japanese).

[Takeuchi et al. 87] Takeuchi,A., K.Takahashi and H.Shimizu, "A An description Language with AND/OR Parallelism for Concurrent Systems and Its Stream-Based Realization," ICOT TR-229,1987.

[Tamaki 86] Tamaki,H., "Stream-based Compilation of Ground I/O Prolog into Committed-choice Languages," Proc. of 4th Int. Conf. on Logic Programming, pp.376-393,1987.

[Ueda 86a] Ueda,K., "Guarded Horn Clauses," PhD. Thesis, The University of Tokyo, 1986.

[Ueda 86b] Ueda,K., "Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard," ICOT TR-208, 1986.

[Ueda 86c] Ueda,K., "Making Exhaustive Search Programs Deterministic," Proc.of 3rd Int. Conf. on Logic Programming, LNCS 225, Springer, pp.270-282, 1986.

[Ueda 87] Ueda,K., "Making Exhaustive Search Programs Deterministic, Part II," Proc.of 4th Int. Conf. of Logic Programming pp.356-375, 1987.

[Yang and Aiso 86] Yang,R. and H.Aiso, "P-Prolog: A Parallel Logic Language Based on Exclusive Relation," Proc.of 3rd Int. Conf. of Logic Programming pp.255-269, 1986.

APPENDIX

```
%% Block Loading Problem in ANDOR-II

:- mode pf1(-), block(+,-,-,+,+,+,+,+,-), action(+,-,-,+,+,+,+,-),
        or_merge(+,+,-), lmerge(+,+,-), rmerge(+,+,-),
        act_action(+,-,-,+,+,+,+,-), pas_action(+,-,-,+,+,+,+,-),
        term_action(+,+,-), send_msg(+,+,+,-,-,+,+).

pf1(Plan) :- true |
     block(a, AB,AC, BA,CA, c(clear,table), f(c,b),     [start],Pa),
     block(b, BC,BA, CB,AB, c(c,table),     f(a,table), [start],Pb),
     block(c, CA,CB, AC,BC, c(clear,b),     f(clear,a), [start],Pc),
     Plan=[Pa,Pb,Pc].

block(Id,S1,S2,R1,R2,Current,Final,H,P) :- true |
        or_merge(R1,R2,In),
        action(Id,S1,S2,In,Current,Final,H,P).

:- or_relation action/8.
action(Id,S1,S2,In,Current,Final,H,P) :-
        act_action(Id,S1,S2,In,Current,Final,H,P).
action(Id,S1,S2,In,Current,Final,H,P) :-
        pas_action(Id,S1,S2,In,Current,Final,H,P).

act_action(Id,S1,S2,In,c(On,Under),f(On,Under),H,P) :- true |
        S1=[],S2=[],
        term_action(In,H,P).
act_action(Id,S1,S2,In,c(clear,Under),f(FOn,FUnder),H,P) :- Under\=FUnder |
        get_Event_ID(Ev),    % system predicate which assigns
                             % a unique event identifier to the variable Ev
        send_msg(Id,FUnder,(move(Id),Ev),S1,S2,M1,M2),
        send_msg(Id,Under,(clear,Ev),M1,M2,T1,T2),
        action(Id,T1,T2,In,c(clear,FUnder),f(FOn,FUnder),
                                [(clear,Ev),(move_to(FUnder),Ev)|H],P).
act_action(Id,S1,S2,In,c(On,Under),f(FOn,FUnder),H,P) :-
        Under\=FUnder, On\=clear |
        get_Event_ID(Ev),
        send_msg(Id,On,(remove_from(Id),Ev),S1,S2,T1,T2),
        action(Id,T1,T2,In,c(clear,Under),f(FOn,FUnder),
                                    [(remove(On),Ev)|H],P).

pas_action(Id,S1,S2,[(move(BLK),Ev)|In],c(clear,Under),Final,H,P) :- true |
        action(Id,S1,S2,In,c(BLK,Under),Final,[(accept(BLK),Ev)|H],P).
pas_action(Id,S1,S2,[(move(BLK),Ev)|In],c(On,Under),Final,H,P) :- On\=clear |
        get_Event_ID(Ev),
        send_msg(Id,On,(remove_from(Id),Ev1),S1,S2,T1,T2),
        action(Id,T1,T2,In,c(BLK,Under),Final,
```

```
                                [(accept(BLK),Ev),(remove(On),Ev1)|H],P).
pas_action(Id,S1,S2,[(remove_from(BLK),Ev)|In],c(On,Under),Final,H,P) :-
        BLK\=Under |
        action(Id,S1,S2,In,c(On,Under),Final,[(ignore,Ev)|H],P).
pas_action(Id,S1,S2,[(remove_from(BLK),Ev)|In],c(clear,BLK),Final,H,P) :-
        true |
        action(Id,S1,S2,In,c(clear,table),Final,[(escape,Ev)|H],P).
pas_action(Id,S1,S2,[(remove_from(BLK),Ev)|In],c(On,BLK),Final,H,P) :-
        On\=clear |
        get_Event_ID(Ev1),
        send_msg(Id,On,(remove_from(Id),Ev1),S1,S2,T1,T2),
        action(Id,T1,T2,In,c(clear,table),Final,
                                [(escape,Ev),(remove(On),Ev1)|H],P).
pas_action(Id,S1,S2,[(clear,Ev)|In],c(On,Under),Final,H,P) :- true |
        action(Id,S1,S2,In,c(clear,Under),Final,[(cleared,Ev)|H],P).

term_action([],H,P) :- true | P=[end|H].

:- or_relation or_merge/3.
or_merge(X,Y,Z) :-  lmerge(X,Y,Z).
or_merge(X,Y,Z) :-  rmerge(X,Y,Z).

lmerge([X|X1],Y,Z) :- true | Z=[X|Z1], or_merge(X1,Y,Z1).
lmerge([],    Y,Z) :- true | Z=Y.

rmerge(X,[Y|Y1],Z) :- true | Z=[Y|Z1], or_merge(X,Y1,Z1).
rmerge(X,[],    Z) :- true | Z=X.

send_msg(a,b,Msg,S1,S2,T1,T2) :- true | S1=[Msg|T1],S2=T2.
send_msg(a,c,Msg,S1,S2,T1,T2) :- true | S2=[Msg|T2],S1=T1.
send_msg(b,c,Msg,S1,S2,T1,T2) :- true | S1=[Msg|T1],S2=T2.
send_msg(b,a,Msg,S1,S2,T1,T2) :- true | S2=[Msg|T2],S1=T1.
send_msg(c,a,Msg,S1,S2,T1,T2) :- true | S1=[Msg|T1],S2=T2.
send_msg(c,b,Msg,S1,S2,T1,T2) :- true | S2=[Msg|T2],S1=T1.
send_msg(_,table,Msg,S1,S2,T1,T2) :- true | S1=T1,S2=T2.
```