ICOT Technical Report: TR-229

TR-229

A Description Language with AND/OR Parallelism for Concurrent Systems and Its Stream-Based Realization

by

A. TAKEUCHI, K. TAKAHASHI
and H. SHIMIZU
(Mitsubishi Electric Corp.)

February, 1987

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

AND/OR並列性を備えた並列動作系記述言語とそのストリームによる実現

A Description Language with AND/OR Parallelism for Concurrent Systems and Its Stream-Based Realization

竹内 彰一　　高橋 和子　　清水 弘之

Akikazu TAKEUCHI　Kazuko TAKAHASHI　Hiroyuki SHIMIZU

三菱電機(株)　中央研究所

Central Research Laboratory, Mitsubishi Electric Corporation

Abstract　This paper describes the description language *ANDOR-II* for concurrent system and its realization by stream. Our purpose is an amalgamation of committed choice language with OR-parallelism, and giving a full AND/OR process model of logic programs. *ANDOR-II* is designed for modeling and simulating the nondeterministic system. Our compiler translates the program written in *ANDOR-II* into the one in FGHC so that both AND- and OR-parallelism are realized. In the transformed program, OR-parallel processes are executed in the independent worlds from one another. Their solutions are sent as a data to other processes in a stream form, and each solution is associated with its own color. Fault diagnosis of a simple circuit is shown as an example. Although the overhead of the color check is burdensome now, we can make it smaller by optimization.

1. INTRODUCTION

Recently, a lot of researches have been done both on parallel logic programming and on parallel problem solving. Many parallel logic programming languages such as CP[Shapiro 83], PARLOG[Clark and Gregory 84] and FGHC[Ueda 85] have been designed and developed. From the viewpoint of parallel problem solving, description and reasoning about concurrent systems is one of interesting subjects.

Generally speaking, the system with nondeterministic behaviors has many possibilities depending on the action. For example, if a circuit has a faulty component which behaves indefinitely, then the behavior of the circuit has many possibilities. And if the circuit has some faulty components, the possibility of the behavior of the circuit is productively increased. Therefore, when we deal with such a nondeterministic system, we have to search all the possible worlds. We usually want to perform multi-simulation or all solution search. Several researches are undertaken as a main area of qualitative reasoning. [deKleer and Brown 83][Kuipers 86]

On the treatment of a lot of worlds simultaneously, the realization of OR-parallelism is needed as well as AND-parallelism. On the other hand, committed choice languages such as FGHC do not realize OR-parallelism. That is, every conjunctive goals are executed in parallel, but if a clause is once selected, alternative resolutions are abolished. It follows that it is difficult to express all solution search problem in FGHC. Currently some interesting researches are done on this topics. Ueda has proposed the transformation method from exhaustive search program in Prolog into deterministic FGHC/Prolog program by using continuation.[Ueda 86a][Ueda 86b] Tamaki presented alternative method based on stream execution model.[Tamaki 86] It transforms the program written in the language with AND/OR parallelism into the one in committed choice language. However, their method is not sufficient enough for the treatment of nondeterministic systems. Therefore, it is necessary to present a method in which behaviors of such systems can be described naturally and several reasoning can be done.

In this paper, we propose a language *ANDOR-II* for modeling and simulating such systems and present the compilation method from the *ANDOR-II* program to the FGHC program. It is an amalgamation of committed choice language with OR-parallelism, and give a full AND/OR parallel execution model of logic programs.[Takeuchi 84][Conery 85]

For example, consider the following program of simple arithmetic in Prolog.

```
compute(X,Z) :- pickup(X,Y),
    double(Y,DY), triple(Y,TY), add(DY,TY,Z).

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

double(X,Y) :- Y:=X*X.

triple(X,Y) :- Y:=X*X*X.

add(X,Y,Z) :-  Z:=X+Y.
```

When a list X is given, the definition *compute*(X, Z) picks up an arbitrary element Y from the list X and computes $Y^2 + Y^3$. But there are several solutions for this computation because of the nondeterminacy of *pickup*. Although the primitive '*setof*' or '*bagof*' provides all solutions, isn't there any other ways to process several cases in parallel and generates answers at the same time ? *ANDOR-II* gives one solution. See the following program in *ANDOR-II* .

```
:- mode compute(+,-), pickup(+,-),
    double(+,-), triple(+,-), add(+,+,-).

:- and_relation compute/2.
compute(X,Z) :- true | pickup(X,Y),
    double(Y,DY), triple(Y,TY), add(DY,TY,Z).

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

:- and_relation double/2.
double(X,Y) :- true | Y:=X*X.

:- and_relation triple/2.
triple(X,Y) :- true | Y:=X*X*X.

:- and_relation add/3.
add(X,Y,Z) :- true | Z:=X+Y.
```

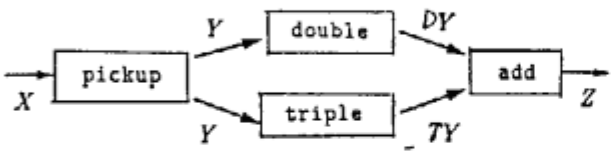

Fig 1. Data Flow Graph of Compute

In the program, the definition of the mode declaration is based on that in Prolog (see section 2, in detail.) Intuitively, OR-relation clause corresponds to Prolog clause and AND-relation clause corresponds to FGHC clause. 'OR-relation' indicates that the predicate *pickup* is nondeterministic, and the world branches depending on the selected clause. The worlds are independent from one another. This program is transformed into the FGHC program through the compiler. Assume that we call the goal *compute*$([1,2,3], A)$. In this case, the call of *pickup*$([1,2,3], Y)$ causes the case split, since either clause can be selected, three worlds are created depending on whether 1, 2, or 3 is picked up. Every solution is associated with its own color as an identifier. All the solutions are folded in a stream form and propagated to the conjunctive goals *double* and *triple*. They are processed in parallel, producing the streams as solutions $\{v(1,c1), v(4,c2), v(9,c3)\}$ and $\{v(1,c1), v(8,c2), v(27,c3)\}$, respectively, And *add* is invoked by receiving these values. However, if the values are associated with completely different color, then they cannot be computable. As the order of the elements in a stream is nondeterministic, we find a computable pair by checking these colors. We can add the values 1 and 1, since they are associated with the same color, We can also add the values 4 and 8, but we cannot add 1 and 4, since they are associated with completely different colors. Finally we get a set of the solutions [2,12,36].

In this case, the process which causes the world split is only *pickup*. However, if there are many such processes, then we have to treat all the split world and check the computability of data. Therefore, we need the mechanism which multi-simulation can be treated in a simple way. *ANDOR-II* supports such function.

This paper is organized as follows. We give a specification and computation mode of *ANDOR-II* in section 2, and describe the compiler in section 3. In section 4, we apply the method to fault diagnosis of a simple circuit, and in section 5, we compare the method with other works and also discuss the problem to be solved.

2. ANDOR-II Language

2.1. Syntax

Firstly, we describe the syntax of *ANDOR-II* .

```
Sentence            ::= Mode-Declaration |
                        Relation-Declaration |
                        Clause
Mode-Declaration    ::= (:- mode PModes)
PModes              ::= (PModes, PModes) | PMode
PMode               ::= Functor(Modes)
Modes               ::= (Modes,Modes) | Mode
Mode                ::= '+' | '-'
Relation-Declaration  ::=
   (:- and_relation Functor/Arity) |
    (:- or_relation Functor/Arity)
Clause          ::= (Head :- Goals '|' Goals) |
                    (Head :- Goals)
```

As for *Head*, *Goal*, *Arity* and *Functor* are defined similarly with the definitions in Prolog. As for *Functor*, we should not use the capital letter immediately after '_'(underbar).

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

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

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

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

is called an *AND-clause*.

In an AND-clause, H is called a *head goal*, G_1,...,G_n are called *guard goals*, and B_1,..., B_m are called *body goals*. The part before the commit operator '|' is called *passive part* of the clause and the part after the operator, *active part*. We extend this definition to OR-clauses, regarding it as a clause without guard goals.

Definition(AND-predicate,OR-predicate)

If a predicate has an AND-relation declaration, then it is called an *AND-predicate*, and if a predicate has an OR-relation declaration, then it is called an *OR-predicate*.

An *ANDOR-II* program is a finite set of sentences which satisfies the following conditions:

(1) Mode-Declaration is put at the top of the program.
(2) Relation-Declaration of a predicate is put before the clauses defining the predicate.
(3) Each predicate has a corresponding mode declaration and relation declaration.
(4) Only the built-in predicates are allowed as guard goals.
(5) An AND-predicate is defined only by AND-clauses, and an OR-predicate is defined only by OR-clauses.

Note that a clause can contain as the body goals both of AND-predicate and OR-predicate.

Since an automatic mode analysis is not performed in the compilation, mode should be declared by the user.

Besides, we impose some restrictions on the program which we deal with.

[Restrictions]

(1) *decidable mode*

For every predicate, the mode of every argument can be decidable. That is, the input argument must be ground when the predicate is called. And only variable is permitted as an output argument of a head goal.

(2) *prohibition of multiple writers*

More than two body goals cannot share a variable in the output mode in the same clause.

(1) is introduced to clarify the direction of data flow. It requires that all variables should be bound at the end of the computation. and prevents the use of partially instantiated data structure such as d-list. (2) is introduced to decrease the overhead in the execution of the transformed program.

Example.

```
:- mode permute(+,-), delete(+,-,-).

:- and_relation permute/2.
permute([],Y) :- true | Y=[].
permute(X,Y) :- true |
    delete(X,E,R),
    permute(R,Y1), Y=[E|Y1].

:- or_relation delete/3.
delete([X|X1],E,R) :-
    E=X, R=X1.
delete([X|X1],E,R) :-
    delete(X1,E,R1), R=[X|R1].
```

2.2. Computation Model

Next, we consider the computation model of *ANDOR-II* program. An AND-clause, is executed similarly in FGHC. Namely, it is suspended until the guard goals succeed on the condition that the variables in the caller are not bound to terms other than variables. And once a clause is committed, possibilities of trying the other clauses are abandoned. However, as for an OR-clause, it searches all solutions in parallel after the commitment of a clause. We realize the execution by stream model.

If an AND-predicate is called, then head unifications of the AND-clauses are tried, and if a clause is committed, then its body goals are invoked in parallel.(AND-parallelism) On the other hand, if an OR-predicate is called, all the clauses defining the predicate are committed if the head unification succeeds. As a result, the execution world branches.(OR-parallelism) Worlds are independent from one another.

Here, we introduce a basic concept *color*. *Color* is an identifier of these worlds. A variable is bound to a stream of solutions, each of which is associated with its own color. We do not know which data arrives first, or in which world deadlock appeared. And if there are several streams, we have to pick up the data belonging to the same world from every stream.

For example, pick up the permutation program in the above example and examine how the operation proceeds. Assume that we want to compute *permute*($[1,2], P$).

(0) A goal *permute*($[1,2], P$) is called.
(1) The second clause of *permute* is invoked.
(2) The goals *delete*($[1,2], E, R$), *permute*($R, Y1$) and *cons*($E, Y1, Y$) are invoked.
(3) As *delete* is an OR-predicate, the execution world of the goal *delete*($[1,2], E, R$) branches.
  (3a) In the first world, the answer E=1, R=[2] associated with a color [c1] is returned.
  (3b-1) In the second world, goals *delete*($[2], E, R1$) and *cons*($1, R1, R$) are invoked with a color [c2].
  (3b-2) The execution goal of the goal *delete*($[2], E, R1$) branches again.
  (3b-3) In the first world the answer E=2, R=[ ] is returned associated with a color [c2,c3] And in the second one the goals *delete*($[\,], E, R1$) and *cons*($2, R1, R$) are invoked with the color [c2,c4]. In this case, the call of *delete*($[\,], E, R1$) fails, since there are no unifiable head.

(3b-4) Only the answer from the first world is sent to the goal $cons(1, R1, R)$.

(3b-5) $Cons(1, [\,], R)$ returns the value R=[1] associated with the color [c2,c3].

(4) The output stream E and R is composed, and the goal $delete([1,2], E, R)$ returns the streams E={ v(1,[c1]),v(2,[c2,c3]) }, R={ v([2],[c1]),v([1],[c2,c3]) }, which are sent to $permute(R, Y1)$.

(5) The stream R is decomposed into the worlds.

(6) The goal $permute([2], Y)$ is called in the world with a color [c1].

(6a-1) $Permute([2], Y)$ returns the answer $Y = [2]$ with a color [c1].

(6a-2) The goals $delete([2], E, R)$, $permute(R, Y1)$ and $cons(E, Y1, Y)$ are invoked.

(6a-3) $Delete([2], E, R)$ returns the answer $E = 2, R = [\,]$.

(6a-4) $Permute([\,], Y1)$ returns the answer $Y1 = [\,]$.

(6a-5) $Cons(2, [\,], Y)$ returns the answer $Y = [2]$.

(6a-6) Finally, $permute([2], Y)$ returns the answer $Y = [2]$ with the color [c1].

(6b) By the similar discussion, $permute([1], Y)$ is called and returns the answer $Y = [1]$ with a color [c2,c3].

(7) The output stream $Y1$ of $delete([1,2], E, R)$ is composed. And $permute(R, Y1)$ returns the answer Y1={ v([2],[c1]),v([1],[c2,c3]) }, which is sent to $cons(E, Y1, Y)$.

(8) The streams E={ v(1,[c1]),v(2,[c2,c3]) } and Y1={ v([2],[c1]),v([1],[c2,c3]) } are decomposed.

(9) *Cons* is called on only pairs of computable colored values.

(9a) Cons(1,[2],A1) is called and returns the answer A1=[1,2] associated with the color [c1].

(9b) Cons(2,[1],A2) is called and returns the answer A2=[2,1] associated with the color [c2,c3].

(10) The output stream Y is composed. And $cons(E, Y1, Y)$ returns the answer Y={ v([1,2],[c1]), v([2,1],[c2,c3]) }.

Finally, we get a set of all solutions { [1,2],[2,1] }.

Computation tree model can be constructed dynamically according to the computation.

On a computation tree model, we can find three types of nodes.

fork: If P is an OR-predicate, then the execution world branches. And for each case, a new primitive color is added. (*delete*)

merge: If P has more than two input streams, then pick up the values whose colors are consistent with one another. (*cons*)

pass: Otherwise, execute the goal with the received color in AND-parallel. (*permute*)

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

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

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

where *id* is the counter that is characteristic to each call of OR-predicate, and *clause-number* is the ordinal number of definition clauses of that predicate. Variables appeared in the original program are bound to a stream of colored values, denoted by { v(V1,C1),v(V2,C2), ..., v(Vn,Cn) }. Each *id* corresponds to each fork point and *clause-number* corresponds to to the branches from that fork point in the computation tree model.

In the above example, c1 is $(\#1, 1)$ which denotes that the first clause of *delete* is selected on the call of number #1. And c2 is $(\#1, 2)$, which denotes that the second clause of *delete* is selected on the call of number #1. #1 is an id-number given by the system. [c2,c3] is $[(\#1, 2), (\#2, 1)]$ which shows that the second clause is selected at the first fork point and the first clause is selected at the second fork point. Color is an incremental set, indicating the history of which clause is selected at each fork point.

We mentioned before that computability is checked by examining colors. In the followings, we show how to check consistency between colors. For any pair of colors C1 and C2, their relation is defined either as same, orthogonal or productive.

Definition (same,orthogonal,productive)

Let C1 and C2 be colors. Then the relation between C1 and C2 are defined as follows.

(1) If there exists such color (*id*,*clause-number*) that is included both in C1 and in C2, then C1 and C2 are said to be *same*.

(2) If there exists an *id* such that $(id, n1)$ is included in C1 and $(id, n2)$ is included in C2 where $n1 \neq n2$, then C1 and C2 are said to be *orthogonal*.

(3) Otherwise, C1 and C2 are said to be *productive*.

Intuitively, values with the same color have the common branch at some fork point and values with the orthogonal color have the different branch at the same fork point, and values with product colors have no common fork point in the computation tree model.

Definition(consistency)

For colored values v(V1,C1) and v(V2,C2), if C1 and C2 are either same or productive, then C1 and C2 are said to be *consistent*. For colored values v(V1,C1), v(V2,C2), ..., v(Vn,Cn), if any $i, j (i \neq j)$, Ci and Cj are consistent, then C1,C2,...,Cn are consistent.

When a goal receives the set of values v(V1,C1), v(V2,C2), ..., v(Vn,Cn), each of which is received from the different streams, if C1,C2, ...,Cn are consistent, then the goal is computable with the values V1,V2,...,Vn. Let R be the result. Then, the color associated with R is defined as the union of C1,C2,...,Cn. It is called *joint*

color.

3. Compilation

When *ANDOR-II* program is given, our compiler translates it into FGHC program. The compiler consists of two main modules : DFA module and TRA module.

DFA is an analyzing part, In this procedure, the compiler reads the basic program, and makes a graph for each clause which shows the data flows in the clause. Regarding this graph, it analyses the data types and generates a pre-transformed codes with the requisite information. TRA is a translation part. It translates the pre-transformed code into FGHC program using these informations.

3.1. DFA

Data flow analysis(DFA) means the examination of data flows for each clause using a data flow graph(DFG). It picks up such variables (channel) that possibly have a stream flow.

In this procedure, we have the following subprocedures:

(1) making a basic DFG
(2) pick up OR-nodes
(3) finding a shell-covered goals
(4) judgment of output variable types
(5) making pre-transformed code

First of all, we make a basic DFG of each clause which shows the data flow in the clause. Secondly, we have to pickup OR-nodes which may produce a stream. Next, we find a shell covered goals and judge the types of output variables for each predicate. And lastly, we make the output information. Described later, we will make a transformed program using these informations.

3.1.1. DFG

We must grasp the data flows among goals through shared variables and the flowing data types. A data flow graph is introduced so as to support the analysis.

Definition (Data Flow Graph of a clause)

Let C be a clause whose head goal is H and whose body goals are $B_1, ..., B_m$. For a clause C, *Data Flow Graph (DFG) of C* is a minimal graph which satisfies the following conditions:

(1) Each node has a different label.
(2) Each node is labeled with $B_1, ..., B_n$, respectively.
(3) Each edge is labeled with a variable or a constant appearing in the body goals. (Strictly speaking, there also exist some supplement variables, explained later)
(4) For goals B_i and B_j $(i \neq j)$, if a variable X appears both in an output argument of the goal B_i and an input argument of B_j, then there is an edge labeled with X from the node labeled with B_i to the node B_j.
(5) There is a special edge that lacks either a starting node or an ending node and whose label is the variable or constant appearing in the head goal H.

Definition (global input, global output)

In the above (5), the label put on the edge that lacks a starting node is called a *global input* and the label put on the edge that lacks an ending node is called a *global output*.

DFG is constructed statistically for a given source program, and it has the following properties.

(1) It is acyclic.
(2) There are no edges with the same label whose starting nodes are different.

The property (1) is because of the determinacy of modes, and the property (2) is based on the prohibition of multiple writers. On the other hand, there exists edges with the same label whose starting nodes are same. (Fig.2a) And we sometime reduce the graph to the form shown in Fig.2b.

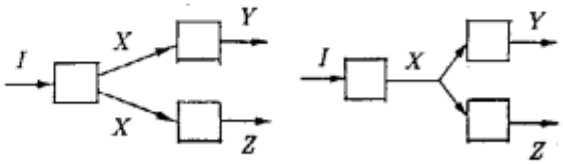

Fig 2a. DFG **Fig 2b. Reduced DFG**

Example.

Fig.3 shows the DFG of the clause:

```
permute(X,Y) :- true |
  delete(X,E,R), permute(R,Y1), Y=[E|Y1].
```

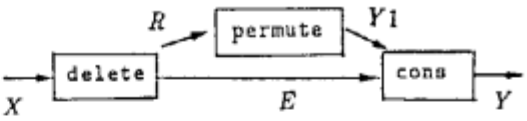

Fig 3. DFG of permute

Definition(DFG's of a predicate)

If a predicate P is defined by a set of clauses $C_1, ..., C_n$, then a set of DFG's of $C_1, ..., C_n$ is said to be *DFG's of P*.

[Notation]

We sometimes say 'node' instead of 'goal' and vice versa. And we use 'channel' instead of 'edge'.

Definition(*Pseudo OR-predicate*)

An AND-predicate P is said to be *pseudo OR-predicate* if there exists such clause in the definition of P that includes, as a body goal, either an OR-predicate or a pseudo OR-predicate.

Definition(AND-node,OR-node)

For a DFG, if a node corresponds either to an OR-predicate or to a pseudo OR-predicate, then it is said to be an *OR-node*. Otherwise, it is said to be an *AND-node*.

Example.

In Fig.3, *delete* and *permute* OR-nodes and *cons* is an AND-node.

As a clause in a source program has the one-to-one relation with DFG, we will proceed our discussion on DFG's instead of the source program, hereafter.

3.1.2. Shell

After DFG's are constructed by statistical analysis of the source program, we do dynamical analysis on the data flowing on the execution.

We have to examine data types flowing channels (edges) in DFG's, and if a node has a stream input, add a shell onto that node which deals with the stream inputs properly.

Definition (channel types)

If the channel always has single data flow, then it is called a *scalar type*. If it possibly has a stream data flow, then it is called a *vector type*.

Example.

In the Fig.3, the channel X is a scalar type, and the channel $R, E, Y1$, and Y are vector types.

Definition(descendant)

For a DFG, if there exists an edge from a node N1 to a node N2, then it is said that *N2 is an immediate descendant of N1*. The node N2 that satisfies either of the following condition is called a *descendant of N1*.

(1) N2 is an immediate descendant of N1

(2) There exists a node N such that N is an immediate descendant of N1 and that N2 is a descendant of N.

We judge the channel types in a DFG according to the following rules:

[Rules]

(1) A channel whose starting node is OR-node is the vector type.

(2) A channel whose starting node is the descendant of an OR-node is the vector type.

(3) The rest of channels are the scalar types.

The most important case is that more than two input vector type channels flow into one node, since we can compute only on the data whose colors are consistent. *Shell* is introduced in order to deal with vector type channels.

Shell has the following functions:

(1) *decomposition*

It dispatches the element from the stream.

(2) *consistency check*

After the decomposition, if there are more than two input streams, then check the consistency of the colors of the colored values. If they are consistent, then pass them to the core process with the joint color. If they are not consistent, then abolish the computation for those inputs.

(3) *composition*

Put the solutions in each world together into the stream form again.

Core process executes the original computation in an execution world.

We judge which node needs a shell, according to the following rules:

[Rule]

If the goal has an input channel of vector type, then the node needs a shell. We call the node a *shell-covered node*.

Note that shell-covered node is a descendant of an OR-node.

We also examine the shell type, since different shell is created depending on which channel is vector type. Shell type is represented in a set of channel types. For example, the shell type [vector,scalar,vector] denotes the shell whose first and third arguments are vector, and the second is scalar.

We also have to judge the global output data type of each predicate P. In general, there are multiple DFG's of a predicate P. Therefore, it may happen that in some DFG's of P, the channels of the global output are scalar type, while in other DFG's of P, the corresponding channels are vector type. In this case, we take safer judgment, namely, judge the global output data as a stream type.

Example.

There are two DFG's of *delete*.(Fig.4) As for the second argument, according to the first graph, the channel is the scalar type, and to the second graph, it is the vector type. We judge the second argument as the stream type.

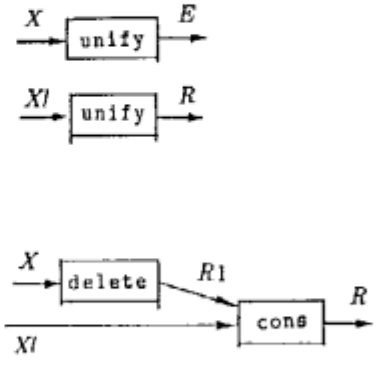

Fig 4. DFG's of delete

3.1.3. Treatment of Term

In order to lessen the overhead of the transformed program, we deal with a term in somewhat tricky way when constructing a DFG. That is, if a compound term (i.e. terms other than constants, variables) appears as the argument of a body goal, then we rewrite the goal so that the compound term is not used as labels of edges. We will give an informal explanation of the procedure.

(I) a compound term appears in the input argument

Assume that a goal $P(t(X,Y),Z)$ appears as a body goal of some clause where the mode is $P(+,-)$. And assume that X has a vector type flow, while Y has a scalar type. Then, we rewrite the goal to $P^*(X,Y,Z)$ where the mode is $P^*(+,+,-)$. At the same time, we add the new clause

```
P*(X,Y,Z) :- true | P(t(X,Y),Z).
```

(II) a compound term appears in the output argument

Assume that a goal $P(X,t(Y,Z))$ appears as a body goal of some clause where the mode is $P(+,-)$. In this case, we rewrite the goal to two goals $P(X,NV)$ and $out\_Term^*(NV,Y,Z)$ where NV is a new variable that does not appear in the clause and the mode is $out\_Term^*(+,-,-)$. At the same time, add the new clause

```
out_Term(NV,Y,Z) :- true | t(Y,Z)=NV.
```

After rewriting, for all clauses other than the newly generated clauses in reading the source program, the number of arguments of each goal is equal to the number of channels of the corresponding node. Note that these newly generated clauses have only one body goal. It means that the body goal never has a shell.

3.1.4. Output Pre-Transformed Code

Finally in the DFA procedure, pre-transformed code is generated. Pre-transformed code consists of three parts : predicate list part, information part and definition part. Predicate list part is the list of predicates appeared in the original program and newly generated predicates. Information part represents the information for each predicate, such as arity, relation, global outputs type and shell type. Definition part is almost as same as the original program, only the information about shell is associated.

3.2. TRA

TRA(TRAnsformation) is the procedure which transforms the pre-transformed code into FGHC program. It consists of the following four subprocedures:

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

3.2.1. Shell Creation

Firstly, we make shells according to the information. For each shell covered node, create the corresponding *shell_creation* clauses. Note that all the output channels of shell covered nodes are always *vector*. *Shell_creation* clauses decompose the input stream into a single input value with its own color, pass them to the core process, and put the output values for each color together into the stream again. Core process for each data is executed independently. We do not know whether each core process succeeds, fails or deadlocks. Neither do we know the order of solution we can get. If a core process succeeds, then the answer is added to the tail of the output stream. If it fails or deadlocks, no answers are added. Therefore, we can get all the solutions without being disturbed by failure or deadlock. It is implemented by the fair merge technique. We will illustrate the procedure by the permutation example.

Example.

According to the information in the pre-transformed code, two types of shells for *cons* are created: [scalar,vector,vector] and [vector,vector,vector].

(I) First type : [scalar,vector,vector]

In this case, since only one input channel is a vector type, shell decomposes the second argument and passes each value to the core with its own color. And output values are put into one stream by fair merge. Therefore, the following clauses are created:

```
cons_Shell_1_1(X,[v(Y,Cy)|Ys],Z) :- true |
    cons_Core(X,Y,Z0,w(Cy)),
    Z1=[v(Z0,Cy)],
    cons_Shell_1_1(X,Ys,Z2),
    out_Merge(Z1,Z2,Z).
cons_Shell_1_1(_,[],Z) :- true |  Z=[].
```

(II) Second type [vector,vector,vector]

In this case, we decompose the first argument first, then second argument. Moreover, we check the consistency of colors before passing the data to the core process. Therefore, the following clauses are created:

```
cons_Shell_2_1([v(X,Cx)|Xs],Y,Z) :-
    true |
    cons_Shell_2_2(v(X,Cx),Y,Z1),
    cons_Shell_2_1(Xs,Y,Z2),
    out_Merge(Z1,Z2,Z).
cons_Shell_2_1([],_,Z) :- true |  Z=[].

cons_Shell_2_2(v(X,Cx),[v(Y,Cy)|Ys],Z) :-
    true |
    cons_Check_2_1(v(X,Cx),v(Y,Cy),Z1),
    cons_Shell_2_2(v(X,Cx),Ys,Z2),
    out_Merge(Z1,Z2,Z).
cons_Shell_2_2(_,[],Z) :- true |  Z=[].
```

3.2.2. Check Creation

As stated before, we have to check the consistency among colors in the shell. If the colors are consistent, then the values are passed to the core process and the

joint color is created. Otherwise, do nothing. This process is called from *shell_creation* clause.

Example.

```
cons_Check_2_1(v(X,Cx),v(Y,Cy),Z) :- true |
    same_Color([Cx,Cy],R),
    cons_Check_2_2(R,X,Y,Z).

cons_Check_2_2(success(C),X,Y,Z) :- true |
    cons_Core(X,Y,Z0,w(C)),
    Z=[v(Z0,C)].
cons_Check_2_2(fail,_,_,Z) :- true | Z=[].
```

In the above program, *same_Color* is the system predicate which checks the consistency among colors. If they are consistent, then it returns the value *success*(C) where C is a joint color. If they are not consistent, then it returns the value *fail*. Note that *cons_Check_2_1* never fails.

3.2.3. OR–AND Transformation

In this step, we transform nondeterministic OR-relation clauses into the deterministic ones. OR-relation in the source program is realized by AND-parallelism. And their solutions are collected by using a fair merge again.

Example.

```
delete_Core(X,Y,Z,w(C)) :- true |
    get_Counter(Ct),
    delete_Core_1(X,Y1,Z1,w(C),Ct),
    delete_Core_2(X,Y2,Z2,w(C),Ct),
    out_Merge(Y1,Y2,Y),
    out_Merge(Z1,Z2,Z).
```

In the above program, *get_Counter* is the system predicate which gives some ID-number to identify the call.

3.2.4. Predicate Transformation

Finally, we transform all predicates. It is performed on the types of clauses according to the following rules. Newly generated clauses other than *out_Term* are regarded as AND-clauses.

(I) AND-clauses
  (1) Change the predicate name p to *p_Core*.
  (2) Add an argument to receive the information of the current color.
  (3) If it includes shell covered node as a body goal, then rewrite the goal to the corresponding shell creation goal, and make the output of the other body goals in the stream form of $[v(V,C)]$ where V is the original output variable and C is the current color.
  (4) Rewrite the goal other than shell covered node to the corresponding goal of **_Core*. (5) If the head unification might fail when the predicate is called, then add an extra clause for failure.

Example.

```
permute_Core([],Y,w(C)) :- true |
     VY=[],
     Y=[v(VY,W)].
permute_Core([X|Xs],Y,w(C)) :- true |
    delete_Core([X|Xs],E,R,w(C)),
    permute_Shell(R,Z),
    cons_Shell_2_1(E,Z,Y).
```

(II) OR-clauses
  (1) Change the predicate name p to *p_Core*.
  (2)-(4) As same as the above statement.
  (5) Add an argument to receive the information of the ID-number of the current call.
  (6) '*true*' is added as a guard goal.
  (7) The body goal which updates the color information is added.

Example.

```
delete_Core_1([X|Xs],E,Y,w(C),Ct) :-
    true |
    append(W,[(c1,Ct)],NewC),
    VE = X,
    E=[v(VE,NewC)],
    VY = Xs,
    Y=[v(VY,NewC)].
delete_Core_1(_,E,Y,w(_),_) :- true |
    E=[], Y=[].
```

(III) Clauses in the form of *out_Term**,
We do not have to transform the clauses except for adding an argument for receiving the information of the current color.

The transformed program is shown in Appendix A.

4. APPLICATION

In this section, we will give more sophisticated example of fault diagnosis of a simple circuit. Our purpose is the following two hold : to describe dynamical behaviors of such a system that has a lot of behavioral possibilities in a simple way, and to gain a set of solutions via parallel search mechanism.

Fig.5 shows a half adder. It adds two binary digits, but it cannot take care of carries which might be originated in lower order positions when numbers with more than one digital position are added. S is the sum of A and B, and C is the carrier. The outputs C and D are observable, but inputs A and B cannot always be observed. For simplicity, we assume that two inverters and an OR-gate are known to be fault-free, and input A is observable. The system is represented in *ANDOR-II* as follows:

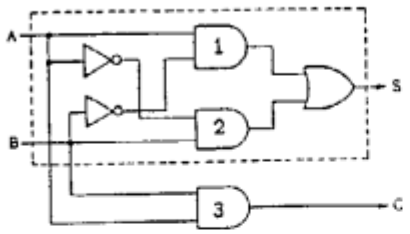

Fig 5. Half Adder

```
:- and_relation circuit/2.
circuit([I1,I2],Output) :- true |
    element(inv_1,inverter,correct,[I1],O1,_),
    element(inv_2,inverter,correct,[I2],O2,_),
    element(and_1,and2,doubt,[I1,O2],O3,St1),
    element(and_2,and2,doubt,[O1,I2],O4,St2),
    element(and_3,and2,doubt,[I1,In2],O5,St3),
    element(or_1,or2,correct,[O3,O4],O6,_),
    Output=[[O6,O5],[St1,St2,St3]].
```

In *element*(*Name*, *Type*, *State*, *In*, *Out*, *State_1*), *Name* denotes the name of component and *Type* shows its type. *State* is the given state of the component. If *State* is *doubt*, we do not know the component behaves correctly or not. Therefore, the behavior is represented by OR-relation. *In* and *Out* show the set of inputs and outputs of the component, respectively. *State_1* is the detected state of the component, which is either *correct* or *error*. In the last goal of *circuit*, the variable *Output* is bound to a set of observed output variables and a set of detected states of the components which might behave incorrectly. The rest of the programA complete program is shown in Appendix B. In the program, the two OR-predicate *set_num* and *element_d* represent the nondeterminacy. *Set_num* causes the execution world split depending on the input *B*. And *element_d* causes the world split depending on the behavior of that component.

Our purpose is to detect a faulty component and determine the unknown inputs. Fault diagnosis is done by generate-and-test strategy in the following manner: *Circuit* is the generator and *check_output* shown below is the tester. Generator performs the multi-simulation with the assumption of faulty component and the input *B*. And tester picks up the cases the generated outputs are equivalent to the observed outputs.

```
:- and_relation test/2.
test([Data_in,Data_out],Ans) :- true |
    set_input(Data_in,Input),
    circuit(Input,Output),
    check_output(Input,Output,Data_out,Ans).

:- and_relation check_output/4.
check_output(In,[Out,State],Data_out,Ans) :-
    Data_out = Out |
    Ans=[In,Out,State].
```

Example.

$C = 0$ and $D = 1$ are observed as outputs, and $A = 1$ is observed as an input. In this case, it is invoked by the call of the goal.

```
test([[1,?],[0,1]],Answer)
```

As the result of the execution of the transformed program, two possible cases are detected.

```
[ [A,B],[C,D],[comp1,comp2,comp3] ] =

[ [1,0],[0,1],
  [(and_1,error),(and_2,correct),(and_3,error)]
],
[ [1,1],[0,1],
  [(and_1,correct),(and_2,correct),(and_3,corre
]
```

The first case denotes that the components 1 and 3 behaves incorrectly, for the input $B = 0$, and the second case denotes that all the component behave correctly for the input $B = 1$.

5. DISCUSSION

5.1. Comparison with Other Works

The problem of all solution search and transformation method from nondeterministic program into the deterministic one seem to be significant and fruitful.

Ueda has proposed the transformation method from exhaustive search program in Prolog into deterministic FGHC/Prolog program by using continuation. [Ueda 86a||Ueda 86b| OR-parallelism in the original program is realized by AND-parallelism in the transformed program, while the intrinsic AND-parallelism is realized by sequential AND clauses. His paper reports that transformed program have much more efficiency for some class of programs, and that they do not lose so much efficiency for others. Since the computation model has a statistic scheduling, it is possible to encode coroutine. However, it is not always possible to schedule statistically on the simulation of nondeterministic systems, since the number of split cases depends on the input data.

Tamaki presented alternative method based on stream execution model.[Tamaki 86| He adopts a stream-based compilation with the constraint of ground I/O. It transforms the program written in the language with AND/OR parallelism into the one in committed choice language. In his system, data from alternative worlds are propagated to several AND processes in a stream form Although runtime overhead is high, the use of stream enables dynamic scheduling and the transformed program has more parallelism, since it reflects both the AND- and OR-parallelism in the source program, And the execution of the AND-processes in the same 'block' are synchronized for each data. However, there is a restriction that

if some processes in the same block have a shared variable, then it appears only in input arguments of the processes. That is, we cannot simulate such a phenomenon in that consumer and producer can executed in parallel. It always happen to the real system.

Although both methods have advantages, they are not sufficient to treat a nondeterministic system. Our system designed so that it can suffice the basic requirements for the treatment of nondeterministic system by the introduction of *color*.

(1) Dynamic scheduling is realized in the execution by the utilization of stream.
(2) Simulation of the basic behavior such as the parallel execution of consumer and producer is possible.

We avoid using d-list on collecting the solutions, since we hope the system to support case of 'bad' cases. If a process deadlocks, or goes into the infinite loop, then we cannot get another solution Hence, although we lose the efficiency by using d-list, we can get all the solutions in spite of 'bad' cases using fair merge.

5.2. Future Works

There are some remaining problems for future works.

One is the optimization of transformed program. The most important point is the overhead of color consistency check in the execution of the transformed program. Two ways can be considered for the solutions of this problem. The one is the choice of the necessary information and restriction of color addition. In current system, primitive color information is added at every fork point. But detail information is not always required for the identification of the world. We select only the essential ones so that the system can check less data. The other solution is the choice of the necessary consistency check. For example, we do not have to check consistency of productive colors. It is because that since they do not have common fork point, we can define the join color only by appending the color information without checking their consistency. Automatic mode analysis is also in consideration so that it can decrease the users' burden.

Suppression of irrelevant computations is also an important problem to increase efficiency. If a process fail, then the conjunctive goals need not to be computed any more. However, the current system which does not have such a mechanism completes all the computations. In order to realize such a mechanism, for instance, we make the system so that if an element of conjunctive goals fails, then an agent is sent to abort the other goals.

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

An enlargement of application domain is For example, elimination of the prohibition of multiple writers can enlarge the class of programs to be handled. Moreover, since an actual system often has a cyclic data flow, it is required to simulate such systems.

6. CONCLUDING REMARKS

We have proposed a parallel logic programming language *ANDOR-II* for modeling and simulating the system with the nondeterministic behavior. We have presented the compilation method from *ANDOR-II* to FGHC. Both AND- and OR-parallelism are realized based on the stream. The compiler is developed by DEC10-Prolog on DEC-2060. The transformation program of the half adder example shown in the section 5 takes about 3500 msec by FGHC compiler on DEC10-Prolog. Although the overhead of color check is burdensome now, we can make it smaller by optimization.

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, Chief of the 1st Laboratory of ICOT, for his advice and encouragement.

REPERENCES

[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.

[deKleer and Brown 83] deKleer,J. and J.S.Brown, "The Origin, Form, and Logic of Qualitative Physical Laws," Proc.of IJCAI-83, pp.1158-1169.

[Kuipers 86] Kuipers,B.J., "Qualitative Simulation," Artificial Intelligence, Vol.29, No.3, pp.289-338, 1986.

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

[Shapiro 84] Shapiro,E.Y., "Systems Programming in Concurrent Prolog," Proc.11th Annual ACM Symposium on Principles of Programming Languages, pp.93-105,1984.

[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).

[Tamaki 86] Tamaki,H., "Stream-Based Compilation of Ground I/O Prolog into Committed Choice Languages, Technical Report No.86-5, Dept. of Information Science, Ibaraki University, 1986.

[Ueda 85] Ueda,K., "Guarded Horn Clauses," Proc.of Logic Programming 85, LNCS 221, Springer pp.168-

179, 1986.

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

[Ueda 86b] Ueda,K., "Making Exhaustive Search Programs Deterministic(II)," Proc.of 3rd Conf. of Japan Society of Software Science and Technology, pp.1-8, 1986 (in Japanese).

Appendix A Transformed program of permuation.

```
permute_Shell([],Y) :- true |  Y=[].
permute_Shell([v(X,C)|Xs],Y) :- true |
    permute_Core(X,Y1,w(C)),
    permute_Shell(Xs,Y2),
    out_Merge(Y1,Y2,Y).

permute_Core([],Y,w(C)) :- true |
     VY=[],
     Y=[v(VY,C)].
permute_Core([X|Xs],Y,w(C)) :- true |
    delete_Core([X|Xs],E,R,w(C)),
    permute_Shell(R,Z),
    cons_Shell_2_1(E,Z,Y).

delete_Core(X,Y,Z,w(C)) :- true |
    get_Counter(Ct),
    delete_Core_1(X,Y1,Z1,w(C),Ct),
    delete_Core_2(X,Y2,Z2,w(C),Ct),
    out_Merge(Y1,Y2,Y),
    out_Merge(Z1,Z2,Z).

delete_Core_1([X|Xs],E,Y,w(C),C) :- true |
    append(C,[(n1,C)],NewC),
    VE = X,
    E=[v(VE,NewC)],
    VY = Xs,
    Y=[v(VY,NewC)].
delete_Core_1(_,E,Y,w(_),_) :- true | E=[], Y=[].

delete_Core_2([X|Xs],E,Y,w(C),Ct) :- true |
    append(C,[(n2,Ct)],NewC),
    delete_Core(Xs,E,R,w(NewC)),
    cons_Shell_1_1(X,R,Y).
delete_Core_2(_,E,Y,w(_),_) :- true | E=[], Y=[].

cons_Shell_1_1(X,[v(Y,CY)|Ys],Z) :- true |
    cons_Core(X,Y,Z0,w(CY)),
    Z1=[v(Z0,Cy)],
    cons_Shell_1_1(X,Ys,Z2),
    out_Merge(Z1,Z2,Z).
cons_Shell_1_1(_,[],Z) :- true |  Z=[].

cons_Shell_2_1([v(X,C)|Xs],Y,Z) :- true |
    cons_Shell_2_2(v(X,C),Y,Z1),
    cons_Shell_2_1(Xs,Y,Z2),
    out_Merge(Z1,Z2,Z).
cons_Shell_2_1([],_,Z) :- true |  Z=[].

cons_Shell_2_2(v(X,CX),[v(Y,CY)|Ys],Z) :- true |
    cons_Shell_2_3(v(X,CX),v(Y,CY),Z1),
    cons_Shell_2_2(v(X,CX),Ys,Z2),
    out_Merge(Z1,Z2,Z).
cons_Shell_2_2(_,[],Z) :- true |  Z=[].

cons_Shell_2_3(v(X,CX),v(Y,CY),Z) :- true |
    same_Color([CX,CY],R),
    cons_Shell_2_4(R,X,Y,Z).

cons_Shell_2_4(success(C),X,Y,Z) :- true |
    cons_Core(X,Y,Z0,w(C)),
    Z=[Z0].
cons_Shell_2_4(fail,_,_,Z) :- true | Z=[].
```

Appendix B Half Adder Program in *ANDOR-II*

```
:- mode test(+,-), circuit(+,-), element(+,+,+,+,-,-), get_input(+,-),
    element_1(+,+,+,+,-,-), element_d(+,+,+,-,-), element_2(+,+,+,-),
    inverter(+,+,-), and2(+,+,-), or2(+,+,-), set_input(+,-),
    set_in(+,-), set_num(-), check_output(+,+,+,-).
%-----------------------------------------------

:- and_relation test/2.
test([Data_in,Data_out],Ans) :- true |
    set_input(Data_in,Input),                % make a set of inputs
    circuit(Input,Output),                   % generator
    check_output(Input,Output,Data_out,Ans).  % tester

%-------------------------------------------------------------------------
%    half_adder circuit definition

:- and_relation circuit/2.
circuit([In1,In2],Output) :- true |
    element(inv_1,inverter,correct,[In1],Out1,_),
    element(inv_2,inverter,correct,[In2],Out2,_),
    element(and_1,and2,doubt,[In1,Out2],Out3,State1),
    element(and_2,and2,doubt,[Out1,In2],Out4,State2),
    element(and_3,and2,doubt,[In1,In2],Out5,State3),
    element(or_1,or2,correct,[Out3,Out4],Out6,_),
    Output=[[Out6,Out5],[State1,State2,State3]].
```

```
:- and_relation element/6.
element(Name,Type,State,Input,Output,State_1) :- true |
    get_input(Input,Input_1),
    element_1(Name,Type,State,Input_1,Output,State_1).

:- and_relation get_input/2.
get_input([Input1_1|Input1],Input2) :- Input1_1\=(_,_) |
    Input2 = [Input1_1|Input2_1],
    get_input(Input1,Input2_1).
get_input([Input1_1|Input1],Input2) :- Input1_1=(Input1_2,_) |
    Input2 = [Input1_2|Input2_1],
    get_input(Input1,Input2_1).
get_input([],Input2) :- true | Input2=[].

:- and_relation element_1/6.
element_1(Name,Type,doubt,Input,Output,State) :- true |
    element_d(Name,Type,Input,Output,State).
element_1(Name,Type,State,Input,Output,State_1) :- State\=doubt |
    element_2(Type,State,Input,Output),
    State_1=(Name,State).

:- or_relation element_d/5.
element_d(Name,Type,Input,Output,State) :- true |
    element_1(Name,Type,correct,Input,Output,State).
element_d(Name,Type,Input,Output,State) :- true |
    element_1(Name,Type,error,Input,Output,State).

%-----------------------------------------------------------------------
%    element definition

:- and_relation element_2/4.
element_2(inverter,State,Input,Output) :- true |
    inverter(State,Input,Output).
element_2(and2,State,Input,Output) :- true |
    and2(State,Input,Output).
element_2(or2,State,Input,Output) :- true |
    or2(State,Input,Output).

      * inverter, and2, and or2 are defined in the lower level.
%-----------------------------------------------------------------------

:- and_relation set_input/2.
set_input([Data_in|Data_ins],Input) :- true |
    set_in(Data_in,In),
    set_input(Data_ins,Input_1),
    Input=[In|Input_1].
set_input([],Input) :- true | Input=[].

:- and_relation set_in/2.
set_in(Data_in,In) :- Data_in='?' | set_num(In).
set_in(Data_in,In) :- Data_in\='?' | In := Data_in.

:- or_relation set_num/1.
set_num(Num) :- true | Num:=0.
set_num(Num) :- true | Num:=1.

%-----------------------------------------------------------------------

:- and_relation check_output/4.
check_output(Input,[Output,State],Data_out,Ans) :-
    Data_out = Output |
    Ans=[Input,Output,State].
```