ICOT Technical Memorandum: TM-0171

TM-0171

On Parallel Programming Methodology in GHC
—Experience in Programming of A Proof Procedure of Temporal Logic—

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
8-1-1, Tsukaguchi-Honmachi
Amagasaki, Hyogo, JAPAN 661

ABSTRACT

This paper discusses the parallel programming methodology in GHC based on our experience in programming of a proof procedure of temporal logic. GHC (Guarded Horn Clauses) is a parallel programming language designed for execution on highly parallel architecture. 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 (sequential) Prolog and GHC, we compare the thinking style in sequential programming and that in parallel programming. General principles for enhancing concurrency, programming paradigms in GHC and programming style in GHC are discussed based on the experience.

Keywords : Parallel Logic Programming, GHC, Parallel Programming Methodology, Proof Procedure, Temporal Logic

CONTENTS

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 programming style. 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.

This paper is organized as follows. In section 2, we give a general introduction of GHC. In section 3, we give the syntax and the semantics of propositional temporal logic. In section 4, we explain our proof procedure of temporal logic and present its sequential implementation in Prolog. In section 5, we show its parallel implementation in GHC. And in section 6, we compare the differences of thinking style between the sequential version and the parallel version and discuss the parallel programming methodology in GHC.

2. PROLOG AND GUARDED HORN CLAUSES (GHC)

In the research of logic programming, Guarded Horn Clauses (GHC) is developed as a language for parallel logic prorамming. It is a machine-independent core of Kernel Language One (KL1). It can express simply the characteristic features of parallel programming such as processes, communication and synchronization. It is almost like Prolog syntactically, except that only one new construct '|' called *trust operator* is added to Horn Clauses. It needs no inheritance of multiple environment, which provides us a simpler implementation compared with other parallel programming languages such as Concurrent Prolog or PARLOG.

2.1. Syntax of GHC

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 formuals defined as usual. H, G_i's and B_i's are called *clause head, guard goals* and *body goals*, respectively. The part of the clause before '|' is called a *passive part* or a *guard*, and the part after '|' is called an *active part*. The special predicate *true* is used for denoting the empty goal, and it is sometimes omitted. The clauses

H_1 :- B_1.

H_2.

represent

H_1 :- *true* | B_1.

H_2 :- *true* | *true*.

respectively.

2.2. Semantics of GHC

Informally the 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 guard goals of these candidate clauses are tried in parallel, and if guard goals of some clause succeed, then the active part of the clause is executed. The execution is done accrording to the following rules.

Rules of Suspension

(1)[synchronization]

Any piece of unification invoked directly or indirectly in the passive part of a clause cannot bind a variable appearing in the caller of that clause with

(a) a non-variable term or

(b) another variable appearing in the caller

until that clause is trusted.

(2) [sequencing]

Any pieace of unification invoked directly or indirectly in the active part of a clause cannot bind a variable appearing in the passive part of that clause with

(a) a non-variable term or

(b) another variable appearing in the passive part

until that clause is trusted.

A piece of unification which can succeed only by making such bindings is *suspended* until it can succeed without making such bindings or it turns out to fail.

Rule of Trust

When some clause succeeds in solving its passive part, that clause tries to be *trusted*.

It must confirm that no other clause that belongs to the same procedure has been trusted for the same call, and if confirmed, that clause is trusted immediately. We say that a set of goals *succeeds* (or is *solved*) if it is reduced to the empty set of goals by using only trusted clauses.

The execusion of conjunctive goals are done in parallel ('AND' parallelism) and the search of the clauses to be trusted are also done in parallel ('OR' parallelism). In the programming in GHC, we make use of these mechanisms as well as communication through streams. The lack of backtracking forces us to change our thinking style and investigate a new methodology of programming. Later, we discuss on these subjects in detail.

3. PROPOSITIONAL TEMPORAL LOGIC (PTL)

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

First, we will present the syntax and the semantics of PTL.

3.1. Language of PTL

The language of PTL uses the following four classes of symbols.
(1) The propositional constants *true*, *false*
(2) The propositional variables $P, Q, R, \ldots$
(3) The Boolean connectives $\neg, \wedge, \vee, \supset, \equiv$
(4) The temporal operators $\Box, \Diamond, \bigcirc$

An *atomic formula* is either a propositional constant or a propositional variable. A *literal* is either an atomic formula or the negation of an atomic formula. *Formulas* are defined inductively as follows :
(1) An atomic formula is a formula.
(2) $\neg F$, $F \wedge G$, $F \vee G$, $F \supset G$ and $F \equiv G$ are formulas when F and G are formulas.
(3) $\Box F$, $\Diamond F$ and $\bigcirc F$ are formulas when F is a formula.

Example 3.1. The followings are formulas of PTL.
$(\Diamond P \supset \Diamond(Q \wedge \neg\Diamond R)) \wedge \neg\Box S$, $\Diamond P \wedge \Diamond\neg P$, $\Box\Diamond P \wedge \Diamond\Box\neg P$

A temporal literal is either a literal or the formula whose outermost operator is $\bigcirc$.

3.2. Models for PTL

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

$$K_0, K_1, K_2, \ldots$$

We define the truth value assignment for formulas in the usual way. Let G be a subformula of F and M be a model $K_0, K_1, \ldots$. We define the assignment for G by K_i inductively as follows. When G is a propositional constant *true*, G is always assigned **t** by K_i. When G is a propositinal constant *false*, G is always assigned **f** by K_i. When G is a propositinal variable, G is assigned **t** or **f** by K_i following the complete assignment K_i. When G is of the form $\neg G'$, G is assigned **t** (**f**) by K_i if and only if G' is assigned **f** (**t**) by K_i. When G is of the form $G_1 \wedge G_2$, G is assigned **t** by K_i if both G_1 and G_2 are assigned **t** by K_i; otherwise G is assigned **f** by K_i. When G is of the form $G_1 \vee G_2$, G is assigned **t** by K_i if either G_1 or G_2 is assigned **t** by K_i; otherwise G is assigned **f** by K_i. When G is of the form $G_1 \supset G_2$, G is assigned **t** by K_i if either G_1 is assigned **f** or G_2 is assigned **t** by K_i; otherwise G is assigned **f** by K_i. When G is of the form $G_1 \equiv G_2$, G is assigned

t by K_i if both of G_1 and G_2 are assigned t or both of them assigned f by K_i; otherwise G is assigned f by K_i. When G is of the form $\Box G'$, G is assigned t by K_i if every $K_j(j \geq i)$ assigns t to G'; otherwise G is assigned f by K_i. When G is of the form $\Diamond G'$, G is assigned t by K_i if there exists some $K_j(j \geq i)$ that assigns t to G'; otherwise G is assigned f by K_i. When G is of the form $\circ G'$, G is assigned t(f) by K_i if and only if K_{i+1} assigns t(f) to G'.

Let F be a formula and M a model $K_0, K_1, \ldots$ for F.

(1) F is said to be *true(false) in* M if F is assigned t(f) by K_0.
(2) If F is true in a model M, we say that M *satisfies* F, and denote it by $M \models F$.
(3) F is *satisfiable* if there exists a model which satisfies F.
(4) F is *unsatisfiable* if it is not satisfiable
(5) F is *valid* if it is true in every model and we denote it by $\models F$

Example 3.2. 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 $N_0 \rightarrow N_1 \rightarrow N_2 \rightarrow N_1 \rightarrow N_2 \rightarrow N_1 \rightarrow N_2 \rightarrow \ldots$. 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,...).

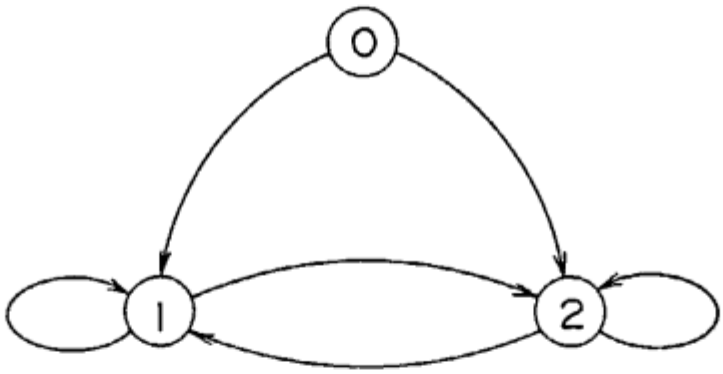


Figure 3.2. Graph Representation for $\Box\Diamond$P

Lemma.

When $F \equiv G$ is valid, F and G are said to be *logically equivalent.*

(1) The followings hold.
  (1-1) $F \supset G$ is logically equivalent to $\neg F \vee G$.
  (1-2) $F \equiv G$ is logically equivalent to $(F \wedge G) \vee (\neg F \wedge \neg G)$.
  (1-3) $\neg(F \wedge G)$ is logically equivalent to $\neg F \vee \neg G$.
  (1-4) $\neg(F \vee G)$ is logically equivalent to $\neg F \wedge \neg G$.
  (1-5) $\neg\Box F$ is logically equivalent to $\Diamond\neg F$.
  (1-6) $\neg\Diamond F$ is logically equivalent to $\Box\neg F$.
  (1-7) $\neg\circ F$ is logically equivalent to $\circ\neg F$.
  (1-8) $\neg\neg F$ is logically equivalent to F.
(2) The followings hold.
  (2-1) $\Box G$ is logically equivalent to $G \wedge \circ\Box G$.
  (2-2) $\Diamond G$ is logically equivalent to $G \vee \circ\Diamond G$.
(3) The followings hold.
  (3-1) $F \wedge (G \vee H)$ is logically equivalent to $(F \wedge G) \vee (F \wedge H)$.

(3-2) $(F \vee G) \wedge H$ is logically equivalent to $(F \wedge H) \vee (G \wedge H)$.
(3-3) $\circ F \wedge \circ G$ is logically equivalent to $\circ(F \wedge G)$.
(3-4) $F \wedge F$ is logically equivalent to F.
(3-5) $P \wedge \neg P$ is logically equivalent to *false* when P is an atomic formula.
(3-6) $\neg P \wedge P$ is logically equivalent to *false* when P is an atomic formula.
(3-7) *false* $\wedge F$ is logically equivalent to *false*
(3-8) $F \wedge$ *false* is logically equivalent to *false*.
(3-9) *false* $\vee F$ is logically equivalent to F.
(3-10) $F \vee$ *false* is logically equivalent to F.
(3-11) E is logically equivalent to $E \wedge \circ\Box$*true*
when E is a formula which does not contain temporal operators.

(4) Let $F[G_1]$ and $F[G_2]$ be two formulas obtained from F by replacing some subformula of F with G_1 and G_2 respectively. When G_1 is logically equivalent to G_2, $F[G_1]$ is valid iff $F[G_2]$ is valid.

Proof. Trivial. For example,(2-2) is proved as follows.

We will show that $\diamond G \equiv G \vee \circ\diamond G$. Assume that $M = K_0, K_1, \ldots$ is any model, and we will show that $M \models \diamond G \equiv G \vee \circ\diamond G$, that is, the formula is assigned **t** by K_0. By the definition of assignment on $\equiv$, it is sufficient to show that for any model $M = K_0, K_1, \ldots$, $\diamond G$ is assigned **t** by K_0 iff $G \vee \circ\diamond G$ is assigned **t** by K_0. We will prove if-part. Assume that $\diamond G$ is assigned **t** by K_0. Then there exists K_i such that G is assigned true by K_i. If $i = 0$, G is assigned **t** by K_0. Otherwise, there exists $j (j \geq 1)$, such that G is assigned **t** by K_j. Therefore $\diamond G$ is assigned **t** by K_1, that is, $\circ\diamond G$ is assigned **t** by K_0 Hence, $G \vee \circ\diamond G$ is assigned **t** by K_0. Only-if-part is trivial.

4. ω- GRAPHS REFUTATION IN PTL

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 F_0 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 the ω-loop freeness of the constructed ω-graph. If it is ω-loop free,the given formula is valid.

4.1. Computation of Initial Node Formula

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

Definition 4.1.1. Negation Normal Form
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

$$F \supset G \implies \neg F \vee G$$
$$F \equiv G \implies (F \wedge G) \vee (\neg F \wedge \neg G)$$

[Rule NNF2] move negation inwards

$$\neg(F \wedge G) \implies \neg F \vee \neg G$$
$$\neg(F \vee G) \implies \neg F \wedge \neg G$$
$$\neg\Box F \implies \Diamond\neg F$$
$$\neg\Diamond F \implies \Box\neg F$$
$$\neg\bigcirc F \implies \bigcirc\neg F$$
$$\neg\neg F \implies F$$

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. (See lemma (1) and (4).)

Example 4.1.1. Let F be $(\Diamond P \supset \Diamond(Q \wedge \neg\Diamond R)) \wedge \neg\Box S$. F is converted as follows.

$$\underline{(\Diamond P \supset \Diamond(Q \wedge \neg\Diamond R))} \wedge \neg\Box S$$
$$\Downarrow \text{ Rule NNF1}$$
$$(\underline{\neg\Diamond P} \vee \Diamond(Q \wedge \underline{\neg\Diamond R})) \wedge \underline{\neg\Box S}$$
$$\Downarrow \text{ Rule NNF2}$$
$$(\Box\neg P \vee \Diamond(Q \wedge \Box\neg R)) \wedge \Diamond\neg S$$

The last formula is the negation normal form of F.

Later,we need to check whether we have constructed an ω-graph corresponding to a model of PTL in which the eventualities in a negation normal form F_0 are satisfied.

Definition 4.1.2. Eventuality Set
Let F be a formula in a negation normal form. The set of all subformulas G such that $\Diamond G$ is a subformula of F is called *eventuality set of F*.

Example 4.1.2. The eventuality set of the formula $(\Box\neg P \vee \Diamond(Q \wedge \Box\neg R)) \wedge \Diamond\neg S$ is $\{(Q \wedge \Box\neg R), \neg S\}$. The eventuality set of the formula $\Diamond P \wedge \Diamond\neg P$ is $\{P, \neg P\}$. The eventuality set of the formula

$(\Box\Diamond P \wedge \Box\Diamond Q)$ is $\{P, Q\}$.

From now on, we denote the negation normal form of the negation of the given formula by F_0 and the eventuality set of F_0 by ES_0.

Definition 4.1.3. Initial Node Formula

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.

An initial node formula $[F_0]_{\{\}}$ is logically equivalent to the formula F_0 and it is a special form of a node formula defined later.

The computation of initial node formula is implemented in Prolog as follows.

```
compute_initial_node_formula(F,NNF,EveSet) :-
        negation_normal_form(F,NNF),
        compute_eventuality_set(NNF,EveSet).

negation_normal_form(F,G) :-
        remove_implication_and_equivalence(F,F1),
        move_not_inwards(F1,G).

remove_implication_and_equivalence(imply(F,G),or(not(F1),G1)) :-
        remove_implication_and_equivalence(F,F1),
        remove_implication_and_equivalence(G,G1).
remove_implication_and_equivalence(equivalent(F,G),
                or(and(F1,G1),and(not(F1),not(G1)))) :-
        remove_implication_and_equivalence(F,F1),
        remove_implication_and_equivalence(G,G1).
remove_implication_and_equivalence(and(F,G),and(F1,G1)) :-
        remove_implication_and_equivalence(F,F1),
        remove_implication_and_equivalence(G,G1).
remove_implication_and_equivalence(or(F,G),or(F1,G1)) :-
        remove_implication_and_equivalence(F,F1),
        remove_implication_and_equivalence(G,G1).
remove_implication_and_equivalence(always(F),always(G)) :-
        remove_implication_and_equivalence(F,G).
remove_implication_and_equivalence(eventually(F),eventually(G)) :-
        remove_implication_and_equivalence(F,G).
remove_implication_and_equivalence(next(F),next(G)) :-
        remove_implication_and_equivalence(F,G).
remove_implication_and_equivalence(not(F),not(G)) :-
        remove_implication_and_equivalence(F,G).
remove_implication_and_equivalence(F,F).

move_not_inwards(and(F,G),and(F1,G1)) :-
        move_not_inwards(F,F1), move_not_inwards(G,G1).
move_not_inwards(or(F,G),or(F1,G1)) :-
        move_not_inwards(F,F1), move_not_inwards(G,G1).
move_not_inwards(always(F),always(G)) :-
```

```
        move_not_inwards(F,G).
move_not_inwards(eventually(F),eventually(G)) :-
        move_not_inwards(F,G).
move_not_inwards(next(F),next(G)) :-
        move_not_inwards(F,G).
move_not_inwards(not(and(F,G)),or(F1,G1)) :-
        move_not_inwards(not(F),F1), move_not_inwards(not(G),G1).
move_not_inwards(not(or(F,G)),and(F1,G1)) :-
        move_not_inwards(not(F),F1), move_not_inwards(not(G),G1).
move_not_inwards(not(always(F)),eventually(G)) :-
        move_not_inwards(not(F),G).
move_not_inwards(not(eventually(F)),always(G)) :-
        move_not_inwards(not(F),G).
move_not_inwards(not(next(F)),next(G)) :-
        move_not_inwards(not(F),G).
move_not_inwards(not(not(F)),F).
move_not_inwards(F,F).

compute_eventuality_set(and(F,G),S) :-
        compute_eventuality_set(F,SF),
        compute_eventuality_set(G,SG),
        union(SF,SG,S).
compute_eventuality_set(or(F,G),S) :-
        compute_eventuality_set(F,SF),
        compute_eventuality_set(G,SG),
        union(SF,SG,S).
compute_eventuality_set(always(F),S) :-
        compute_eventuality_set(F,S).
compute_eventuality_set(eventually(F),S) :-
        compute_eventuality_set(F,SF),
        union([F],SF,S).
compute_eventuality_set(next(F),S) :-
        compute_eventuality_set(F,S).
compute_eventuality_set(F,[]).
```

Figure 4.1. Computation of Initial Node Formula

4.2. Expansion of Node Formulas

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

Definition 4.2.1. Node Formulas

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* (or *h-set* for short). $\diamond^* G$ is semantically identical to $\diamond G$. $\diamond^*$ is called *marked eventuality* and $\diamond^* G$ is called *marked formula*.

From now on,we call node formulas simply formulas and we sometimes use $[F]_H$ in place of F. A node formula $[F]_H$ is logically equivalent to the formula F. The intuitive meanings of the mark and the history sets will be explained later.

Example 4.2.1. The followings are node formulas.

$[\diamond P \wedge \diamond \neg P]_{\{\}}$, $[\diamond^* \neg P]_{\{P\}}$, $[\diamond^* P]_{\{\neg P\}}$, $[\diamond^* P \wedge \diamond^* \neg P]_{\{\}}$

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. Suppose we are trying to expand a node labelled 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}.

Definition 4.2.2. Next Prefix Form

Let F be a formula in negation normal form and G be a formula obtained from F by applying the rules below

[Rule NPF1] postpone $\Box$

$\Box G \quad \Longrightarrow G \wedge \bigcirc\Box G$

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

$\diamond G \quad \Longrightarrow G \vee \bigcirc\diamond^* G$

$\diamond^* G \quad \Longrightarrow G \vee \bigcirc\diamond^* G$

as far as possible to subformulas not inside the next operator $\bigcirc$. Then G is called *next prefix form* of F.

The conversion to a next prefix form is essentially based on the *tableau methods* [Wolper 81]. Note that the next prefix form of a formula is valid if and only if the original formula is valid. (See lemma (2) and (4).)

Example 4.2.2. A formula $(\diamond P \wedge \diamond \neg P)$ is converted as follows.

$$\begin{array}{ccc} \underline{(\diamond P} & \wedge & \underline{\diamond \neg P)} \\ \Downarrow & & \Downarrow \\ P \vee \bigcirc\diamond^* P & & \neg P \vee \bigcirc\diamond^* \neg P \end{array}$$

The next prefix form is $(P \vee \bigcirc\diamond^* P) \wedge (\neg P \vee \bigcirc\diamond^* \neg P)$

Example 4.2.3.

A formula $(\Box\neg P \vee \diamond(Q \wedge \Box\neg R)) \wedge \diamond\neg S$ is converted as follows.

$$\begin{array}{ccccc} \underline{(\Box\neg P} \vee \underline{\diamond(Q \wedge \Box\neg R))} & & \wedge & & \underline{\diamond\neg S} \\ \Downarrow \qquad \Downarrow & & & & \Downarrow \\ \neg P \wedge \bigcirc\Box\neg P \quad (Q \wedge \underline{\Box\neg R)} \vee \bigcirc\diamond^*(Q \wedge \Box\neg R) & & & & \neg S \vee \bigcirc\diamond\neg S \\ \Downarrow & & & & \\ \neg R \wedge \bigcirc\Box\neg R & & & & \end{array}$$

The next prefix form is

$$((\neg P \wedge \circ\Box\neg P) \vee ((Q \wedge (\neg R \wedge \circ\Box\neg R)) \vee \circ\Diamond^*(Q \wedge \Box\neg R))) \wedge (\neg S \vee \circ\Diamond^*\neg S)$$

Example 4.2.4. A formula $\Box\Diamond P \wedge \Box\Diamond Q$ is converted as follows.

$$\begin{array}{ccc} \underline{\Box\Diamond P} & \wedge & \underline{\Box\Diamond Q} \\ \Downarrow & & \Downarrow \\ \underline{\Diamond P} \wedge \circ\Box\Diamond P & & \underline{\Diamond Q} \wedge \circ\Box\Diamond Q \\ \Downarrow & & \Downarrow \\ P \vee \circ\Diamond^* P & & Q \vee \circ\Diamond^* Q \end{array}$$

The next prefix form is $((P \vee \circ\Diamond^* P) \wedge \circ\Box\Diamond P) \wedge ((Q \vee \circ\Diamond^* Q) \wedge \circ\Box\Diamond Q)$.

Definition 4.2.3. Disjunctive Normal Form

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

$$(E_1 \wedge \circ F_1) \vee (E_2 \wedge \circ F_2) \vee .... \vee (E_n \wedge \circ F_n)$$

where $E_1, E_2, ..., E_n$ and $F_1, F_2, ..., F_n$ are formulas other than $false$. $E_1, E_2, ..., E_n$ 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 in a *disjunctive normal form of* F. (Note that there exists as a special case G is the unique propositional constant $false$. It is also considered as in a dijunctive normal form.)

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

$F \wedge (G \vee H) \Longrightarrow (F \wedge G) \vee (F \wedge H)$

$(F \vee G) \wedge H \Longrightarrow (F \wedge H) \vee (G \wedge H)$

[Rule DNF2] eliminate simple contradictions and duplication

$P \wedge \neg P \Longrightarrow false$ where P is an atomic formula

$\neg P \wedge P \Longrightarrow false$ where P is an atomic formula

$false \wedge F \Longrightarrow false$

$F \wedge false \Longrightarrow false$

$false \vee F \Longrightarrow F$

$F \vee false \Longrightarrow F$

$F \wedge F \Longrightarrow F$

[Rule DNF3] supplement next part

Current formula F is in the form $C_1 \vee C_2 \vee ... \vee C_n$ where each C_i is a conjunction of temporal literals.

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

[Rule DNF4] ordering

$\circ G \wedge F \Longrightarrow F \wedge \circ G$ where F isn't in the form of $\circ F'$

[Rule DNF5] merge $\circ$

$\circ F \wedge \circ G \Longrightarrow \circ(F \wedge G)$

[Rule DNF6] combination by next part

$(F \wedge \circ H) \vee (G \wedge \circ H) \Longrightarrow (F \vee G) \wedge \circ H$

Example 4.2.5. Consider the following four formulas.

$(P \wedge \circ Q) \vee ((\neg P \vee R) \wedge \circ(Q \wedge S))$

$(P \wedge \circ Q) \vee (\neg P \wedge \circ\Box true)$.

$(P \wedge \circ Q) \vee (R \wedge \circ Q)$

$(P \wedge \circ Q) \vee \Box R$

The first two formulas are in disjunctive normal form, while the last two are not.

Assume that $(E_1 \wedge \circ F_1) \vee (E_2 \wedge \circ F_2) \vee \cdots \vee (E_n \wedge \circ F_n)$

is a disjunctive normal form of F. If F is in a negation normal form, each F_i is in a negation normal form. Note that the disjunctive normal form of a formula is valid if and only if the original formula is valid. (See lemma (3) and (4).)

Example 4.2.6. Let F be a formula in next prefix form

$$(P \vee \bigcirc\Diamond^* P) \wedge (\neg P \vee \bigcirc\Diamond^* \neg P)$$

It is converted as follows.

$\underline{(P \vee \bigcirc\Diamond^* P) \wedge (\neg P \vee \bigcirc\Diamond^* \neg P)}$

⇓ Rule DNF1

$\underline{(P \wedge \neg P)} \vee (P \wedge \bigcirc\Diamond^* \neg P) \vee (\bigcirc\Diamond^* P \wedge \neg P) \vee (\bigcirc\Diamond^* P \wedge \bigcirc\Diamond^* \neg P)$

⇓ Rule DNF2

$(P \wedge \bigcirc\Diamond^* \neg P) \vee \underline{(\bigcirc\Diamond^* P \wedge \neg P)} \vee (\bigcirc\Diamond^* P \wedge \bigcirc\Diamond^* \neg P)$

⇓ Rule DNF4

$(P \wedge \bigcirc\Diamond^* \neg P) \vee (\neg P \wedge \bigcirc\Diamond^* P) \vee \underline{(\bigcirc\Diamond^* P \wedge \bigcirc\Diamond^* \neg P)}$

⇓ Rule DNF5

$(P \wedge \bigcirc\Diamond^* \neg P) \vee (\neg P \wedge \bigcirc\Diamond^* P) \vee (\bigcirc(\Diamond^* P \wedge \Diamond^* \neg P))$

The last formula is the disjunctive normal form of F.

$$(P \wedge \bigcirc\Diamond^* \neg P) \vee (\neg P \wedge \bigcirc\Diamond^* P) \vee (\bigcirc(\Diamond^* P \wedge \Diamond^* \neg P))$$

Example 4.2.7. A formula in next prefix form

$$\{(\neg P \wedge \bigcirc\Box\neg P) \vee ((Q \wedge (\neg R \wedge \bigcirc\Box\neg R)) \vee \bigcirc\Diamond^*(Q \wedge \Box\neg R))\} \wedge (\neg S \vee \bigcirc\Diamond^* \neg S)$$

is converted to its disjunctive normal form

$(\neg P \wedge \neg S \wedge \bigcirc\Box\neg P) \vee (\neg P \wedge \bigcirc(\Box\neg P \wedge \Diamond^* \neg S)) \vee (Q \wedge \neg R \wedge \neg S \wedge \bigcirc\Box\neg R) \vee$
$(Q \wedge \neg R \wedge \bigcirc(\Box\neg R \wedge \Diamond^* \neg S)) \vee (\neg S \wedge \bigcirc(\Diamond^*(Q \wedge \Box\neg R))) \vee ((\bigcirc(\Diamond^*(Q \wedge \Box\neg R) \wedge \Diamond^* \neg S))$

Example 4.2.8. A formula in next prefix form

$$((P \vee \bigcirc\Diamond^* P) \wedge \bigcirc\Box\Diamond P) \wedge ((Q \vee \bigcirc\Diamond^* Q) \wedge \bigcirc\Box\Diamond Q)$$

is converted to its disjunctive normal form

$(P \wedge Q \wedge \bigcirc(\Box\Diamond P \wedge \Box\Diamond Q)) \vee (P \wedge \bigcirc(\Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q)) \vee$
$(Q \wedge \bigcirc(\Diamond^* P \wedge \Box\Diamond P \wedge \Box\Diamond Q)) \vee (\bigcirc(\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q))$

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.

Definition 4.2.4. Expansion of Node Formula

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

Each H_i is called a *history set*(*h-set*, in short)*of* F_i. H-sets are introduced to ensure that eventuality will actually be realized. Each element of h-set indicates the history of the realization.

Example 4.2.9. Let F_0 be $\Diamond P \wedge \Diamond\neg P$ in a negation normal form. The eventuality set ES_0 is $\{P, \neg P\}$ from the example 4.1.2. By the example 4.2.6, a node formula $[\Diamond P \wedge \Diamond\neg P]_{\{\}}$ is converted to

$$(P \wedge \bigcirc(\Diamond^* \neg P)) \quad \vee \quad (\neg P \wedge \bigcirc(\Diamond^* P)) \quad \vee \quad (\bigcirc(\Diamond^* P \wedge \Diamond^* \neg P))$$

Let S_1, S_2 and S_3 be node formulas corresponding to the formulas

$\diamond^* \neg P$,

$\diamond^* P$,

$\diamond^* \neg P \wedge \diamond^* P$

respectively. For each S_i, we define the corresponding h-set H_i. For each S_i, ES_i is computed as follows.

$$ES_1 = \{\neg P\}$$
$$ES_2 = \{P\}$$
$$ES_3 = \{P, \neg P\}$$

Since $\{\} \neq ES_0$, H_1 is computed as follows.

$$H_1 = (ES_0 - ES_1) \bigcup H = (\{P, \neg P\} - \{\neg P\}) \bigcup \{\} = \{P\}.$$

It means that $\diamond P$ is fufilled. Similarly H_2 and H_3 are computed as follows.

$$H_2 = (ES_0 - ES_2) \bigcup H = (\{P, \neg P\} - \{P\}) \bigcup \{\} = \{\neg P\}.$$
$$H_3 = (ES_0 - ES_3) \bigcup H = (\{P, \neg P\} - \{P, \neg P\}) \bigcup \{\} = \{\}.$$

Thus, three node formulas

$$S_1 = [\diamond^* \neg P]_{\{P\}},$$
$$S_2 = [\diamond^* P]_{\{\neg P\}},$$
$$S_3 = [\diamond^* P \wedge \diamond^* \neg P]_{\{\}}$$

are generated.

Example 4.2.10. Let F_0 be $(\Box\diamond P \wedge \Box\diamond Q)$ in negation normal form. The eventuality set ES_0 is $\{P, Q\}$ from the example 4.1.2. By the example 4.2.8, a node formula $[\Box\diamond P \wedge \Box\diamond Q]_{\{\}}$ is converted to

$$(P \wedge Q \wedge \circ(\Box\diamond P \wedge \Box\diamond Q)) \quad \vee \quad (P \wedge \circ(\Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q)) \quad \vee$$
$$(Q \wedge \circ(\diamond^* P \wedge \Box\diamond P \wedge \Box\diamond Q)) \quad \vee \quad (\circ(\diamond^* P \wedge \Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q))$$

Let S_1, S_2, S_3 and S_4 be node formulas corresponding to the formulas

$\Box\diamond P \wedge \Box\diamond Q$,

$\Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q$,

$\diamond^* P \wedge \Box\diamond P \wedge \Box\diamond Q$,

$\diamond^* P \wedge \Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q$

respectively. For each S_i, ES_i is computed as follws.

$$ES_1 = \{\}$$
$$ES_2 = \{Q\}$$
$$ES_3 = \{P\}$$
$$ES_4 = \{P, Q\}$$

Since $\{\} \neq ES_0$, we compute the corresponding h-set H_i as $H_i = (ES_0 - ES_i) \bigcup H$.

$$H_1 = (\{P, Q\} - \{\}) \bigcup \{\} = \{P, Q\}.$$
$$H_2 = (\{P, Q\} - \{Q\}) \bigcup \{\} = \{P\}.$$
$$H_3 = (\{P, Q\} - \{P\}) \bigcup \{\} = \{Q\}.$$
$$H_4 = (\{P, Q\} - \{P, Q\}) \bigcup \{\} = \{\}.$$

Thus, four node formulas

$$S_1 = [\Box\diamond P \wedge \Box\diamond Q]_{\{P,Q\}},$$
$$S_2 = [\Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q]_{\{P\}},$$
$$S_3 = [\diamond^* P \wedge \Box\diamond P \wedge \Box\diamond Q]_{\{Q\}},$$
$$S_4 = [\diamond^* P \wedge \Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q]_{\{\}}$$

are generated.

Example 4.2.11. Now,consider expansion of node formulas from the node formula

$$[\Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q]_{\{P\}}.$$

Suppose the eventuality set ES_0 is $\{P, Q\}$. The formula is converted to the next prefix form,

then to the same disjunctive normal form with that of $\Box\Diamond P \wedge \Box\Diamond Q$. Let S_1, S_2, S_3, S_4 and H_1, H_2, H_3, H_4 are formulas and h-sets defined in the example 4.2.10. Each H_i is computed by $(ES_0 - ES_i) \bigcup H$.

$$H_1 = (\{P,Q\} - \{\}) \bigcup \{P\} = \{P,Q\}$$
$$H_2 = (\{P,Q\} - \{Q\}) \bigcup \{P\} = \{P\}$$
$$H_3 = (\{P,Q\} - \{P\}) \bigcup \{P\} = \{P,Q\}$$
$$H_4 = (\{P,Q\} - \{P,Q\}) \bigcup \{P\} = \{P\}$$

Thus four node formulas

$$S_1 = [\Box\Diamond P \wedge \Box\Diamond Q]_{\{P,Q\}},$$
$$S_2 = [\Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{P\}},$$
$$S_3 = [\Diamond^* P \wedge \Box\Diamond P \wedge \Box\Diamond Q]_{\{P,Q\}},$$
$$S_4 = [\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{P\}}$$

are generated.

Our purpose is to construct an ω-graph by expanding node formulas repeatedly. In this procedure, we repeatedly apply the rule NPF2. But it may continuously postpone the instance in which G is true by satisfying the $\bigcirc\Diamond^* G$ part of the disjunction. In order to escape such a case, $\Diamond^*$ and h-set are introduced.

$\Diamond^* G$ indicates that the instance in which G is true is postponed and h-set indicates that there need not exist the instance in which G should be true any more since it is already realized. h-set represents an incremental information about the realization of the eventualities. We will give an intuitive explanation a little more.

See examples 4.2.10 and 4.2.11. In the examples, disjunctive normal forms of the given formulas are identical. In the example 4.2.10, the h-set of the given node formula is { } which means that neither P nor Q is realized yet. For the node formula S_3, for example, Q is added to the h-set as the information that Q is realized. On the other hand, in 4.2.11, the h-set of the given formula is $\{P\}$ which means that P is already realized. For the node formula S_3, Q is added to the h-set as the information that Q is realized. And together with the former information, H_3 becomes $\{P,Q\}$. It means that both P and Q are realized and that there need not exist the state in which P is true nor Q is true any more. However, the given node formula $\Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q$ requires that P and Q should be realized infinitely often respectively. We have to guanrantee that both P and Q are realized after the instance corresponding to S_3. (It is not necessary that P and Q are realized at the same instant.) Therefore, the information about the realization of the eventuality is once cleared. It is a special instant. The node formula whose h-set is equivalent to ES_0 corresponds to the special instant in which all the eventualities in ES_0 have at least once realized by continuing the expansion from the initial state or latest special instant.

The expansion procedure is implemented in Prolog as follows.

```
expand_node_formula([F,H],ES0,NewNodeFormulas) :-
        next_prefix_form(F,NPF),
        disjunctive_normal_form(NPF,DNF),
        generate_new_node_formulas(DNF,ES0,H,NewNodeFormulas).

%%% (1) Next Prefix Form

next_prefix_form(and(F,G),and(F1,G1)) :-
        next_prefix_form(F,F1),
```

```
        next_prefix_form(G,G1).
next_prefix_form(or(F,G),or(F1,G1)) :-
        next_prefix_form(F,F1),
        next_prefix_form(G,G1).
next_prefix_form(eventually(F),or(F1,next(eventually*(F)))) :-
        next_prefix_form(F,F1).
next_prefix_form(eventually*(F),or(F1,next(eventually*(F)))) :-
        next_prefix_form(F,F1).
next_prefix_form(always(F), and(F1,next(always(F)))) :-
        next_prefix_form(F,F1).
next_prefix_form(F,F).

%%% (2) Disjunctive Normal Form

disjunctive_normal_form(NPF,DNF) :-
        distribute_and_over_or(NPF,Disjunction),
        simplify_and_gather_next_formulas(Disjunction,DNF).

distribute_and_over_or(or(F,G), or(F1,G1)) :-
        distribute_and_over_or(F,F1),
        distribute_and_over_or(G,G1).
distribute_and_over_or(and(F,G), H) :-
        distribute_and_over_or(F,F1),
        distribute_and_over_or(G,G1),
        distribute(and(F1,G1),H).
distribute_and_over_or(F,F).

distribute(and(or(F,G),H),or(F1,G1)) :-
        distribute_and_over_or(and(F,H),F1),
        distribute_and_over_or(and(G,H),G1).
distribute(and(F,or(G,H)),or(F1,G1)) :-
        distribute_and_over_or(and(F,G),F1),
        distribute_and_over_or(and(F,H),G1).
distribute(F,F).

simplify_and_gather_next_formulas(or(F,G),Fs) :-
        simplify_and_gather_next_formulas(F,F1),
        simplify_and_gather_next_formulas(G,G1),
        union1(F1,G1,Fs).
simplify_and_gather_next_formulas(Conjunction,Fs) :-
        simplify_conjuction_of_temporal_literals(Conjunction,Fs).

simplify_conjuction_of_temporal_literals(F,[G]) :-
        flatten_and_partition(F,Current,Next),
        eliminate_confliction(Current,Ct),
        merge_next_formulas(Ct,Next,G).

flatten_and_partition(and(F,G),Ct,Nt) :-
        flatten_and_partition(F,Ct1,Nt1),
        flatten_and_partition(G,Ct2,Nt2),
        union(Ct1,Ct2,Ct),
```

```
        union(Nt1,Nt2,Nt).
flatten_and_partition(next(F),[ ],[F]).
flatten_and_partition(F,[F],[ ]).

eliminate_confliction(Ct,Ct1) :-
        eliminate_confliction(Ct,Ct,Ct1).
eliminate_confliction(Ct,[F|Fs],false) :-
        member(not(F),Fs).
eliminate_confliction(Ct,[not(F)|Fs],false) :-
        member(F,Fs).
eliminate_confliction(Ct,[F|Fs],Ct1) :-
        eliminate_confliction(Ct,Fs,Ct1).
eliminate_confliction(Ct,[ ],Ct).

merge_next_formulas(false,_,false).
merge_next_formulas(_,[ ],always(true)).
merge_next_formulas(_,Nt,Nt1) :-
        mg_next_formulas(Nt,Nt1).

mg_next_formulas([F|[ ]],F).
mg_next_formulas([F,G],and(F,G)).
mg_next_formulas([F|Fs],and(F,G)) :-
        mg_next_formulas(Fs,G).

union1([false],L,L).
union1(L,[false],L).
union1(L,M,N) :- union(L,M,N).

%%% (3) Generate New Node Formulas

generate_new_node_formulas([F|Fs],ES0,H0,[NF1|NFs]) :-
        gen_new_node_formula(F,ES0,H0,NF1),
        generate_new_node_formulas(Fs,ES0,H0,NFs).
generate_new_node_formulas([ ],_,_,[ ]).

gen_new_node_formula(F,ES0,H0,[F,H1]) :-
        compute_marked_eventuality_set(F,ES1),
        define_h_set(H0,ES0,ES1,H1).

compute_marked_eventuality_set(and(F,G),S) :-
        compute_marked_eventuality_set(F,SF),
        compute_marked_eventuality_set(G,SG),
        union(SF,SG,S).
compute_marked_eventuality_set(or(F,G),S) :-
        compute_marked_eventuality_set(F,SF),
        compute_marked_eventuality_set(G,SG),
        union(SF,SG,S).
compute_marked_eventuality_set(always(F),S) :-
        compute_marked_eventuality_set(F,S).
compute_marked_eventuality_set(eventually*(F),S) :-
        compute_marked_eventuality_set(F,SF),
```

```
        union([F],SF,S).
compute_marked_eventuality_set(next(F),S) :-
        compute_marked_eventuality_set(F,S).
compute_marked_eventuality_set(F,[ ]).

define_h_set(H,ES0,ES1,H1) :-
        equivalent_set(H,ES0),
        complement(ES0,ES1,H1).
define_h_set(H,ES0,ES1,H1) :-
        complement(ES0,ES1,H0),
        union(H0,H,H1).
```

Figure 4.2. Expansion of Node Formulas

4.3. Construction of ω-Graphs

Definition 4.3. ω-Graph

Let F_0 be a formula obtained by converting the negation of 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 $N_1, N_2, \ldots, N_m$.

The ω-graph of a formula F_0 is constructed as follows.

Construction of ω-graph of F_0
Let F_0 be a formula in negation normal form.

(1) Create an initial node N_0 labelled with the initial node formula $[F_0]_{\{\}}$. Initialize the set of unexpanded nodes *Unexpanded* to $\{N_0\}$.
(2) Repeat the following (3) until *Unexpanded* $= \{\}$.
(3) Take one of nodes in *Unexpanded*. Suppose it is corresponding to $[F]_H$. Generate node formulas $[F_1]_{H_1}, [F_2]_{H_2}, \ldots, [F_n]_{H_n}$ from $[F]_H$. If there already exists a node corresponding to $[F_i]_{H_i}$, create an edge to the node. If there exists no node corresponding to $[F_i]_{H_i}$, create a new node N_i labelled with $[F_i]_{H_i}$ and an edge to the node and add the node to *Unexpanded*.

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

Example 4.3.1. Let F_0 be

$$\Box\Diamond P \wedge \Box\Diamond Q.$$

The construction of the ω-graph of F_0 proceeds as follows.

Create an initial node N_0 labelled with $[\Box\Diamond P \wedge \Box\Diamond Q]_{\{\}}$. Initialize the set *Unexpanded* to $\{N_0\}$. Take from *Unexpanded* the node N_0 labelled with $[\Box\Diamond P \wedge \Box\Diamond Q]_{\{\}}$ and generate node formulas from it. Then the following four node formulas are generated : (See example 4.2.10).

$S_1 = [\Box\Diamond P \wedge \Box\Diamond Q]_{\{P,Q\}}$,
$S_2 = [\Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{P\}}$,
$S_3 = [\Diamond^* P \wedge \Box\Diamond P \wedge \Box\Diamond Q]_{\{Q\}}$,
$S_4 = [\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{\}}$

Since there exists no node corresponding to these node formulas, N_1, N_2, N_3 and N_4 are corresponding to S_1, S_2, S_3 and S_4 , respectively, and edges to these nodes are created. And N_1, N_2, N_3 and N_4 are added to *Unexpanded*. Then, since *Unexpanded* $= \{N_1, N_2, N_3, N_4\} \neq \{\}$, take N_1 labelled with $[\Box\Diamond P \wedge \Box\Diamond Q]_{\{P,Q\}}$ as one of nodes from *Unexpanded* and generate node formulas from it. In this case, the following four node formulas are generated :

$S_5 = [\Box\Diamond P \wedge \Box\Diamond Q]_{\{P,Q\}}$,
$S_6 = [\Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{P\}}$,
$S_7 = [\Diamond^* P \wedge \Box\Diamond P \wedge \Box\Diamond Q]_{\{Q\}}$,
$S_8 = [\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{\}}$

Since there exist the node N_1, N_2, N_3 and N_4 corresponding to S_5, S_6, S_7 and S_8, respec-

tively, create edges from N_1 to those nodes. Since $Unexpanded = \{N_2, N_3, N_4\} \neq \{\ \}$, take N_2 one of nodes and repeat the procedure in this way (Appendix 1). The constructed ω-graph is shown in Figure A1.

Example 4.3.2. Let F_0 be $\Box\Diamond P \wedge \Diamond\Box\neg P$. The eventuality set ES_0 is $\{P, \Box\neg P\}$. The construction of the ω-graph of F_0 proceeds as follows. The next prefix form of the formula is

$((P \vee \bigcirc\Diamond^* P) \wedge \bigcirc\Box\Diamond P) \wedge ((\neg P \wedge \bigcirc\Box\neg P) \vee \bigcirc\Diamond^*\Box\neg P)$.

The construction of the ω-graph of F_0 proceeds as follows.

Create an initial node N_0 labelled with $[\Box\Diamond P \wedge \Diamond\Box\neg P]_{\{\}}$. Initialize the set *Unexpanded* to $\{N_0\}$. Take from *Unexpanded* the node N_0 labelled by $[\Box\Diamond P \wedge \Diamond\Box\neg P]_{\{\}}$. Then three node formulas are generated :

$S_1 = [\Box\Diamond P \wedge \Diamond^*\Box\neg P]_{\{P\}}$,
$S_2 = [\Diamond^* P \wedge \Box\Diamond P \wedge \Box\neg P]_{\{\Box\neg P\}}$
$S_3 = [\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^*\Box\neg P]_{\{\}}$

Since there exists no node corresponding to these nodes, N_1, N_2 and N_3 are created corresponding to S_1, S_2 and S_3, respectively, and edges to these nodes are also created. And add N_1, N_2 and N_3 to *Unexpanded*. Then, since $Unexpanded = \{N_1, N_2, N_3\} \neq \{\ \}$, take N_1 labelled by $[\Box\Diamond P \wedge \Diamond^*\Box\neg P]_{\{P\}}$, as one of nodes from *Unexpanded* and generate node formulas from it. In this case, three node formulas are generated :

$S_4 = [\Box\Diamond P \wedge \Diamond^*\Box\neg P]_{\{P\}}$
$S_5 = [\Diamond^* P \wedge \Box\Diamond P \wedge \Box\neg P]_{\{P, \Box\neg P\}}$
$S_6 = [\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^*\Box\neg P]_{\{P\}}$

Since there exists the node N_1 corresponding to S_4, create edges from N_1 to N_1, and since there exist no nodes corresponding to S_5 and S_6, create N_4 and N_5 corresponding to these nodes, respectively, and edges to these nodes are created. Since N_4 and N_5 are added to *Unexpanded*, it becomes $\{N_2, N_3, N_4, N_5\}$. Take N_2 from *Unexpanded* as one of nodes and repeat the procedure in this way (Appendix 2). The constructed ω-graph is shown in Figure 4.4.1.

Construction of ω-graphs is implemented in Prolog as follows.

```
construct_omega_graphs(F0,ES0,WGraph) :-
        construct([[F0,[ ]]],ES0,[ ],[[F0,[ ]]],Connect,ExistNodes),
        rename_and_list_up_omega_nodes(ExistNodes,graph([ ],Connect,[ ]),WGraph,ES0,0).

construct([ ],ES0,Connect,ExistNodes,Connect,ExistNodes).
construct([NF|Unexp],ES0,Cnt,Ext,Cnt1,Ext1) :-
        expand_nodes(NF,ES0,NFs,Cnt,Cnt0),
        create_nodes(NFs,Unexp,Ext,Unexp0,Ext0),
        construct(Unexp0,ES0,Cnt0,Ext0,Cnt1,Ext1).

expand_nodes(NF,ES0,NFs,Cnt,[[NF,NFs]|Cnt]) :-
        expand_node_formula(NF,ES0,NFs).

create_nodes([ ],Unexp,Ext,Unexp,Ext).
create_nodes([F|NFs],Unexp,Ext,Unexp1,Ext1) :-
        member(F,Ext),
        create_nodes(NFs,Unexp,Ext,Unexp1,Ext1).
create_nodes([F|NFs],Unexp,Ext,Unexp1,Ext1) :-
```

```
        member(F,Unexp),
        create_nodes(NFs,Unexp,[F|Ext],Unexp1,Ext1).
create_nodes([F|NFs],Unexp,Ext,Unexp1,Ext1) :-
        create_nodes(NFs,[F|Unexp],[F|Ext],Unexp1,Ext1).

rename_and_list_up_omega_nodes([ ],graph(Ns,Egs,Ws),graph(Ns,Egs,Ws),_,_).
rename_and_list_up_omega_nodes([NForm|NForms],graph(NFs,Cnt,WFs),
                graph(Nodes,Edges,Wnodes),ES0,Number) :-
        rename_and_listup(NForm,graph(NFs,Cnt,WFs),graph(Ns0,Egs0,Ws0),ES0,Number),
        N1 is Number+1,
        rename_and_list_up_omega_nodes(NForms,graph(Ns0,Egs0,Ws0),
                graph(Nodes,Edges,Wnodes),ES0,N1).

rename_and_listup([F,ES],graph(NFs,Cnt,WFs),
                graph([Nbr|NFs],Cnt1,[Nbr|WFs]),ES0,Nbr) :-
        equivalent_set(ES,ES0),
        replace_symbols_in_the_connections([F,ES],Nbr,Cnt,Cnt1).
rename_and_listup(NF,graph(NFs,Cnt,WFs),graph([Nbr|NFs],Cnt1,WFs),_,Nbr) :-
        replace_symbols_in_the_connections(NF,Nbr,Cnt,Cnt1).

replace_symbols_in_the_connections(NF,Nbr,[ ],[ ]).
replace_symbols_in_the_connections(NF,Nbr,[C|Cnt],[E|Egs]) :-
        replace_symbols(NF,Nbr,C,E),
        replace_symbols_in_the_connections(NF,Nbr,Cnt,Egs).

replace_symbols(NF,Nbr,[NF,NFs],[Nbr,ConnectedNodes]) :-
        replace(NF,Nbr,NFs,ConnectedNodes).
replace_symbols(NF,Nbr,[NF1,NFs],[NF1,ConnectedNodes]) :-
        replace(NF,Nbr,NFs,ConnectedNodes).

replace(NF,Nbr,[ ],[ ]).
replace(NF,Nbr,[NF|NFs],[Nbr|CntNodes]) :-
        replace(NF,Nbr,NFs,CntNodes).
replace(NF,Nbr,[NF1|NFs],[NF1|CntNodes]) :-
        replace(NF,Nbr,NFs,CntNodes).
```

Figure 4.3. Construction of ω-Graphs

4.4. Check of ω-Loop Freeness of ω-Graphs

Definition 4.4. 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 visit several nodes). If there is no ω-loop, then the graph is said to be *ω-loop free*.

Example 4.4.1. In the Appendix 1, N_1, N_4, N_6 are ω-nodes and $N_1 \rightarrow N_1$ is an ω-loop and $N_1 \rightarrow N_4 \rightarrow N_1$ is also an ω-loop. In the Appendix 2 (Figure 4.4.1), N_4 is an ω-node and the graph is ω-loop free.

The ω-loop freeness of the ω-graph is checked by the following procedure.

Checking the ω-Loop Freeness of ω-Graphs

(1) Initialize Ω to the set of all ω-nodes in the ω-graph of F_0.
(2) While $\Omega \neq \{\}$, repeat the following (3). If the repetition stops with $\Omega = \{\}$, the graph is ω-loop free.
(3) Take one of the ω-node W in Ω. Check whether there exists an ω-loop of W. If there exists such a loop, the ω-graph is not ω-loop free. Otherwise, remove W from Ω.

Whether there exists an ω-loop of W is checked as follows. Though we present a nondeterministic one below for simplicity of description, the actual implementation in Prolog searches the path by using the backtracking mechanism.

Finding An ω-Loop of W

(1) Initialize *Traced* to $\{W\}$, N_0 to W and i to 0 (A node in *Traced* is said to be *traced*. A node not in *Traced* is said to be *untraced*.)
(2) Repeat the following (3).
(3) If there is no node to which an edge outgoing from N_i leads, stop with failure. Otherwise take a node N_{i+1} to which an edge outgoing from N_i leads. If $N_{i+1} = W$, stop with answer "there exists an ω-loop of W". Otherwise, trace N_i (add N_i to *Traced*) and increment i by 1.

As we gave an intuitive explanation in example 3.2, 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 unsatisfiable.

Example 4.4.2. We will illustrate the above procedure on the ω-graph of $\Box\Diamond P \wedge \Diamond\Box\neg P$.

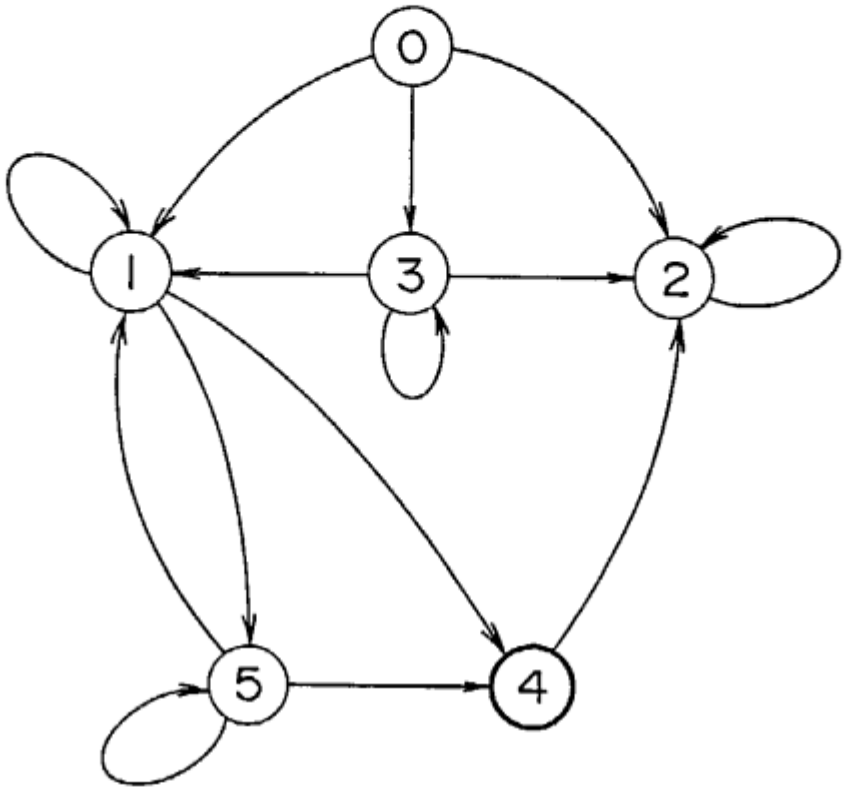


Figure 4.4.1. The ω-Graph of $\Box\Diamond$P $\wedge$ $\Diamond\Box\neg$P

(1) Find an ω-loop of N4.

0. Initially *Traced* = { N4 }.
1. Take a node N2 as one of such nodes that have an edge from N4.
2. As N2$\neq$N4 and N2 is an untraced node, add N2 to *Traced*. Then *Traced* becomes {N4, N2}.
3. Take a node N2 as one of such nodes that has an edge from N2.
4. As N2$\neq$N4 and N2 is a traced node, check another node which has an edge from N2.
5. Since there is no node that has an edge from N2, stop with failure.

(2) Select another ω-node and check

Check ω-node other than N4. However, there is no other ω-node. Thus the graph is ω-loop free.

It is easily implemented in Prolog as follows.

```
check_omega_loop_freeness (graph(Ext,Egs,[W|Ws]),'not omega-loop free') :-
        find_an_omega_loop(W,Ext,Egs).
check_omega_loop_freeness (graph(Ext,Egs,[W|Ws]),A) :-
        check_omega_loop_freeness(graph(Ext,Egs,Ws),A).
check_omega_loop_freeness (graph(Ext,Egs,[ ]),'omega-loop free').
```

```
find_an_omega_loop(W,Untraced,Egs) :-
        find_path(W,W,Untraced,Egs,Egs).

find_path(N,M,Untraced,[[N,CNs]|Egs0],Egs) :-
        take_one_node(N1,CNs),
        test(N,N1,M,Untraced,Egs).
find_path(N,M,Untraced,[[Node,_]|Egs0],Egs) :-
        find_path(N,M,Untraced,Egs0,Egs).
          .
test(N,M,M,_,_).
test(N,N1,M,Untraced,Egs) :-
        member(N1,Untraced),
        delete(N,Untraced,Untraced1),
        find_path(N1,M,Untraced1,Egs,Egs).

take_one_node(N,[N|rNs]).
take_one_node(N,[_|Ns]) :-
        take_one_node(N,Ns).
```

Figure 4.4.2. Check of ω-Loop Freeness of ω-Graphs

4.5. Sequential ω-Graphs Refutation Procedure

Lastly in this section, we show the top level of the sequential ω-graphs refutaion procedure for checking the validity of the given formula.

Sequential ω-Graphs Refutation Procedure

Negate a given formula F, compute initial node formula of the negation of given formula. Then construct the ω-graph $\mathcal{G}$ and check ω-loop freeness of $\mathcal{G}$. If $\mathcal{G}$ is ω-loop free, F is valid. Otherwise, F is not valid.

The procedure is implemented in Prolog as follows.

```
prove(F) :-
        refute(not(F),A),
        write_answer(A,F).

refute(F,A) :-
        compute_initial_node_formula(F,F0,ES0),
        construct_omega_graphs(F0,ES0,G),
        check_omega_loop_freeness(G,A).

write_answer('omega-loop free,F) :-
        pretty_print(F),
        pretty_print('is valid').
write_answer('not omega-loop free,F) :-
        pretty_print(F),
        pretty_print('is not valid').
```

Figure 4.5 Sequential ω-Graphs Refutation in Prolog

5. PARALLEL ω-GRAPHS REFUTATION PROCEDURE IN GHC

First, we discuss the parallel programming for each procedure *compute_initial_node_formula*, *expand_node_formulas*, *construct_omega_graphs* and *check_omega_loop_freeness*.

5.1. Parallel Computation of Initial Node Formula

The implementation of the procedure *compute_initial_node_formula* in GHC is quite easy. Three subprocedures *remove_implication_and_equivalence*, *move_not_inwards* and *compute_eventuality_set* were written in general recursive style in sequential version. They are easily transformed to GHC programs with a few syntactical modifications. The implementation is as follows.

For example, take the first clause of *remove_implication_and_equivalence*. If the head unification of the clause succeeds, then it forks three subprocesses which run in parallel. The two of them try to solve *remove_implication_and_equivalence*($P, P1$) and *remove_implication_and_equivalence*($Q, Q1$), while the third one tries to bind A to *and*($P1, Q1$) which is propagated to unify with a head of some other process.

```
compute_initial_node_formula(F,NNF,EveSet) :-
        negation_normal_form(F,NNF),
        compute_eventuality_set(NNF,EveSet).

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

remove_implication_and_equivalence( and(P,Q), A ) :-
        remove_implication_and_equivalence(P,P1),
        remove_implication_and_equivalence(Q,Q1),
        A=and(P1,Q1).
remove_implication_and_equivalence( or(P,Q), A ) :-
        remove_implication_and_equivalence(P,P1),
        remove_implication_and_equivalence(Q,Q1),
        A=or(P1,Q1).
remove_implication_and_equivalence( always(P), A ) :-
        remove_implication_and_equivalence(P,P1),
        A=always(P1).
remove_implication_and_equivalence( eventually(P), A ) :-
        remove_implication_and_equivalence(P,P1),
        A=eventually(P1).
remove_implication_and_equivalence( next(P), A ) :-
        remove_implication_and_equivalence(P,P1),
        A=next(P1).
remove_implication_and_equivalence( imply(P,Q), A ) :-
        remove_implication_and_equivalence(P,P1),
        remove_implication_and_equivalence(Q,Q1),
        A=or(not(P1),Q1).
remove_implication_and_equivalence( equivalence(P,Q), A ) :-
        remove_implication_and_equivalence(P,P1),
        remove_implication_and_equivalence(Q,Q1),
        A=or(and(P1,Q1),and(not(P1),not(Q1))).
```

```
remove_implication_and_equivalence( not(P), A ) :-
        remove_implication_and_equivalence( P, P1 ),
        A=not(P1).
remove_implication_and_equivalence( P, A ) :- atom(P) | A=P.

move_not_inwards( and(P,Q), A ) :-
        move_not_inwards(P,P1),
        move_not_inwards(Q,Q1), A=and(P1,Q1).
move_not_inwards( or(P,Q), A ) :-
        move_not_inwards(P,P1),
        move_not_inwards(Q,Q1), A=or(P1,Q1).
move_not_inwards( always(P),A ) :-
        move_not_inwards(P,P1), A=always(P1).
move_not_inwards( eventually(P), A ) :-
        move_not_inwards(P,P1), A=eventually(P1).
move_not_inwards( next(P), A ) :-
        move_not_inwards(P,P1), A=next(P1).
move_not_inwards( not(and(P,Q)), A ) :-
        move_not_inwards(not(P),P1),
        move_not_inwards(not(Q),Q1), A=or(P1,Q1).
move_not_inwards( not(or(P,Q)), A ) :-
        move_not_inwards(not(P),P1),
        move_not_inwards(not(Q),Q1), A=and(P1,Q1).
move_not_inwards( not(always(P)),A ) :-
        move_not_inwards(not(P),P1), A=eventually(P1).
move_not_inwards( not(eventually(P)), A ) :-
        move_not_inwards(not(P),P1), A=always(P1).
move_not_inwards( not(next(P)), A ) :-
        move_not_inwards(not(P),P1), A=next(P1).
move_not_inwards( not(not(P)), A ) :-
        move_not_inwards(P,A).
move_not_inwards( not(P),A ) :- atom(P) | A=not(P).
move_not_inwards( P, A ) :- atom(P) | A=P.

compute_eventuality_set(and(A,B),ES) :-
        compute_eventuality_set(A,ES1),
        compute_eventuality_set(B,ES2),union(ES1,ES2,ES).
compute_eventuality_set(or(A,B), ES) :-
        compute_eventuality_set(A,ES1),
        compute_eventuality_set(B,ES2),union(ES1,ES2,ES).
compute_eventuality_set(always(A),ES) :-
        compute_eventuality_set(A,ES).
compute_eventuality_set(next(A), ES) :-
        compute_eventuality_set(A,ES).
compute_eventuality_set(not(A), ES) :-
        compute_eventuality_set(A,ES).
compute_eventuality_set(eventually(A), ES) :- ES=[A].
compute_eventuality_set(A, ES) :- atom(A) | ES=[ ].
```

Figure 5.1. Parallel Computation of Initial Node Formula

5.2. Parallel Expansion of Node Formulas

As the procedure *expand_node_formulas* is rather complicated, we divide it into some modules and make clear what processes should run in parallel. Among the rules in the convertion from NPF to DNF, DNF2~DNF6 can be applied in parallel, and h-set can be computed at the same time. Therefore, we divide *expand_node_formulas* into the following three sequential subprocesses, and consider a parallel programming of each subprocess.

1. Negation Normal Form (negation_normal_form)
   convertion from NNF to NPF
2. Distribute $\wedge$ over $\vee$ (distribute_and_over_or)
   convertion by using DNF1
3. Simplification (simplify_formulas)
   conversion by using DNF2~DNF6

The process of *negation_normal_form* is implemented in GHC similar to the process in recursive style such as *remove_implication_and_equivalence* .

As for the process of *distribute_and_over_or*, we use the Wand's algorithm[Wand 80] in order to get high efficiency. Though it is suitable rather for sequential execution and lacking clarity, it provides high efficiency. As this subprocedure has more sequential characters.

The output of *distribute_and_over_or* is a disjunction of the conjunction of temporal literals. Each conjunction is of the form

$F_1, ..., F_n, \circ G_1, ..., \circ G_m$

where $F_1, ..., F_n$ are formulas which include no temporal operators The process *simplify_formulas* executes the convertion of the formulas by the rules DNF2~DNF6 and generate a set of node formulas. Do the following procedure sequentially for each conjunction.

(1) Initially set $NextFormulas = \{\}$. For each temporal literal F in the conjunction, do the following in parallel.
   If $\neg F$ is included, stop all the other processes and return the result *false*.
   If F is included as a formula F_i, stop the process of F_i.
   If F is in the form of $\circ G_i$, G_i is added to $NextFormulas$.
   Otherwise, do nothing.
(2) For $NextFormulas$, do the following in parallel.
   If $NextFormulas = \{\}$, then return $\Box true$ as a result and compute the h-set.
   If $NextFormulas = \{G_1, ..., G_m\}$, then return $G_1 \wedge ... \wedge G_m$ and compute the h-set.

The implemantation is shown below.

In the program, '&' denotes the sequential execution. Note that subprocesses of *simplify_formulas* have 'JudgeStop' as an argument. It is a termination flag in order to terminate other processes related to a conjunction C at the moment when C is found to be *false*.

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

%%% Next Prefix Form
```

```
next_prefix_form(and(P,Q),A) :-
        next_prefix_form(P,P1),
        next_prefix_form(Q,Q1), A=and(P1,Q1).
next_prefix_form(or(P,Q), A) :-
        next_prefix_form(P,P1),
        next_prefix_form(Q,Q1), A=or(P1,Q1).
next_prefix_form(always(P),A) :-
        next_prefix_form(P,P1),
        A=and(P1,next(always(P))).
next_prefix_form(eventually(P), A) :-
        next_prefix_form(P,P1),
        A=or(P1,next(m_eventually(P))).
next_prefix_form(m_eventually(P),A) :-
        next_prefix_form(P,P1),
        A=or(P1,next(m_eventually(P))).
next_prefix_form(not(P),A) :- A=not(P).
next_prefix_form(P, A) :- atom(P) | A=P.

%%% Distribute ∧ over ∨ (Wand's Algorithm)

distribute_and_over_or(Formula,Answer) :-
        distribute_and_over_or(Formula,[ ],[ ],Answer).

distribute_and_over_or( or(X1,X2),Lits,Rest,Ans ) :-
        distribute_and_over_or(X1,Lits,Rest,A1),
        distribute_and_over_or(X2,Lits,Rest,A2),
        append(A1,A2,Ans).
distribute_and_over_or( and(X1,X2),Lits,Rest,Ans ) :-
        collect_and(and(X1,X2),NewRest),
        union(Rest,NewRest,[Rest1|Rest2]),
        distribute_and_over_or(Rest1,Lits,Rest2,Ans).
distribute_and_over_or(X,Lits,Rest,Ans) :- otherwise |
        distribute_and(X,Lits,Rest,Ans).

collect_and( and(X1,X2),Rest ) :-
        collect_and(X1,Rest1),
        collect_and(X2,Rest2),
        union(Rest1,Rest2,Rest).
collect_and( X,Rest ) :- otherwise | Rest=[X].

distribute_and(X,Lits,Rest,Ans) :-
        member(X,Lits,yes) |
        distribute_next_and(Lits,Rest,Ans).
distribute_and(X,Lits,Rest,Ans) :-
        otherwise |
        distribute_next_and([X|Lits],Rest,Ans).
distribute_next_and( Lits,[ ],Ans ) :- Ans=[Lits].
distribute_next_and( Lits,[Rest1|Rest2],Ans ) :-
        distribute_and_over_or(Rest1,Lits,Rest2,Ans).

%%% Simplification
```

```
simplify_formulas(ES0,H0,Xn,[F|DNF],NodeFormulas) :-
        ( gather_next(JudgeStop,F,Next),
        check_contradiction(JudgeStop,F,[ ]) ) &
        supplement_next_part(JudgeStop,Next,NewNext) &
        make_node_formula_list(ES0,H0,Xn,NewNext,NodeFormulas,EndNodeFs),
        simplify_formulas(ES0,H0,Xn,DNF,EndNodeFs).
simplify_formulas(ES0,H0,Xn,[ ],NodeFormulas) :-
        NodeFormulas=[[end,end]].

gather_next(JudgeStop,[next(F)|Fs],Next) :-
        prolog(var(JudgeStop)) |
        Next=[F|EndNext],
        gather_next(JudgeStop,Fs,EndNext).
gather_next(JudgeStop,[F|Fs],Next) :-
        prolog(var(JudgeStop)),
        F\=next(_) |
        gather_next(JudgeStop,Fs,Next).
gather_next(stop,_,_).
gather_next(JudgeStop,[ ],Next) :- Next=[ ].

check_contradiction(Judge,[ ],_).
check_contradiction(Judge,[not(F)|Fs],C) :-
        member(F,C,yes) |
        Judge='stop'.
check_contradiction(Judge,[F|Fs],C) :-
        F\=not(G),
        member(not(F),C,yes) |
        Judge='stop'.
check_contradiction(Judge,[F|_],_) :-
        F=='false' |
        Judge='stop'.
check_contradiction(Judge,[F|Fs],C) :- otherwise |
        check_contradiction(Judge,Fs,[F|C]).

supplement_next_part(stop,_,NewF) :-
        NewF='false'.
supplement_next_part(Judge,[ ],NewF) :-
        prolog(var(Judge)) |
        NewF=[always(true)]. supplement_next_part(Judge,F,NewF) :-
        prolog(var(Judge)), F\=[ ] |
        NewF=F.

make_node_formula_list(ES0,H0,Xn,F,NF,EndOfNF) :- F\=false |
        merge_next(F,F1),
        compute_h_set(ES0,H0,F1,H),
        NF=[[Xn,(F1,H)]|EndOfNF].
make_node_formula_list(_,_,_,false,NF,EndOfNF) :-
        NF=EndOfNF.

merge_next([F|Fs],F1) :- Fs\=[ ] |
        F1=and(F2,F),
```

```
        merge_next(Fs,F2).
merge_next([F],F1) :- F1=F.

compute_h_set(ES0,H0,F,H) :-
        equal(ES0,H0,no) |
        compute_marked_eventuality_set(F,ES1),
        delete(ES0,ES1,H1),
        union(H0,H1,H).
compute_h_set(ES0,H0,F,H) :-
        otherwise |
        compute_marked_eventuality_set(F,ES1),
        delete(ES0,ES1,H).

compute_marked_eventuality_set(and(P,Q),H) :-
        compute_marked_eventuality_set(P,H1),
        compute_marked_eventuality_set(Q,H2),
        union(H1,H2,H).
compute_marked_eventuality_set(or(P,Q),H) :-
        compute_marked_eventuality_set(P,H1),
        compute_marked_eventuality_set(Q,H2),
        union(H1,H2,H).
compute_marked_eventuality_set(always(P),H) :-
        compute_marked_eventuality_set(P,H).
compute_marked_eventuality_set(m_eventually(P),H) :-
        H=[P].
compute_marked_eventuality_set(F,H) :-
        otherwise | H=[ ].
```

Figure 5.2. Parallel Expansion of Node Formulas

5.3. Parallel Construction of ω-Graphs

In the *construct_omega_graphs* procedure, we try to execute *expand_node_formulas* procedure for each node 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 *expand_node_formulas* 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 manages creation and abortion of node processes.

The process *construct_omega_graph* forks three subprocesses *graph_manager*, *node_process* and *mutiplexer*. *Graph_manager* 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. *Node_process* expands the corresponding node formula. *Multiplexer* arranges generated node formulas in order by using the Kusalik's algorithm (6.1.4). 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 node number, $OutStrm$ is the 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 the output in this form.

Let F_0 be a given formula in NNF.

(1) Create the processes of graph manager GM, multiplexer MUX, and node process NP_0 correspondoing 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.

Then terminate.

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

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

Take a node formula NF_i from *MergedStream*.

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*.

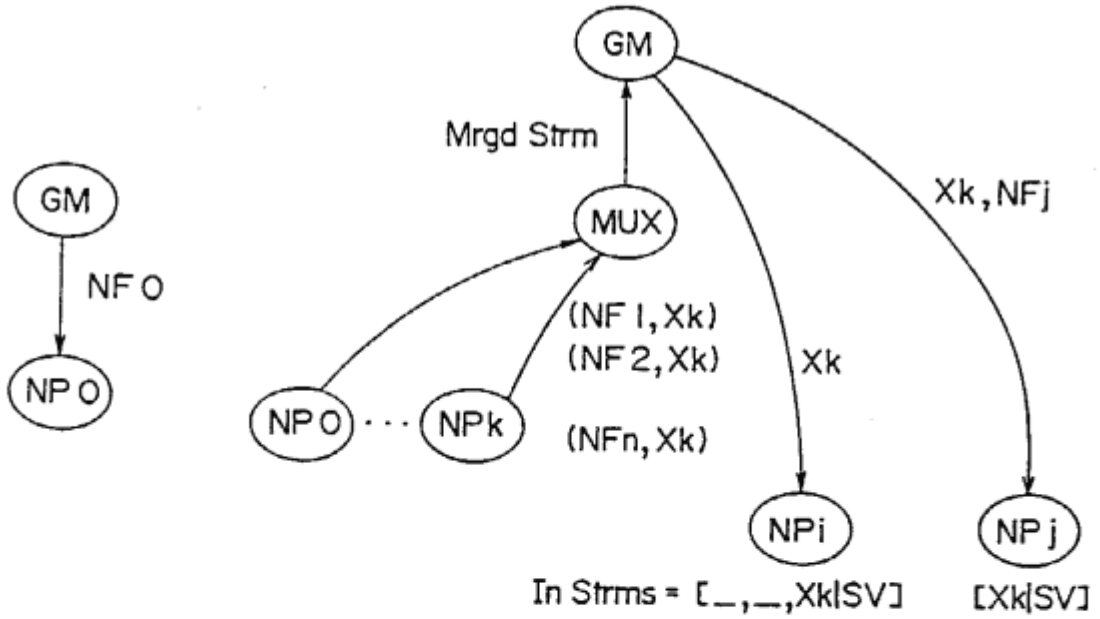


Figure 5.3.1. Stream Flow in Parallel Construction of ω-Graph

```
construct_omega_graphs(JudgeStop,ES0,F,Graph) :-
        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).

%%% Node Process

node_process(JudgeStop,ES0,NodeNo,[[No,Xn,_,(F,H)]|Exist],StrmList) :-
        prolog(var(JudgeStop)) |
        expand_node_formulas(F,H,Xn,ES0,ExpFormulas),
        No:=NodeNo,
        StrmList=[ExpFormulas|EndStrm],
        NodeNo1:=NodeNo+1,
        node_process(JudgeStop,ES0,NodeNo1,Exist,EndStrm).
node_process(_,_,_,[ ],StrmList) :- StrmList=[ ].
node_process(stop,_,_,_,_).

%%% Mutiplexer

multiplexer(JudgeStop,StrmList,Mrg) :-
        merge(JudgeStop,StrmList,Mrg).

merge(JudgeStop,StrmList,Mrg) :-
        prolog(var(JudgeStop)),
        wait_strm(JudgeStop,StrmList,ActiveStrm,CheckStrms) |
        arrival(JudgeStop,ActiveStrm,CheckStrms,Mrg).
merge(JudgeStop,[ ],E) :- E=[ ].
merge(stop,_,_).

arrival( JudgeStop,[[E,F]|ActiveStrm],CheckStrms,OutStrm ) :-
        prolog(var(JudgeStop)) |
        OutStrm=[[E,F]|Mrg],
        busy_wait( JudgeStop,CheckStrms,[ActiveStrm],Mrg ).
arrival( JudgeStop,[ ],CheckStrms,OutStrm ) :-
        prolog(var(JudgeStop)) |
        OutStrm=Mrg,
        busy_wait( JudgeStop,CheckStrms,[ ],Mrg ).
arrival( stop,_,_,_).

busy_wait( JudgeStop,[[[E,F]|Strm]|CheckStrms],CheckedStrms,OutStrm ) :-
        prolog(var(JudgeStop)) |
        OutStrm=[[E,F]|Mrg],
        busy_wait( JudgeStop,CheckStrms,[Strm|CheckedStrms],Mrg ).
busy_wait( JudgeStop,[[ ]|CheckStrms],CheckedStrms,OutStrm ) :-
        prolog(var(JudgeStop)) |
        OutStrm=Mrg,
        busy_wait( JudgeStop,CheckStrms,CheckedStrms,Mrg ).
busy_wait( JudgeStop,[Strm|CheckStrms],CheckedStrms,Mrg ) :-
        prolog(var(JudgeStop)),
        prolog(var(Strm)) |
        busy_wait( JudgeStop,CheckStrms,[Strm|CheckedStrms],Mrg ).
busy_wait( JudgeStop,[ ],CheckedStrms,OutStrm ) :-
        prolog(var(JudgeStop)) |
        OutStrm=Mrg,
```

```
        merge( JudgeStop,CheckedStrms,Mrg ).
busy_wait( JudgeStop,Strms,CheckedStrms,Mrg ) :-
        prolog(var(JudgeStop)),
        prolog(var(Strms)) |
        make_stream( JudgeStop,Strms,CheckedStrms,CheckStrms ),
        merge( JudgeStop,CheckStrms,Mrg ).
busy_wait( stop,_,_,_).

make_stream( JudgeStop,X,[Y|L],L1 ) :-
        prolog(var(JudgeStop)) |
        L1=[Y|S],
        make_stream( JudgeStop,X,L,S ).
make_stream( JudgeStop,X,[ ],L1 ) :-
        prolog(var(JudgeStop)) |
        L1=X.
make_stream( stop,X,_,L1 ) :- L1=X.

wait_strm(JudgeStop,[[[E,F]|Strm]|StrmList],St1,St2) :-
        prolog(var(JudgeStop)) |
        St1=[[E,F]|Strm],
        St2=StrmList.
wait_strm(JudgeStop,[[ ]|StrmList],St1,St2) :-
        prolog(var(JudgeStop)) |
        St1=[ ],
        St2=StrmList.
wait_strm(JudgeStop,[Strm|StrmList],Active,St) :-
        prolog(var(JudgeStop)),
        prolog(var(Strm)),
        wait_strm(JudgeStop,StrmList,ActiveStrm,CheckStrms) |
        Active=ActiveStrm,
        St=[Strm|CheckStrms].
wait_strm(stop,_,_,_).

%%% Graph Manager

graph_manager(JudgeStop,ES0,No,[[Xn,(F,H)]|Fs],Graph,Exist) :-
        prolog(var(JudgeStop)) |
        exist_node(JudgeStop,ES0,[Xn,(F,H)],Graph,Exist) &
        graph_manager(JudgeStop,ES0,No,Fs,Graph,Exist).
graph_manager(JudgeStop,ES0,No,[[end,end]|Fs],Graph,Exist) :-
        prolog(var(JudgeStop)) |
        No1:=No+1,
        check_graph_termination(JudgeStop,No1,Fs,Graph,Exist) &
        graph_manager(JudgeStop,ES0,No1,Fs,Graph,Exist).
graph_manager(_,_,_,[ ],_,_).
graph_manager(stop,_,_,_,_,_).

exist_node(stop,_,_,_,_).
exist_node(JudgeStop,ES0,[Xn,(F,H)],Graph,Exist) :-
        prolog(var(JudgeStop)) |
        exist_node(ES0,[Xn,(F,H)],Graph,Exist).
```

```
exist_node(ES0,[Xn,(F,H)],Graph,Exist) :-
        check_member((F,H),Exist,InStrm,yes) |
        search_end(InStrm,EndIn),
        EndIn=[Xn|NewEndIn].
exist_node(ES0,[X0,(F,H)],Graph,Exist) :-
        otherwise |
        check_type(ES0,H,Type),
        In=[X0|EndIn],
        search_end(Exist,EndExist),
        EndExist=[[No,Xn,In,(F,H)]|EndExist1],
        search_end(Graph,EndGraph),
        EndGraph=[[No,Xn,In,Type]|EndGraph1].

search_end([G|Gs],EndG) :- search_end(Gs,EndG).
search_end(G,EndG) :- prolog(var(G)) | EndG=G.

check_type(ES0,H,Type) :- equal(ES0,H,yes) | Type='omega'.
check_type(ES0,H,Type) :- otherwise | Type='not-omega'.

check_member((F,H),[[_,_,In,(Fn,Hn)]|Exist],Instrm,Ans) :-
        F==Fn,H==Hn |
        Instrm=In,
        Ans='yes'.
check_member((F,H),Exist,Instrm,Ans) :-
        prolog(var(Exist)) |
        Instrm='no',Ans='no'.
check_member((F,H),[_|Exist],Instrm,Ans) :- otherwise |
        check_member((F,H),Exist,Instrm,Ans).

check_graph_termination(JudgeStop,Num0,Fs,Graph,Exist) :-
        prolog(var(JudgeStop)),
        prolog(var(Fs)) |
        check_graph_termination(Num0,0,Exist,Graph).
check_graph_termination(stop,_,_,_,_).
check_graph_termination(JudgeStop,Num0,[[_,_]|Fs],_,_) :-
        prolog(var(JudgeStop)) | true.
check_graph_termination(Num0,Num1,Exist,Graph) :-
        prolog(var(Exist)),
        Num0 < Num1 | true.
check_graph_termination(Num0,Num1,Exist,Graph) :-
        prolog(var(Exist)),
        Num0=:=Num1 |
        Exist=[ ],
        terminate_graph(Graph).
check_graph_termination(Num0,Num,[[No,_,_,_]|Exist],Graph) :-
        integer(No) |
        Num1:=Num+1,
        check_graph_termination(Num0,Num1,Exist,Graph).
check_graph_termination(_,_,_,_,_) :- otherwise | true.

terminate_graph([[_,_,In,_]|Graph]) :-
```

```
        search_end(In,EndIn) &
        EndIn=[ ],
        terminate_graph(Graph).
terminate_graph(Graph) :-
        prolog(var(Graph)) |
        Graph=[ ].
```

Figure 5.3.2. Parallel Construction of ω-Graphs in GHC

5.4. Parallel Check of ω-Loop-Freeness of ω-Graphs

The program of *check_of_omega_loop_freeness* in Prolog is in typical backtracking style. 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. With each node $NodeNmbr$, we associate a stream variable $OutStrm$. Let M be the number of nodes in the graph. The graph is represented as a list of M quadruples $(NodeNmbr, OutStrm, InStrms, NodeType)$ where $NodeNmbr$ is the node number, $OutStrm$ is its associated stream variable, $InStrms$ is a list of the stream variables associated with the node which has an edge flowing into the node $NodeNmbr$ and $NodeType$ is either *omega* or *not_omega*. Moreover, extra argument 'Judgestop' is added as a termination flag. It can stop other processes as soon as an ω-loop is found.

Example 5.4. Let $X1, X2$ and $X3$ be stream variables.
[(0,X0,[ ],not-omega),(1,X1,[X0,X1,X2],omega),(2,X2,[X0,X1,X2],not-omega)]
represents the graph shown in example 3.2.(Figure 3.2).

The following procedure shows the parallel checking the ω-loop freeness of the ω-graph.

Parallel Check of ω-Loop-Freeness of ω-Graphs
(1) Initialize C as the set of all nodes in the ω-graph. 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, let L_n be [ ] and repeat the following (3).
(3) If $C = \{\ \}$, then stop with failure.
If $JudgeStop = stop$, then stop with 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 Ln, extract a node from C.
(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) :-
        check_omega_loop_freeness(JudgeStop,Graph,Graph) &
        write_result(JudgeStop).

write_result(JudgeStop) :-
        prolog(var(JudgeStop)) |
        prolog(pretty_print('The graph is omega-loop free')).
write_result(stop) :-
```

```
        prolog(pretty_print('The graph is not omega-loop free')).

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

find_omega_loop(JudgeStop,[G|Graph],Ln,N,A) :-
        prolog(var(JudgeStop)) |
        node1(JudgeStop,Graph,Ln,N,A).
find_omega_loop(_,[ ],_,_,_).
find_omega_loop(stop,_,_,_,_).

node1(JudgeStop,Graph,Ln,N,[A|X]) :-
        prolog(var(JudgeStop)) |
        node2(JudgeStop,Graph,Ln,N,A),
        node1(JudgeStop,Graph,Ln,N,X).
node1(_,_,_,_,[ ]).
node1(stop,_,_,_,_).

node2(JudgeStop,Graph,Ln,N,[A|X]) :-
        N=\=A,
        member(A,Ln,no) |
        append([A],Ln,NewLn) &
        find_omega_loop(JudgeStop,Graph,NewLn,N,X).
node2(JudgeStop,_,Ln,N,[A|_]) :-
        N=:=A,
        prolog(var(JudgeStop)) |
        JudgeStop='stop'.
node2(_,_,_,_,_) :- otherwise | true.
```

Figure 5.4. Parallel Check of ω-Loop Freeness of ω-Graphs

5.5. Parallel ω-graphs Refutation Procedure

The parallel ω-graphs refutation procedure for checking the validity of the given formula is implemented as follows.

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

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

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

Figure 5.5. Parallel ω Graphs Refutation in Prolog

Refute consists of three parallel processes *compute_initial_node_formula*, *construct_omega_graphs* and *check_omega_loop_freeness*. Each process 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.

6. PARALLEL PROGRAMMING METHODOLOGY IN GHC

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

6.1. General Principles for Enhancing Concurrency

First, we discuss on general principles for enhancing concurrency.

6.1.1. Early Publication

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

Example 6.1.1. Negation Normal Form
Consider the program of *negation_normal_form*.

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

Compare the following two programs of *remove_implication_and_equivalence*.

[Program (A)]

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

[Program (B)]

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

In program (A), the active part is not executed before trust operation, that is, publication of *Answer* must wait until both *remove_implication_and_equivalence*$(F, F1)$ and *remove_implication_and_equivalence*$(G, G1)$ succeed. As far as *Answer* is not yet instantiated to a non-variable term, the head unification of *move_not_inwards* is suspended. Though program (A) is more suitable for debugging on sequential machines because of its simplicity of computation trace, it is inefficient for parallel execution. *move_not_inwards* process must wait for all implications and equivalences to be eliminated from inside *not* in order to proceed. On the other hand, in program (B), three processes can run in parallel so that *Answer* is propagated as soon as *Answer* $= or(not(F1), G1)$ is executed, which allows the head unification of *move_not_inwards*. In spite of the complicated computation trace, program (B) provides high concurrency. We should adopt program (B) after debugging.

6.1.2. Early Commitment

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

Example 6.1.2. Union

Compare the following two programs of *union*.

[Program (C)]

```
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) :- S=S2.
union(S1,[ ],S) :- 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) :- Answer=no.
```

[Program (D)]

```
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) :- S=S2.
union(S1,[ ],L1,L2,S) :- 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) :- Answer=no.
```

In program (C), if *member* in the passive part succeeds with an anwer *no*, *union* does not return the answer until all the elements of $S2(S1)$ are checked. Therefore, if the stream $S2(S1)$ is partially instantiated, the unification of *member* in the passive part is suspended. On the other hand, in program (D), 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, though these additional arguments consume additional memory space. 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.

6.1.3. Decision Distribution

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

Example 6.1.3. Bounded Buffer

Consider the bounded buffer problem [Furukawa and Takeuchi 85],[Ueda 85].

[Program (E)]

```
bounded_buffer_communication :- produce(0,100,B), consume(B).
```

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

consume([H|Strm]) :- H\='EOS' | write(H), consume(Strm).
consume([H|Strm]) :- H=='EOS' | true.
```

[Program (F)]

```
bounded_buffer_communication :- 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=[ ].
```

In program (E), *produce* generates a stream of integers and sends it to the buffer. When an integer is generated, a slot in the buffer is created and the integer is putted. *Consume* reads the value from the head of the buffer. In program (F),*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. In program (E), *consume* is suspended until *produce* generates the value for some slot, and processing of the next slot also have to wait. On the other hand, in program (F), *buffer* only manages the relations of slots and the values put to each slot is decided independently. This is a typical example which shows the effectiveness of decision distribution by using difference lists.

6.1.4. Efficient Communication Network

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. Stream is dynamic flow of an arbitrary data structure. The length of stream is not fixed, while that of list is fixed. Parallel processes shares common stream variables to communicate each other by sending messages by partially instantiating streams. When a message arrives at one process, the process computes with this message and proceeds to the next stage. Process communication can be supported only by this mechanism and special elements such as superviser or controller are unneccessary.

The communication takes various form depending on the problem, *no communication, one-way one-time communication, one-way stream communication, two-way stream communication, communication through a dedicated merge process for multiplexing streams to shared resource.*

Example 6.1.4. Merge

We show below two kinds of 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].

[Program (G)]

```
merge_streams :- generate_streams([S1,S2,...,Sn]), merge(S1,S2,...,Sn,MrgdStrm).

merge([X|X1],X2,...,Xn,MrgdStrm) :-
        MrgdStrm=[X|NewStrm], merge(X2,...,Xn,X1,NewStrm).
    ..........
merge(X1,...,[X|Xi],...,Xn,MrgdStrm) :-
        MrgdStrm=[X|NewStrm], merge(X1,...,X_{i-1},X_{i+1},..,Xn,Xi,NewStrm).
    ..........
merge(X1,X2,...,[X|Xn],MrgdStrm) :-
        MrgdStrm=[X|NewStrm], merge(X1,X2,...,Xn,NewStrm).

merge([[ ]|X1],X2,...,Xn,MrgdStrm) :- merge(X2,...,Xn).
    ..........
merge(X1,...,Xi,...,[[ ]|Xn],MrgdStrm) :- merge(X1,...,X_{n-1}).
```

[Program (H)] multiplexer

```
merge_streams :- generate_streams(StrmList), merge(StrmList,MrgdStrm).

merge(StrmList,Mrg) :- wait_strm(StrmList,Out,CheckList) |
        merge1(Out,CheckList,Mrg).
merge([ ],E) :- E=[ ].

merge1([E|StrmList],CheckList,Out) :- busy_wait(CheckList,Mrg,NewStrmList,NewMrg) |
        Out=[E|Mrg], merge([StrmList|NewStrmList],NewMrg).
merge1([ ],CheckList,Out) :- busy_wait(CheckList,Mrg,NewStrmList,NewMrg) |
        Out=Mrg, merge(NewStrmList,NewMrg).

busy_wait([[E|Strm]|CheckList],Out,St,NewMrg) :-
        Out=[E|Mrg], St=[Strm|NewStrmList],
        busy_wait(CheckList,Mrg,NewStrmList,NewMrg).
busy_wait([[ ]|CheckList],Mrg,NewStrmList,NewMrg) :-
        busy_wait(CheckList,Mrg,NewStrmList,NewMrg).
busy_wait([Strm|CheckList],Mrg,St,NewMrg) :- prolog(var(Strm)) |
        St=[Strm|NewStrmList], busy_wait(CheckList,Mrg,NewStrmList,NewMrg).
busy_wait([ ],Mrg,E,M) :- E=[ ], M=Mrg.
busy_wait(Strms,Mrg,St,NewMrg) :- prolog(var(Strms)) |
        St=Strms, NewMrg=Mrg.

wait_strm([[E|Strm]|StrmList],St1,St2) :-
        St1=[E|Strm], St2=StrmList.
wait_strm([[ ]|StrmList],St1,St2) :-
        St1=[ ], St2=StrmList.
wait_strm([Strm|StrmList],Active,St) :-
```

```
prolog(var(Strm)), wait_strm(StrmList,ActiveStrm,CheckStrms) |
Active=ActiveStrm, St=[Strm|CheckStrms].
```

Program (G) shows the method of merging of n processes (n is fixed) by building communication network specific to the problem. When a message is sent through the output stream of a sender process, it is received by the receiver process and added to $MrgdStrm$, and the priority of the sender process is lowered. On the other hand, program (H) shows the method of merging by multiplexer based on the Kusalik's algorithm.

Multiplexer is used for fair merge of streams in communication from several *sender* processes to one *receiver* process. Multiplexer manages a stream of stream variables, each of which is output from each sender process. It receives messages from the current sender processes and return the merged stream to the receiver process. 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. Multiplexer has the following functions :

1. $receive(X, M, NewM)$: A message X is received through the multiplexer M when the message X was sent through some stream variable in multiplexer M before. M is updated to a new multiplexer $NewM$.

2. $add(SV, M, NewM)$: A new stream variable SV is added to the multiplexer M. M is updated to $NewM$.

If some sender process generates some output, a *merge process* in his solution receives it, makes the priority of this sender process lower and check other processes whether they have generated output. If the merge process receives [ ] as a sign of end_of_stream from some sender process, it aborts that process. Multiplexer is used to treat multiple node processes in the *construct_omega_graph* procedure.

In general, the less communication we have,the more concurrency we can enjoy. But each problem has the lower bound of the amount of necessary communication specific to the problem. We have to balance the burden to build communication network specific to the problem and the burden to merge messages from sender processes the number of which changes dynamically.

6.1.5. Equal Opportunity

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 process.

Example 6.1.5. Member

Consider the following two programs.

[Program (I)]

```
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 (C)]

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

Program (I) is a direct translation from Prolog version. The predicate *otherwise* succeeds when the passive part of all other OR-parallel clauses have failed. Compare the definitions of *member* in program (I) and that in program (C). In program (I), *member* has two arguments and use *otherwise*, which is harmful since the execution depends on the passive part of other clauses. On the other hand, program (C) realizes fair OR-parallel execution in the passive part. Therefore, if we put a predicate like *member* in the passive part, we try to use the predicate *otherwise* as less as possible since it is against the principles of parallel programming. To avoid the use, we should write passive parts symmetrically, which is reduced to the equal opportunity of decision, add an argument, which is seen in the definition of *member* in previous section.

6.2. Programming Paradigms in GHC

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

6.2.1. Synchronization in Passive Parts

It is an essential mechanism of GHC that any piece of unification invoked in the passive part of a clause cannot instantiate a variable appeaing in the caller. It gives GHC a remarkable characteristic as a programming language. we should not violate this synchronizationn mechanism when we apply some technique such as partial evaluation to GHC programs.

Example 6.2.1. Partial Evaluation

We consider an example of *remove_implication_and_equivalence* again.

[Program (J)]

```
remove_implication_and_equivalence(imply(F,G), Answer) :-
        remove_implication_and_equivalence(not(F),F1),
```

```
        remove_implication_and_equivalence(G,G1)
        Answer=or(F1,G1).
```

[Program (B)]

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

In program (J), 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 instanciating $F, F1$. Therefore, we can apply partial evaluation. Program (B) is the result, in which evaluation proceeds one more step ahead than that in program (J).

On the other hand, the following is the GHC program of the problem of stream communication discussed in [Brock and Ackerman 81].

[Program (K)]

```
t1(In1,Out) :- s1(In1,In2,Out), plus1(Out,In2).
t2(In1,Out) :- s2(In1,In2,Out), plus1(Out,In2).

s1(In1,In2,Out) :- duplicate(In1,M1), duplicate(In2,M2), merge(M1,M2,M), p1(M,Out).
s2(In1,In2,Out) :- duplicate(In1,M1), duplicate(In2,M2), merge(M1,M2,M), p2(M,Out).

p1([ ],Out) :- Out=[ ].
p1([X|NewIn],Out) :- Out=[X|NewOut],first(NewIn,NewOut).
p1([X,Y|NewIn],Out) :- Out=[X,Y].

p2([ ],Out) :- Out=[ ].
p2([X,Y|NewIn],Out) :- Out=[X,Y].

first([Y|In],Out) :- Out=[Y].

duplicate([ ],Out) :- Out=[ ].
duplicate([X|_],Out) :- Out=[X,X].

merge([ ],In2,Out) :- Out=In2.
merge(In1,[ ],Out) :- Out=In1.
merge([X|In1],In2,Out) :- Out=[X|NewOut], merge(In1,In2,NewOut).
merge(In1,[X|In2],Out) :- Out=[X|NewOut], merge(In1,In2,NewOut).

plus1([X|In],Out) :- Y:=X+1, Out=[Y|Out1], plus1(In,Out1).
plus1([ ],Out) :- Out=[ ].
```

Both of $p1$ and $p2$ generate as output the first two values received from input streams. However, $p1$ can generate if it receives at least one value, while $p2$ waits until it receives two values. $s1$ and $s2$ generate as output the first two values received from the merged stream of two input streams by using the processes $p1$ and $p2$, respectively. At last, $s1$ and $s2$

generate teh same output, but the difference of the behaviors of these two processes causes the difference between the outputs of $t1$ and $t2$. $t1$ and $t2$ show a kind of feedback system in which the output value effects on the next value of input stream. In the process $t1$, $s1$ receives the first value from the stream $In1$, generates it and it calls the goal $first$. Then it can receive the second value either from the stream $In1$ or $In2$. On the other hand, in the process $t2$, the head unification of $p2$ suspends until $s2$ receives two values from the stream $In1$. therefore, $t2$ always generates as output the first two values received from $In1$.

As for this program, we cannot apply partial evaluation by which we replace the goal $first$ in the second clause of $p1$ by $NewOut = [Y]$.

6.2.2. Communication through Shared Variables

In logic programming, communication though shared variables provides interseting programming paradigms as was investigated by Shapiro, though the mechanism of communication though shared variables itself is common to many parallel programming language. Here, we show a problem encountered in our programming, *termination-flag*.

Example 6.2.2. 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.

Although the introduction of $JudgeStop$ forces each predicate to have an extra argument and one extra clause for termination, it can make some kind of parallel programs efficient because it releases the system from executing superfluous computations as soon as an answer is found. Therefore it makes such a program effective that finds a solution among a lot of candidates.

In the ω-graphs refutation procedure, it makes the system very effective 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* with no more superfluous expansion of nodes.

For example, the complete ω-graph of the example 4.3.2. (Appendix 2) consists of 9 nodes and 36 edges. If we execute two processes sequentially, such a large graph is treated and the execution explodes with *work-space-full*. However,we can find an ω-loop when 5 nodes are created and only 2 nodes are expanded. Therefore, we can perform the whole process without *work-space-full* by parallel execution.

6.2.3. Use of Partially Specified Data Structures

Partially specified data structures are especially useful for utilizing potential concurrency.

Example 6.2.3. Check of ω-Loop Freeness

As was described in 6.2.2, we can sometimes find an ω-loop before the ω-graph is completely constructed. Therefore, we can check the 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. It is an important concept to write better programs together with that of communication through shared varibales.

6.2.4. Use of Decision Distributible Data Structures

Difference list is a typical data structure which is suited for decision distribution. Its use provides us with possibility to increase efficiency of GHC programs, though it might cost much in some cases.

Example 6.2.4. Union with Difference Lists

[Program (L)]

```
union(d([X|H1],T1),d(H2,T2), d(H,T)) :-
        insert(X,d(H2,T2),d(H,I)), union(d(H1,T1),d(H2,T2),d(I,T)).
union(d(H1,T1),d([X|H2],T2), d(H,T)) :-
        insert(X,d(H1,T1),d(H,I)), union(d(H1,T1),d(H2,T2),d(I,T)).
union(d([ ],[ ]), d(H2,T2),d(H,T)) :- H=H2, T=T2.
union(d(H1,T1),d([ ],[ ]), d(H,T) ):- H=H1, T=T1.

insert(X,d([Y|H],T), d(A,B)) :- X==Y | A=B.
insert(X,d([Y|H],T), d(A,B)) :- X\=Y | insert(X,d(H,T),d(A,B)).
insert(X,d([ ],[ ]),d(A,B)) :- A=[X|B].
```

The above is another GHC program of *union* by using difference list. The process *union* reads out the data X from the head of a stream and the process *insert* decides the output data for X. These two phases are done independently for each data X. Difference list enables the processes to be distributed into each step and decide the output data independently.

6.2.5. Paradigms in Sequential Programming Revisited

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 recursion style. Since sequential programs in general recursive style are almost directly tranlated to corrsponding GHC programs with AND-parallel processes, this paradigm is suited for GHC programming, especially when the synthesis from subsolutions works well for partially obtained subsolutions. We used this paradigm in *compute_initial_node_formula* procedure.

Example 6.2.5.1. Flatten and Mc-Flatten

When the synthesis from subsolutions does not work for partially specified data structures, direct parallelism in GHC might not enhance the concurrency as expected. In such a case, program transformation technique for sequential programs are still useful. Consider the following GHC program (M), which is a direct translation from the well-known *flatten* in Prolog.

[Program (M)]

```
flatten(tip(A),Z) :- Z=[A].
flatten(tree(L,R),Z) :- flatten(L,X),flatten(R,Y),append(X,Y,Z).

append([ ],M,N) :- M=N.
```

```
append([X|L],M,N) :- N=[X|N1],append(L,M,N1).
```

Because *append* does not work until its first argument is instantiated to a non-variable, the subsolutions obtained in the children process are not returned back to the parent process if some children processes are delayed. The program (N) below is a direct translation of *mc-flatten*, which can be obtained by transformation of *flatten* in sequential Prolog.

[Program (N)]

```
flatten(tip(A),Z) :- Z=d([A|T],T).
flatten(tree(L,R),d(H,T)) :- flatten(L,d(H,I),flatten(R,d(I,T)).

append([ ],M,N) :- M=N.
append([X|L],M,N) :- N=[X|N1],append(L,M,N1).
```

This GHC program works well without being blocked by other process's delay.

Dynamic programming is a technique used to convert non-tail-recursive programs with redundant computation into tail-recursive ones with tables to store the results when they are once computed. (Hence, this technique is called *tabulation technique*.) The converted program computes the desired final result in the bottom-up manner by storing intermediate results and consulting them if necessary. Though the converted program must have the common table as an extra argument in general, we sometimes need just part of the table, e.g., computation of Fibonacci sequence.

One might think that conversion to repetitive (tail-recursive) programs is the theme just effective for sequential programs. However, 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 comminucation 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 in 6.1.4. is a useful programming concept.

Example 6.2.5.2. Fibonacci Sequence and Common Table

[Program (O)]

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(s(s(X)),F) :- fibonacci(X,F1),fibonacci(s(X),F2),add(F1,F2,F).
```

[Program (P)]

```
fibonacci(0,0,1).
fibonacci(1,1,1).
fibonacci(s(s(X)),F1,F) :- fibonacci(s(X),F0,F1),add(F0,F1,F).
```

In program (O), more than two processes with identical arguments are generated. For example, in computing *fibonacci*(4, 7), two *fibonacci*(2, 2) processes compute the same result twice. In program (P), the second arguments corresponds to the table in dynamic programming. But here, we need only one entry of the table, which is now a shared variable

for spanning the communication network.

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 cadidates at the same time and if one succeeds, then stop the whole processes." The search of candidates should trace all paths in a search tree from the root to leaves in parallel and fork at every branching node. If there exists some succeeding path, the whole search terminates. However, we can sometimes obtain a more efficient parallel program by totally changing the algorithm itself. For example, some problems on graphs, such as finding connected components and searching a path, can be solved more effectively by assigning processes to nodes of the graph.

Example 6.2.5.3. Check of ω-Loop Freeness

The program of *check_omega_loop_freeness* is in typical backtracking style in Prolog. We describe in some details how we reached the final GHC program after several "try and error". We wrote three programs, following-message oriented one, stream oriented one and structure oriented one. Each algorithm has its advatage, but at last, we have adopted the third one because of better parallelism and simpler behavior of processes (See 5.4). In all these algorithms, each node is considered as a process sending messages each other.m

(a) Following-Message Oriented Algorithm

In the three algorithms following-message oriented algorithm is most close to the algorithm for sequential Prolog. Each ω-node sends a message through each path and all the paths are checked in parallel. The whole graph is represented as a list of triples

(NodeNmbr,AdjacentNodes,NodeType).

We call the node which has an edge outgoing from the node N *adjacent node of* N.

Example 6.2.5.4. The graph shown in example 3.2. is represented by

Graph=[(0,[1,2],not-omega), (1,[1,2],omega), (2,[1,2],not-omega)].

The algorithm and program are shown below .

Each ω-node sends the associated node number as a message and propagates it through all the paths outgoing from that node. In the program, C_ω shows the number of nodes the message sent initially from the ω-node have visited. The process *return_of_the_message* checks whether the message each ω-node sent before is returned to itself or not. The process *proceed_message_farther* passes the message to its adjacent nodes that each ω-node initially sent. Each ω-node sends the associated node number as a message and propagates it through all the paths outgoing from that node. Each ω-node checks whether the message it initially sent is returned or not. Every node passes the message to its adjacent nodes if the node receives it.

(1) Set MAX to the number of nodes in the ω-graph. For each ω-node, initialize *Count* to 1.
For every ω-node, send its node number $NodeNmbr$ as a message to its adjacent nodes.

(2) For each node whose node number is N, if it receives a message, then repeat the following (3).

(3) If $Count > MAX$, then stop with failure.
If *JudgeStop* is instantiated to 'stop', then stop with the answer " There exists an ω-loop."
Otherwise, do the following (4).

(4) If ω-node receives the message it initially sent, then set *JudgeStop* to 'stop'. Otherwise, increment *Count* by 1 and send the message to its adjacent nodes.

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

This algorithm have much to do with sequential programming and it is suitable for our conventional thinking style. However, it is not appropriate for parallel programming. Then we will introduce another algorithm more appropriate for parallel programming, in which each node is taken as a process.

[Program (Q)]

```
check_omega_loop_freeness(MAX,Graph) :-
        check_omega_loop_freeness(JudgeStop,MAX,Graph,Graph),
        write_result(JudgeStop).

check_omega_loop_freeness(JudgeStop,MAX,[(N,Out,omega)|G],Graph) :-
        find_omega_loop(JudgeStop,N,Graph,Out,MAX,1),
        check_omega_loop_freeness(JudgeStop,MAX,G,Graph).
check_omega_loop_freeness(JudgeStop,MAX,[(N,Out,not-omega)|G],Graph) :-
        check_omega_loop_freeness(JudgeStop,MAX,G,Graph).
check_omega_loop_freeness(_,_,[ ],_).

find_omega_loop(JudgeStop,W,Graph,[N|NodeList],MAX,Count) :-
        Count<MAX, prolog(var(JudgeStop)) |
        return_of_the_message(JudgeStop,W,Graph,N,MAX,Count),
        find_omega_loop(JudgeStop,W,Graph,NodeList,MAX,Count).
find_omega_loop(_,_,_,_,MAX,Count) :- MAX=<Count | true.
find_omega_loop(stop,_,_,_,_,_) :- true.
find_omega_loop(_,_,_,[ ],_,_).

return_of_the_message(JudgeStop,W,Graph,N,MAX,Count) :- N==W |
        JudgeStop='stop'.
return_of_the_message(JudgeStop,W,Graph,N,MAX,Count) :- N\=W |
        Count1 := Count + 1,
        proceed_message_farther(JudgeStop,N,Graph,OutNodes),
        find_omega_loop(JudgeStop,W,Graph,OutNodes,MAX,Count1).

proceed_message_farther(JudgeStop,N,[(M,Out,Type)|Graph],OutNodes) :- N==M |
        OutNodes=Out.
proceed_message_farther(JudgeStop,N,[(M,Out,Type)|Graph],OutNodes) :- N\=M |
        proceed_message_farther(JudgeStop,N,Graph,OutNodes).
proceed_message_farther(JudgeStop,N,[ ],OutNodes) :- OutNodes=[ ].

write_result(stop) :- write('The graph is not omega-loop free').
write_result(JudgeStop) :- prolog(var(JudgeStop)) |
        write('The graph is omega-loop free').
```

(b) Stream Oriented Algorithm

We use a programming technique similar to one in [Shapiro 83]. Each node is considered as a process which has some input channels and output channels.

A set of node numbers is sent as a message through output channel. Each node sends a message and receives a set of messages at each time. A node does not send the same message with the one it sent before. The whole graph is represented by a list of the quadruples

(NodeNmbr,OutStream,InStreams,NodeType),

which is as same as the in case of structure oriented algorithm (See 5.4).

The algorithm and program are shown below.

In the program, variable L_n is an incremental set of node numbers that have been received by the node whose associated number is N, $NewMes$ is a current set of node numbers received by that node and not included by L_n. C_n behaves as a counter. The process *pickup* checks the current message.

(1) For each node, send the associated node number to its adjacent nodes.
(2) For each node whose node number is N, initialize C_n to the set of all nodes in the ω-graph and let L_n be [ ], repeat the following (3).
(3) If $C_n = \{\}$, then stop with failure.
If $JudgeStop$ is instantiated to 'stop', then stop with the answer " There exists an ω-loop."
Otherwise, if the node receives the message, then let $NewMes$ be a set of received node numbers which are not included by L_n, and for $NewMes$, do the following (4).
(4) If the node is an ω-node and N is a member of $NewMes$, then set $JudgeStop$ to 'stop'.
Otherwise, extract a node from C_n send $NewMes$ to its adjacent nodes and add $NewMes$ to L_n.

If all the processes for the nodes stop with failure, then answer " The graph is ω-loop free." Note that if $InStreams$ of a node is $\{\}$, then $NewMes$ of that node is always [ ].

[Program (R)]

```
check_omega_loop_freeness(Graph) :-
        check_omega_loop_freeness(JudgeStop,Graph,Graph) &
        write_result(JudgeStop).

check_omega_loop_freeness(_,_,[ ]).
check_omega_loop_freeness(JudgeStop,Cn,[(N,Xn,INs,omega)|Gs]) :-
        Xn=[[N]|Xn1], Ln=[ ],
        node(JudgeStop,N,Type,Cn,Xn1,INs,Ln),
        check_omega_loop_freeness(JudgeStop,Cn,Gs).

node(_,_,_,[ ],_,_,_).
node(stop,_,_,_,_,_,_).
node(JudgeStop,NodeNmbr,Type,[_|Cn],Xn,INs,Ln) :-
        prolog(var(JudgeStop)) |
        receive(JudgeStop,NodeNmbr,Type,Xn,Xn1,INs,INs1,Ln,Ln1) &
        node(JudgeStop,NodeNmbr,Type,Cn,Xn1,INs1,Ln1).

receive(JudgeStop,NodeNmbr,Type,Xn,Xn1,INs,INs1,Ln,Ln1) :-
        prolog(var(JudgeStop)) |
        pickup(INs,INs1,Ln,NewMes) & append(NewMes,Ln,Ln1),
        check_omega_member(JudgeStop,NodeNmbr,Type,NewMes),
        Xn=[NewMes|Xn1].
```

```
receive(stop,_,_,_,_,_,_,_,_).

check_omega_member(JudgeStop,NodeNmbr,omega,NewMes) :-
        member(NodeNmbr,NewMes,yes) |
        JudgeStop='stop'.
check_omega_member(_,_,omega,_) :-
        member(NodeNmbr,NewMes,no) | true.
check_omega_member(_,_,not-omega,_).

pickup([[A|Avar]|INs],NewINs,Ln,NewMes) :-
        pickup1(A,Ln,New),
        pickup(INs,Avars,Ln,NewMes1) & NewINs=[Avar|Avars],
        union(New,NewMes1,NewMes).
pickup([ ],NewINs,_,NewMes) :- NewINs=[ ], NewMes=[ ].

pickup1([A1|An],Ln,New) :- member(A1,Ln,yes) | pickup1(An,Ln,New).
pickup1([A1|An],Ln,New) :- member(A1,Ln,no) | New=[A1|New1], pickup1(An,Ln,New1).
pickup1([ ],_,New) :- New=[ ].

write_result(stop) :- write('The graph is not omega-loop free.').
write_result(JudgeStop) :- prolog(var(JudgeStop)) |
        write('The graph is omega-loop free.').
```

(c) Structure Oriented Algorithm

This algorithm is almost same as the stream oriented one. However, in this case, each node process propagates the structure of the graph by connecting the input stream to the tail of the output stream and check the list whether or not it contains loops. It is an infinite recursive list if the graph contains a loop. Intuitively, paths flowing into an ω-node are checked backwardly. The algorithm is expressed in section 5.4. This algorithm treats the static data of a graph structure sent as a message, while stream oriented algorithm treats the dynamic one of received message at every stage. It makes the behavior of processes in this algorithm easier to understand.

6.3. 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.

6.3.1. Intentional Sequentialization

If we borrow the viewpoint by Kowalski, GHC programs also consist of *logic* part and *control* part. The control part in Prolog programs is not so burdensome that we can construct Prolog programs taking the both parts into consideration simultaneously. However, 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.

(1) Computation trace of GHC programs are, in general, so complicated that following the traces is very tedious and hard (especially under the breadth-first search scheduling). We can confirm the logic part of GHC programs more easily by following simpler computation trace of the corresponding sequentialized one.

(2) Even very efficient parallel algorithms in GHC do not show high performance if they are executed on the sequential machines (especially under the breadth-first search scheduling), because the scheduler always check the goals even if the suspension of the goal is released in long future. We can confirm the logic part of GHC programs in shorter time by executing the corresponding sequentialized one.

Example 6.3.1. Compare the following two clauses where goals p and q have a shared variable X.

```
[ Program (S) ]
        h :- p, q.

[ Program (T) ]
        h :- p | q.
```

In program (S), since the processes p and q run in parallel, the unification of the goals p and q are scheduled in turn under the breadth-first search scheduling. Assume that the shared variable X is instantiated in the 10th step of p and the unification of the goal q is enabled by the propagation of the instantiation of the common variable X. Then q is checked unnecessarily 10 times before the head unification of q succeeds. In program (T),however, this check is not done at all, which makes the execution fast.

6.3.2. Incremental Parallelization

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, behaves as we expected, the whole program might behaves quite differently from our expectation. The more increases the number of processes which run in parallel, the more serious this problem becomes. However, it is important in programming to trace computation procedure and understand the behavior of programs.

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.

Example 6.3.2. Refute

We use this programming style in the upper level parallelism for making the ω-graphs refutation. On the top level of ω-graphs refutation, at first we devide the process *refute* into three modules *compute_initial_node_formula*, *construct_omega_graph* and *check_omega_loop_freeness*. Next, we perform a parallel programming within each module. We have to be careful for the treatment of shared variable such as *Graph* and also for interface between modules in order to perform a parallel computation of the three modules later. Each module may be divided into some submodules, if necessary. For example, *construct_omega_graph* is divided into three submodules of *node_process*, *multiplexer*, and *graph_manager*. Then we try to realize an uper level parallelism.

6.3.3. Interpreter + Editor + Incremental Compiler ?

Since the current GHC system has neither interpreter nor incremental compiler, it takes quite much time and energy for programming and debugging. Though the interpreter of GHC might be slow, it is convinient for interactive programming and debugging with screen editor like *ledit* in Lisp. As for the incremental compiler, we are not sure now whether it is possible for parallel programming language. The KL1 (Kernel Language One), which includes GHC as its core, is still under development. Further investigation of better programming environment is needed.

6.3.4. Visual Debugging Aids

Current GHC debugger is rather weak so that we can neither see the whole execution tree, nor activate and trace process by process nor see output of each process separatedly on the screen of terminals. 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.

If the window system used widely now is adapted for debugging of parallel programs, our parallel programming would be much more comfortable. For example, one window always shows the current execution tree and active process and accept the user's direction about which process is activated next. Other windows are assigned to each process dynamically in order to show the trace of the process and print out the output of the process.

6.3.5. Performance Measurement of Parallel Execution

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 and 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 rough base for evaluation of parallelism. The less is the number of cycles compared with a program size, the higher is the parallelism thought to be. What measurement should we choose to judge whether a given GHC program is better or not ?

At first, we discuss on processing time and space.

Example 6.3.5. Consider *union* program again.

[Program (C)]

```
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) :- S=S2.
union(S1,[ ],S) :- 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) :- Answer=no.
```

[Program (U)]

```
union([X|S1],S2,S) :- S=[X|NewS],delete(X,S2,NewS2), union(S1,NewS2,NewS).
union(S1,[X|S2],S) :- S=[X|NewS],delete(X,S1,NewS1), union(NewS1,S2,NewS).
union([ ],S2,S) :- S=S2.
union(S1,[ ],S) :- S=S1.

delete(X,[Y|S],T) :- X==Y | T=S.
delete(X,[Y|S],T) :- X\= Y | T=[Y|NewT],delete(X,S,NewT).
delete(X,[ ],T) :- T=[ ].
```

Program (C) needs more processing time since *member* in the passive part may check all the elements of the completely instantiated list before trust. On the other hand, in program (U), another definition of *union*, *delete* in the passive part can succeeds when the head of list is instantiated, and the trusted clause is executed in parallel. Therefore, it needs less processing time. However, it consumes more space since *delete* must reconstruct some of lists.

In general, we give higher priority to time efficiency than less space-consuming because development of parallel execution machine for GHC such as Parallel Inference Machine(PIM) in near future will solve the space problem.

Next, we discuss about the time and parallelism.

[Program (V)]

```
h :- p, q.
```

[Program (V)]

```
h :- p & q.
```

In program (V), two processes of p and q run in parallel, while they run sequentially in program (W). Therefore, (V) provides higher concurrency and needs less processing time theoretically. But with the current compiler, most programs which provide higher concurrency need more time because of the scheduler's overhead. This problem will also be resolved if the compiler is revised. Currently, we give parallelism a higher priority than time efficiency as far as the program does not cost an extravagant time.

Example 6.3.5.2. Distribute And over Or

The following is a program of distribute_and_over_or.

[Program (X)]

```
distribute_and_over_or(or(P,Q),A) :-
        distribute_and_over_or(P,P1), distribute_and_over_or(Q,Q1),
        A=or(P1,Q1).
distribute_and_over_or(and(P,Q),A) :-
        distribute_and_over_or(P,P1), distribute_and_over_or(Q,Q1) |
        distribute_andor(and(P1,Q1),A).
```

```
distribute_and_over_or(not(P),A) :- A=not(P).
distribute_and_over_or(always(P),A) :- A=always(P).
distribute_and_over_or(eventually(P),A) :- A=eventually(P).
distribute_and_over_or(m_eventually(P),A) :- A=m_eventually(P).
distribute_and_over_or(next(P),A) :- A=next(P).
distribute_and_over_or(P,A) :- atomic(P) | A=P.

distribute_andor(and(R,or(P,Q)),A) :-
        distribute_and_over_or(and(R,P),A1), distribute_and_over_or(and(R,Q),A2),
        A=or(A1,A2).
distribute_andor(and(or(P,Q),R),A) :-
        distribute_and_over_or(and(P,R),A1), distribute_and_over_or(and(Q,R),A2),
        A=or(A1,A2).
distribute_andor(and(P,Q),A) :-
        P\=or(P1,P2), Q\=or(Q1,Q2) |
        A=and(P,Q).
```

Pay attention to the trust operator '|' in the second clause of *distribute_and_over_or*. Theoretically, a faster parallel execution is possible without this '|'. In fact, this is not always true depending on data. We show below an example.

Assume that computation is executed under the breadth-first search scheduling. Suppose this '|' is eliminated. If the goal

(0) distribute_and_over_or(and(and(and(f1,f2),f3),f4),Answer)

is called, then the head of the second clause is unified and it forks the following three subprocesses.

(1) distribute_and_over_or(and(and(f1,f2),f3), A1)
(2) distribute_and_over_or(f4,A2)
(3) distribute_andor(A1,A2,Answer)

On the first cycle, although (2) returns f4 as the value of A2 immediately, A1 still remains as a variable. Therefore, the clause called by (3) is suspended. On the second cycle, (1) invokes the following three subprocesses again.

(1-1) distribute_and_over_or(and(f1,f2), A11)
(1-2) distribute_and_over_or(f3,A12)
(1-3) distribute_andor(A11,A12,A1)

In this case, since A11 remains as a variable, A1 is not instantiated yet. Therefore, the clause called by (3) is still suspended. It is suspended until the unification invoked by the calls

(1-1-1) distribute_and_over_or(f1, A111)
(1-1-2) distribute_and_over_or(f2, A112)

succeed, which means that both calls (1) and (2) succeed. Therefore, even if the trust operator is eliminated, it does not give a better program. What is worse, it makes the program worse inefficient by burdening the scheduler with constant check of the superflous process (3). Thus, it is prudent to use '|'(trust operator) and '&'(sequential AND) for the essentially sequential problem or the clause including an independent process which costs much time.

Our program of ω-graphs refutation procedure in GHC consists of about 500 lines. The result of execution of some examples is shown in Appendix 3. Some of the performance measurements are shown below. (The CPU time required for the execution of the Prolog version is 144,395,3087,1147 and 2763 ms for ex1,ex2,ex3,ex4 and ex5, respectively.)

Table 6.3.5. Evaluation of the Execution of Examples

	formula	CPU time(ms)	global stack	cycle
ex1	$\neg\Diamond p$	8780	13360	39
ex2	$\Box\neg p \vee \Box p$	63758	19985	102
ex3	$\Box\Diamond p \supset \Diamond\Box\neg q$	15541	22684	49
ex4	$\Diamond\Box\neg p \vee \Box\Diamond p$	125367	16564	147
ex5	$\Box\Diamond p \supset \Diamond\Box p$	108688	39947	125

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 chance of this research and Dr.K.Furukawa, Chief of the 1st Laboratory of ICOT, for his advice and encouragement.

REFERENCES

[Brock and Ackerman 81] Brock,J.D. and W.B.Ackerman, "Acenarios : A Model of Non-Deterministic Computation," pp.252-259,Lecture Note in Computer Science,Vol.107,Springer-Verlag,1981.

[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," a paper of Technical Group on Automata and Languages, 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.

[Manna 81] Manna,Z., " Verification of Sequential Programs: Temporal Axiomatization," Stanford TR 81-877,1981.

[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.

[Takeuchi and Furukawa 83] Takauchi,A. and K.Furukawa, "Interprocess Communication in Concurrent Prolog," Proc. of Logic Programming Workshop 83,Universidade nova de Lisboa,1983.

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

[Wand 80] Wand,M., "Continuation-Based Program Transformation Strategies," JACM,Vol.27,No.1,pp.164-180,1980.

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

Appendix 1.

The construction of ω-graph of $\Box\Diamond P \wedge \Box\Diamond Q$. (See example 4.3.1)

$Unexpanded = \{N_0\}$

$N_0 : [\Box\Diamond P \wedge \Box\Diamond Q]_{\{\}}$

$$\begin{array}{lll} \Longrightarrow & (\ P \wedge Q \wedge & \circ[\Box\Diamond P \wedge \Box\Diamond Q]_{\{P,Q\}} \ ) \quad : N_1 \\ \vee & (\ P \wedge & \circ[\Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{P\}} \ ) \quad : N_2 \\ \vee & (\ Q \wedge & \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Box\Diamond Q]_{\{Q\}} \ ) \quad : N_3 \\ \vee & (& \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{\}} \ ) \quad : N_4 \end{array}$$

$Unexpanded = \{N_1, N_2, N_3, N_4\}$

$N_1 : [\Box\Diamond P \wedge \Box\Diamond Q]_{\{P,Q\}}$

$$\begin{array}{lll} \Longrightarrow & (\ P \wedge Q \wedge & \circ[\Box\Diamond P \wedge \Box\Diamond Q]_{\{P,Q\}} \ ) \quad : N_1 \\ \vee & (\ P \wedge & \circ[\Box\Diamond P \wedge \Diamond^* \wedge \Box\Diamond Q]_{\{P\}} \ ) \quad : N_2 \\ \vee & (\ Q \wedge & \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Box\Diamond Q]_{\{Q\}} \ ) \quad : N_3 \\ \vee & (& \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{\}} \ ) \quad : N_4 \end{array}$$

$Unexpanded = \{N_2, N_3, N_4\}$

$N_2 : [\Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{P\}}$

$$\begin{array}{lll} \Longrightarrow & (\ P \wedge Q \wedge & \circ[\Box\Diamond P \wedge \Box\Diamond Q]_{\{P,Q\}} \ ) \quad : N_1 \\ \vee & (\ P \wedge & \circ[\Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{P\}} \ ) \quad : N_2 \\ \vee & (\ Q \wedge & \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Box\Diamond Q]_{\{P,Q\}} \ ) \quad : N_5 \\ \vee & (& \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{\}} \ ) \quad : N_6 \end{array}$$

$Unexpanded = \{N_3, N_4, N_5, N_6\}$

$N_3 : [\Diamond^* P \wedge \Box\Diamond P \wedge \Box\Diamond Q]_{\{Q\}}$

$$\begin{array}{lll} \Longrightarrow & (\ P \wedge Q \wedge & \circ[\Box\Diamond P \wedge \Box\Diamond Q]_{\{P,Q\}} \ ) \quad : N_1 \\ \vee & (\ P \wedge & \circ[\Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{P,Q\}} \ ) \quad : N_7 \\ \vee & (\ Q \wedge & \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Box\Diamond Q]_{\{Q\}} \ ) \quad : N_3 \\ \vee & (& \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{Q\}} \ ) \quad : N_8 \end{array}$$

$Unexpanded = \{N_4, N_5, N_6, N_7, N_8\}$

$N_4 : [\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{\}}$

$$\begin{array}{lll} \Longrightarrow & (\ P \wedge Q \wedge & \circ[\Box\Diamond P \wedge \Box\Diamond Q]_{\{P,Q\}} \ ) \quad : N_1 \\ \vee & (\ P \wedge & \circ[\Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{P\}} \ ) \quad : N_2 \\ \vee & (\ Q \wedge & \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Box\Diamond Q]_{\{Q\}} \ ) \quad : N_3 \\ \vee & (& \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{\}} \ ) \quad : N_4 \end{array}$$

$Unexpanded = \{N_5, N_6, N_7, N_8\}$

$N_5 : [\Diamond^* P \wedge \Box\Diamond P \wedge \Box\Diamond Q]_{\{P,Q\}}$

$$\begin{array}{lll} \Longrightarrow & (\ P \wedge Q \wedge & \circ[\Box\Diamond P \wedge \Box\Diamond Q]_{\{P,Q\}} \ ) \quad : N_1 \\ \vee & (\ P \wedge & \circ[\Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{P\}} \ ) \quad : N_2 \\ \vee & (\ Q \wedge & \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Box\Diamond Q]_{\{Q\}} \ ) \quad : N_3 \\ \vee & (& \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^* Q \wedge \Box\Diamond Q]_{\{\}} \ ) \quad : N_4 \end{array}$$

$Unexpanded = \{N_6, N_7, N_8\}$

$$
\begin{array}{lll}
N_6 : [\diamond^* P \wedge \Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q]_{\{P\}} & & \\
\Longrightarrow & (\; P \wedge Q \wedge \; \bigcirc[\Box\diamond P \wedge \Box\diamond Q]_{\{P,Q\}} \;) & : N_1 \\
\vee & (\; P \wedge \; \bigcirc[\Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q]_{\{P\}} \;) & : N_2 \\
\vee & (\; Q \wedge \; \bigcirc[\diamond^* P \wedge \Box\diamond P \wedge \Box\diamond Q]_{\{P,Q\}} \;) & : N_5 \\
\vee & (\; \bigcirc[\diamond^* P \wedge \Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q]_{\{P\}} \;) & : N_6
\end{array}
$$

$Unexpanded = \{N_7, N_8\}$

$$
\begin{array}{lll}
N_7 : [\Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q]_{\{P,Q\}} & & \\
\Longrightarrow & (\; P \wedge Q \wedge \; \bigcirc[\Box\diamond P \wedge \Box\diamond Q]_{\{P,Q\}} \;) & : N_1 \\
\vee & (\; P \wedge \; \bigcirc[\Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q]_{\{P\}} \;) & : N_2 \\
\vee & (\; Q \wedge \; \bigcirc[\diamond^* P \wedge \Box\diamond P \wedge \Box\diamond Q]_{\{Q\}} \;) & : N_3 \\
\vee & (\; \bigcirc[\diamond^* P \wedge \Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q]_{\{\}} \;) & : N_4
\end{array}
$$

$Unexpanded = \{N_8\}$

$$
\begin{array}{lll}
N_8 : [\diamond^* P \wedge \Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q]_{\{Q\}} & & \\
\Longrightarrow & (\; P \wedge Q \wedge \; \bigcirc[\Box\diamond P \wedge \Box\diamond Q]_{\{P,Q\}} \;) & : N_1 \\
\vee & (\; P \wedge \; \bigcirc[\Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q]_{\{P,Q\}} \;) & : N_7 \\
\vee & (\; Q \wedge \; \bigcirc[\diamond^* P \wedge \Box\diamond P \wedge \Box\diamond Q]_{\{Q\}} \;) & : N_3 \\
\vee & (\; \bigcirc[\diamond^* P \wedge \Box\diamond P \wedge \diamond^* Q \wedge \Box\diamond Q]_{\{Q\}} \;) & : N_8
\end{array}
$$

$Unexpanded = \{\}$

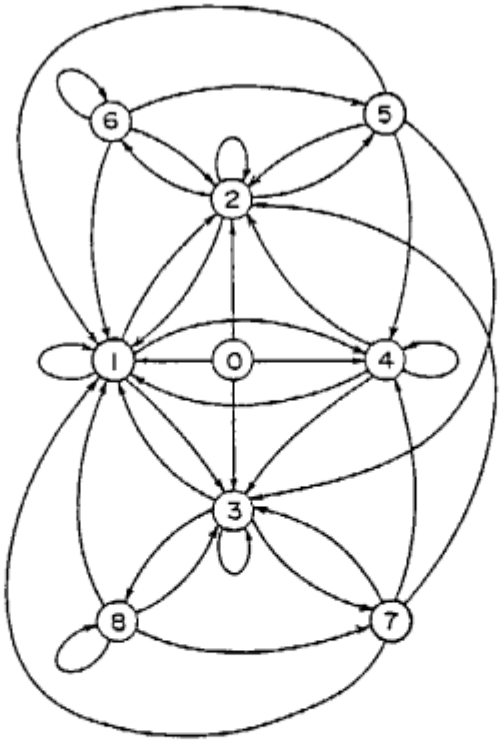


Figure A1. ω-Graph of □◇P ∧ □◇Q

Appendix 2.
The construction of ω-graph of $\Box\Diamond P \wedge \Diamond\Box\neg P$. (See example 4.3.2 and 4.4.2)

$Unexpanded = \{N_0\}$

$N_0 : [\Box\Diamond P \wedge \Diamond\Box\neg P]_{\{\}}$

$$\begin{array}{lll} \Longrightarrow & (\ P \wedge & \circ[\Box\Diamond P \wedge \Diamond^*\Box\neg P]_{\{P\}}\ ) \quad : N_1 \\ \vee & (\ \neg P \wedge & \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Box\neg P]_{\{\Box\neg P\}}\ ) \quad : N_2 \\ \vee & (\ & \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^*\Box\neg P]_{\{\}}\ ) \quad : N_3 \end{array}$$

$Unexpanded = \{N_1, N_2, N_3\}$

$N_1 : [\Box\Diamond P \wedge \Diamond^*\Box\neg P]_{\{P\}}$

$$\begin{array}{lll} \Longrightarrow & (\ P \wedge & \circ[\Box\Diamond P \wedge \Diamond^*\Box\neg P]_{\{P\}}\ ) \quad : N_1 \\ \vee & (\ \neg P \wedge & \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Box\neg P]_{\{P,\Box\neg P\}}\ ) \quad : N_4 \\ \vee & (\ & \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^*\Box\neg P]_{\{P\}}\ ) \quad : N_5 \end{array}$$

$Unexpanded = \{N_2, N_3, N_4, N_5\}$

$N_2 : [\Diamond^* P \wedge \Box\Diamond P \wedge \Box\neg P]_{\{\Box\neg P\}}$

$$\Longrightarrow \quad (\ \neg P \wedge \quad \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Box\neg P]_{\{\Box\neg P\}}\ ) \quad : N_2$$

$Unexpanded = \{N_3, N_4, N_5\}$

$N_3 : [\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^*\neg P \wedge \Box\neg P]_{\{\}}$

$$\begin{array}{lll} \Longrightarrow & (\ P \wedge & \circ[\Box\Diamond P \wedge \Diamond^*\Box\neg P]_{\{P\}}\ ) \quad : N_1 \\ \vee & (\ \neg P \wedge & \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Box\neg P]_{\{\Box\neg P\}}\ ) \quad : N_2 \\ \vee & (\ & \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^*\Box\neg P]_{\{\}}\ ) \quad : N_3 \end{array}$$

$Unexpanded = \{N_4, N_5\}$

$N_4 : [\Diamond^* P \wedge \Box\Diamond P \wedge \Box\neg P]_{\{P,\Box\neg P\}}$

$$\Longrightarrow \quad (\ \neg P \wedge \quad \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Box\neg P]_{\{\Box\neg P\}}\ ) \quad : N_2$$

$Unexpanded = \{N_5\}$

$N_5 : [\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^*\Box\neg P]_{\{P\}}$

$$\begin{array}{lll} \Longrightarrow & (\ P \wedge & \circ[\Box\Diamond P \wedge \Diamond^*\Box\neg P]_{\{P\}}\ ) \quad : N_1 \\ \vee & (\ \neg P \wedge & \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Box\neg P]_{\{P,\Box\neg P\}}\ ) \quad : N_4 \\ \vee & (\ & \circ[\Diamond^* P \wedge \Box\Diamond P \wedge \Diamond^*\Box\neg P]_{\{P\}}\ ) \quad : N_5 \end{array}$$

$Unexpanded = \{\}$

Appendix 3.

Execution Examples of Parallel ω-Graphs Refutation Procedure.

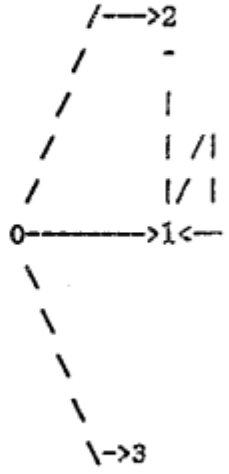
Example 3.

```
| ?- ghc ex3.
Given formula is []<>p => <>[]~q
== Negation normal form is ==
[]<>p & []<>q
Eventuality set is [ q, p ]
== A new node is created. ==
( []<>p & []<>q, [ p, q ] )
== A new node is created. ==
( []<>p & <>*q & []<>q, [ p ] )
+ Path ( 0-1 ) was found.
== A new node is created. ==
( <>*p & []<>p & []<>q, [ q ] )
== The node already exists. ==
( []<>p & []<>q, [ p, q ] )
+ Path ( 1-1 ) was found.
!!! A loop was found. !!!

== The node already exists. ==
( []<>p & []<>q, [ p, q ] )
+ Path ( 1-1 ) was found.
== The node already exists. ==
( []<>p & <>*q & []<>q, [ p ] )
!!! A loop was found. !!!
The graph is not omega-loop free.
```

```
Given formula is []<>p => <>[]~q
Eventuality set is [ q, p ]
Given formula is not valid.
```

```
0 ( []<>p & []<>q, [] )
1 ( []<>p & []<>q, [ p, q ] )
2 ( []<>p & <>*q & []<>q, [ p ] )
3 ( <>*p & []<>p & []<>q, [ q ] )
```



Example 4.

```
| ?- ghc ex4.
Given formula is <>[]~p V []<>p
== Negation normal form is ==
[]<>p & <>[]~p
Eventuality set is [ []~p, p ]
== A new node is created. ==
( []<>p & <>*[]~p, [ p ] )
== A new node is created. ==
( []<>p & <>*[]~p & <>*p, [] )
== A new node is created. ==
( []~p & []<>p & <>*p, [ []~p ] )
== The node already exists. ==
( []<>p & <>*[]~p, [ p ] )
== A new node is created. ==
( []<>p & <>*[]~p & <>*p, [ p ] )
== The node already exists. ==
( []<>p & <>*[]~p, [ p ] )
== A new node is created. ==
( []~p & []<>p & <>*p, [ p, []~p ] )
+ Path ( 1-5 ) was found.
== The node already exists. ==
( []<>p & <>*[]~p & <>*p, [] )
+ Path ( 0-1-5 ) was found.
== The node already exists. ==
( []~p & []<>p & <>*p, [ []~p ] )
+ Path ( 2-1-5 ) was found.
+ Path ( 0-2-1-5 ) was found.
== The node already exists. ==
( []~p & []<>p & <>*p, [ []~p ] )
== The node already exists. ==
( []<>p & <>*[]~p, [ p ] )
+ Path ( 4-1-5 ) was found.
== The node already exists. ==
( []<>p & <>*[]~p & <>*p, [ p ] )
== The node already exists. ==
( []~p & []<>p & <>*p, [ p, []~p ] )
+ Path ( 4-5 ) was found.
```

```
== The node already exists. ==
( □¯p & □<>p & <>*p, [ □¯p ] )
+ Path ( 1-4-5 ) was found.
+ Path ( 0-1-4-5 ) was found.
+ Path ( 2-1-4-5 ) was found.
+ Path ( 0-2-1-4-5 ) was found.
== Terminate graph construction. ==
The graph is omega-loop free.
```

```
0 ( □<>p & <>□¯p, □ )
1 ( □<>p & <>*□¯p, [ p ] )
2 ( □<>p & <>*□¯p & <>*p, □ )
3 ( □¯p & □<>p & <>*p, [ □¯p ] )
4 ( □<>p & <>*□¯p & <>*p, [ p ] )
5 ( □¯p & □<>p & <>*p, [ p, □¯p ] )
```

```
Given formula is <>□¯p V □<>p
Eventuality set is [ □¯p, p ]
Given formula is valid.
```