ICOT Technical Report: TR-608

TR-608

A Debugger for AND/OR Parallel Logic Programming Language ANDOR-II

by
K. Takahashi & A. Takeuchi (Mitsubishi)

December, 1990

© 1990, ICOT

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

A Debugger for AND/OR Parallel Logic Programming Language ANDOR-II

Kazuko TAKAHASHI Akikazu TAKEUCHI

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

(TEL) +81-6-497-7141
(FAX) +81-6-497-7289

takahashi@sys.crl.melco.co.jp takeuchi@sys.crl.melco.co.jp

Abstract

This paper discusses a debugger for an AND/OR parallel logic programming language *ANDOR-II* . Debugging a parallel programming language is complicated and burdensome, since the user has to handle the computation on multiple worlds. We propose a method that selects a representative world and peels it from the multiple worlds. Computation of *ANDOR-II* in this single world is equivalent to that of committed-choice languages. Therefore, debugging in a single world is attributed to that of committed-choice languages. Although this method alleviates the burden in debugging, if we perform reasoning over the worlds, more effective bug location can be realized.

1 Introduction

Due to the rapid progress of parallel architecture, there has been increased anticipation for parallel computation, which accelerates the research on concurrent programming techniques as well as efficient and sophisticated algorithms suitable for parallel architecture. In the field of logic programming, new languages are developed which exploit both AND- and OR-parallelism [Barget and Gregory 89][Clark and Gregory 87][Haridi et al. 88] [Naish 87] [Takeuchi et al. 88][Yang and Aiso 86]. They provide declarative descriptions for determinate and nondeterminate phenomena, and the programs written in these languages are executed in parallel utilizing powerful reasoning mechanisms. However, it is very difficult

to debug these languages. If one writes/runs a program in such an AND/OR parallel logic programming language, what kind of environment is desired?

In parallel programming, the burden of debugging is so heavy, because: (1) It is hard to follow multiple reductions executed in parallel with a sequential interface; we have to follow several reductions in an interleaving manner; (2) The time to bind the value to variables and the time of the binding's propagation are unknown and nondeterminate.

However, in committed-choice languages, much research has been undertaken to realize a declarative debugger [Huntbach 87][Lichtenstein and Shapiro 89][Lloyd 87][Lloyd and Takeuchi 86] [Takeuchi 86][Tatemura and Tanaka 89][Ueno and Kanamori 90].

For AND/OR parallel logic programming languages, debugging is much harder because the user has to trace all possibilities. And almost no debugging tools are developed for this class of languages so far. It is necessary to provide some environment for these languages, and it is high time to discuss this issue.

In this paper, we propose a debugger for an AND/OR parallel logic programming language *ANDOR-II* [Takahashi et al. 90][Takeuchi et al. 88] [Takeuchi 90]. The idea is a debugger which focuses on "one possible world." In *ANDOR-II* , computation proceeds based on "colored worlds." All possible computations are executed on the colored worlds independent of one another, and solutions from all the worlds are collected. A buggy program may generate an incorrect answer or fail unexpectedly. In this case, we want to inspect the single world that generates an incorrect answer or fails, rather than inspect the behaviors of all the worlds. The reason is twofold: (1) it is easier to inspect a single world and (2) several worlds may share a computation, and fixing a bug in one world may simultaneously fix bugs in other worlds. When debugging, at first we perform the computation to record the history called *computation forest*, then we peel a tree corresponding to a representative world from the whole forest, and reconstruct the computation along this tree. This tree is equivalent to the computation tree of committed-choice languages. Therefore, debugging in a single world is equivalent to debugging in committed-choice languages. It implies that we can use the debugging methodology or techniques developed for

committed-choice languages.

This method of debugging greatly alleviates the burden of debugging. However, if we survey more than one world, new facts can be obtained by reasoning on solutions and their associated colors. For instance, assume that a color consists of two parts, and that the following two facts are obtained as a result of computation.

(1) Computation succeeds if we take the *red* path in the first part and the *white* path in the second part.

(2) Computation fails if we take the *red* path in the first part and the *blue* path in the second part.

In this case, we can infer the fact that the *red* path in the first part is safe and the *blue* path on the second part causes a failure. Therefore, we examine only the *blue* path instead of examining all paths. Such meta reasoning realizes more effective bug location.

This paper is organized as follows. In section 2, we overview the language *ANDOR-II* and its computation model. In section 3, a framework of a debugger is provided. In section 4, meta reasoning over worlds is discussed. And in section 5, conclusion and future works are shown.

Familiarity with GHC [Ueda 86] is assumed.

2 Language ANDOR-II

2.1 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 AND-clause has the same syntax as that of GHC [Ueda 86]. An OR-predicate definition consists of a mode declaration and a set of OR-clauses. An OR-clause has no guard goals. A predicate defined by an AND(OR)-predicate definition is called an *AND(OR)-predicate*. A clause of either type can contain both AND-predicates and OR-predicates in its body part. An *ANDOR-II* program can be read in the same way as GHC except for the OR-predicates. Similar to flat languages, a goal in a guard part is restricted

to a test predicate.

Mode declaration is in the form of

$:-mode\ P(m_1,\ldots,m_N).$

where P, N, m_i are predicate symbol, arity, mode of n-th argument of P, respectively. m_i is either + or − which stands for *reference-only(read)* or *write*, respectively. Reference-only mode means that the corresponding arguments are never instantiated during the computation of P [1], and write mode has the complementary meaning. A variable appearing in reference-only mode in a head never appears in the write mode in the body in the same clause.

Let us show an example of a program of *cycle*, which consists of two processes interacting with each other. If a component which forms a cycle has nondeterminacy, it is troublesome for a conventional language to give them a declarative expression, since the user has to be concerned about expressing synchronization. *ANDOR-II* provides a declarative description for such a problem.

Example 2.1

```
%% AND-predicates
  :- mode cycle(-).
  cycle(Y) :- true | p1([2|X],Y), p2(Y,X).                    % C1
  :- mode p1(+,-).
  p1([stop],Y) :- true | Y=[ ].                               % C2
  p1([X|X1],Y) :- X\ =stop | multi(X,A), Y=[A|Y1], p1(X1,Y1). % C3
  :- mode p2(+,-).
  p2([X|X1],Y) :- X<20 | wave(X,A), Y=[A|Y1], p2(X1,Y1).      % C4
  p2([X|X1],Y) :- X>=20 | Y=[stop].                           % C5
%% OR-predicates
  :- mode wave(+,-).
```

[1] The computation of an atom means the whole computation tree, the root of which is the atom. And if the predicate symbol of the atom is P, we simply call it the computation of P.

```
wave(X,Y) :- Y:=X-1.                    % C6
wave(X,Y) :- Y:=X+1.                    % C7
:- mode multi(+,-).
multi(X,Y) :- Y:=X*X.                   % C8
multi(X,Y) :- Y:=X*X*X.                 % C9
```

In the clause defining *cycle*, processes $p1$ and $p2$ form a cyclic structure with communication channels X and Y. $p1$ receives the stream via its first argument, and executes the goal *multi* on the received element. *multi* has two possibilities, generating a squared value or generating a cubed value. $p1$ sends the results to $p2$ via its second argument. $p2$ receives the stream via its first argument, executes the goal *wave* 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.

2.2 Semantics

Next, we will show an operational semantics of *ANDOR-II* . Further discussion is given in [Takeuchi 90].

The computation starts from the initial world and proceeds by reducing the goals in parallel. The world is a set of conjunctive goals. Reductions are controlled by the following four rules.

The reduction rule of AND-predicates is similar to that of GHC [Ueda 86]. Here, by the term *guard computation of a clause C*, we mean both head unification and execution of the guard part.

Rule of Suspension Unification invoked directly or indirectly in guard computation of an AND-clause C called by a goal atom G cannot instantiate G. A piece of unification that can succeed only by violating the above rules is suspended until it can succeed without such violation.

Rule of Commitment When some AND-clause C called by a goal G succeeds in solving its guard, clause C tries to be selected for subsequent execution of G. To be selected, C must confirm at first 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, and the reduction is said to be *AND-reduction between* G *and* C.

As for the reduction of OR-predicates, two rules are imposed.

Rule of Suspension Head unification between a goal atom G and an OR-clause C cannot instantiate G. A piece of unification that can succeed only by violating the rules above is suspended until it can succeed without such violation.

Rule of Proliferation When N OR-clauses $C_1, \ldots, C_N$ called by a goal atom G succeed in head unifications, they all try to be selected for subsequent execution of G. To continue the execution, N copies of the current world must be made indivisibly, and the current world is said to *proliferate into* N *worlds*, and the reduction is said to be *OR-reduction between* G *and* $C_1, \ldots, C_N$.

2.3 Implementation

Although the semantics shown above implies an eager copying of the world at every OR-reduction, a lazy copying scheme called coloring scheme is adopted in the actual implementation [Takeuchi 90]. The debugger discussed here is also developed based on this coloring scheme.

2.3.1 Colored World

In *ANDOR-II* , computation starts from a given goal on an initial world with an initial color and proceeds as follows. All conjunctive goals in a world are executed in parallel. AND-predicates are reduced similarly as these in GHC. In case of OR-predicates, instead of immediately making all copies of the current world, copy worlds are constructed incrementally: first new distinct colors are determined for each new world, then the computations

corresponding to all definition clauses are executed in parallel. As a result, if a variable comes to have multiple bindings, it is represented in a form of a special data structure called *colored vector*. An element of the colored vector is a pair of a value and a color, which denotes the value of the variable corresponding to the color. And on the other hand, when the goal to be reduced has a colored vector in its argument, copies of the goal are made in the same number as that of the elements of the colored vector, and reductions are performed for each one. Incremental copying of the world is realized in this way.

Now, we will present a formal definition of a color.

Definition 2.1 (primitive color) *A primitive color is a pair of symbols (P,S) where P and S are called a branching point and a branching arc, respectively. Two primitive colors are defined to be orthogonal with each other if and only if they share the same branching point, but have different branching arcs.*

A branching point is a unique identifier of the event invoking an OR-predicate, and a branching arc is an identifier of the selected clause at that invocation.

Definition 2.2 (color) *A color is defined to be a set of primitive colors, in which no element is orthogonal with each other. Two colors α, β are defined to be orthogonal iff $\exists p_1 \in \alpha, p_2 \in \beta$, such that p_1 and p_2 are orthogonal with each other.*

Every goal has a color of the world where it is invoked and it has a channel between itself and the manager which is responsible for identifying branching points. At an invocation of an OR-predicate, a new primitive color is added to the current color. New primitive colors are determined as follows: the branching point is given by the manager and the branching arcs are given as the numbers corresponding to each definition clause.

Definition 2.3 (colored vector) *A colored vector is a finite set of pairs of terms and colors. A colored vector is denoted by*

$$< T^1|\alpha^1, \ldots, T^k|\alpha^k >$$

where k is the size of the set, T^i is a term and α^i's are orthogonal colors.

As a result of the computation of an OR-predicate, the variables in write mode are bound to the colored vector $< T^1|\alpha^1, \ldots, T^k|\alpha^k >$ where $T^i (i = 1, \ldots, k)$ is the binding to the variable in the world whose color is α^i. When a goal G receives a colored vector, the computation is possible only for the value whose attached color is not orthogonal with that of the world where G is called.

Let G be a goal $p(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$ where $X_1, \ldots, X_n$ are the arguments in reference-only mode, and $Y_1, \ldots, Y_m$ are in write mode. Assume that arguments $X_1, \ldots, X_n$[2] receive the following colored vectors in the world with the color α_0.

$$< T_1^1|\alpha_1^1, \ldots, T_1^{k_1}|\alpha_1^{k_1} >$$

$$< T_2^1|\alpha_2^1, \ldots, T_2^{k_2}|\alpha_2^{k_2} >$$

$$\vdots$$

$$< T_n^1|\alpha_n^1, \ldots, T_n^{k_n}|\alpha_n^{k_n} >$$

The following computation is needed for each set of colors $\alpha_1^{F_1}, \ldots, \alpha_n^{F_n}$ such that no two of them are orthogonal with each other, where for each $i (i = 1, \ldots, n)$, $\alpha_i^{F_i} \in \{\alpha_i^1, \ldots, \alpha_i^{k_i}\}$ and $\alpha_i^{F_i}$ is not orthogonal with α_0. Let α be the union of the colors $\alpha_0, \alpha_1^{F_1}, \ldots, \alpha_n^{F_n}$, and θ be the substitution $\{X_1/T_1^{F_1}, \ldots, X_n/T_n^{F_n}\}$. If p is an AND-predicate, then the reduction between the goal $G\theta$ and a clause C is carried out in the world with the color α. If p is an OR-predicate, then the reduction between the goal $G\theta$ and the clauses $C_1, \ldots, C_N$ is carried out in the world with the color α. Note that α's are orthogonal with one another.

Here, we define the concepts *success*, *failure* and *suspension* of the computation. *success*, *failure* and *suspension* in a single world are defined similar to committed-choice languages. If the computation of a goal Q succeeds in some world, then the computation of Q is defined to be *success*; if it fails in all the worlds, then the computation of Q is defined to be *failure*; otherwise, the computation of Q is defined to be *suspension*.

[2] Actually, only the arguments which are referred to in at least one clause during head unification or guard computation are to be considered. Here, to simplify the problem, it is assumed that all the arguments are these types.

3 Debugger for ANDOR-II

As the execution of *ANDOR-II* is extended over multiple worlds, its debugging seems to be complicated and burdensome. However, it is "a bug" that has en effect on bugs in several worlds, and it is enough to examine a representative world. At first, we perform the computation to record the history called *computation forest*, peel a tree corresponding to a representative world from the whole forest, and reconstruct the computation along this tree.

Following [Lloyd and Takeuchi 86], we consider the following three erroneous computations:

(1) success with an incorrect answer

(2) unexpected failure

(3) unexpected suspension

3.1 Meta Interpreter

First of all, we show a fundamental meta interpreter in KL1 in order to capture a behavior of the execution of *ANDOR-II* program. Then, we will enhance it for a debugger.

An *ANDOR-II* program is preprocessed so that it can make an interface for a meta interpreter.

An AND-clause

Head :- Guard | Body.

is transformed into the clause

reduction(Head, *Clause*, IDc, *Ctl*) :- Guard | *Clause*=Body.

An OR-predicate definition

$p(X_{11},\ldots,X_{1n})$:- $Body_1$.

$\vdots$

$p(X_{m1},\ldots,X_{mn})$:- $Body_m$.

is transformed into the clauses

reduction($p(Y_1,\ldots,Y_n)$, *Clause*, IDc, *Ctl*) :- true |

$Clause$ = [$p_1(Y_1,\ldots,Y_n),\ldots,p_m(Y_1,\ldots,Y_n)$].

reduction($p_1(X_{11},\ldots,X_{1n})$, $Clause$, IDc, Ctl) :- true | $Clause$=$Body_1$.

$\vdots$

reduction($p_m(X_{m1},\ldots,X_{mn})$, $Clause$, IDc, Ctl) :- true | $Clause$=$Body_m$.

where p_i's ($i = 1,\ldots,m$) are new distinct predicate symbols.

For each predicate, the clause handling the failure case is added to the end of the transformed program preceded by 'otherwise' clause so that it is invoked when all the other clauses fail to be committed.

reduction(_, $Clause$, _, _) :- true | $Clause$ = '$failure$'.

Here, Col denotes the color attached to the world where the goal is called. Ids denotes the channel expanded between goals and the manager called *bp_handler*. This manager is required to assign a unique number to each invocation of an OR-predicate, since such invocations may occur at several places in parallel. Each goal sends a request to the *bp_handler* via this channel when it invokes an OR-predicate. These requests are merged and sent to *bp_handler*. The request is in the form of

$$get\_bp(Bp)$$

where Bp is a variable used for a back communication. When *bp_handler* receives the request, it determines the unique identifier and sends it back to the goal process. Ctl is a channel onto which the user sends a contol message. IDc is the clause identifier. Ctl and IDs are not used in the fundamental meta interpreter.

Below, we show the top level of a fundamental meta interpreter. The predicate *reduce* plays the main role which is a goal reduction. In the program, user-defined predicates are placed in the guard part for making the description simpler. '– >' is a macro definition of KL1 which indicates the conditional.

```
reduce( (G1,G2), Col, Ids ) :- true |
  merge( { Ids1,Ids2 }, Ids ),        % division of the stream
```

```
   reduce(G1, Col, Ids1 ),              % reduction of a goal G1
   reduce(G2, Col, Ids2 ).              % reduction of a goal G2
reduce( Goal, Col, Ids ) :- is_colored_vector(Goal) |
                                        % receiving a colored vector
   pickup_non_orthogonal( Goal, Col, SlctdGoals ).
                                        % picking up the elements of the colored vector
                                        % whose colors are not orthogonal with Col
   do_for_each_goal( SlctdGoals, Ids ).
                                        % execution of the goal with the union color
                                        % for each element
reduce( Goal, Col, Ids ) :- is_or_pred(Goal) |
                                        % OR-reduction
   reduction(Goal,Clause,_,_),
   do_for_each_clause(Clause,Col,1,Ids).
                                        % setting of the initial value of the branching arc
                                        % and reduction for each clause
otherwise.
reduce( Goal, Col, Ids ) :- true |      % for AND-clause
   reduction( Goal,Clause, _, _ ),
   ( Clause = '$failure$' - > substitute_void(Goal), Ids=[ ] ;
     otherwise; true - > reduce(Clause,Col,Ids) ).
 do_for_each_clause([C|Cls],Col,BArc,Ids) :- true |
   Ids=[get_bp(Bp)|Idss],               % request for giving the branching point
   Idss={Ids1,Ids2},                    % division of the stream
   add_color_element(Col,(Bp,BArc),NewCol),
                                        % addition of a new color element
   BArc1:=BArc+1,                       % branching arc for the next chosen clause
   reduce(C,NewCol,Ids1),               % reduction of a clause
```

```
  do_for_each_clause(Cls,Col,BArc1,Ids2).
do_for_each_clause([ ],_,_,Ids) :- true |
  Ids=[ ].                          % closing the stream
do_for_each_goal( [(G,Col)|Gs], Ids ) :- true |
  Ids={Ids1,Ids2},                  % division of the stream
  reduce(G,Col,Ids1),               % reduction of a goal
  do_for_each_goal(Gs,Ids2).
do_for_each_goal( [ ], Ids ) :- true |
  Ids=[ ].                          % closing the stream
```

If a goal receives a colored vector, it performs the reduction only for the value with the color which is not orthogonal with the current color. This is the most important process in the execution of *ANDOR-II* . We have to handle all cases in which each variable is bound to a colored vector. There are several patterns depending on the variable which is bound to a colored vector. Therefore, in fact, several clauses are created in this case, and the actual implementation realizes the mechanism in the same way as that in the compiler [Takeuchi 90].

3.2 Meta Interpreter Recording A History

Now, we will introduce a meta interpreter enhanced for debugging. It is the meta interpreter which records the history of computation. Assume that all the clauses in the program are given distinct numbers and the goals in a clause are given locally distinct numbers. When a goal with a history H is invoked, it accumulates its own identifier (C, G) onto H where C, G are a clause identifier and a goal identifier, respectively.

Similar to the fundamental meta interpreter, an *ANDOR-II* program is preprocessed in the same way, except that a clause for handling the deadlock case is added [3].

reduction(_, *Clause*, _, *deadlock*) :- true | *Clause* = '$suspension$'.

[3]The user can know deadlock, for example, by checking the number of reductions executed so far. It is assumed that deadlock is informed to each process by the user.

The top level of the meta interpreter which records the history written in KL1 is shown below. In this case, *reduce* should have three additional arguments: *Forest*, *Label* and *Ctl*. *Forest* denotes a history of the computation. *Label* denotes the label of the current node in a form of the triple (C, G, α), where C,G and α are a clause identifier, a goal identifier and a current color, respectively. *Ctl* is a channel onto which the user sends a control message.

```
reduce( (G1,G2), Col, Ids, Forest, node(IDc,IDg,Col), Ctl ) :- true |
   merge( { Ids1,Ids2 }, Ids ),                          % division of the stream
   Forest = ( LF, RF ),                                  % branch of the forest
   reduce(G1, Col, Ids1, LF, node(IDc,IDg,Col), Ctl),% reduction of a goal G1
   IDg1:=IDg+1,                                          % increment a goal ID
   reduce(G2, Col, Ids2, RF, node(IDc,IDg1,Col), Ctl).
                                                         % reduction of a goal G2
reduce( Goal, Col, Ids, Forest, Node, Ctl ) :- is_colored_vector(Goal) |
                                % receiving a colored vector
   pickup_non_orthogonal( Goal, Col, SlctdGoals ).
                                % picking up the elements of the colored vector
                                % whose colors are not orthogonal with Col
   do_for_each_goal( SlctdGoals, Ids, Forest, Node, Ctl ).
                                % execution of the goal with the union color
                                % for each element
reduce( Goal, Col, Ids, Forest, Node, Ctl ) :- is_or_pred(Goal) |
                                % OR-reduction
   reduction(Goal,Clause,_,Ctl),
   do_for_each_clause(Clause,1,Ids,Forest,Node,Ctl).
                                % setting of the initial value of the branching arc
                                % and reduction of each clause
```

```
otherwise.
reduce( Goal, Col, Ids, Forest, node(IDc,IDg,_), Ctl ) :- true |
                                                    % for AND-clause
  reduction( Goal, Clause, IDc1, Ctl),
  ( Clause = '$failure$' - >
      substitute_void(Goal), Ids=[ ],
      Forest=[node(IDc,IDg,Col),failure] ;           % storing the result
    Clause = '$suspension$' - >
      substitute_suspend(Goal), Ids=[ ],
      Forest = [node(IDc,IDg,Col),suspension] ;      % storing the result
    otherwise; true - >
      Forest = [node(IDc,IDg,Col),F1],               % storing the result
      reduce(Clause,Col,Ids,F1,node(IDc1,0,Col),Ctl ).
                                    % reset of the goal ID
 do_for_each_clause([C|Cls],Col,BArc,Ids,Forest,Label,Ctl) :- true |
  Ids=[get_bp(Bp)|Idss],
                                    % request for giving the branching point
  Idss={Ids1,Ids2},                                  % division of the stream
  add_color_element(Col,(Bp,BArc),NewCol),           % addition of a new color element
  BArc1:=BArc+1,                                     % branching arc for the next chosen clause
  copy_node(Forest,F1,F2),                           % copy of the node for the next clause
  reduce(C,NewCol,Ids1,F1,Label,Ctl),                % reduction of a clause
  do_for_each_clause(Cls,Col,BArc1,Ids2,F2,Label,Ctl).
do_for_each_clause([ ],_,_,Ids,Forest,_,_) :- true |
  Ids=[ ],                                           % closing the ID-stream
  end_of_copying(Forest).                            % termination of copying of the node
do_for_each_goal( [(G,Col)|Gs], Ids, Forest, Label, Ctl ) :- true |
  Ids={Ids1,Ids2},                                   % division of the stream
```

```
    copy_node(Forest,F1,F2),                    % copy of the node for the next goal
    reduce(G,Col,Ids1,F1,Label,Ctl),            % reduction of a goal
    do_for_each_goal(Gs,Ids2,F2,Label,Ctl).
do_for_each_goal( [ ], Ids, Forest, _, _ ) :- true |
    Ids=[ ],                                    % closing the ID-stream
    end_of_copying(Forest).                     % termination of copying of the node
```

Now, we explain how to record a history of computation based on this program.

Rule of Labelling

Let IDc and IDg denote identifiers of a clause C and a goal G.

(1) Add the label *root* to the root node.

(2) When the AND-reduction between a goal G and a clause C is carried out in a world with a color α, and the goals $G_1, \ldots, G_k$ are invoked, then add the labels $(IDc, IDg_1, \alpha), \ldots, (IDc, IDg_k, \alpha)$ to the nodes corresponding to $G_1, \ldots, G_k$.

(3) When the OR-reduction between a goal G and clauses $C_1, \ldots, C_m$ is carried out in a world with a color α, and the goals $G_{11}, \ldots, G_{1k_1}, \ldots, G_{m1}, \ldots, G_{mk_m}$ are invoked, let the new colors corresponding to $C_1, \ldots, C_m$ be $\alpha_1, \ldots, \alpha_m$. Add the labels $(IDc_1, IDg_{11}, \alpha_1), \ldots, (IDc_1, IDg_{1k_1}, \alpha_1), \ldots, (IDc_m, IDg_{m1}, \alpha_m), \ldots, (IDc_m, IDg_{mk_m}, \alpha_m)$ to the nodes corresponding to $G_{11}, \ldots, G_{1k_1}, \ldots, G_{m1}, \ldots, G_{mk_m}$, respectively.

(4) When a goal G receives a colored vector $< T^1|\alpha^1, \ldots, T^n|\alpha^n >$ in a world with a color α, let $\beta^1, \ldots, \beta^m$ be the colors which are not orthogonal with α where $\beta^j (j = 1, \ldots, m) \in \{\alpha^1, \ldots, \alpha^n\}$, and $G\theta^j$ be the goal obtained by applying the substitution θ^j corresponding to β^j to the goal G. If G is a goal of an AND-predicate and for each $j (j = 1, \ldots, m)$, the reduction between the goal $G\theta^j$ and a clause C_j is carried out and the goals $G_{j1}, \ldots, G_{jk_j}$ are invoked, then add the labels $(IDc_j, IDg_{j1}, \beta^j), \ldots, (IDc_j, IDg_{jk_j}, \beta^j)$ to the nodes corresponding to $G_{j1}, \ldots, G_{jk_j}$, respectively. If G is

a goal of an OR-predicate and for each $j(j = 1, \ldots, m)$, the reduction between the goal $G\theta^j$ and clauses $C_1, \ldots, C_N$ are carried out and the goals $G_{11}, \ldots, G_{1k_1}, \ldots, G_{N1}, \ldots, G_{Nk_N}$ are invoked, then let the new colors corresponding to $C_1, \ldots, C_N$ be $\beta_1^j, \ldots, \beta_N^j$. Add the labels $(IDc_1, IDg_{11}, \beta_1^j), \ldots, (IDc_1, IDg_{1k_1}, \beta_1^j), \ldots, (IDc_N, IDg_{N1}, \beta_N^j), \ldots, (IDc_N, IDg_{Nk_N}, \beta_N^j)$ to the nodes corresponding to $G_{11}, \ldots, G_{1k_1}, \ldots, G_{N1}, \ldots, G_{Nk_N}$, respectively.

(5) When a built-in goal G succeeds, fails or suspends, add the label *success*, *failure* or *suspension* to the leaf node corresponding to *success*, *failure* or *suspension*, respectively. ∎

When the computation terminates, the history of the computation is obtained in the form of a forest whose node corresponds to each goal. For a goal of an OR-predicate and a goal which receives a colored vector, several reductions are called. Therefore, the corresponding node stands for a set of reductions. We call these nodes *multi-nodes*.

The structure of the forest is shown below.

Forest ::= [Label, Nodes] | [Label, < Nodes_1, ..., Nodes_k >]

Nodes ::= (Forest_1, ..., Forest_n) | *success* | *failure* | *suspension*

Label ::= node(ClauseID,GoalID,Color) | root

When Forest is in the form of [Label, < Nodes_1, ..., Nodes_k >], the root node of this forest is a multi-node.

The obtained computation forest is equivalent to an AND/OR-tree and, a history in a single world is equivalent to an AND-tree.

The computation forest of the example *cycle* is shown in Figure 1.

3.3 Reconstruction of A Computation with A Designated Color

If in some world, an expected solution is not obtained, we start debugging. Debugging consists of two phases, reconstruction of buggy computation and its diagnosis. We adopt

two methods for reconstruction: reconstruction with a designated color and reconstruction by stepwise selection.

In this subsection, we explain the former reconstruction method. This method is useful in case the world in which a bug appears is clear. If an incorrect answer is obtained in some world, we know that a bug exists in the computation in that world. If *failure/suspension* is obtained in some world despite the fact that all possible computations are expected to succeed, we also know that a bug exists in the computation in that world. In these cases, we take one buggy world as a representative and start reconstruction with its color. The reason why a representative world is taken is: (1) it is easier to inspect a single world, (2) several worlds may share a computation, and fixing a bug in one world may simultaneously fix bugs in other worlds.

Reconstruction of computation in the world specified by a color is achieved by re-executing the initial goal. The execution is guided by the special meta interpreter, which takes the computation forest and the color as the additional arguments. At each branching point, the meta interpreter traces only the world with the designated color. As for the other worlds, no computation is performed. On the multi-nodes in the computation forest, only the nodes with the designated color are re-executed, and the other nodes remain untraced. Thus, the computation with the designated color is reconstructed just as the tree with the color is peeled from the forest. Formal description of the meta interpreter is shown below.

Rule of Reconstruction of the Computation

To a given computation forest F and a designated color α_0, apply the following procedure. Let IDc and IDg denote the identifiers of clause C and goal G.

[initial state]

If a root node is not a multi-node and a forest is in the form of $[root, (Forest_1, \ldots, Forest_n)]$, then let $Forest_1$ be $[node(IDc', IDg', \alpha), Nodes]$, and perform the reduction between the given goal and clause C'. If a root node is a multi-node and a forest is in the form of $[root, < Nodes_1, \ldots, Nodes_k >]$, then peel the multi-node.

[expansion of a node]

If a forest is in the form of $[node(IDc, IDg, \alpha), (Forest_1, \ldots, Forest_n)]$, let $Forest_1$ be $[node(IDc', IDg', \alpha), Nodes]$, then perform the reduction between goal G and clause C'.

[peeling of a multi-node]

If a forest is in the form of $[node(IDc, IDg, \alpha), < Nodes_1, \ldots, Nodes_k >]$, let $\alpha_i (i = 1, \ldots, k)$ be a color in the label of $Nodes_i$. If (IDc, IDg) is a goal of an OR-predicate and α_0 does not contain the branching point of this node, then do nothing. Otherwise, peel the nodes $Nodes_F (F \in \{1, \ldots, k\})$ where α_F is not orthogonal with α_0. That is, if we assume that $Nodes_F$ is $(Forest_{F1}, \ldots, Forest_{Fm_F})$ and $Forest_{F1}$ is $[node(IDc_F, IDg', \alpha_F), Nodes]$, then perform the reduction between goal G and clause C_F.

[termination]

For a forest $[node(IDc, IDg, \alpha), Node]$ where $Node$ is either *success*, *failure* or *suspension*, execute the goal G. ∎

The above rule guarantees the reconstruction of the computation in the world. It is proved as follows. We assume without losing generality that every predicate has at least one argument in write mode to make the discussion simpler.

For a multi-node, let $Nodes_1, \ldots, Nodes_k$ correspond to $\alpha_1, \ldots, \alpha_k$, respectively.

Assume that the multi-node corresponds to the goal of an OR-predicate and the designated color α_0 does not contain the branching point of this node. Since all the history of the concerned branching points are passed to the final solution and reflected in its color, the node is unrelated to obtaining the solution, and the branching arc at that point does not affect the computation which gets the solution. Therefore, the computation which gets the solution is reconstructed even if such a node is ignored.

Assume that the multi-node corresponds to the goal of an OR-predicate and α_0 contains the branching point P of this node. Let $S_1, \ldots, S_k$ be the branching arcs of that branching point. Note that $\alpha_i (i = 1, \ldots, k)$ has (P, S_i) as a primitive color. It is proved that there is only one $Nodes_F$ whose color is not orthogonal with α_0, since if α_0 has (P, Arc) as a

primitive color, Arc is one of $S_1, \ldots, S_k$, and the primitive colors $(P, S_1), \ldots, (P, S_k)$ are orthogonal with one another.

Assume that the multi-node corresponds to the goal receiving a colored vector. In this case, there is only one element that is not orthogonal with α_0. This is shown by deriving contradiction where there are two such elements. Suppose that there exist two colors α_i and $\alpha_j (i, j = 1, \ldots, k)$ which are not orthogonal with α_0. As α_i and α_j are orthogonal with each other, they have different branching arcs S_i and S_j at the branching point P. Let α_0 have (P, S) as a primitive color. S and S_i are not orthogonal, S and S_j are not orthogonal, while S_i and S_j are orthogonal with each other. This is a contradiction. Thus, there is only one $Nodes_F$ whose color is not orthogonal with α_0.

Therefore, no colored vector appears in reconstruction, since only one world is selected at each branching point. Most branches are pruned, and only the branches on a peeled tree can be performed.

In this method, which clause is used in each reduction is determinate. Therefore, the computation can be reconstructed without considering the synchronization. Note that the reconstructed forest is an AND-tree which has no multi-nodes. Therefore, it is equivalent to the computation tree of GHC.

Ignoring the unrelated nodes implies that the solutions other than the target one are not generated even in the same world.

For a forest F, take a node N whose color is α and the corresponding goal is A. Let $R_1, \ldots, R_n$ be the solutions in the worlds whose colors are not orthogonal with α. Assume that the subtree whose root node is N contains a node corresponding to the goal of an OR-predicate. If we take α as the designated color, then the computation which generates A is reconstructed. However, branching points under N are undefined, and no reduction corresponding to these is performed. As a result, the solutions $R'_1, \ldots, R'_m (m \leq n)$ are obtained where R'_i is possibly a partially specified form of R_j for some $j (j = 1, \ldots, n)$.

The above discussion implies that the following theorem holds.

Theorem 3.1 *Assume that the execution of a goal G terminates. Let α be the color of the world in which a goal atom A appears, and let F be the computation forest of goal G. Then, the computation which generates A can be reconstructed from F and α.*

We illustrate the debugging procedure by Example 2.1. Each definition clause is labelled with the identifier $Ci(i = 1, ..., 9)$. And for each clause, goals are labelled with the identifiers $G1, ..., Gn$ from left to right where n is the number of body goals of the clause.

Invoke the goal $cycle(Y)$, then the following solutions are gained.

```
Y = [4,9,64],
    [4,9,512],
    [4,9,100],
    [4,9,1000],
    [4,27],
    [4,25],
    [4,125],
    [8,49],
    [8,343],
    [8,81],
    [8,729]
```

Suppose that the clause C7 is replaced by

C7':: wave(X,Y) :- true | Y:=X+10.

Then, the following solutions R1,R2,...,R11 would be gained. The sequence of digits $d_0 d_1 \cdots d_n$ denotes the color of the corresponding world, where $d_i(i = 0, 1, \ldots, n)$ corresponds to the branching arc of the i-th branching point. $d_j = 0$ means that j-th branching point is not selected yet.

```
Y = [4,9,64],          1101000110   ::R1
    [4,9,512],         1101000120   ::R2
```

```
[4,9,361],          1101000201  ::R3*
[4,9,6859],         1101000202  ::R4*
[4,27],             1102000000  ::R5
[4,196],            1200100000  ::R6*
[4,2744],           1200200000  ::R7*
[8,49],             2010010000  ::R8
[8,343],            2010020000  ::R9
[8,324],            2020001000  ::R10*
[8,6832]            2020002000  ::R11*
                  * denotes incorrect answers
```

Among the solutions, R3,R4,R6,R7,R10,R11 are the incorrect solutions. We select one of them, say, R3, and start debugging. The designated color α is "1101000201." The first element "1" of the color "1101000201" means that the branching arc "1" is selected at the branching point "0." Similar analysis can be done for the other elements. Given a computation forest, the branching point of a multi-node which corresponds to the goal of an OR-predicate can be gained in the following manner. Consider a multi-node has a label [Label, < Nodes_1,...,Nodes_k >]. Let $\alpha_1, \ldots, \alpha_k$ be the colors of Nodes_1,...,Nodes_k, respectively. $\alpha_1, \ldots, \alpha_k$ are orthogonal with one another and differ only in one branching point. If i is such a point, then i is the branching point of the node. The computation forest for Example 2.1 is shown in Figure 1. The number added to the top-left of a multi-node denotes the branching point of that node. For example, since the colors of the nodes $(6, 1, \alpha_3)$ and $(7, 1, \alpha_4)$ are "1100000000" and "1200000000," respectively, the branching point of their parent node $(4, 1, \alpha_1)$ is "1."

Reconstruction starts from the root node *root* with the designated color "1101000201." As the root node is not a multi-node, and its child nodes are $(1, 1, \alpha_0)$ and $(1, 2, \alpha_0)$, perform the reduction between the goal *cycle*(Y) and the clause $C1$. As a result, the child nodes $(1, 1. \alpha_0)$ and $(1, 2, \alpha_0)$ are traced. Then, for the forest whose root node is $(1, 1, \alpha_0)$

which has the child nodes $(3,1,\alpha_0)$, $(3,2,\alpha_0)$ and $(3,3,\alpha_0)$, perform the reduction between the goal corresponding to the node $(1,1,\alpha_0)$ and the clause $C3$. As a result, the child nodes $(3,1,\alpha_0)$, $(3,2,\alpha_0)$ and $(3,3,\alpha_0)$ are traced. As $(3,1,\alpha_0)$ is a multi-node, check its child nodes $(8,1,\alpha_1)$ and $(9,1,\alpha_2)$, where α_1 and α_2 are the colors "1000000000" and "2000000000," respectively. Peel the node $(8,1,\alpha_1)$ since α_1 is not orthogonal with the designated color. In this way, the reconstruction proceeds.

In Figure 1, the nodes in the computation forest are located in the several layers. And in the reconstruction with the color "1101000201," only the nodes in the surface are peeled off, and the computation in this world is reconstructed.

3.4 Reconstruction by Stepwise Selection

Another method of reconstruction is the reconstruction by stepwise selection. This method is useful for the case in which the world in which a bug appears is unclear. It is hard to distinguish *unexpected* failure from *expected* failure since only "*failure*" is gained as a result of that world. In this case, the desirable method is to choose a color incrementally as computation proceeds. At every invocation of an OR-predicate, we designate the clause to be used, namely, the branch to be followed. The remaining branches are pruned. This method is similarly realized with the previous method.

4 Discussion

4.1 Meta Reasoning

The method of debugging described so far greatly alleviates the burden of debugging. However, if we survey more than one world, more effective bug location is possible.

Execution of *ANDOR-II* gets all solutions in parallel for all possible cases. In addition, not only solutions but also their histories can be obtained in the form of a color. Therefore, it is possible to perform meta reasoning on these solutions and colors to accelerate bug location algorithm.

Suppose that, a bug manifests in some world while there also exists a world in which the

computation terminates successfully. In such a case, we can derive some useful information by comparing their colors, and hence narrow the bug's location.

Consider the *cycle* example again. Several interesting observations are obtained directly from their colors.

First of all, note that colors attached to the answers are orthogonal with one another. Any pair of them have different branching arcs at only one branching point. Let α and β be colors with an incorrect answer and correct answer, respectively. Comparing α and β, the selected branching arc at a branching point causes the difference of getting a correct answer and an incorrect answer.

For example, comparing the color of R3 and the colors with correct answers, O1, O2 and O3 are observed.

- O1: Selecting "2" at branching point "7" causes the difference. (from R3 and R1(or R2))

- O2:: Selecting "1" at branching point "3" causes the difference. (from R3 and R5)

- O3:: Selecting "1" at branching point "0" causes the difference. (from R3 and R8(or R9))

If we check other answers, there is no success world in which branching arc "2" is selected at branching point "7." Therefore, for O1, we consider that the observation is *admissible*. However, since R1 is a success case in which branching arc "1" is selected at branching point "0," and "1" is selected at "3," O2 and O3 are considered to be *not admissible*.

This procedure is summarized as follows. Let C_α and C_β be a set of colors attached to incorrect answers and correct answers, respectively. For a pair of $\alpha \in C_\alpha$ and $\beta \in C_\beta$, let

P be the branching point both α and β have, and let α have (P, S_α) as a primitive color. If there exists no element in C_β that has (P, S_α) as a primitive color, then the observation

Obs:: Selecting S_α at branching point P causes the difference.

is *admissible*. In this case, the selection S_α at P is a candidate for the cause of the bug.

If there exists an element in C_β that has (P, S_α) as a primitive color, then the above observation is *not admissible*. In this case, the selection S_α at P does not directly cause the bug. As further computation proceeds after passing branching point P, there exists another branching point at which the selection may be a candidate for the cause of the bug. Namely, P is too far from the bug.

For example, there are three admissible observations in this example.

- O1:: Selecting "2" at branching point "7" causes the difference. (from R3 and R1)

- O4:: Selecting "2" at branching point "2" causes the difference. (from R10 and R8)

- O5:: Selecting "2" at branching point "1" causes the difference. (from R6 and R5)

In the given computation forest, all the branching points "1," "2" and "7" correspond to the goal $(4, 1)$. Hence, reasoning on these observations and the forest brings up a new fact:

- F1:: Reduction between the goal $(4, 1)$ and the clause corresponding to branching arc "2," namely, the reduction of the leftmost body goal of clause C4 and the second definition clause of that predicate, causes a bug.

This is a procedure to find a set of nodes of branching points which are as close as possible to the nodes which correspond to the incorrect goal atoms in the computation

forest, and then to get the fact that narrows the bug's location. Tracing starts from the reduction indicated by the fact since computation before the point is guaranteed to be correct. However, the bug is not always located in the subtree whose root node corresponds to that goal. It may be located in another subtree, because reduction itself is correct, and the imported binding to the goal at a distance causes the bug.

4.2 Coordination with Other Methods

Several years have passed since algorithmic debugger was proposed as a declarative debugging method instead of conventional tracing [Shapiro 84]. Our approach is similar to an algorithmic debugger in the sense of constructing a computation forest(tree) and narrowing a bug's location. The algorithmic debugger of *ANDOR-II* would be the one traversing the forest, and several nodes or arcs are detected as buggy reductions, since a bug affects several worlds. This process seems quite complicated and the method of picking up a target world seems to be simpler and more suitable. It would be interesting to pick up a representative world and carry out an algorithmic debugging with the obtained knowledge by meta reasoning.

5 Concluding Remarks

In this paper, we have proposed a debugger for an AND/OR parallel logic programming language *ANDOR-II* . We proposed the method of selecting a representative world and peeling it from the multiple worlds. Debugging of the *ANDOR-II* computation on this single world is equivalent to the debugging of the computation of committed-choice languages. Furthermore, we showed that if we survey over the worlds, we can perform meta reasoning over the observations, and a more effective bug location algorithm can be realized by taking advantage of the results.

Debugging of AND/OR parallel languages has scarcely been studied. The debugger proposed here is under development, but it leaves lots of room for future works. The most urgent issue to consider is the adoption of some strategies for taking the designated color.

If we take some color, the bug is found easily while if we take another color, the bug is hardly found or takes a long time to find. Some strategies should be adopted for obtaining the designated color. Furthermore, formalization of meta reasoning over the worlds is required. An efficient algorithm to get a set of *admissible* observations and to obtain the fact that narrows bug's location from these observations and the computation forest need to be considered.

Acknowledgements

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, and Dr. R. Hasegawa, the Chief of Fifth Laboratory, for their advice and encouragement.

REFERENCES

[Bahgat and Gregory 89] Bahgat,R. and S.Gregory, "Pandora: Non-deterministic Parallel Logic Programming," Proc. of 6th International Conference on Logic Programming, pp.471-486,1989.

[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.

[Haridi et al. 88] Haridi,S. and P.Brand, "ANDORRA Prolog - An Integration of Prolog and Committed Choice Languages," Proc. of International Conference on Fifth Generation Computer Systems, pp.745-754, 1988.

[Huntbach 87] Huntbach,M.M., "Algorithmic PARLOG Debugging," Proc. of Symposium on Logic Programming, pp.288-297, 1987.

[Lloyd 87] Lloyd,J., "Declarative Error Diagnosis," New Generation Computing, pp.123-154, Vol.5, No.2, 1985.

[Lloyd and Takeuchi 86] Lloyd,J., "A Framework of Debugging GHC," ICOT TR-186, 1986.

[Lichtenstein and Shapiro 89] Lichtenstein,Y. and E.Shapiro, "Abstract Algorithmic

Debugging," Proc. of 4th Int. Conf. on Logic Programming, 1989.

[Naish 87] Naish,L., "Parallelizing NU-Prolog," Proc.of Logic Programming, pp.1546-1564, 1988.

[Shapiro 84] Shapiro,E.Y., "Algorithmic Program Debugging," The MIT Press, 1983.

[Takahashi et al. 90] Takahashi,K., A.Takeuchi and T.Yasui, "A Parallel Problem Solving Language ANDOR-II and Its Parallel Implementation," ICOT TR-558, 1990.

[Takeuchi 86] Takeuchi,A., "Algorithmic Debugging of GHC Programs and Its Implementation in GHC," ICOT TR-185, 1986.

[Takeuchi 90] Takeuchi,A., "Parallel Logic Programming," PhD. Thesis, The University of Tokyo, 1990.

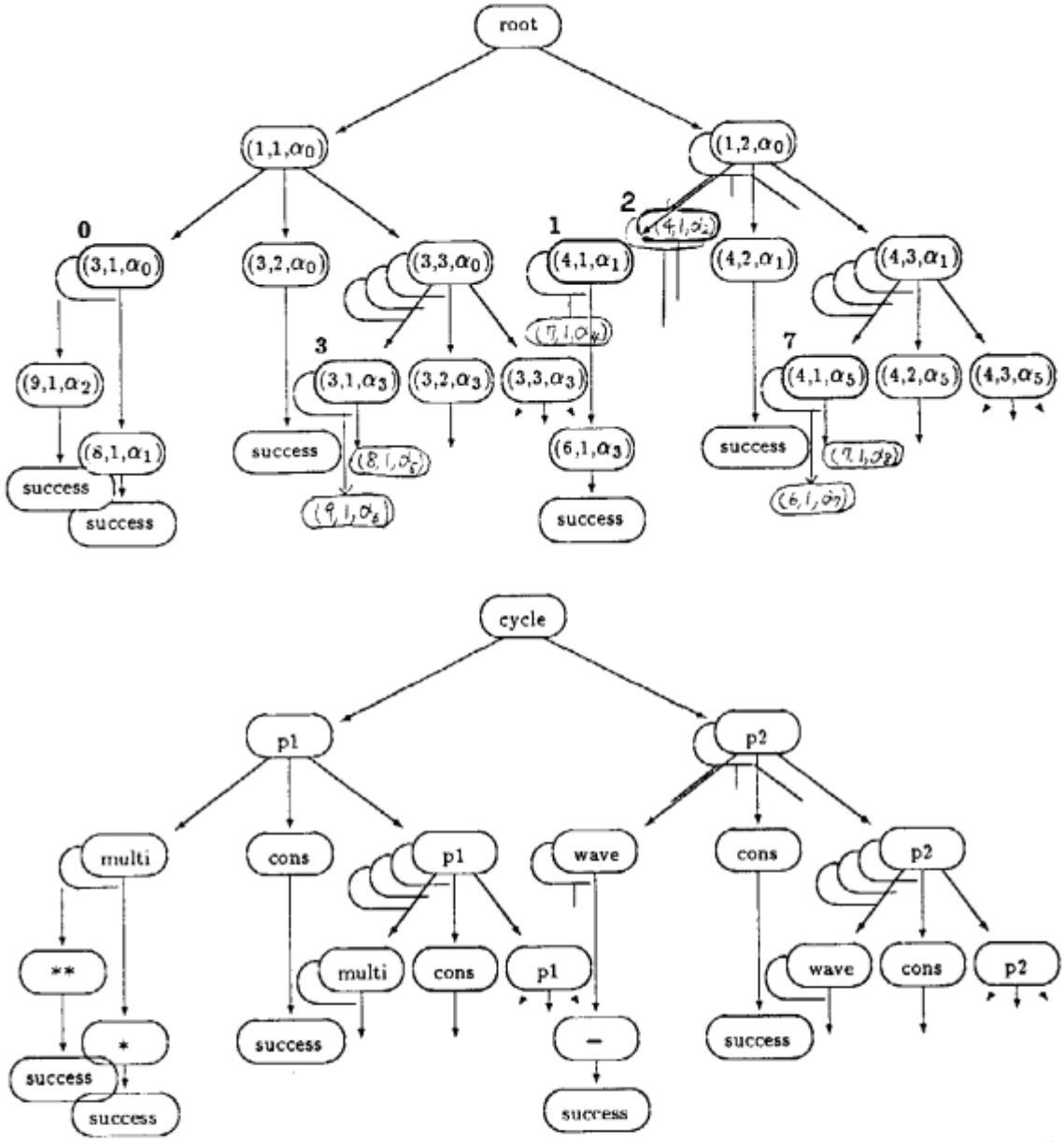
[Takeuchi et al. 88] Takeuchi,A., K.Takahashi and H.Shimizu, "A Parallel Problem Solving Language for Concurrent Systems," ICOT TR-418, 1988, also in Concepts and Characteristics of Knowledge-based Systems, M.Tokoro,Y.Anzai and A.Yonezawa(eds.), North-Holland, 1989.

[Tatemura and Tanaka 89] Tatemura,J. and H.Tanaka, "Debugger for Parallel Logic Programs: FLENG," Proc.of Logic Programming 89, pp.133-142, 1989.

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

[Ueno and Kanamori 90] Ueno,M. and T.Kanamori, "GHC Program Diagnosis Using Atom Behavior," Proc.of Logic Programming 90, Springer(to appear), also in ICOT TR-, 1990.

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



α_0 : "0000000000"
α_1 : "1000000000"
α_2 : "2000000000"
α_3 : "1100000000"
α_4 : "1200000000"
α_5 : "1101000000"
α_6 : "1102000000"
α_7 : "1101000100"
α_8 : "1101000200"
.....

multi-node

Figure 1: Compntation Forest of *cycle*(above) and Corresponding Goals(below)