ICOT Technical Report: TR-184

TR-184

On Parallel Programming Methodology in GHC

by
K. Takahashi and T. Kanamori
(Mitsubishi Electric Corp.)

May, 1986

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

On Parallel Programming Methodology in GHC

— Experience in Programming of A Proof Procedure of Temporal Logic —

Kazuko TAKAHASHI Tadashi KANAMORI
Central Research Laboratory, Mitsubishi Electric Corporation

ABSTRACT

Parallel programming methodology in GHC is discussed based on our experience in programming of a proof procedure of temporal logic. It is said that GHC can express basic constructs of parallel processing such as communication and synchronization very simply, but we have not yet had enough experiences of parallel programming in GHC. By programming a proof procedure of temporal logic in Prolog and GHC, we compare the thinking style in sequential programming and that in parallel programming. Parallel programming methodology is discussed based on the experience.

1. INTRODUCTION

"Guarded Horn Clauses (GHC)" is a language designed for execution on highly parallel architecture [Ueda 85] and regarded as the core of the Kernel Language One (KL1) of the Fifth Generation Computer System (FGCS) project in Japan. GHC is a descendant of other Prolog-like parallel programming languages Concurrent Prolog [Shapiro 84] and PARLOG [Clark and Gregory 84]. It is said that GHC not only provides us with the basic functions for parallel processing such as communication and synchronization but also imposes less burden of implementation than Concurrent Prolog such as multiple environments. But we have not yet had enough experience of parallel programming in GHC. Especially,we don't yet know whether GHC gives us enough expressive power in practice and what transition of programming style is necessary for GHC.

In this paper, we show our experience in programming a proof procedure of temporal logic in GHC. The proof procedure,called ω-graphs refutation,was familiar with us before programming it in GHC [Fusaoka and Takahashi 85]. We had its sequential implementation in Prolog. Fortunately or unfortunately, the sequential version contains subprocedures which embody three typical styles of programs. The first one is general recursive style, the second one is repetitive (tail-recursive) style and the third one is backtracking style. We show what difficulties we have encountered in programming these procedures in GHC and disccuss the parallel programming methodology based on the experience.

2. PRELIMINARIES

A GHC program is a finite set of Horn clauses of the following form ($m \geq 0, n \geq 0$) :

$$H :- G_1, ..., G_m \mid B_1, ..., B_n$$

where G_i's and B_i's are atomic formulas defined as usual. The part of the clause before '|' is called a *passive part* or a *guard*, and the part after '|' is called an *active part*.

Informally execution of a clause is done in the following manner : When a goal is called, the clauses whose heads are unifiable are invoked. Execution of the goals in the passive part of these candidate clauses are tried in parallel, and if goals in the passive part of some clause succeed, then the clause is *trusted* and the active part of the clause is executed. Any piece of unification invoked in the passive part of a clause cannot instantiate a variable appearing in the caller.

Temporal Logic [Manna and Pnueli 81] is an extension of first order logic to include a notion of time and deal with logical description and reasoning on time. It is a branch of *modal logic* [Hughes and Cresswell 68], in which the relation between worlds is considered as a temporal one. The temporal logic we consider in this paper is a propositional one called Propositional Temporal Logic (PTL). Three temporal operators used in PTL have the following intuitive meanings :

$\Box F$ (always F) : F is true in all future instants
$\Diamond F$ (eventually F) : F is true in some future instant
$\bigcirc F$ (next F) : F is true in the next instant

For example, a formula $\Box\Diamond P$ indicates that P will be true infinitely often.

Let F be a formula of PTL. A *complete assignment* for F is a function which assigns truth value (t or f) to every propositinal variable in F. A *model* M for F is an infinite sequence of complete assignments for F

$$K_0, K_1, K_2, ...$$

F is said to be *true(false) in* M if F is assigned t(f) by K_0. F is *satisfiable* if there exists a model in which F is true. F is *valid* if it is true in every model.

Consider the following graph. Intuitively, the edges in the graph correspond to complete assignments. The edge from the node N_0 to the node N_1 corresponds to a complete assignment that assigns t to P, the edge from N_1 to N_2 corresponds to one that assigns f to P and the edge from N_2 to N_1 corresponds to one that assigns t to P. Then, the infinite path of $N_0 \rightarrow N_1 \rightarrow N_2 \rightarrow N_1 \rightarrow N_2 \rightarrow N_1 \rightarrow N_2 \rightarrow ...$ corresponds to a model for $\Box\Diamond P$ where the assignment for P is the sequence in which t and f appear alternately (t,f,t,f,t,f,...).

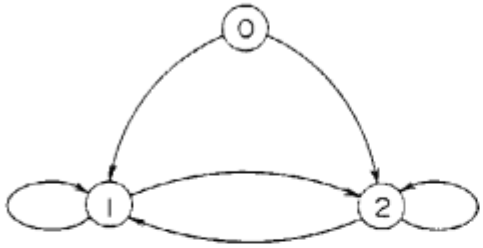

Figure 1 Graph Representation for □◇P

3. ω- GRAPHS REFUTATION PROCEDURE

An *ω-graph* is a graph in which each node is labeled with an expression called *node formula*. When a formula of PTL is given, the ω-graphs refutation procedure shows whether the formula is valid or not as follows.

(1) Negate the given formula.
(2) Compute initial node formula of the negation of given formula
(3) Construct an ω-graph by starting from the initial ω-graph consisting of only one node corresponding to initial node formula and successively expanding nodes in the ω-graph.
(4) Check ω-loop freeness of the constructed ω-graph. If it is ω-loop free,the given formula is valid.

First, we explain each procedure, *compute_initial_node_formula*, *expand_node_formulas*, *construct_omega_graphs* and *check_omega_loop_freeness* and discuss parallel programming for each procedure. Then, we show the top level implemenatation.

3.1. Computation of Initial Node Formula

The negation of a given formula is once converted to its negation normal form in order to compute initial node formula before constructing the ω-graph.

Let G be a formula obtained from a formula F by applying the rules below as far as possible. Then G is called a *negation normal form* of the formula F.

[Rule NNF1] remove implication and equivalence

$A \supset B \implies \neg A \vee B$

$A \equiv B \implies (A \wedge B) \vee (\neg A \wedge \neg B)$

[Rule NNF2] move negation inwards

$\neg(A \wedge B) \implies \neg A \vee \neg B$

$\neg(A \vee B) \implies \neg A \wedge \neg B$

$\neg\Box A \implies \Diamond\neg A$

$\neg\Diamond A \implies \Box\neg A$

$\neg\bigcirc A \implies \bigcirc\neg A$

$\neg\neg A \implies A$

For example, $\neg\Diamond\Box\neg P$ is converted into $\Box\Diamond P$ by applying NNF2. Negation normal forms are unique. Note that the negation normal form of a formula is valid if and only if the original formula is valid.

Later,we need to check whether we have constructed an ω-graph corresponding to a set of models of PTL in which the eventualities in a negation normal form are satisfied. We compute the eventuality set of the formula F_0 in a negation normal form in order to use it at that time. The set of all subformulas G such that $\Diamond G$ is a subformula of F_0 is called *eventuality set of* F_0. For example, the eventuality set of the formula $\Box\Diamond P$ is $\{P\}$.

Let F be a formula. *An initial node formula* $[F_0]_{\{\}}$ of F is a formula suffixed by { }, where F_0 is in a negation normal form of F. The top level of the procedure *compute_initial_node_formula* in GHC is implemented as follows.

```
compute_initial_node_formula(F,NNF,EveSet) :- true |
    negation_normal_form(F,NNF),
    compute_eventuality_set(NNF,EveSet).
negation_normal_form(F,G) :- true |
    remove_implication_and_equivalence(F,F0),
    move_not_inwards(F0,G).
```

Three subprocedures *remove_implication_and_equivalence*, *move_not_inwards* and *compute_eventuality_set* were written in a general recursive style in Prolog. They are easily transformed to GHC programs with a few syntactical modifications.

3.2. Expansion of Node Formulas

An ω-graph is a graph whose nodes are labelled with expressions called *node formula*.

Let ES_0 be the given fixed eventuality set. A node formula $[F]_H$ is a formula F suffixed by a subset H of ES_0, where F is in a negation normal form and may contain $\Diamond^*$ in stead of $\Diamond$ and the suffix H is called *history set*. For example, $[\Box\Diamond P]_{\{\}}$, $[\Box\Diamond P]_{\{P\}}$ and $[\Diamond^* P \wedge \Box\Diamond P]_{\{\}}$ are node formulas. $\Diamond^* G$ is semantically identical to $\Diamond G$. $\Diamond^* G$ is called *marked formula*. A node formula $[F]_H$ is logically equivalent to the formula F. An initial node formula is a special node formula. The intuitive meanings of the mark and the history sets will be explained later.

In our proof procedure, we construct the ω-graph of F_0 by starting from an initial ω-graph and successively expanding nodes in the graph. It is based on the tableau method [Wolper 81]. Suppose we are trying to expand a node labeiled with $[F]_H$. New node formulas are obtained by converting F to its next prefix form F_{NPF} and then converting F_{NPF} to its disjunctive normal form F_{DNF} and expanding F_{DNF}. Validity of the formulas is kept throughout these transformations.

Let F be a formula in a negation normal form and G be a formula obtained from F by applying the rules below as far as possible to subformulas not inside the next operator ○, then G is called *next prefix form* of F.

[Rule NPF1] postpone □

$\bigcirc A \quad \Longrightarrow A \wedge \bigcirc\Box A$

[Rule NPF2] postpone $\circ$ and $\circ^*$

$\Diamond A \quad \Longrightarrow A \vee \bigcirc\circ^* A$

$\circ^* A \quad \Longrightarrow A \vee \bigcirc\circ^* A$

For example, a formula $\Box\Diamond P$ is converted into $(P \vee \bigcirc\circ^* P) \wedge \bigcirc\Box\Diamond P$ by applying NPF1 first and then NPF2.

Let F be a formula in a next prefix form and G is a formula in the form

$(E_1 \wedge \bigcirc F_1) \vee (E_2 \wedge \bigcirc F_2) \vee \ldots \vee (E_m \wedge \bigcirc F_m)$

where $E_1, E_2, \ldots, E_m$ and $F_1, F_2, \ldots, F_m$ are formulas other than *false*. $E_1, E_2, \ldots, E_m$ do not contain temporal operators, and F_i and F_j are not literally identical if $i \neq j$. If G is obtained from F by applying the following rules to subformulas of F as far as possible, then G is said to be a *disjunctive normal form of* F. (Note that each F_i is in a negation normal form.)

[Rule DNF1] distribute $\wedge$ over $\vee$

$A \wedge (B \vee C) \Longrightarrow (A \wedge B) \vee (A \wedge C)$

$(A \vee B) \wedge C \Longrightarrow (A \wedge C) \vee (B \wedge C)$

[Rule DNF2] eliminate simple contradictions and duplication

[Rule DNF3] supplement next part

Current formula A is in the form $C_1 \vee C_2 \vee \ldots \vee C_n$ where each C_i is a conjunction of literals or the formula whose outermost operator is $\bigcirc$.

$C_i \Longrightarrow C_i \wedge \bigcirc\Box true$ where C_i is a conjunction of literals (i.e.including no $\bigcirc$-formulas)

[Rule DNF4] ordering in each conjunction

$\bigcirc B \wedge A \Longrightarrow A \wedge \bigcirc B$ where A isn't in the form of $\bigcirc A'$

[Rule DNF5] merge $\bigcirc$ in each conjunction

$\bigcirc A \wedge \bigcirc B \Longrightarrow \bigcirc(A \wedge B)$

[Rule DNF6] combination by next part

$(A \wedge \bigcirc C) \vee (B \wedge \bigcirc C) \Longrightarrow (A \vee B) \wedge \bigcirc C$

For example, $(P \vee \bigcirc\circ^* P) \wedge \bigcirc\Box\Diamond P$ is converted into $(P \wedge \bigcirc\Box\Diamond P) \vee \bigcirc(\circ^* P \wedge \Box\Diamond P)$ by applying DNF1 first and then DNF5.

Expansion of node formulas is the basic operation in constructing ω-graphs, and defined by using next prefix forms and disjunctive normal forms as follows.

Let ES_0 be the given fixed eventuality set, $[F]_H$ be a node formula and

$(E_1 \wedge \bigcirc F_1) \vee (E_2 \wedge \bigcirc F_2) \vee \ldots \vee (E_m \wedge \bigcirc F_m)$,

be a disjunctive normal form of the next prefix form of F. Then $[F_i]_{H_i}$ is *an expansion* of $[F]_H$ if and only if

$$H_i = \begin{cases} ES_0 - ES_i & \text{if } H = ES_0 \\ (ES_0 - ES_i) \bigcup H & \text{otherwise} \end{cases}$$

where $ES_i = \{G \mid \circ^* G \text{ is a subformula of } F_i\}$.

$\circ^* G$ denotes that the realization of G is postponed in that expansion. Each H_i is called *history set*(*h-set*, in short) *of* F_i. It is introduced to ensure that eventuality will actually be realized in expanding nodes succesively in the construction of an ω-graph. Each element of h-set indicates the history of the realization in a sequence of expansions.

The expansions of the node formula $[\Box\Diamond P]_{\{\}}$ where the eventuality set is $\{P\}$ are $[\Box\Diamond P]_{\{P\}}$ and $[\circ^* P \wedge \Box\Diamond P]_{\{\}}$.

We discuss parallel programming of the procedure *expand_node_formulas*. As it is rather complicated, we divide it into the following three subprocesses which run in parallel : (1) convertion from NNF to NPF (2) convertion by using DNF1 (3) conversion by using DNF2~DNF6 and compute h-sets. The top level of expansion of node formulas is implemented as follows.

```
expand_node_formulas(F,H,Xn,ES0,NodeFormulas) :-
    true |
    next_prefix_form(F,NPF),
    distribute_and_over_or(NPF,DNF1),
    simplify_formulas(ES0,H,Xn,DNF1,NodeFormulas).
```

3.3. Construction of ω-Graphs

Let F_0 be a formula obtained by converting the negation of a given formula to its negation normal form and ES_0 the eventuality set of F_0. An *ω-graph* of F_0 is the minimum graph satisfying the following conditions.

(1) Each node is labelled with different node formulas.

(2) There is a special node N_0 called *initial node* labelled with $[F_0]_{\{\}}$.

(3) When there exists a node N labelled with $[F]_H$ and $[F_1]_{H_1}, [F_2]_{H_2}, \ldots, [F_m]_{H_m}$ are all expansions of $[F]_H$, there exist m nodes $N_1, N_2, \ldots, N_m$ labelled with $[F_1]_{H_1}, [F_2]_{H_2}, \ldots, [F_m]_{H_m}$ and m directed edges from N to $N_1, N_2, \ldots, N_m$.

In parallel programming of construction of the ω-graph of a formula, each node is considered as a process and we try to execute expansion of each node formula in parallel. Since new node formulas are generated as an output stream of each node process at the same time, it is impossible for each node process to decide whether the node formula is an existing one or not. It is necessary the system to introduce some graph manager which controls all node formulas. It creates a *node_process* if a new node formula is generated and aborts it if the expansion of the node formula is over. It also stores a current list of node formulas and checks whether newly generated node formula is an existing one or not. If all node formulas are expanded then *graph_manager* terminates.

A graph is represented as a list of quadruples $(NodeNmbr, OutStrm, InStrms, NodeType)$ where $NodeNmbr$ is the associated node number, $OutStrm$ is the associated stream variable, and $NodeType$ is either *omega* or *not_omega*. $InStrms$ is a list of the stream variables associated with the node which has an edge flowing into the node $NodeNmbr$. Thus, at the end of *construct_omega_graphs* procedure, *graph_manager*

generates an output in this form.

For example, the ω-graph of a formula $\Box\Diamond P$ is represented as :

```
[ (0,  X0,  [ ],          not-omega),
  (1,  X1,  [X0,X1,X2], omega),
  (2,  X2,  [X0,X1,X2], not-omega) ]
```

and it is shown in Figure 1.

Parallel Construction of An ω-Graph of F_0

Let F_0 be a given formula in NNF.

(1) Create the processes of graph manager GM, multiplexer MUX, and node process NP_0 corresponding to the node formula $[F_0]_{\{\}}$. Initialize *Exist* to { } and *Graph* to { }. For each process, do the following (2).

(2) NP : For each node process NP, let NF be the corresponding node formula. Do the followings.

Expand NF. Assume that $NF_1, ..., NF_k$ are the node formulas generated from NF .

For each i, send MUX a pair of (NF_i, X) where X is the stream variable corresponding to NF_i.

MUX : Merge the input streams into *MrgdStrm* and send it to GM. If every stream gets to the end_of_stream, it terminates.

GM : Repeat the following procedure until *MrgdStrm* is { }.

Take a node formula NF_i from *MrgdStrm*.

If NF_i is a member of *Exist*, then send a message to the corresponding NP_i (as a result, X is added to the tail of *InStrms* of NF_i).

If NF_i is not a member of *Exist*, then register NF_i to *Exist* as a new node formula, and create the corresponding node process NP_i. (the head of the *InStrms* of NP_i is X). Add the node to *Graph*.

This procedure terminates in a finite steps, since there exist only a finite number of node formulas generated from F_0.

The top level of parallel construction of ω-graphs is implemented as follows, while it is in a repetitive style in sequential version.

```
construct_omega_graphs(JudgeStop,ES0,F,Graph) :-
    true |
    Exist=[[No0,X0,I0,F]|NewExist],
    Graph=[[No0,X0,I0,not-omega]|NewGraph],
    node_process(JudgeStop,ES0,0,Exist,StrmList),
    multiplexer(JudgeStop,StrmList,MrgdStrm),
    graph_manager(JudgeStop,ES0,0,MrgdStrm,Graph,Exist).
```

3.4. Check of ω -Loop Freeness

Let F_0 be a formula in a negation normal form and ES_0 its eventuality set. A node N labelled with $[F]_H$ in the ω-graph of F_0 is called *ω-node* when $H = ES_0$. The loop which starts from an ω-node W and returns to the same ω-node W is called *ω-loop of* W. (A loop may pass through several nodes). If there is no ω-loop, then the graph is said to be *ω-loop free*. For example, in Figure 1, N_1 is an ω-node, $N_1 \rightarrow N_1$ and $N_1 \rightarrow N_2 \rightarrow N_1$ are ω-loops of N_1.

As we gave an intuitive explanation in example in section 2, some infinite paths in the ω-graph of F_0 correspond to models of F_0. Moreover, we can show that ω-graph of F_0 is not ω-loop free if there is a model of F_0. Hence, ω-loop freeness of the ω-graph indicates that F_0 is not satisfiable.

In Prolog programming, for an ω-node, every path outgoing from that node is checked one by one via backtracking mechanism. Because GHC has no backtracking mechanism, we have to change the algorithm for GHC program. We use a programming technique similar to one in [Shapiro 83]. Each node is considered as a process sending messages each other. Moreover, extra argument 'Judgestop' is added as a termination flag. It can stop other processes as soon as an ω-loop is found.

Parallel Check of ω-Loop Freeness of ω-Graphs

(1) For each node, instantiate the head of X_N by its node number N and add *InStrms* to the tail of X_N. (As a result, *InStrms* becomes to the list of paths flowing into that node. The length may be infinite.)

(2) For an ω-node whose node number is N, initialize *CGraph* to the set of all nodes in the ω-graph and *RecMes* as [ ], and repeat the following (3).

(3) If *CGraph* = { }, then stop with failure.

If *JudgeStop* = *stop*,

then stop with the answer "There exists an ω-loop".

Otherwise, assume that *InStrms* is in the form of [[A|X]|Paths], then do (3)-1 and (3)-2 in parallel.

(3)-1 If $N = A$, then set *JudgeStop* to 'stop'.

If $N \neq A$, then append [A] to *RecMes*, extract a node from *CGraph*.

(3)-2 Let *InStrms* be *Paths* and repeat (3).

If all the processes for the nodes stop with failure, then answer "The graph is ω-loop free."

```
check_omega_loop_freeness(JudgeStop,Graph) :- true |
    check_omega_loop_freeness(JudgeStop,Graph,Graph).

check_omega_loop_freeness(_,_,[ ]).
check_omega_loop_freeness(stop,_,_).
check_omega_loop_freeness(JudgeStop,CGraph,
                          [[N,Xn,INs,omega]|Gs]) :-
    prolog(var(JudgeStop)) |
    Xn=[N|Xn1], Xn1=INs,
    find_omega_loop(JudgeStop,CGraph,[ ],N,Xn1),
    check_omega_loop_freeness(JudgeStop,CGraph,Gs).
check_omega_loop_freeness(JudgeStop,CGraph,
                          [[N,Xn,INs,not-omega]|Gs]) :-
    prolog(var(JudgeStop)) |
    Xn=[N|Xn1], Xn1=INs,
    check_omega_loop_freeness(JudgeStop,CGraph,Gs).
```

3.5. ω-Graphs Refutation Procedure

Lastly in this section, we show the top level of the parallel ω-graphs refutaion procedure for checking the validity of the given formula. In the program, '&' denotes the sequential execution.

```
prove(F) :- true |
    refute(not(F),JudgeStop) &
    write_answer(JudgeStop,F).

refute(F,JudgeStop) :- true |
    compute_initial_node_formula(F,F0,ES0),
    construct_omega_graphs(JudgeStop,ES0,(F0,[ ]),Graph),
    check_omega_loop_freeness(JudgeStop,Graph).

write_answer(stop,F) :- true |
    pretty_print(F), pretty_print('is valid').
write_answer(JudgeStop,F) :- prolog(var(JudgeStop)) |
    pretty_print(F), pretty_print('is not valid').
```

Refute consists of three parallel processes each of which again consists of many parallel processes. Note that *construct_omega_graphs* and *check_omega_loop_freeness* have a common variable *JudgeStop*. It is set to 'stop' in order to terminate the graph construction if an ω-loop is found in the *check_omega_loop_freeness* process. Although the program contains meta-predicate *var* and sequential AND '&' for simplicity, it is possible to implement without these elements. Our program consists of about 500 lines in total.

4. PARALLEL PROGRAMMING METHODOLOGY IN GHC

In this section, we discuss on parallel programming methodology in GHC.

4.1. General Principles for Enhancing Concurrency

First, we discuss general principles for enhancing concurrency.

Information should be made public to other processes as soon as it is fixed in one process.

```
negation_normal_form(F,G) :- true |
    remove_implication_and_equivalence(F,F0),
    move_not_inwards(F0,G).

remove_implication_and_equivalence(imply(F,G), A) :-
    remove_implication_and_equivalence(F,F1),
    remove_implication_and_equivalence(G,G1) |
    A=or(not(F1),G1).
```

In this program, publication of A must wait until both *remove_implication_and_equivalence*$(F, F1)$ and *remove_implication_and_equivalence*$(G, G1)$ succeed. As far as A is not yet instantiated to a non-variable term, the head unification of *move_not_inwards* is suspended. Without this '|', three processes can run in parallel so that A is propagated as soon as $A = or(not(F1), G1)$ is executed, which allows the head unification of *move_not_inwards*. It gives high concurrency.

Each process should run independently as far as possible without being suspended by the delay of another process, even if they share common variables.

```
union([X|S1],S2,L1,L2,S) :- member(X,L2,yes) |
    union(S1,S2,L1,L2,S).
union([X|S1],S2,L1,L2,S) :- member(X,L2,no) |
    S=[X|NewS], union(S1,S2,[X|L1],L2,NewS).
union(S1,[X|S2],L1,L2,S) :- member(X,L1,yes) |
    union(S1,S2,L1,L2,S).
union(S1,[X|S2],L1,L2,S) :- member(X,L1,no) |
    S=[X|NewS], union(S1,S2,L1,[X|L2],NewS).
union([ ],S2,L1,L2,S) :- true | S=S2.
union(S1,[ ],L1,L2,S) :- true | S=S1.

member(X,[Y|S],Answer) :-
    X=:=Y | Answer=yes.
member(X,[Y|S],Answer) :-
    X=\=Y | member(X,S,Answer).
member(X,[ ],Answer) :- true | Answer=no.
```

In this program, the third and the fourth arguments in *union*, which accumulate the set elements already output so far, are always completely instantiated to lists. Hence, the unification of *member* in the passive part is never suspended, In general,in order to realize early commitment, predicates in the passive part should be written so that the clause is trusted even if the shared variables are partially instantiated.

Decision should be done in the distributed manner as far as possible if it does not increase the overall communication cost excessively.

```
bounded_buffer_communication :- true |
    produce(0,100,H), buffer(N,H,T), consume(H,T).

produce(N,Max,[M|L]) :- N < Max |
    M=N, N1:=N+1, produce(N1,Max,L).
produce(N,Max,[M|_]) :- N >= Max | M='EOS'.

buffer(N,H,T) :- N>0 |
    H=[_|H1], N1:=N-1, buffer(N1,H1,T).
buffer(N,H,B) :- N=:=0 | B=H.

consume([H|Hs],B) :- H\= 'EOS' |
    B=[_|Ts], write(H), consume(Hs,Ts).
consume([H|Hs],B) :- H=='EOS' | B=[ ].
```

This is the bounded buffer problem discussed in [Ueda 85]. *Produce* creates a stream of intergers and puts the integer to the slot if there is a slot in the buffer. The process *produce* itself never creates a slot. If the head of the buffer

is instantiated, *consume* reads it and makes a new slot at the tail. The head and the tail of the stream are initially related by the goal *buffer*. These three processes run in parallel. As *buffer* only manages the relations of slots and the values put to each slot are decided independently. *Consume* does not have to wait until *produce* generates the values for slots. This is a typical example which shows the effectiveness of decision distribution by using difference lists.

Communication network connected by shared variables should be as simple as possible if the cost of devising simple networks pays.

GHC uses streams for process communication similarly to other concurrent programming languages. We show below a fair merge of streams in the communication between several sender processes and one receiver process. The neccesity of *internal merge* in "communication from multiple processes to one process" was also a problem in Concurrent Prolog [Shapiro 83]. Kusalik gave a solution to this problem [Kusalik 84].

In *construct_omega_graph* procedure, we use multiplexer to merge multiple node processes the number of which changes dynamically based on the Kusalik's algorithm. Multiplexer manages a stream of stream variables, each of which corresponds to output from each sender process. In behaves as follows.

If the head of a stream variable from a sender process is instantiated to a non-variable term, then it is eventually received by the receiver process. If some sender process generates some output, a *merge process* in his solution receives it, decreases the priority of this sender process and check other processes whether they have generated output. If a new sender process is generated, multiplexer creates a new channel to communicate that process. If the merge process receives [ ] as a sign of end_of_stream from some sender process, it aborts that process.

Each process should have independent and equal opportunity to decide whether it trusts the selected clauses without being affected by the result of other OR-parallel proceses. Consider the following two programs.

Program (A)

```
union([X|S1],S2,S) :- member(X,S2) | union(S1,S2,S).
union([X|S1],S2,S) :- otherwise |
    S=[X|NewS], union(S1,S2,NewS).
union([ ],S2,S) :- true | S=S2.

member(X,[Y|_]) :- X==Y | true.
member(X,[Y|S]) :- X\=Y | member(X,S).
```

Program (B)

```
union([X|S1],S2,S) :- member(X,S2,yes) | union(S1,S2,S).
union([X|S1],S2,S) :- member(X,S2,no) |
    S=[X|NewS], union(S1,S2,NewS).
union(S1,[X|S2],S) :- member(X,S1,yes) | union(S1,S2,S).
union(S1,[X|S2],S) :- member(X,S1,no) |
    S=[X|NewS], union(S1,S2,NewS).
union([ ],S2,S) :- true | S=S2.
union(S1,[ ],S) :- true | S=S1.

member(X,[Y|S],Answer) :- X==Y | Answer=yes.
member(X,[Y|S],Answer) :- X\=Y | member(X,S,Answer).
member(X,[ ],Answer) :- true | Answer=no.
```

Program (A) is a direct translation from Prolog version. The predicate *otherwise* succeeds when the passive part of all other OR-parallel processes have failed. It is harmful since the execution depends on the passive part of other clauses. On the other hand, program (B) realises fair OR-parallel execution in the passive part. Therefore, we try to use the predicate *otherwise* as less as possible. To avoid the use, we should write passive parts symmetrically, which is reduced to the equal opportunity of decision.

4.2. Programming Paradigms in GHC

Secondly, we discuss on programming paradigms, i.e.,the patterns of representing parallel algorithms in GHC.

Any piece of unification invoked in the passive part of a clause cannot instantiate a variable appearing in the caller. We should not violate this synchronization mechanism when we apply some technique such as partial evaluation to GHC programs. We consider an example of *remove_implication_and_equivalence* again.

```
remove_implication_and_equivalence(imply(F,G), A) :-
    true |
    remove_implication_and_equivalence(not(F),F1),
    remove_implication_and_equivalence(G,G1)
    A=or(F1,G1).
```

In this program, the definition is according to the fact that $F \supset G$ is logically equivalent to $\neg F \vee G$. *remove_implication_and_equivalence*$(F, F1)$ in this clause allows the commitment of other *remove_implication_and_equivalence* process without instantiating $F, F1$. Therefore, we can apply partial evaluation. The program shown in 4.1 is the result, in which evaluation proceeds one more step ahead than that in this program.

In logic programming, communication through shared variables provides interesting programming paradigms as was investigated by Shapiro. Here, we show a problem encountered in our programming, *termination-flag*.

Termination-flag *JudgeStop* is a shared variable among several processes. If it is instantiated to 'stop' by some process, the message is propagated to the other processes to stop them. It can make some kinds of parallel programs efficient because it releases the system from executing superfluous computations as soon as an answer is

found. In the ω-graphs refutation procedure, it makes the system very effective in the way that *construct_omega_graph* and *check_omega_loop_freeness* run in parallel with a common variable *JudgeStop*. When the process *check_omega_loop_freeness* finds an ω-loop early in the computation, it sets *JudgeStop* to 'stop', which terminates subprocesses in *construct_omega_graph*. Therefore, it does not need to treat a large graph nor superfluous expansion of nodes.

Partially specified data structures are especially useful for utilizing potential concurrency. It is an important concept to write better programs together with that of communication through shared varibales.

As was described above, we can sometimes find an ω-loop before the ω-graph is completely constructed. Therefore, we can check ω-loop freeness on the current partial graph while constructing the ω-graph. Two processes *construct_omega_graph* and *check_omega_loop_freeness* have a shared variable *Graph* which is partially specified during the computation.

Difference list is a typical data structure which is suited for decision distribution. It enables processes to be distributed into each step and decide the output data independently. Its use provides us with possibility to increase efficiency of GHC programs, though it might cost much in some cases.

Are the paradigms in sequential programming, such as *devide and conquer*,*dynamic programming* and *generate and test*, completely of no use in parallel programming ? Or are they still useful with some modification ?

Devide and conquer is a paradigm to divide the problem,solve each subproblems independently and synthesize the subsolutions to the solution of the whole problem. This paradigm naturally takes the form of general recursive style. Since sequential programs in general recursive style are almost directly translated to corresponding GHC programs with AND-parallel processes, this paradigm is suited for GHC programming. We used this paradigm in *compute_initial_formula* procedure.

Dynamic programming is a technique used to convert non-repetitive programs with redundant computation into repetitive(tail-recursive) ones with tables to store the results when they are once computed.

No matter how much resource we can assume in parallel computation, we should still avoid limitless redundant computation. In order to utilize the results computed in one process before, we must pass the results either through shared variables directly to other processes or through the common table accessible from other processes. If we use shared variables, we need to span the communication network by the shared variables. The paradigm of dynamic programming help us to figure out the network. If the network is too complicated and we use common tables, *multiplexer* discussed before is a useful programming concept.

Generate and test is a paradigm to find out a solution by enumerating candidates in sequence and testing each candidate whether it is the desired one. When the generated candidate solution is not the desired one, the program must backtrack once and generate another candidate. Because backtracking is not supported in GHC, the principle "if fail, then redo," should be changed to the principle "test all candidates at the same time and if one succeeds, then stop the other processes." Besides, we can sometimes consider an algorithm that is more suitable for parallel execution. For example, in *check_omega_loop_freeness*, each node is considered as a process sending messages each other, which is more efficient than the algorithm got from modifying sequential one.

4.3. Programming Style in GHC

Lastly, we discuss on programming style, i.e.,the patterns of the activities in constructing GHC programs.

We usually start GHC programming by conceiving a rough and still vague parallel algorithm at an appropriate level of modules and develop it in two directions, downwards (refine parallelism within each module) and upwards (adjust and modify inter-module parallelism). In the process to reach the final GHC program, we need several tools and environment devised for parallel programming.

If we borrow the viewpoint by Kowalski, GHC programs also consist of *logic* part and *control* part.

Different from Prolog, the control part in GHC programs contains more subtle problems and needs more concentration of programmers. We usually would like to confirm the logic part as earlier as possible before considering these subtle control problems in parallel programs. In refining the parallelism within each module, we sometimes modify parallel GHC programs to sequential one and test them first. The reason is two fold : complicated tracing and scheduler's overhead (especially under the breadth-first search scheduling on the sequential machine).

In general, it is difficult to trace the computation procedure of parallel programs compared with that of sequential programs, because it is difficult to know when variables are instantiated or clauses are suspended. Although each process, if executed independently, behave as we expected, the whole program might behave quite differently from our expectation. The more increases the number of processes which run in parallel, the more serious this problem becomes. As one of the solutions of this problem, we propose the programming by incremental parallelization. First, we divide the whole problem into modules in a proper size, and make parallel programming in each module with an attention to interface between modules. Next, we try

to accomplish parallelism at the upper level. We use this programming style in the upper level parallelism for making the ω-graphs refutation and its three modules *compute_initial_node_formula*, *construct_omega_graph* and *check_omega_loop_freeness*.

Since the current GHC system does not have an interpreter, it takes quite much time and energy for programming and debugging. Though the interpreter of GHC might be slow, it is convenient for interactive programming and debugging with screen editor like *ledit* in Lisp.

The KL1 (Kernel Language One), which includes GHC as its core, is still under development. Further investigation of better programming environment is needed.

Current GHC debugger is rather weak. We need better human interface which at least has the following functions. (1) show the figure of the current execution tree and which process is now executed. (2) activate the process designated interactively by the programmer. (3) print out the output of each process to the designated place on the screen process by process.

From the different point of view, the algorithmic debugger of GHC programs is developed[Takeuchi 86]. It does not adopt the debugging method of tracing the execution procedure but the one based on the "divide and query" algorithm. This debugger cannot treat some types of debugging currently. The functional extension can provide us more comfortable parallel programming environment.

In constructing GHC programs, we need to check whether the GHC program at hand is efficient enough for parallel execution. There are several measurements of performance e.g.,CPU time, space used and "parallelism". We used the compiler developed on DEC10-Prolog by Miyazaki[Miyazaki 85], which translates GHC source program to Prolog code and compiles it by DEC10-Prolog Compiler. Because the compiler employs the breadth-first scheduling, the system reports the number of cycles in the execution. We can take the number of cycles as a rough base for evaluation of parallelism.

Among these measurement to judge better GHC program, we give higher priority to time efficiency than less space-consuming because development of parallel execution machine for GHC will solve the space problem. And we give higher priority to parallelism than time because currently execution time is not always equivalent to the theoretical one. For example, the ω-graphs refutation on the formula $\circ\Box\neg P$ needs 10064ms CPU time, 42 cycles and the used global stack is 12457.

7. CONCLUDING REMARKS

We have shown our experience in programming of a proof procedure of temporal logic in GHC and discussed the parallel programming methodology in GHC. Through the experience, we have found a lot of interesting facts and encountered some difficulties due to the difference of the thinking style in GHC from that in sequential programming. Further research on the parallel programming methodology and accumulation of experiences are needed to be done simultaneously with the development of the GHC system itself and parallel machines for execution of GHC.

ACKNOWLEDGMENTS

This research was done as one of the subprojects of the Fifth Generation Computer Systems (FGCS) project. Authors 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.

REFERENCES

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

[Fusaoka and Takahashi 85] Fusaoka,A. and K.Takahashi, "On QFTL and the Refutation Procedure on ω-graphs," pp.43-54,TGAL85-31,IECE,Japan,1985.

[Hughes and Cresswell 68] Hughes,G.E. and Cresswell,M.J., " An Introduction to Modal Logic," Methuen and Co. Ltd, 1968.

[Kusalik 84] Kusalik,A.J., "Bounded-Wait Merge in Shapiro's Concurrent Prolog," New Generation Computing,pp.157-169,Vol.2,No.2,1984.

[Kripke 69] Kripke,S.A., "A Completeness Theorem in Modal Logic," The Journal of Symbolic Logic,Vol.24.No.1,March 1969.

[Manna and Pnueli 81] Manna,Z. and A.Pnueli, "Verification of Concurrent Programs, Part1: The Temporal Framework," Stanford TR 81-836,1981.

[Miyazaki 85] Miyazaki,T., "Guarded Horn Clause Compiler User's Guide," unpublished, 1985.

[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.

[Takahashi and Kanamori 86] Takahashi,K. and T.Kanamori, "On Parallel Programming Methodology in GHC," ICOT Technical Memo, to appear.

[Ueda 85] Ueda,K., "Guarded Horn Clauses," ICOT TR-103,1985.

[Wolper 81] Wolper,P.L., "Temporal Logic Can Be More Expressive," Proc.22nd IEEE Symposium on Foundation of Computer Science, pp.340-348,1981.