TR-0836

Bug Detection Method Over AND/OR – Computation Tree

by
K. Takahashi (Mitsubishi)

March, 1993

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191 ~ 5

Institute for New Generation Computer Technology

Bug Detection Method over AND/OR-Computation Tree

Kazuko Takahashi
Central Research Laboratory
Mitsubishi Electric Corporation
8-1-1, Tsukaguchi-Honmachi,
Amagasaki, 661, JAPAN
(TEL) +81-6-497-7141
(FAX) +81-6-497-7289
(E-MAIL) takahashi@sys.crl.melco.co.jp

Abstract

This paper discusses a bug detection method for AND/OR-parallel logic programming languages by traversing the computation tree. We regard the computation as a set of worlds and an invocation of an OR-primitive as a branching point. First of all, we construct a computation tree and record a history of selections at each branching point. Comparing the history associated with a correct answer and the one with an incorrect answer, we detect the direct cause of the difference. We show that all the (manifest) bugs can be found by traversing branching points of the computation tree. Moreover, if a program satisfies some condition, bugs can be found only by inspecting the history associated with the answers. We also discuss about such a class. This method is applicable not only to locate a bug but also to find the point which causes the difference of solutions.

1 Introduction

Recently, design and implementation of logic programming languages which support both AND- and OR-parallelism attract many researchers [C87][HB88][Nai88] [TTS90][YA89]. In these languages, AND-parallelism means the parallel execution of conjunctive goals, and OR-parallelism means the parallel execution of all possibilities. These languages are designed to subsume Prolog and committed-choice languages, and to achieve high parallelism. According to the advancement of the

research on these AND/OR-parallel logic programming languages, there arises a big problem of debugging. Actually, so far almost no debugging tools has been proposed for these class of languages.

In [TT90], we have proposed a framework for debugging AND/OR-parallel logic programming languages. In that framework, the system stores the history of computation and applys an algorithmic debugging method on a world in which an erroneous answer is obtained. We also suggested a possibility of debugging by traversing the whole computation tree. In this paper, we show a new method of bug detection which is derived from the discussion in [TT90]. We call this method *tree-traversing method.*

Consider the following simple program.

```
p :- q.  or  p :- r.

q :- q1. or  q :- q2.

q1.   q2.   r.
```

Note that p and q have two possibilities on selecting a clause to be used. For the computation of the goal p, the computation tree shown in Figure 1 is constructed. Note that N_0 and N_1 are branching points, that denote multiple possibilities of selections.

Example 1.1 Assume that the following three solutions are obtained.

```
ans1 with the color { (1 at N0) (1 at N1) }
ans2 with the color { (1 at N0) (2 at N1) }
ans3 with the color { (2 at N0) }
```

Assume that **ans1** and **ans2** are the incorrect solutions, while ans3 is the correct one. In this case, selecting "1" and "2" at branching point N_0 causes the difference of getting either an incorrect solution or a correct solution. If a program contains

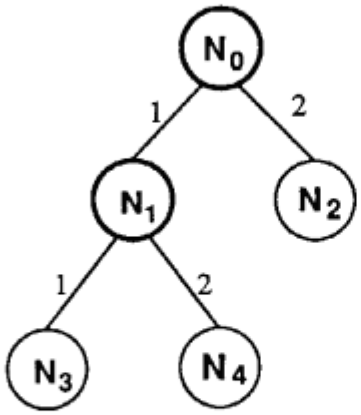

Figure 1: A Computation Tree for an Example

only one bug, then the bug exists in N_1, that is, the reduction between the goal `p` and the clause `p :- q` is incorrect.

In this case, the bug can be detected only by examining the solutions. However, if a program contains more than one bug, they cannot be detected only by doing so. If the above program contains more than one bug, they may exist both in N_3 and N_4. That is, the reduction between `q` and `q :- q1` is incorrect and the reduction between `q` and `q :- q1` is also incorrect. In general, it is necessary to traverse the computation tree.

If several bugs affects each other, the case becomes more complicated.

Example 1.2 In Example 1.1, assume that `ans1` is incorrect and the rests are correct. In this case, there are two possibilities: the one is that a bug exists in N_3, and the other is that bugs exist both in N_1 and N_4. The latter is the case in which more than one bug affect each other to generate the correct result.

These examples show that we have to check not only leaf nodes but also an intermediate node in the tree to detect bugs. However, for some class of bugs, they can be detected only by examining the leaf nodes. In this paper, we show the

algorithm of bug detection by traversing the whole tree, and discuss the class for which the algorithm can be simplified.

This paper is organized as follows. In Section 2, parallel logic programming language is described and the framework for bug detection is shown. In Section 3, the algorithm for bug detection is shown. In Section 4, the class for which the algorithm can be simplified is discussed. In Section 5, comparison with other related works are discussed. Finally, in Section 6, conclusion and future works are shown.

2 Framework

In AND/OR-parallel computation, the whole computation can be viewed as a disjunction of multiple worlds, where each world is doing AND-parallel computation and a world may split into several worlds according to an invocation of an OR-primitive. A set of worlds forms a tree when viewing a splitting relation as an arc. The result of AND/OR-parallel computation is a collection of results from each leaf world. Solutions on multiple worlds are presented together with their associated world identifiers which extracts outlines of their histories of the computation.

Each world is associated with a *color*. Color is defined as follows.

Definition 2.1 *A primitive color* is a pair of symbols (P, A) where P and A 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 color* is defined to be a set of primitive colors, in which no element is orthogonal with each other.

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.

For example, consider the following program of *cycle* written in ANDOR-II, an AND/OR-parallel logic programming lanaguge[TTS90].

Example 2.1 `% cycle`

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

```
p1([stop],Y) :- true | Y=[ ].
p1([X|X1],Y) :- X\=stop | multi(X,A), Y=[A|Y1], p1(X1,Y1).

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

:- or_predicate wave/2.
wave(X,Y) :- Y:=X-1.
wave(X,Y) :- Y:=X+1.

:- or_predicate multi/2.
multi(X,Y) :- Y:=X*X.
multi(X,Y) :- Y:=X*X*X.
```

When a goal $cycle(Y)$ is called, the computation proceeds as follows: Goals $p1([2|X], Y)$ and $p2(Y, X)$ are executed in parallel in the initial world with color α_0. When the goal $p1([2|X], Y)$ is called, the first clause of `p1` is selected. As for $multi$, there are two possibilities of clause selection, and depending on each case, two new worlds with colors α_1 and α_2 are created. Goals $p2([4|Y1], X1)$ and $p2([8|Y1], X1)$ are invoked in the world with the colors α_1 and α_2, respectively. In this way, the computation proceeds.

Suppose that a bug manifests in some world while in another world computation terminates successfully. In such a case, we can derive some useful information by comparing their colors, and hence can narrow the bug's location. Starting from the point, bugs are searched by traversing the whole tree.

Thus, our debugging framework is described as follows:

Stage 1: A buggy program is executed as usual and the history of the whole computation is gathered. Solutions on multiple worlds are presented together with their associated world identifiers, and a user classify all the solutions into cor-

rect ones and incorrect ones.

Stage 2: Compare the colors associated to the solutions, and detect the branchin point which causes the difference of getting either a correct solution or an incorrect solution.

Stage 3: Starting from such a branching point, traverse the tree and examine whether the computation in that world is correct or not until an incorrect computation is found.

For some class of bugs, Stage 3 can be skipped, and bugs can be detected only by examining the finally obtained solutions. We also discuss such a class.

3 Bug Detection Algorithm

First of all, we make the following assumption.

Assumption(single OR-invocation)

We assume that at most one OR-goal is invoked in a world.

This assumption assures that the primitive colors contained in a color are totally ordered.

3.1 Construction of a Reduced AND/OR-tree

Given a buggy program, in the first stage, we execute the program to record the history of the computation. The history is extracted as *reduced AND/OR-tree*. In the reduced AND/OR-tree, a node corresponds to the set of goals executed in a single world, and an arc corresponds to a branching arc. Different nodes correspond to the different worlds . Note that each node includes only one branching point. The world generated as a result of selecting a branching arc S is said to be *the world generated by* S, and the corresponding node is said to be *the node generated by* S. Reduced AND/OR-tree is constructed as follows.

root node Create the root node.

general node When an OR-reduction is carried out in the world W and m new worlds are created, then m new nodes $N_1, \ldots, N_m$ are created, and edges are drawn from the node corresponding to W to $N_1, \ldots, N_m$, respectively.

Intuitively, the goals contained in the same world are in conjunction, and some of them are suspended waiting for the OR-reduction. Assume that an OR-reduction between a goal G and clauses $C_1, \ldots, C_m$ is carried out in the world W, and that worlds $W_1, \ldots, W_m$ are created. Let G_i be a goal contained in W_i. Then, G_i is a goal instance of G obtained by applying the substitutions in the world W_i, where G is either a suspended goal in the world W, or an invoked goal in the world W_i before the next OR-reduction is carried out.

In case of *suspension* or *failure* has occurred, the result is propagated as far as possible.

In Figure 2, we show a part of the reduced AND/OR-tree for the example of *cycle*.

3.2 Detection of Fatal Branching Points

The second stage of this method is the detection of fatal branching points.

Definition 3.1 (fatal selection, fatal branching point, buggy tree)
Let C_s and C_f be sets of colors attached to correct solutions and incorrect solutions, respectively. Also let $(P, S_\alpha) \in \alpha, \alpha \in C_f$ and $(P, S_\beta) \in \beta, \beta \in C_s$. If there exists no element in C_s that has (P, S_α) as a primitive color and there is at least one (P, S_β), we call P *a fatal branching point*, and S_α *a fatal selection*. For a fatal branching point, a subtree generated by a fatal selection is said to be the *buggy tree* for this fatal branching point.

Similar to the representative world method, solutions on multiple worlds are presented together with their associated colors. A user classify all the solutions into correct ones and incorrect ones. Then, fatal branching points are detected according to the following procedure.

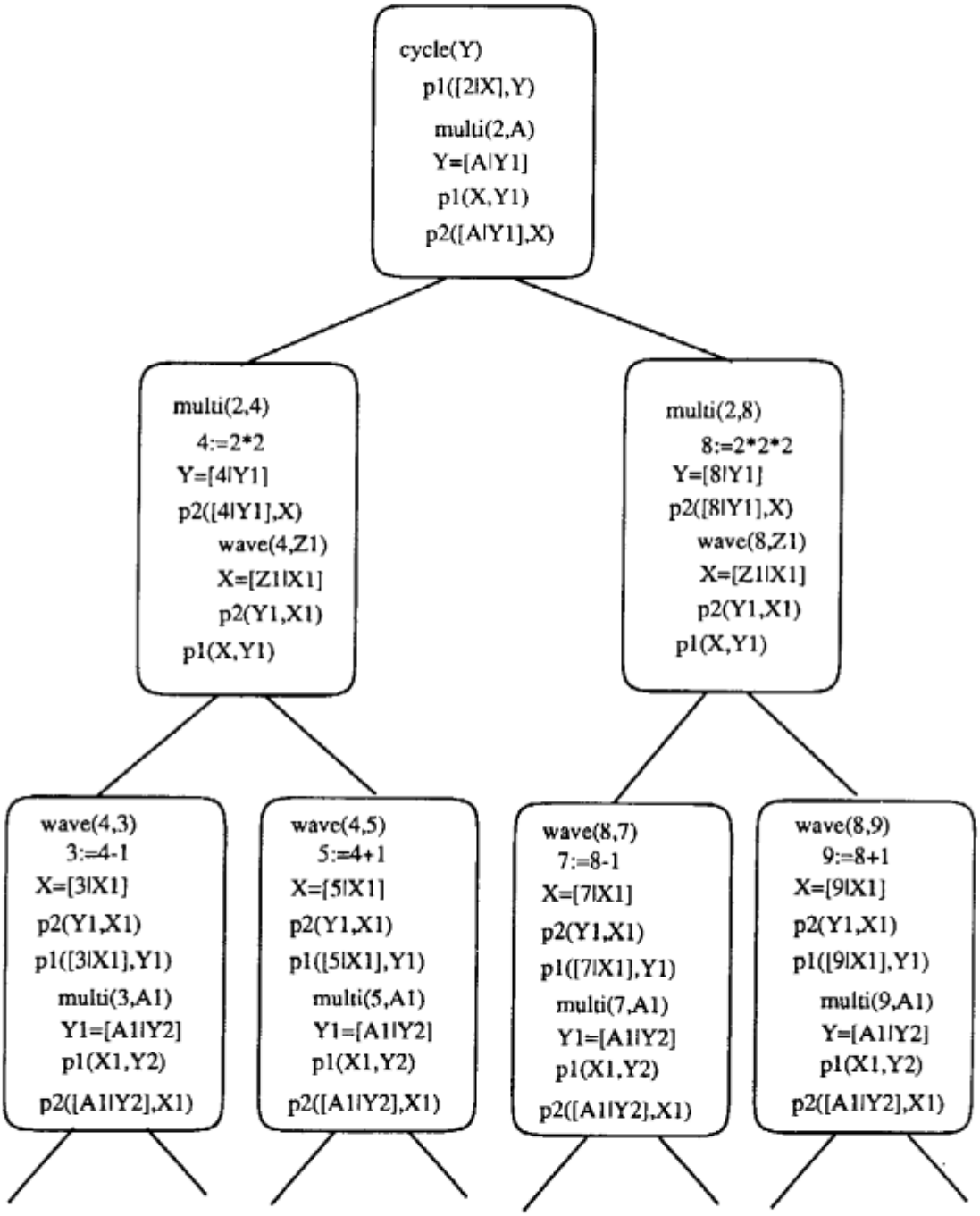

Figure 2: Reduced AND/OR-tree for *cycle*

Let C_s and C_f be the set of colors attached to correct solutions and incorrect solutions, respectively. Let C_f be $\{\alpha_1, \ldots, \alpha_m\}$. For an arbitrary element $\{(P_1, S_1), \ldots, (P_n, S_n)\}$ of C_f, do the following procedure.

Detection of Fatal Branching Points and Fatal Selections

Set $i = 1$.

For a pair (P_i, S_i), do the following procedure.

Procedure $proc_1(i)$

Set $j = 1$.

For α_j, do the following procedure.

Procedure $proc_2(j)$

If $j > m$, then terminate with an answer

"P_i is a fatal branching point and S_i is a fatal selection."

otherwise

If $(P_i, S_i) \in \alpha_j$, then do $proc_1(i + 1)$.

If not $(P_i, S_i) \in \alpha_j$, then do $proc_2(j + 1)$.

3.3 Traverse of the Tree

The third stage of our debugging is traversing the reduced AND/OR-tree. In this stage, starting from the fatal branching points, the whole tree is traversed. The system is taught interactively by a user whether the goal instances in a world are correct or incorrect. As usual, if a goal instance is an intended one, then a user judges it to be *correct*, otherwise, *incorrect*. For a node N, if all the goal instances in the world corresponding to N are correct, then the node is said to be *correct* and denoted by $\mathcal{T}(N)$; otherwise, it is said to be *incorrect* and denoted by $\mathcal{F}(N)$.

For a fatal branching point, let N be the node generated by the fatal selection at that fatal branching point. Also let $\overline{N^1}, \ldots, \overline{N^m}$ be the nodes generated by the branching arcs other than the fatal selection. Do the following procedure.

Bug Location

(1) Take an arbitrary path from N to a leaf node,

and let the path be $N_0(=N), N_1, \ldots, N_k$.

Set $i = 0$.

Procedure(Glancing the buggy tree)

If $i = k$, then do (3).

Otherwise.

If $\mathcal{T}(N_i)$ holds,

then increment i by 1 and repeat this procedure.

If $\mathcal{F}(N_i)$ holds, then do (2).

(2) Set $M = N$.

Do the following procedure for M.

Procedure (Search downwards)

If $\mathcal{F}(M)$, then terminate with an answer "A bug exists in M."

If $\mathcal{T}(M)$,

then for the nodes $M^1, \ldots, M^m$ whose parent is M,

do this procedure for each $M^j (j = 1, \ldots, m)$.

(3) Do the following two procedures.

Procedure(Search upwards)

Let a path from the root node to the node N be $N_0, \ldots, N_k(=N)$.

Set $i = 1$.

If $\mathcal{F}(N_{k-i})$,

then terminate with an answer "A bug exists in N_{k-i}."

Otherwise, increment i by 1 and repeat this procedure.

Procedure(Search outwards)

For all $\overline{N^j} (j = 1, \ldots, m)$, do the followings.

Set $M = \overline{N^j}$

Do the following procedure for M.

If $\mathcal{F}(M)$,

then terminate with an answer "A bug exists in M."

If $\mathcal{T}(M)$,

then for the nodes $M^1, \ldots, M^{m'}$ whose parent is M,

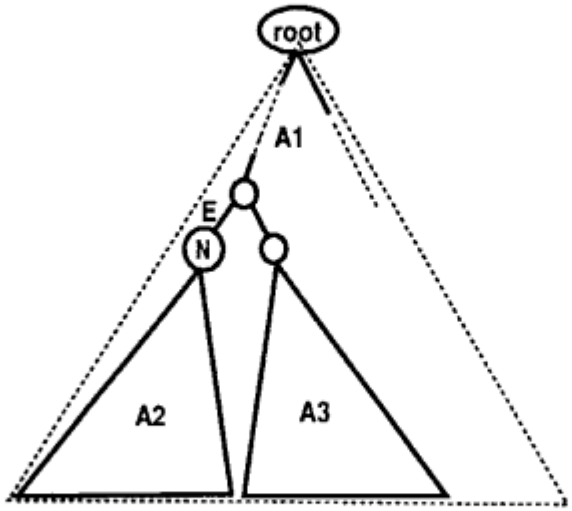

Figure 3: Partition of the Tree

do this procedures for each $M^{j'}(j' = 1, \ldots, m')$.

Firstly, in (1), it is checked whether the buggy tree contains a bug or not. Then the tree is divided into three areas to be searched depending on the case. The first area A_1 is the part from the root node to the branching point N. The second area A_2 is the part from N to the leaf nodes (i. e. buggy tree). And the third area A_3 is the part from $\overline{N^1}, \ldots, \overline{N^k}$ to their leaf nodes (Figure 3).

In case of (2), bugs exist in the area A_2, and in case of (3), bugs exist both A_1 and A_3. In case of (2), it is not determined whether A_1 contains a bug or not. Therefore, this algorithm is sound but incomplete.

Note that in the reduced AND/OR-tree, the result of computation (substitution) in a world does not effect on a world corresponding to the parent node. It is because the synchonization is perfomed at an invocation of OR-primitive, and substitution is realized as far as possible in a world. Thus, if $\mathcal{F}(N)$ holds, then the bug exists on N. Conversely, if a bug exists in a world N, then $\mathcal{F}(N)$ holds.

4 The Class for Which Algorithm Can Be Simplified

Without the assumption of single OR-invocation, the third stage of the framework for a bug detection is complicated, since there are several 'next' branching points.

However, some class of bugs can be found only by checking the colors attached to the final solutions and need not observe the intermediate nodes. Moreover, we can ignore the assumption. In this section, we discuss the conditions to be satisfied.

Cond(1) For each path from the root node to a leaf node, there is at most one node N which satisfies $\mathcal{F}(N)$.

Cond(2) For each branching point, there is at most one node N which satisfies $\mathcal{F}(N)$ in the nodes whose parent is the branching point.

Cond(1) shows that bugs are independent from each other, and Cond(2) assures that if a bug exists in a world, then there are no bugs in the other worlds which share the parent node.

Theorem 4.1 Assume that both of Cond(1) and Cond(2) are satisfied. If there exists a bug, it exists in the world generated by the fatal selection.

Lemma 4.1 Assume that both of Cond(1) and Cond(2) are satisfied. For a node N, let M be a parent node and E be an edge which connects M and N. Let P and S_α be the branching point and the branching arc corresponding to M and E, respectively. If $\mathcal{F}(N)$, then any solution associated with the color α that contains (P, S_α) is incorrect.

Proof. It is trivial from the first condition. ∎

Lemma 4.2 Assume that both of Cond(1) and Cond(2) are satisfied. For a node N, let M be a parent node and $\overline{E}$ be an edge which connects M and N. Let P and S_β be the branching point and the branching arc corresponding to M and $\overline{E}$, respectively. If $\mathcal{T}(N)$, then there exists a solution associated with the color β that contains (P, S_β) is correct.

Proof.

It is proved by the induction on the height h of the tree whose root node is N. If $h = 1$, then from Cond(2), there is at most one node which satisfies $\mathcal{F}(N)$ in the nodes that shares the parent node. Therefore, for all the nodes $N^1, \ldots, N^m$ other than N, $\mathcal{T}(N^j)$ $(j = 1, \ldots, m)$ hold. Consider the tree of height h whose root node is N. Let $N^1, \ldots, N^k$ be the nodes whose parent is N. Then from Cond(2), there exists N^i such that $\mathcal{T}(N^i)$ is satisfied. Let E be an edge from N to N^i. Let P and S_β be the branching point and the branching arc bcorresponding to N and E, respectively. By the induction hypothesis, there exists a correct solution associated with the color β^i that contains (P^i, S^i), where P^i and S^i are the branching point and the branching arc corresponding to N^i and E^i, respectively. Hence, from the first condition, there exists a correct solution associated with the color β that contains both (P, S_β) and (P^i, S^i). ∎

Proof of the Theorem 4.1

If there exists a bug, then there exists a node N such that $\mathcal{F}(N)$ holds. Let M be a parent node of N and $\overline{N^1}, \ldots, \overline{N^m}$ be the nodes which shares the parent node M. Also let $E, \overline{E^1}, \ldots, \overline{E^m}$ be the edges from M to $N, \overline{N^1}, \ldots, \overline{N^m}$, respectively. Let P be the branching point corresponding to M, and $S_\alpha, S_{\beta^1}, \ldots, S_{\beta^m}$ be the branching arcs corresponding to $E, \overline{E^1}, \ldots, \overline{E^m}$, respectively. From lemma 4.1, any solution associated with the color α that contains (P, S_α) is incorrect. And from lemma 4.2, there exists an solution associated with the color β that contains (P, S_β) is correct, where $S_\beta \in \{S_{\beta^1}, \ldots, S_{\beta^m}\}$. Therefore, P is a fatal branching point and S_α is a fatal selection. Thus, the bug exists in the world corresponding to N. ∎

In this case, the world generated by the fatal selection contains a bug, and bugs can be found only by checking the colors attached to the solutions.

5 Comparison with Other Works

5.1 Comparison with Representative World Method

We compare the tree-traversing method discussed in this paper with representative world method, which is discussed in [TT90]. The framework of representative world method consists of three stages: (1) A buggy program is executed, and the history of the whole computation is gathered, (2) The computation of a representative world in which the bug manifests is reconstructed using the above history, and (3) Algorithmic debugging is applied to the computation in this world. The underlying idea is the extraction of an erroneous AND-tree from an AND/OR-tree and application of well established debugging algorithms to this AND-tree.

Representative world method is naive and widely applicable. If a history of branching points which the computation have passed can be stored, the method can be applicable to most of AND/OR-parallel logic programming languages, and the mechanism is not hard to implement .

Tree-traversing method is more sophisticated but less general. Without the assumption of single OR-invocation, reduced AND/OR-tree becomes complicated. Further discussion is required for handling general case.

The assumption of single OR-invocation corresponds to the lazy fork of OR-predicates. Lazy fork allows at most one invocation of OR-process, while eager fork allows the simultaneous invocation of multiple OR-processes. Therefore, this method is applicable to the languages such as Andorra group [HB88] and Pandora [BG89], which employ lazy fork of OR-predicates. As for languages such as ANDOR-II[TTS90], that employ eager fork of OR-predicates, the method is applicable to the restricted class.

Another difference is the size of the computation tree (forest). In representative world method, nodes of a computation forest stores only identifiers of the goals, clauses, and colors. In tree-traversing method, the nodes stores the goal instances which reflect the substitution.

5.2 Comparison with Algorithmic Debugger

Algorithmic debugging is a declarative debugging method proposed by Shapiro[Sha83]. Different from the classical tracer-type debugger, it enables the programmer to locate a bug by answering the query from the system whether or not the goal instance used in the computation is the intended one. Shapiro applied it successfully to Prolog, and following his work, several algorithmic debuggers have been developed for committed-choice languages [Hun87][LS88][LT86][Tak87] [TT91][UKar].

Our approach is similar to them in that bug detection proceeds by checking the goal instances in the computation tree is correct or not. However, in our method, each node in the tree denotes a set of goal instances, while it denotes a single goal instance in the case of algorithmic debuggers. Moreover, in our method, substitution is the independent operation within a world, and does no effects on the parent node.

Furthermore, in algorithmic debugger, bug detection is incrementally proceeds, that is, if a bug is detected and corrected, then re-construct a computation tree.

However, in the computation of AND/OR-parallel logic programming languages, the cost of construction of the tree is so high. It is desirable that once a tree is constructed, bug detection proceeds on the same computation tree, in order to detect as many bugs as possible.

5.3 Comparison with ATMS

In the second stage of the tree-traversing method, we compare colors attached to solutions on multiple worlds.

In the point of propagation of information among possible worlds, there is a similarity with assumption-based truth maintenance system(ATMS)[deK86]. In ATMS, starting from the given fact, forward reasoning is performed, possible worlds are created depending on each assumption. If some world is found to be inconsistent, the world with the inconsistent set of propositions is pruned, and finally (maximal) consistent set of propositions is found to satisfy the goal. ATMS is usually solved in a bottom-up manner, while the computation tree described in this paper

is constructed in a top-down manner.

6 Concluding Remarks

We have proposed a bug detection method for AND/OR-parallel logic programming languages by traversing the computation tree.

There are several advantages in this method.

1. Search spaces for a bug can be reduced.
2. Multiple bugs can be found at the same time.
3. Even if a correct solution is obtained, there may be a bug on this computation. Such an unexpected bugs can be found.

Furthermore, some class of bugs can be found, only by inspecting the histories with the resulting solutions. It is much easier than to examine intermediate goals.

The algorithm described here is applicable not only to locate a bug but also to find the cause of the differences between worlds.

For future works, we try to improve the algorithm more efficient. To omit the duplicated observation, the result of observation should be stored in a certain table. Furthermore, in parallel environment, search for several fatal branching points can be executed in parallel, which enables not only fast detection of a bug but also simultaneous detection of several bugs.

References

[BG89] R. Bahgat and S. Gregory. Pandora: Non-Deterministic Parallel Logic Programming. In *Proceedings of 6th International Conference on Logic Programming*, pp. 471–486, 1989.

[C87] K. L. Clark and S. Gregory 84. PARLOG and Prolog United. In *Proceedings of 4th International Conference on Logic Programming*, pp. 927–961, 1987.

[deK86] J. deKleer. An Assumption-Based TMS. *Artificial Intelligence*, 1986.

[HB88] S. Haridi and P. Brand. ANDORRA Prolog - An Integration of Prolog and Committed Choice Languages. In *Proceedings of International Conference on Fifth Generation Computer Systems*, pp. 745–754, 1988.

[Hun87] M. M. Huntbach. Algorithmic PARLOG Debugging. In *Proceedings of Symposium on Logic Programming*, 1987.

[LS88] Y. Lichtenstein and E. Shapiro. Abstract Algorithmic Debugging. In *Proceedings of 5th International Conference on Logic Programming*, pp. 512–531, 1988.

[LT86] J. Lloyd and A. Takeuchi. A Framework of Debugging GHC. Technical Report TR-186, ICOT, 1986.

[Nai88] L. Naish. Parallelizing NU-Prolog. In *Proceedings of Logic Programming*, pp. 1546–1564, 1988.

[Sha83] E. Shapiro. *Algorithmic Program Debugging.* The MIT Press, 1983.

[Tak87] A. Takeuchi. *Algorithmic Debugging of GHC Programs and Its Implementation in GHC,* In *Concurrent Prolog: Collected Papers*, volume 2, The MIT Press, 1987.

[TT90] K. Takahashi and A. Takeuchi. A Debugger for AND- and OR-Parallel Logic Programming Language ANDOR-II. In *ICOT TR-608*, 1990.

[TT91] J. Tatemura and H. Tanaka. Debugger for a Parallel Logic Programming Language Fleng. In *LNAI-485, Logic Programming 89*, pp. 87–96. Springer-Verlag, 1991.

[TTS90] A. Takeuchi, K. Takahashi, and H. Shimizu. A Parallel Problem Solving Language for Concurrent Systems. In M. Tokoro, Y. Anzai, and A.Yonezawa, editors, *Concepts and Characteristics of Knowledge-Based*

Systems, pp. 267–296. North-Holland, 1990. also appearing as ICOT TR-418,1988.

[UKar] M. Ueno and T. Kanamori. GHC Program Diagnosis Using Atom Behavior. In *Proceedings of Logic Programming 90*. Springer-Verlag, to appear. also appearing in ICOT TR-.

[YA89] R. Yang and H. Aiso. P-Prolog: A Parallel Logic Language Based on Exclusive Relation. *New generation Computing*, Vol. 5, No. 1, pp. 79–95, 1989.