ICOT Technical Report: TR-558

TR-558

A Parallel Problem Solving Language ANDOR-II and Its Parallel Implementation

by
K. Takahashi, A. Takeuchi & T. Yasui
(Mitsubishi)

May, 1990

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

Institute for New Generation Computer Technology

A Parallel Problem Solving Language ANDOR-II and Its Parallel Implementation

Kazuko TAKAHASHI Akikazu TAKEUCHI Terumasa YASUI

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

Abstract This paper presents a parallel problem solving language *ANDOR-II* and its parallel implementation. Our purpose is 1) to realize a full combination of AND- and OR-parallelism suitable for a parallel problem solving, and 2) to find a class which can be transformed into KL1. *ANDOR-II* is born for parallel problem solving on concurrent systems which consist of determinate, indeterminate and nondeterminate components interacting each other. Problems on these systems are complicated since the number of possible bahaviors of some components may increase dynamically, and a behavior of some component may affect its own behavior in future. *ANDOR-II* provides the declarative description for that sort of problems. The execution of *ANDOR-II* exploits both AND- and OR-parallelism based on a concept of a "color." Each possible behavior is associated with a distinct color and a color may be refined as computation proceeds. The fork of nondeterminate processes occurs eagerly, which provides a fine grain parallelism. A program written in *ANDOR-II* is transformed into that in KL1 so that OR-parallelism in *ANDOR-II* is compiled into AND-parallelism in the object code by means of a stream-based compilation. Both the compiler and runtime system are written in KL1 and they are now running on Multi-PSI system (without processor allocation).

1. INTRODUCTION

Recently a lot of researches are being undertaken on parallel logic programming. As one of these researches, design and implementation of a new language which supports high parallelism together with programming paradigms and program transformation techniques with these languages is vigorously studied.

Committed-choice languages play main roles in these research areas[Clark and Gregory 84][Shapiro 88][Ueda 86a]. They are originally investigated to realize a parallel execution of conjunctive goals (AND-parallelism), and in compensation, they gave up the function of all solution search (OR-parallelism) which Prolog provides as the strong backtrack mechanism. Namely, in committed-choice languages, once some clause is selected, the alternatives are abandoned. In order to handle the full parallelism, a language which comprises both feautures is desirable, and several researches have currently appeared aiming at a language which takes advantage of both of Prolog and committed-choice languages. That is an amalgamation of AND-parallelism and OR-parallelism [Yang 86][Naish 87][Clark and Gregory 87][Haridi et al. 88].

In general, a new language designer mainly discusses the following issues.

(1) description

The language has simple and well-formed structure, covers a large class of problems, and its computation model can be well-defined.

(2) implementaion

The compiler/interpreter is easy to implement.

(3) execution

It can provide high performance.

On parallel programming languages, one more factor is added.

(4) parallelism

It can provide an "appropriate" granuality of parallelism.

These factors are interacting with each other deeply. In general, the more expressive power a language has, the more the performance decreases; in order to get high performance, many annoying contols are added to a description language. And an "appropriate" granuality of parallelism depends on a target problem and a machine architecture. Each language consider a different factor as the most significant point. In fact, in case of a langnage which comprises both AND- and OR-parallelism, languages such as CP and Andorra have splitted into a various classes [Shapiro 88][Haridi 89]. On considering the above four issues in designing that class of language, one of the focus is the elegant and efficient treatment of nondeterminism. That is, when a nondeterminate process is invoked and how it is extended.

ANDOR-II was born on these background [Takeuchi et al. 87]. Our purpose is 1) to realize a full combination of AND- and OR-parallelism suitable for a parallel problem solving, and 2) to find a class which can be tranlated into KL1.[†1]

Originally, the target is the problems on concurrent systems which consist of many determinate, indeterminate and nondeterminate components. The behavior of these components have many possibilities depending on the series of actions taken by them. Furthermore, these possibilities increase dynamically when such components take actions repeatedly. From the point of view of computation, handling of nondeterminate components comprises simultaneous consideration of many possible worlds which may grow dynamically. *ANDOR-II* has given a solution to this problem by realization based on "colored world."

[†1] KL1 is a language, developped at ICOT, which is identical to GHC with several controls.

In the execution of *ANDOR-II* , possible worlds are created at each choice point. Each world is painted by distinct color and computation is executed on this colored world. Each data is also colored by its own color. If and only if a pair of data are painted by the same color, then they belong to the same world, and the computations between them can be executed. If multiple solutions are possible these solutions associated with the distinct color are packed in a vector form. Some goals has an extra cover called "shell" to handle these types of data.

Thus, *ANDOR-II* employs an eager exploitation of nondeterminate forks. Several forks may take place at the same time. There is no meta-calls nor special controls on the invocation, such as *delay* declaration or *when* declaration. This makes a language very simple and provides a fine grain parallelism. Although lazy nondeterminate forks might save search spaces, it causes much suspension and benefits little in handling such problems on concurrent systems as stated above.

Both the compiler and runtime system are written in KL1. We adopt the approach of the transfomation into KL1 code rather than that of a direct implementation to machine code. It makes an implementation easier to embed some useful mechanisms of KL1 both in compiler and runtime system.

The system is now running on Multi-PSI system. However, it is running only on a single processor, since the current operating system on Multi-PSI do not support an automatic processor allocation[†2]. We could obtain a reasonable performance, which is a good chart to a realization on a multi-processor machine.

This paper is organized as follows. In the next section, the overview of the language feature and basic idea of execution model are presented. In section 3, detail compilation techniques are explained, and some optimization is discussed in section 4. In section 5, the result of evaluation and discussion are shown. And in section 6, the conclusion and future works are described. This paper assumes the familiarity with Prolog[Sterling and Shapiro 86] and GHC[Ueda 86a].

[†2] Multi-PSI is a machine with multiple processor elements developed at ICOT. For detail explanation on KL1 and pseudo Multi-PSI, see [Uchida et al. 88]

2. LANGUAGE ANDOR-II

2.1. Syntax

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

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 with that of GHC. An OR-predicate definition consists of a mode declaration, OR-relation 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. A goal in a guard part is restricted to a test predicate.

Mode is attached to each argument of each predicate. User must declare at the top of a program, and each predicate has a unique mode declaration. There are three modes: *input*, *output* and *neutral*.

Before giving the definition of mode, we will introduce a concept of *inspect*.

Definition (inspect)

Let A be an argument of the head goal of a clause. If either of the following conditions holds, then it is said that this clause *inspects* the argument A.

(1) A is a nonvariable term.

(2) A is a variable, and there exists a variable that appears both in A and in some guard goal.

(3) A is a variable, and there exists a variable that appears both in A and in the right hand side of the body goal in the form of $lhs := rhs$.

Definition (mode)

Let A be an argument of a predicate P and $D_1, D_2, ..., D_n$ be clauses defining P.

(1) *input mode*

If there exists such $D_i (1 \leq i \leq n)$ that satisfies either of the followings, then the mode of A is *input*.

(i) D_i inspects A.

(ii) A is a variable V, and there exists a body goal in which V appears in an argument in an *input mode*.

(2) *output mode*

If in all $D_i (1 \leq i \leq n)$, there exists a goal called directly or indirectly in the body of the clause that instantiates A to a nonvariable term, then the mode of A is *output*.

(3) *neutral mode*

If all $D_i (1 \leq i \leq n)$ satisfy all of the followings, then the mode of A is *neutral*.

(i) D_i does not inspect A.

(ii) A is a variable V and there exists no body goal in which V appears in an argument in an *input mode*.

(iii) There exists no goal called directly or indirectly in the body of the clause that instantiates A to a nonvariable term.

Intuitively, the input mode and the output mode in *ANDOR-II* have the same meaning with those in Prolog, respectively. Neutral mode appears when a process creates a cons-cell in a stream. Detail discussion using an example is shown in the next section.

Note that the mode of an argument is decidable, and the argument whose mode cannot be

defined by the above definition is prohibited. Therefore, user must not write a clause such as

```
p(f(X)) :- true | X=a.
```

since the mode of the argument of *p* is both *input* and *output*. This restriction is required for a simple implementation. Input mode, output mode, neutral mode are denoted by '+', '−' and '?', respectively. and we call arguments in these mode *input argument*, *output argument* and *neutral argument*, respectively.

The following is an *ANDOR-II* program of *compute*. For a given input list X, *compute* picks up an arbitrary element and returns the sum of its squared value and cubed value.

Example 1.

```
%% Compute

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

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

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

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

2.2. Computation Model

The computation rule of AND-predicates is similar to that of GHC [Ueda 86a]. That is, two rules are imposed: *rule of suspension* and *rule of commitment*. And only rule of suspension is imposed on the execution of OR-predicates.

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

Fig. 2.1 shows the computation model of *compute* in the *Example 1*.

Suppose the execution of *compute* with the list [1,2,3]. Similar to the other committed-choice languages, each predicate can be regarded as a process and each shared variable can be regarded as a communication channel among processes. When *compute* is invoked with the list [1,2,3], *pickup* is invoked with the list [1,2,3]. *Pickup* is a nondeterminate process and has three possible ways of generating the values of 1, 2 and 3 depending on which definition is selected. They are painted by distinct colors and packed in an arbitrary order in a vector form. It is passed to two processes *square* and *cube*. These processes have extra covers called "shell." When these processes receive the vector, the shell generates computational worlds for each element of the vector. A world is painted with the same color as that of the corresponding element. Computation for each data is executed in this colored world. The output of each

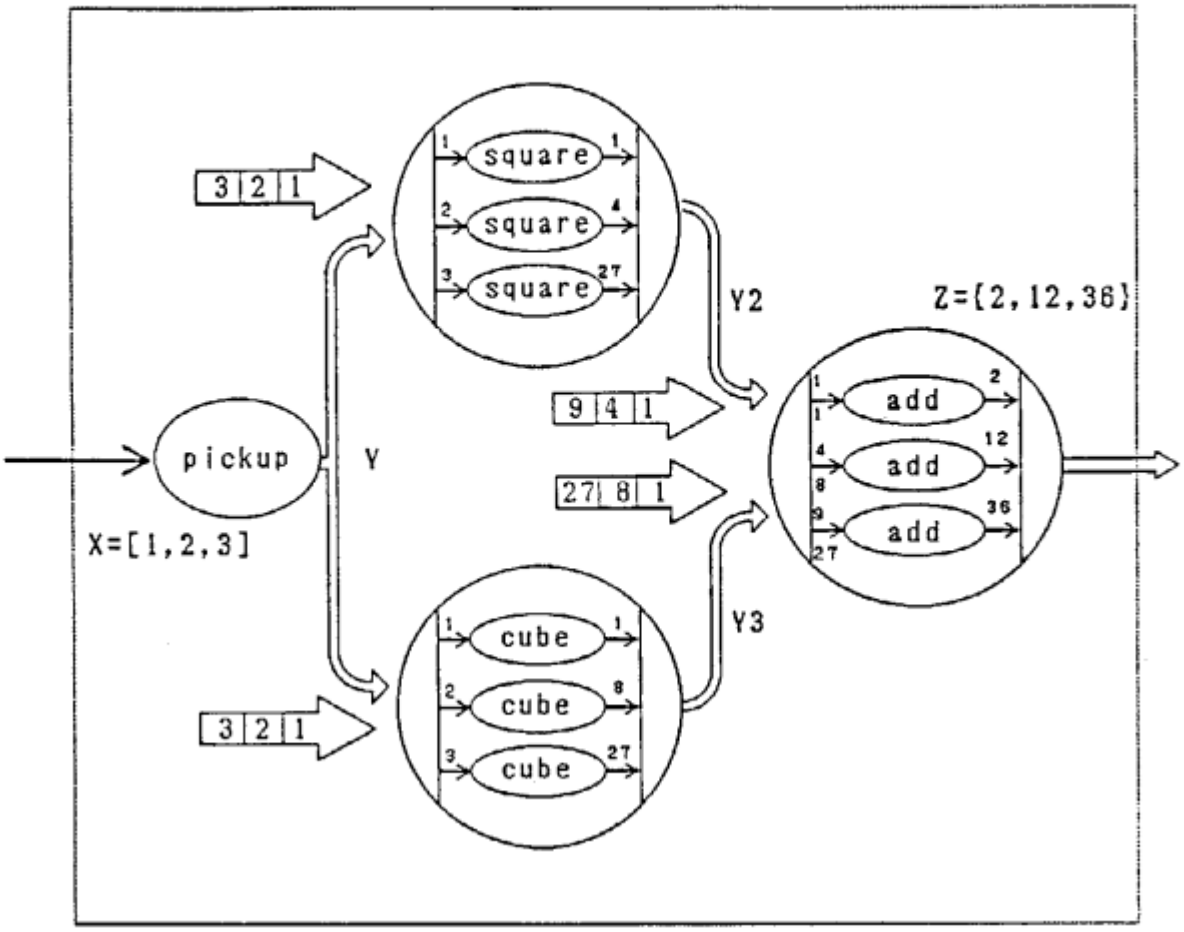

Fig 2.1. Computation model of *compute*

world is painted by its own color and all these colored values are recomposed into a vector form. A process *add* receives vectors from two processes *square* and *cube*. By the shell of *add*, two vectors are preprocessed to make a set of pairs with the same color, and for each pair of the set, the computation is executed in the corresponding world.

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

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

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

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

When an OR-predicate including a variable, say X, is invoked in a world, conceptually the world splits into several worlds along OR-clauses. In coloring scheme, instead of actually creating new worlds, a new color C_i is generated for each OR-clause. In computation of each OR-clause whose associated color is C_i, the variable X might be instantiated to a value V_i. Such multiple binding to the variable X is realized by instantiating X to a vector of colored values, denoted by $\{v(V_1, C_1), v(V_2, C_2), ..., v(V_n, C_n)\}$. We use a term *colored vector* or simply *vector* to denote the data of this type and distinguish it from the stream, which is a different

concept for stream programming. A data without a color (i.e. simple value) is called *scalar*.

Definition (same,orthogonal,productive)
For a pair of colors C_1 and C_2, one of the following three relations holds.
(1) If there exists a branching point bp such that $(n1, bp)$ is included in C_1 and $(n2, bp)$ is included in C_2 where $n1 \neq n2$, then C_1 and C_2 are defined to be *orthogonal*.
(2) Let $bp_i (i = 1, ..., k)$ be a branching point appearing both in C_1 and C_2 and let n_i and m_i be associated clause numbers in C_1 and C_2, respectively. If $n_i = m_i$ for all $i (i = 1, ..., k)$, then C_1 and C_2 are defined to be *the same*.
(3) If C_1 and C_2 share no branching point, C_1 and C_2 are defined to be *productive*.

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

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

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

Color is not asscoiated only to a ground term but also to a variable in a neutral argument. Let us show another example of *ANDOR-II* .

Example 2.

```
%%  Simple Cycle

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

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

p1([X|X1],Y) :- X>20  | Y=[stop].
p1([X|X1],Y) :- X=<20 | add(X,1,A), Y=[A|Y1], p1(X1,Y1).

p2([stop],Y) :- true    | Y=[].
p2([X|X1],Y) :- X\=stop | multi(X,A), Y=[A|Y1], p2(X1,Y1).
                                    --------
:- or_relation multi/2.
multi(X,Y) :- square(X,Y).
multi(X,Y) :- cube(X,Y).

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

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

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

Pay attention to the goal Y=[A|Y1] in the second clause of $p2$. Receiving an input via the channel A from the process *multi* in a vector from, this process generates the vector of variables associated with its own color onto the channel $Y1$. However, each variable is not instantiated to a nonvariable term by this process but by the other process (its conjunctive goal) $p2$. It means that this process only creates a slot on each colored world and another process puts the values onto this slot. This goal is regarded as a special goal of arity 3 *make_Slot*($Y, A, Y1$), whose mode is (−,+,?).

3. COMPILATION

An *ANDOR-II* program is compiled into a KL1 program using a coloring scheme. OR-parallelism in *ANDOR-II* is realized by AND-parallelism in the object program, whereas intrinsic AND-parallelism among conjunctive goals is preserved. In this section, we will show the detail compilation techniques. The fundemantal idea is described in [Takeuchi et al. 87][Takeuchi et al. 88].

The main job of the compiler is an analysis of shell types and their creation. The cover of "shell" is put onto the goal that might receive a vector type data. There are some kinds of shells depending on the data type. Compiler analyzes the goal which needs a shell and its shell types. Each goal is transformed to the one covered with a proper shell.

In the following subsections, we will show how each clause is transformed and what type of shell is created.

3.1. Creation of an OR-manager

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

Consider the OR-predicate *pickup* appearing in the *Example 1*. The *Program 1* shows its transformed code (KL1). *pickup_Core_1* and *pickup_Core_2* correspond two definition clauses of *pickup*, respectively, and they are transformed similarly with AND-clauses. As KL1 has a modularity, *andor:goal* in the program denotes the call of *goal* in the module *andor*.

Program 1.

```
pickup_Core(X,Y,w(C),BP) :- true |
    BP=[BP0|BPs],                    % getting the ID of branching point
    BPs={BP1,BP2},                   % dividing branching point stream
    andor:set_Color(C,(c1,BP0),C1),  % refinement of the color
                                     %       for the first clause
    pickup_Core_1(X,Y1,w(C1),BP1),   % computation of the
                                     %       first clause
    andor:set_Color(C,(c2,BP0),C2),  % refinement of the color
                                     %       for the second clause
    pickup_Core_2(X,Y2,w(C2),BP2),   % computation of the
                                     %       second clause.
    andor:merge(Y1,Y2,Y).
```

In this example, two arguments are added to the predicate *pickup* in *ANDOR-II* . The third argument is for current color which is annotated with the goal, and the fourth is for the stream to obtain the identifier of branching point. For each branching point, a unique number should be assigned. However, the exeuction of OR-relation goals may happen in parallel in a distributed environment. Therefore, some manager is required who gives the identical number to each branching point. Manager expands streams called *branching point stream* to each process that may call an OR-predicate, and each process requests the manager to give its own number via this stream.

A manager is defined below.

Program 2.

```
bp_handler([BP|Strm],N) :- true |
        BP=N, N1:=N+1,
        bp_handler(Strm,N1).
bp_handler([],N) :- true | true.
```

The top level of the transformed code consists of the top level of *ANDOR-II* source program and this handler. For example, the top level of the transformed code of the *Example 2* is as follows.

Program 3.

```
simple_cycle_Top :- true |
        simple_cycle_Core(w([]),BP),
        bp_handler(BP,0).
```

3.2. Transformation of Clauses

Next, we explain the transformation of each clause. Roughly speaking, each clause is transformed according to the following rule.

(1) Add the argument for color information to the head goal and to all the body goals whose predicates are neither '=' nor ':='.

(2) Add the argument for branching point stream to the head goal and all the goals that may call an OR-predicate directly or indirectly.

(3) If the argument for branching point stream is added to more than two body goals, then add the body goal which divides branching point stream.

(4) Call body goals directly or with a cover of a shell, depending on the analysis.

The following example shows the transformed code of two clauses defining $p2$ in the *Example 2*. The second clause contains two shells: *make_Slot_NShell* and *p2_AShell*. In the next subsection, we will discuss about these shells.

Program 4.

```
p2_Core([stop],Y,w(C),BP) :- true |
    Y1=[],                          % single solution in this world
    Y=[v(Y1,C)],                    % solution Y1 is associated with
                                    %    its color C
    BP=[].                          % closing the branching point stream
p2_Core([X|X1],Y,w(C),BP) :- X\=stop |
    BP={BP1,BP2},                   % dividing branching point stream
    multi_Core(X,A,w(C),BP1),
    make_Slot_NShell(Y,A,Y1,w(C)),  % normal shell for
                                    %    a goal of cons cell
    p2_AShell(X1,Y1,w(C),BP2).      % abnormal shell for p2
```

3.3. Shell

Shell is created in order to handle a set of data when a nondeterminate process creates multiple solutions. Several types of shell are introduced. The shell for input vector is called "normal shell" and for output vector is called "abnormal shell." Considering a synchronization, "passing

shell" is introduced. The goals which need these shell types are determined by the static data flow analysis at a compile time, and for the goals whose I/O data type are dynamically determined at a runtime, "bilingal shell" is introduced.

3.3.1. Normal Shell

Normal shell is a basic shell structure which handles a set of input vectors. It decomposes a set of input vectors into a tuple of values, passes them to the corresponding core processes, and puts the output values together into a set of vectors again. Each core process corresponds to computation with one color. Core processes are executed in parallel. Some core processes may succeed and return solutions, while others may fail or deadlock and return no solution. In principle, solutions are put into output channels as soon as they are generated by fair merge operators, so that all the solutions are obtained without being disturbed by failure or deadlock in some worlds.

Program 5.

```
%% normal shell for add in the Example 1

add_NShell_2_1([v(X,Cx)|Xs],Y,Z,w(C0),BP) :-
    true |
    BP={ BP1,BP2 },                      % dividing branching point stream
    andor:consistent_Color([Cx,C0],R),   % check of color consistency
  ( R=success(C) ->
        add_NShell_2_2(X,Y,Z1,w(C),BP1) ;
    R=fail -> Z1=[], BP1=[] ),
    add_NShell_2_1(Xs,Y,Z2,w(C0),BP2),
    andor:merge(Z1,Z2,Z).                          % merging solutions
add_NShell_2_1([],_,Z,_,BP) :- true |  Z=[], BP=[].

add_NShell_2_2(X,[v(Y,Cy)|Ys],Z,w(C0),BP) :-
    true |
    BP={ BP1,BP2 },                     % dividing branching point stream
    andor:consistent_Color([Cy,C0],R),  % check of color consistency
  ( R=success(C) ->
    add_Core(X,Y,Z0,w(C),BP1),                % call of core process
    Z=[v(Z0,C)];                              % solution Z0 is
                                              %    associated with its
                                              %    color C
    R=fail -> Z=[], BP1=[] ),
    add_NShell_2_2(X,Ys,Z2,w(C0),BP2),
    andor:merge(Z1,Z2,Z).                       % merging solutions
add_NShell_2_2(_,[],Z,_,BP) :- true |  Z=[], BP=[].
```

The above example shows the normal shell for the process *add* in the *Example 1*. It has two input vectors via its first and second arguments. *add_Shell_2_1* and *add_Shell_2_2* extract the single data from these input vectors, respectively. If the colors associated with the both input data and the color associated with the goal itself are consistent with one another, then each pair is passed to its core process *add_Core*, otherwise, no data is generated.

3.3.2. Passing Shell

Normal shell is invoked with a set of input data in a vector form. It causes an unexpected synchronization, which does not exist in an *ANDOR-II* source program. Let us show an example.

```
:- mode coat_of_p1(+,-).
coat_of_p1(X,Y) :- true | X=[2|X1], p1(X,Y).
```

This clause is expected to commit immediately without synchanization. Consider the execution of the *Example 3* which is gained by replacing the clause *simple_cycle* shown in the *Example 2*.

Example 3.

```
%% Coated Simple Cycle

:- mode simple_cycle, p1(+,-), p2(+,-), multi(+,-),
        square(+,-), cube(+,-), add(+,+,-),
        coat_of_p1(+,-).

simple_cycle :- true | coat_of_p1(X,Y), p2(Y,X).

coat_of_p1(X,Y) :- true | X=[2|X1], p1(X,Y).

p1([X|X1],Y) :- X>20  | Y=[stop].
p1([X|X1],Y) :- X=<20 | add(X,A), Y=[A|Y1], p1(X1,Y1).

p2([stop],Y) :- true    | Y=[].
p2([X|X1],Y) :- X\=stop | multi(X,A), Y=[A|Y1], p2(X1,Y1).

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

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

At first, the cycle is expected to start when X is instatiated to $[2|X]$ by the goal of *coat_of_p1*. However, it is judged that x a variable X in *simple_cycle* might be bound to a vector, since *p2* calls an OR-predicate *multi*. Therefore, an input shell is put onto the goal *coat_of_p1* in *simple_cycle*. As a result, the core process of *coat_of_p1* is never invoked until its first argument is instantiated to a nonvariable term. *p2* is neither invoked until it first argument is instantiated to a nonvariable term. Neither *coat_of_p1* nor *p2* is invoked, and the computation of *simple_cycle* falls into deadlock. Actually, shell should not put onto the goal *coat_of_p1*, but onto the goal *p1* called from its body part. In another word, shell should be put onto the goal which has a definition clause *inspecting* the argument.

We put a virtual shell onto the goal receiving a vector, and real shell should be put onto the innermost goal which needs a synchronization. Virtual shell does nothing other than passing

the input data to the lower level. This shell is called "passing shell". The passing shell of the *cout_of_p1* in the *Example 3* is as follows.

Program 6.

```
coat_of_p1_Pass(X,Y,w(C),BP) :- true | X=[2|X1], p1_NShell(X,Y,w(C),BP).
```

Let us consider the behavior of the passing shell on an OR-predicate.

Example 4.

```
%%  Complicated Cycle

:- mode complicated_cycle, p1(+,-), p3(+,-), multi(+,-),
        square(+,-), cube(+,-), add(+,+,-),
        or_merge(+,+,-), distribute(+,-,-).

complicated_cycle :- true |
    p1([3|X1],Y1), p3([3|X2],Y2), or_merge(Y1,Y2,Y), distribute(Y,X1,X2).

p1([X|X1],Y) :- X>=20 | Y=[stop].
p1([X|X1],Y) :- X<20  | add(X,1,A), Y=[A|Y1], p1(X1,Y1).

p3([X|X1],Y) :- X>=20 | Y=[stop].
p3([X|X1],Y) :- X<20  | multi(X,A), Y=[A|Y1], p3(X1,Y1).

distribute([stop],Y1,Y2) :- true | Y1=[],Y2=[].
distribute([X|X1],Y1,Y2) :- X\=stop,X>=10 |
    Y1=[X|Y1s], distribute(X1,Y1s,Y2).
distribute([X|X1],Y1,Y2) :- X\=stop,X<10 |
    Y2=[X|Y2s], distribute(X1,Y1,Y2s).

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

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

:- or_relation or_merge/3.
or_merge([X|X1],Y,Z) :- Z=[X|Z1], or_merge(X1,Y,Z1).
or_merge([],    Y,Z) :- Z=Y.
or_merge(X,[Y|Y1],Z) :- Z=[Y|Z1], or_merge(X,Y1,Z1).
or_merge(X,[],    Z) :- Z=X.
```

or_merge in this example is a nondeterminate merge. *or_merge* can be invoked if at least either of first or second argument is instantiated to a nonvariable term. The first two clauses need a synchronization on the first argument, while the others need on the second argument. When *or_merge* is judged to have a passing shell, OR-manager is embedded in it while normal

OR-predicate has the OR-manager in the core process level.
The passing-shell of *or_merge* is shown below.

Program 7.

```
or_merge_Pass(X,Y,Z,w(C),BP) :- true |
            BP=[BP0|BPs],                 % getting the ID of the branching point
            BPs={BP1,BP2,BP3,BP4},        % dividing branching point stream
    andor:set_Color(C,(c1,BP0),C1),       % refinement of the color
                                          %      for the first clause
    or_merge_Pass_1(X,Y,Z1,w(C1),BP1),    % computation of the first clause
    andor:set_Color(C,(c2,BP0),C2),
    or_merge_Pass_2(X,Y,Z2,w(C2),BP2),
    andor:set_Color(C,(c3,BP0),C3),
    or_merge_Pass_3(X,Y,Z3,w(C3),BP3),
    andor:set_Color(C,(c4,BP0),C4),
    or_merge_Pass_4(X,Y,Z4,w(C4),BP4),
    andor:merge([Z1,Z2,Z3,Z4],Z).

or_merge_Pass_1(X,Y,Z,w(C),BP) :- true |
     or_merge_NShell_1(X,Y,Z,w(C),BP).        % NShell for the first argument
or_merge_Pass_2(X,Y,Z,w(C),BP) :- true |
     or_merge_NShell_2(X,Y,Z,w(C),BP).        % NShell for the first argument
or_merge_Pass_3(X,Y,Z,w(C),BP) :- true |
     or_merge_NShell_3(X,Y,Z,w(C),BP).        % NShell for the second argument
or_merge_Pass_4(X,Y,Z,w(C),BP) :- true |
     or_merge_NShell_4(X,Y,Z,w(C),BP).        % NShell for the second argument
```

or_merge_NShell_i($i = 1, 2, 3, 4$) is a normal shell which calls each corresponding clause as a core process†.

3.3.3. Abnormal Shell

Abnormal shell is a shell for handling output vectors.

See the *Example 2* again. In the second clause of *p2*, The goal Y=[A|Y1] receives an input via the channel *A* from the process *multi* in a vector form, so a normal shell is put onto this goal. This process generates the vector of variables associated with its own color onto the channel *Y1*. This goal is regarded as a special goal of arity 3 *make_Slot*. Normal shell for *make_Slot* is shown below.

Program 8.

```
make_Slot_NShell(Lout,[v(H,C)|Hs],Tout,w(Cp)) :-
        true |
        andor:consistent_Color([C,Cp],R),
          ( R=success(C1) ->
               L=[H|T],                   % core process
               T1=[v(T,C)],               % tail part with its color C
```

† As passing shell does nothing, these four definition clauses can be eliminated by unfolding.

```
            L1=[v(L,C)] ) ;              % cons-cell with its color C
          R=fail        -> L1=[],T1=[] ),
        make_Slot_NShell(L2,Hs,T2,w(Cp)),
        andor:merge(T1,T2,Tout),     %  vector of variables
        andor:merge(L1,L2,Lout).     %  output vector of cons-cells
make_Slot_NShell(L,[],T,_) :-
        true |
        T=[], L=[].
```

Therefore, the conjunctive goal $p2$ has an output channel which is instantiated to the vector of variables. Abnormal shell decomposes these output vectors into a world with its own color and checks its input whether it has a consistent color or not. The abnormal shell for $p2$ is as follows.

Program 9.

```
%%  abnormal shell for p2 in Example 2

p2_AShell(Xs,[v(Y,Cy)|Ys],w(C0),BP) :- true |   % C is the color for one stream
        BP={ BP1,BP2 },                         % dividing branching point stream
        andor:consistent_Color([Cy,C0],R),      % check whether
                                                %   the color with the input
                                                %   data is consistent with that
                                                %   of the prepared world
      ( R=success(C) ->
            p2_NShell(Xs,Y,w(C),BP1) ;
        R=fail -> Y1=[],BP1=[] ),
        p2_AShell(Xs,Ys,w(C0),BP2).
p2_AShell(Xs,[],_,BP) :- true | BP=[].
```

p2_NShell called from *p2_AShell* is defined in a normal way.

Program 10.

```
p2_NShell([v(X,Cx)|Xs],Y,w(C0),BP) :- true |
      BP={ BP1,BP2 },                       % dividing branching point stream
      andor:consistent_Color([Cx,C0],R),    % check whether the color
                                            %     with the input data is
                                            %     consistent with that of
                                            %     the prepared world
    ( R=success(C) -> p2_Core(X,Y1,w(C),BP1) ;
      R=fail -> Y1=[],BP1=[] ),
      p2_NShell(Xs,Y2,w(C),BP2),
      andor:merge(Y1,Y2,Y).
p2_NShell([],Y,_,BP) :- true | Y=[],BP=[].
```

p2_Core called from *p2_NShell* generates an output in a vector form again, which means a prepared slot in each colored world is stuffed with a vector again. This slot is created by a core process of *make_Slot* as the tail part of each cons-cell. Therefore, the output of the core process called from *make_Slot_NShell* is instantiated to a vector of streams whose head is an

integer and the remaining part is a vector. Therefore, the two channels X and Y connecting two processes in *simple_cycle* have data form of layered vector.

3.3.4. Bilingal Shell

We will prepare yet another shell. Consider again the *Example 2*. According to the above discussion, the input of core process of $p1$ is a vector of streams whose head is an integer and the remaining part is a vector. As the body goal $p1$ called from $p1$ recursively receives the remaining part, shell should be put onto this goal.

It is difficult to judge which goal has such an input, since it requires a compilcated data flow analysis crossing clauses. In general, the goal that has an argument of the tail of a stream might receive the vector type data, and we put the shell called *bilingal shell* onto those goals, since it can handle both scalar type and vector type data.

Biligal shell judges input data type at a runtime, and calls the coorresponding process. For example, the bilingal shell for a goal $p1$ called from $p1$ recursively is shown below.

Program 11.

```
p1_BI(X,Y,w(C),BP) :- true |
      andor:type(X,Type),                          % judgement of data type
    ( Type=vector ->  p1_NShell(X,Y,w(C),BP);      % calling the corresponding
                                                   %     shell
      Type=scalar ->  p1_Core(X,Y,w(C),BP) ).      % calling core process
```

3.3.5. Judgment of Shell Type

It is determined at a compile time which shell should be put on each goal. Multiple shells may be put onto a goal. In that case, passing shell is the outermost shell, and the order is passing shell, bilingal shell, abnormal shell and normal shell from the outside. Therefore, and normal shell is sometimes called from abnormal shell or from passing shell, and so on.

The basic rules for judgment of shell types is as follows:

First of all, the channel which may have a vector type data flow (namely, the variable appearing in *ANDOR-II* program which may bound to vector in the execution) is detected according to the following rule.

Rules (vector type channel)

(1) Variable appearing in an output or neutral argument of an OR-predicate is bound to vector.
(2) Variable appearing in an output or neutral argument of the predicate that calls OR-predicate directly or indirectly is bound to vector.
(3) Variable appearing in an output or neutral argument of the predicate which has a variable bound to vector in an input argument is bound to vector.

Shell types are judged using this result.

Rules (shell type)

(1) If a variable in an input argument of a goal G is bound to vector, and no definition clause of the corresponding predicate inspect the corresponding argument, then G has a *passing shell*.
(2) If a head goal has an input argument of [Car|Cdr] where Cdr is a variable, and a goal G has Cdr as an input argument, then G has a *bilingal shell*.
(3) If a variable in an output argument of a goal G also appears in an neutral argument of

another goal in the same clause, and this variable is bound to vector, then G has an *abnormal shell*.

(4) If a variable in an input argument of a goal G is bound to vector, then G has a *normal shell*

4. Optimization

Next, we will show some optimization technique used in the compiler.

4.1. Utilization of "shoen"

We utilize a convenient function of "shoen" in KL1. Shoen is a group of the execution of the program. It controls the execution such as start and abort jobs, manages the resource, and processes the failure case. Shoen can call children shoen recursively, and the failure in some shoen does not affect the outside. We make use of this function in prunning. If a conjunctive goal fails, all the siblings fails. They need not to be computed any more. On the other hand, a goal in OR-relation fails iff the computation in all the colored worlds fail. These treatment of failure is realized by shoen in the following way.

At a branching point, the world manager is created, which controls a set of shoens under it. A set of new shoens is created, each of which corresponds to each colored world and has a report stream to the world manager. Each shoen expands report streams to its child processes. If a conjunctive goal fails, then it sends a failure message to its parent shoen. Receiving the message, the parent shoen suicides with killing all their child processes. At the same time, it reports the failure to the world manager. If a goal fails in some world, failure information is passed to the world manager. If the world manager receives failure messages from every world forking at that branching point, then it suicides, sending a message to its upper manager. It means that the OR-predicate fails. The use of shoen prevents a failure on some world from affecting the computation on the other worlds.

4.2. Clustering

The compiler is based on coloring scheme. But in coloring scheme it is expensive to decompose/reconstruct vectors and to check color consistency. It is possible to combine several nodes together in a cluster which has a common shell.

See the *Example 1* again. It is easy to combine processes of AND-predicates such as *square*, *cube* and *add* in the same cluster in a naive manner. However, can we combine the process of OR-predicate *pickup* in this same cluster? We can do it if we apply a continuation-based compilation method in the cluster.

Continuation-based compilation technique is proposed by Ueda [Ueda 86c]. The main idea is constructing the data of the solution in a bottom-up manner. It does not suffer from the copy environment since a creation of variants when collecting solutions is unnecessary.

In this method, OR-parallelism in the original program is realized by AND-parallelism in the transformed program, while the intrinsic AND-parallelism is executed sequentially by passing a continuation around. Moreover, stronger restriction on the input/output data is imposed compared with stream-based compilation with colored vector†.

We adopt continuation-based method in a cluster. In another word, the processes to which continuation-based method can be applied are combined in the same cluster. Originally, continuation-based method is developed in order to make a Prolog program including nondeterminism determinate one. The restriction imposed on input/output in the original program is that they should be *ground terms*. The neutral argument in *ANDOR-II* clearly does not satisfy this condition. This occurs when more than two processes which have channels connect-

† Ueda also proposed the extension of continuation-based transformation to a nondeterminate program with coroutine.[Ueda 87] It is realized by statically determining scheduling of coroutines by compile-time analysis of a source program. However, in his method, only one communication channel is allowed for generator and tester, so the program such as Example 2 cannot be treated.

ing with each other constitute a loop structure. shown in *Example 2*. In our compiler, only the predicates that satisfy the condition can be combined as the cluster.

Clustering is performed at the beginning of the compile procedure. First of all, the compiler reads the source program and makes the basic data structure DFG(Data Flow Graph) and a common information table PL(Prediacte List). DFG is a graph representation for a clause in which nodes correspond to goals appearing in the clause and edges to shared variables. PL is a table which information such as mode and shell type is put on or get from by hashing with a key of a predicate name. Then extract the goals which can be bound in the cluster with a common shell.

We partition all the predicate in *ANDOR-II* into two classes, C-based and S-based, according to the following rule. C-based predicate is the one which can be combined in a cluster and continuation-based compilation is applied. S-based predicate is the one to which stream-based compilation using colored vector is applied.

Rule (partition of predicates)
(1-1)[loop constituent]
If a goal has such a variable that constitute a loop, the corresponding predicate of that goal is S-based.
(1-2)[downward propagation]
If a goal shares the variable appearing in output argument of the head goal which may satisfy the condition (1-1), the corresponding predicate is S-based.
(2-1)[neutral mode]
If a predicate other than *make_Slot_i* has a variable in a neutral argument, the predicate is S-based.
(2-2)[propagation to the writer]
If a predicate has the variable in an output argument which also appears in a neutral argument of another goal in the same clause, then the predicate is S-based.
(2-3)[propagation to the reader]
If a predicate has a variable in an input argument which also appears in an output argument of the goal whose predicate is S-based in the same clause, then the predicate is S-based.
(3)[upward propagation]
If a predicate has such a definition that there is a body goal which calls directly or indirectly a goal whose predicate is S-based is S-based.
(4) The predicate which satisfies neither (1),(2) nor (3) is C-based.

For example, *pickup*, *multi*, *square*, *cube* and *add* in *Example 1* is judged to C-based and they are combined into the same cluster.

C-based predicate satisfies the input/output restriction. Input/output argument of a C-based predicate in *ANDOR-II* can be regarded as a ground input/output in the meaning described in [Ueda 86c]. As for neutral mode, only a predicate of cons-cell creation, that is, *make_Slot_i* is posssibly judged to C-based. Let a variable in a neutral argument be V. If there is no goal that has the variable V in an output argument in the same clause, then V must be exported. It means that V appears in a neutral argument of the head goal and in this case, *make_Slot_i* is judged to S-based. If there is a goal that has the variable V in an output argument, it is the writer on that variable. Then V can be regarded as a ground input in the meaning described in [Ueda 86c], under the scheduling where *make_Slot_i* is performed after the writer process. In this case, *make_Slot_i* is judged to C-based.

In stream-based compilation a large class can be handled and massive parallelism can be induced in a object program, while continuation-based compilation provides higher efficiency.

Utilization of the continuation-based method in a part of the compiler preserves both advatages.

4.3. Other Techniques

There are some other techniques to increase efficiency.

(1) Runtime support

Actual transformed code includes calls of runtime support routine such as core, pass, shell and so on. This prevents the object code from becoming exravagant, and provides an ease of maintainence.

(2) String realization of color

Check of color consistency is expensive and it happens frequently. To reduce the overhead, color is actually implemented as a fixed-length string. Thus, it needs a constant time at each check.

5. DISCUSSION

5.1. Performance Evaluation

The compiler runs on multi-PSI, and the transformed code in KL1 can be compiled by KL1 compiler. Multi-PSI is a machine with multiple processor elements developed at ICOT. However, it is running only on a single processor with 12MW memory, since the current operating system on Multi-PSI do not support an automatic processor allocation. Currently, neither compiler and the object code has a part which imposes explicit allocation on multi processors. We show below the result of the execution of the example 1 and fault dignosis of a circuit of half adder which is shown in the appendix.

Compute(Example 1)

Length of Input Data	CPU time (ms)	number of solutions	number of reductions
10	173	10	1581
50	832	50	10357
100	1935	100	28356

Fault Diagnosis of a Circuit(Appendix)

```
Input data :
  (a) ha( [ [1,'?'],   [1,0] ], Answer )
  (b) ha( [ ['?','?'], [1,0] ], Answer )
  (c) ha( [ ['?','?'], [0,0] ], Answer )
```

Input Data	CPU time (ms)	number of solutions	number of reductions
(a)	204	6	2353
(b)	397	12	4839
(c)	389	4	4815

5.2. Comparison with Other Works

Design and implementation of a new language which has both features of AND-parallelism and OR-parallelism are studied intensively.

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

Clark and Gregory pointed out the importance of further research in these areas and they suggested the combination of PARLOG and Prolog [Clark and Gregory 87]. One of the outcome is a new language Pandora [Bahgat and Gregory 89] whose semantics is based on an Andorra model.

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

Naish proposed a parallel NU-Prolog which is an extension of NU-Prolog [Naish 88]. It can express AND-parallelism together with nondeterminism. A nondeterminate code can call AND-parallel code which (in restricted way) can call nondeterminate code. However, nondeterminism is only handled sequentially.

Saraswat handles both parallelism in a framework of constraint. He proposed a language cp(!,|,&) as one of the CP family [Saraswat 87]. It employs an eager invocation of nondeterminate forks. The basic idea resembles to our system: each possible selection of a clause is expressed in the binding itself as a constraint while it is expressed as a color associated to the data in our system; and early cut of failure world by constraint solver is realized by shoen mechanism in *ANDOR-II* .

Program transformation from a nondeterminate program to a determinate program which collects all the solutions is also intensively studied [Ueda 86b][Tamaki 86][Okumura and Matsumoto 87] [Bansel and Sterling 88]. A good result can be obtained using the proposed techniques for a certain class of programs such as generate-and-test type programs.

In comparison, the main characteristics of *ANDOR-II* can be said as follows. *ANDOR-II* has no execution control other than OR-relation declaration and mode declaration. Execution controls such as *when*, *delay*, *where* and so on are subrogated by various types of shell. It makes a description language simple and friendly. In *ANDOR-II* , eager fork of nondeterminate goals can explore fine grain paralelism, at the same time, it may cause a combinatorial explosion. However, we will take advatage in the execution on multi-processor machine, rather than the execution by lazy fork with a lot of suspensions.

6. CONCLUDING REMARKS

To sum up, our contribution is as follows:

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

There are some remaining problems for future works.

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

Another issue is the realization with processor allocation. KL1 is embedded as a firmware of multi-PSI machine, We can expect a much more efficiency by putting a good allocation both on compiler and object code. A version with processor allocation is now under development.

ACKNOWLEDGMENTS

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

REFERENCES

[Bansel and Stering 88] Bansel A.K. and L.S.Sterling, "Compiling Enumerable-and-Filter Programs for Efficient Execution under Committed-Choice AND-Parallelism," Proc. of Internatinal Conference on Parallel Processing, pp.22-26,1988.

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

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

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

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

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

[Haridi and Janson 89] Haridi,S. and S.Janson, "Kernel ANDORRA Prolog and Its Computation Model," SICS Research Report, 1989.

[Kliger et al. 88] Kliger,S., E.Yardeni, K.Kahn and E.Shapiro, "The Language FCP(:,?)," Proc. of Internatinal Conference on Fifth Generation Computer Systems, pp.763-773, 1988.

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

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

[Saraswat 88] Saraswat,V., "CP as a General-purpose Constraint-Language," Proc. of National Conference on Artificial Intelligence, pp.53-58, 1987.

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

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

[Takeuchi et al. 89] Takeuchi,A., "On Parallel Problem Solving Language and Its Applicaion," ICOT TR-418, 1988, also in Concepts and Characteristics of Knowledge-based Systems. M.Tokoro,Y.Anzai and A.Yonezawa(eds.), North-Holland, 1990.

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

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

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

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

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

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

APPENDIX

```
%% Fault diagnosis of the Circuit of Half Adder
%%
%%   A circuit contains a faulty element.
%%   Given unknown input and observed output,
%%       find a possible faulty elements.

:- module ha.
:- public ha/2.

:- mode ha(+,-),                    circuit(+,-),
    element(+,+,+,+,-,-),           elementDoubt(+,+,+,-,-),
    primitiveElement(+,+,+,-),      inverter(+,+,-),
    and2(+,+,-),                    or2(+,+,-),
    setInput(+,-),                  setIn(+,-),
    setUnknownIn(-),                checkOutput(+,+,+,-),
    check(+,+,+,-).

%------    top level predicate definition ------------------------------

ha([DataIn,DataOut],Answer) :- true |
    setInput(DataIn,Input),
    circuit(Input,Output),
    checkOutput(Input,Output,DataOut,Answer).

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

circuit([In1,In2],Output) :- true |
    element(inv1,inverter,correct,[In1],Out1,_),
    element(inv2,inverter,correct,[In2],Out2,_),
    element(and1,and2,doubt,[In1,Out2],Out3,State1),
    element(and2,and2,doubt,[Out1,In2],Out4,State2),
    element(and3,and2,doubt,[In1,In2],Out5,State3),
    element(or1,or2,correct,[Out3,Out4],Out6,_),
    Output=[[Out6,Out5],[State1,State2,State3]].

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

element(Name,Type,doubt,Input,Output,State) :- true |
    elementDoubt(Name,Type,Input,Output,State).
element(Name,Type,State,Input,Output,State1) :- State\=doubt |
    primitiveElement(Type,State,Input,Output),
    State1=(Name,State).

:- or_relation elementDoubt/5.
elementDoubt(Name,Type,Input,Output,State) :-
    element(Name,Type,correct,Input,Output,State).
```

```
elementDoubt(Name,Type,Input,Output,State) :-
    element(Name,Type,error,Input,Output,State).

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

primitiveElement(inverter,State,Input,Output) :- true |
    inverter(State,Input,Output).
primitiveElement(and2,State,Input,Output) :- true |
    and2(State,Input,Output).
primitiveElement(or2,State,Input,Output) :- true |
    or2(State,Input,Output).

inverter(correct,[0],Out) :- true | Out=1.
inverter(correct,[1],Out) :- true | Out=0.
inverter(error,[0],Out) :- true | Out=0.
inverter(error,[1],Out) :- true | Out=1.

and2(correct,[0,0],Out) :- true | Out=0.
and2(correct,[0,1],Out) :- true | Out=0.
and2(correct,[1,0],Out) :- true | Out=0.
and2(correct,[1,1],Out) :- true | Out=1.
and2(error,[0,0],Out) :- true | Out=1.
and2(error,[0,1],Out) :- true | Out=1.
and2(error,[1,0],Out) :- true | Out=1.
and2(error,[1,1],Out) :- true | Out=0.

or2(correct,[0,0],Out) :- true | Out=0.
or2(correct,[0,1],Out) :- true | Out=1.
or2(correct,[1,0],Out) :- true | Out=1.
or2(correct,[1,1],Out) :- true | Out=1.
or2(error,[0,0],Out) :- true | Out=1.
or2(error,[0,1],Out) :- true | Out=0.
or2(error,[1,0],Out) :- true | Out=0.
or2(error,[1,1],Out) :- true | Out=0.

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

setInput([DataIn|DataIns],Input) :- true |
    setIn(DataIn,In),
    setInput(DataIns,Input1),
    Input=[In|Input1].
setInput([],Input) :- true | Input=[].

setIn(DataIn,In) :- DataIn='?' | setUnknownIn(In).
setIn(DataIn,In) :- DataIn\='?' | In = DataIn.

:- or_relation setUnknownIn/1.
setUnknownIn(In) :- true | In=0.
setUnknownIn(In) :- true | In=1.
```

```
%-----------------------------------------------------------------------

checkOutput(Input,[Output,State],DataOut,Answer) :-
    true | check(Output,DataOut,[Input,Output,State],Answer).

check([O|Output],[D|DataOut],RET,Answer) :- O = D |
    check(Output,DataOut,RET,Answer).
check([O|Output],[D|DataOut],RET,Answer) :- O \= D |
    Answer = [].
check([],[],RET,Answer) :- true | Answer = RET.
```