

A Prover for Parallel Processes

Masahiro Hirata
Department of Mathematics
University of Tsukuba
Sakura-mura Niihari-gun
Ibaraki-ken 300-31, JAPAN

Toshio Nishimura
Department of Mathematics
University of Tsukuba
Sakura-mura Niihari-gun
Ibaraki-ken 300-31, JAPAN

We give an automatic prover for verifying logical properties of parallel processes. The prover bases on a subsystem of the system given in [8]. We mechanize it so that a proof-tree is automatically constructed. The prover reduces a parallel program into possible serial ones by applying rules of inference 'interruption' etc.. This paper includes some examples processed by the prover.

1. INTRODUCTION

The need for proofs of properties of parallel program is very greater. A number of research results for parallel program have been given. [1],[2],[3],[4],[5],[8],[9],[10] In parallel processes, several processes which communicate each other run on several processors. So parallel processes cause complicated states. However, as shown in [8], we can consider them the compositions of indivisible operations. So, by decomposition of parallel programs, we can analyze their movements and effects on states. Syntactic analyses of this type are powerful for those of properties of parallel programs. Our prover processes these analyses.

Since the formal system given in [8] includes infinitely long expressions such as F^* (the abbreviation of $F^0VF^1VF^2V\ldots$), we mechanize its subsystem which contains decomposition of F^* with respect to 'interruption (2.3.4)' but does not contain the rules of inference which decompose $G(F^*)$ into $G(F^0), G(F), \dots, G(F^n), \dots$ (cf. [8] 3.1 (i) *-left and *-right) We implement it by the KLISP[6] system with about 12K free cells on PDP11/55 computer in a similar way to [7].

We roughly explain our basic idea in the simplest cases. In our system, we introduce the operator $F;G$ (G is executed after F), but we do not introduce the 'clock'. Let f, g and h be indivisible operations. f and g , sometimes, run in parallel on different processors (denoted by $f//g$), but $f//g$ causes the same result as $f;g$ or $g;f$ (denoted by $(f;g) \vee (g;f)$). For example, $(f;g)//h$ or $(f;g)*//h$ causes the same result as $(h;f;g) \vee (h;g;f)$ or $(f;g)*;h; (f;g)* \vee$

$(f;g)*;f;h;g; (f;g)*$, respectively. The general form of a rule of inference will be given in 2.3.4 below. In such a way, we could reduce parallel processes into possible serial ones.

In our system, we prove the equivalence of parallel programs in the form of sequent. In order to show the equivalence between two programs F and G , our prover proves two sequents $F \rightarrow G$ and $G \rightarrow F$. For example, let a process be able to enter the critical section C if it satisfies the condition Q . It can not enter C eternally, if it satisfies 'not Q ' (denoted by $\neg Q$) eternally (denoted by $(\neg Q)^*$). So, in order to show that a parallel program P is deadlock free, it is sufficient to show that P is equivalent to a program E , where E does not contain infinite loop such as X^* . And, a parallel program P which contains two critical sections $CS1$ and $CS2$ are mutually exclusive, if P is equivalent to a program E , where E does not contain any parallel run of $CS1$ and $CS2$ such as $CS1//CS2$.

2. IMPLEMENTED FORMAL SYSTEM

2.1. An expression and an indivisible expression.

The symbols used in the propositional calculus have the usual meanings for logical formulas (e.g. $\vee, \wedge, \supset, \equiv$ or $\neg$ represents 'or', 'and', 'implies', 'equivalent' or 'not' respectively). And we introduce symbols of expression-constants. Especially, the expression-constant "I" plays the role of the identity function and means the truth value "true" for a formula. And " $\perp$ " represents "impossible", and the truth value "false" for a formula. A formula in the propositional calculus, extended by adding the clause

"if A and B are formulas, so are A;B and A//B", is an expression. An expression-constant is an expression. A figure of the form $x:=E$ is an expression, called an assignment statement to x, where x is a variable and E is a term in the predicate calculus. If F and G are expressions, so are $F//G$, FVG and $F;G$. If F is an expression, so are F^m , F^∞ and F^* , where m is a variable for integers. If F is an expression, then $[F]$ is an indivisible expression. An indivisible expression is an expression. Only those obtained in the above are expressions.

We briefly explain the intuitive meaning of the above definition. An expression means an instruction, a decision, a series of them or a formula in the propositional calculus. And a decision is denoted by a formula. $F;G$ means that G is executed after the execution of F. $F//G$ represents that F and G are executed in parallel. If F and G are formulas, $F;G$, $F//G$ and FVG have the same meaning as "F and G" in the propositional calculus. An expression FVG represents that at least one of F or G is executed. If F and G are formulas, FVG has the same meaning as "F or G". An expression F^* means that F will be executed n-times for an arbitrary n. F^∞ represents the infinite loop of F.

Generally, an expression F is of the form $F_1;F_2;\dots;F_n$, where every F_i is an indivisible expression, an expression-constant or an expression of the form $\neg H$, HVK , $H//K$, H^m , H^* or H^∞ . This is called the canonical form of F.

In the prover, $F;G$, FVG , $F//G$, $F;G$, $F\equiv G$, $\neg F$, F^n , F^* , F^∞ or $[F]$ is denoted by (CONC F G), (*AND F G), (*OR F G), (PAR F G), (*IMPLIES F G), (*EQUIV F G), (*NOT F), (LABEL F n), (* F), (INF F) or (IND F), respectively.

2.2 A sequent, an axiom and a proof

A figure of the form

$$A_1, \dots, A_m, E \longrightarrow E_1, \dots, E_n$$

is called a sequent. This means that, under the assumptions $A_1, \dots, A_m$, the results of the execution of E are included in the results of those of $E_1, \dots, E_{n-1}$ and E_n . Sometimes we denote a finite series of expressions $E_1, \dots, E_n$ by a Greek capital letter Γ , Δ etc.. Axioms are sequents of the form $\Xi \longrightarrow \Delta$, where Ξ is a special expression-constant, or $\Gamma \longrightarrow \Delta$, where $\Gamma \cap \Delta \neq \emptyset$. A proof-tree is the figure of a tree-form which satisfies the following conditions.

- (1) The uppermost sequent is an arbitrary sequent.
- (2) Every lowermost sequent is an axiom.
- (3) The upper sequent and the lower sequents are connected by a rule of inference given in the paragraph 2.3 below.

Let S be a sequent. Then a proof-tree, which has S as the uppermost sequent, is called the proof-tree to S. If we have a proof-tree to S, we say that S is provable.

Let $A_1, \dots, A_n$ and $\neg B_1, \dots, \neg B_k$ be assumptions.

For the prover, a sequent

$$A_1, \dots, A_n, \neg B_1, \dots, \neg B_k, E \longrightarrow E_1, \dots, E_n$$

is divided into two parts. One is the set of assumptions given in the form $(A_1 \text{ I.1}) \dots$

$(A_n \text{ I.h}) (B_1 \text{ F.h+1}) \dots (B_k \text{ F.h+k})$. Other is

$E \longrightarrow E_1, \dots, E_n$ and this is given in the form

(*ARROW (E) ($E_1 \dots E_n$)).

2.3 Rules of Inference

Rules of inference of the subsystem are divided into the following parts.

2.3.1 Rules of inference for the propositional calculus.

We omit those here. (cf. [7])

2.3.2 Rules of inference with respect to ∇

Let $F(e)$ be an arbitrary expression which contains the indicated expression-variable e in at most one place in $F(e)$ except for the inside of any subexpression of the form K^* , K^∞ or $\neg L$.

V-left Given a sequent $F(BVC) \longrightarrow \Delta$, we can give two sequents $F(B) \longrightarrow \Delta$ and $F(C) \longrightarrow \Delta$.

V-right Given a sequent $\Gamma \longrightarrow \Delta, F(BVC)$, we can give the sequent $\Gamma \longrightarrow \Delta, F(B), F(C)$.

2.3.3 Replacement of expression

(i) Replacement by assumption

Let $F(e)$ be an arbitrary expression. Here the indicated e may be contained in several parts of $F(e)$. Let a sequent $F(A) \longrightarrow G(A)$, be given, where A is a formula. Then, according that the set of assumptions contains A or $\neg A$, we can give the sequent $F(I) \longrightarrow G(I)$, or $F(\Xi) \longrightarrow G(\Xi)$, respectively.

(ii) Inherent replacement

In our system, the followings are built in as

schemes of assumptions.

$G^0 = I, 7\bar{I} = I, \bar{I}^* = I, I^* = I, I//G = G, G//I = G, I;G = G,$
 $G;I = G, \bar{I}VG = G, G\vee\bar{I} = G, \bar{I}^\infty = \bar{I}, \bar{I};G = \bar{I}, \bar{I}//G = \bar{I},$
 $G//\bar{I} = \bar{I}, 7I = \bar{I},$
 if G does not contain an expression of the form
 $K^\infty, G; \bar{I} = \bar{I},$
 $K^\infty; H = K, (K^\infty//G); H = K^\infty//G, (G//K^\infty); H = G//K^\infty,$
 $(G^*)^\infty = (G^*; I^\infty)VG^\infty$ and $G^m; G^\infty = G^\infty.$

Given a sequent, every figure of the same form as the left-hand side of '=' in the sequent is replaced by the right-hand side by matching.

We roughly explain the meanings of some of them. Let the following parallel program be given:
perbegin P1: begin L1: if Q then goto L1; S end;
 P2: begin G end

parend.

In P1, one of the following (i) and (ii) occurs:
 (i) Q is satisfied finite times and after that $7Q$ is satisfied and then S is executed (denoted by $Q^*; 7Q; S$) and (ii) Q is satisfied eternally (denoted by Q^∞). So in P1//P2, either $(Q^*; 7Q; S)//G$ or $Q^\infty//G$ is executed. If $(Q^*; 7Q; S)//G$ is

executed, Q is satisfied finite times (i.e. $\bar{I}^0$ or I^*), the value of Q finally becomes $\bar{I}$. So the value of $\bar{I}^0$ or I^* is I. And then the infinite loop Q^∞ (i.e. $\bar{I}^\infty$) is impossible, i.e. the value of $\bar{I}^\infty$ is $\bar{I}$. Since (i) is executed, the execution of $Q^\infty//G$ is impossible, i.e. the value of $\bar{I}^\infty//G$ is $\bar{I}$. If $Q^\infty//G$ is executed, then the value of $7Q$ is $\bar{I}$ and the execution of $Q^*; 7Q; S$ is impossible, i.e. the value of $I^*; \bar{I}; S (= \bar{I}; S)$ is $\bar{I}$. Since K^∞ is the infinite loop of K, everything after K^∞ is unable to be executed. So the value of $K^\infty; H$ is K^∞ . In $K^\infty//G, G$ may eternally wait for the completion of K^∞ after its completion. Then the value of $(K^\infty//G); H$ is $K^\infty//G$.

2.3.4 Interruption

In order to describe the rule of inference for interruption, we define two ordered sets $\text{comp}(E)$ and $\text{lete}(E)$ for an arbitrary expression E, which does not contain //, as follows.

If F is an indivisible expression or an expression-constant, then

$\text{comp}(F) = (I, F)$ and $\text{lete}(F) = (F, I)$.
 Let F and G be arbitrary expressions, where
 $\text{comp}(F) = (F_{10}, \dots, F_{1m})$, $\text{lete}(F) = (F_{20}, \dots, F_{2m})$,
 $\text{comp}(G) = (G_{10}, \dots, G_{1n})$ and $\text{lete}(G) = (G_{20}, \dots, G_{2n})$.

Then

$\text{comp}(F; G) = (F_{10}, \dots, F_{1m}, F; G_{11}, \dots, F; G_{1n})$,
 $\text{lete}(F; G) = (F_{20}; G_{20}, \dots, F_{2m}; G_{21}, \dots, G_{2n})$,

$\text{comp}(FVG) = (I, F_{10}, \dots, F_{1m}, G_{10}, \dots, G_{1n}, FVG)$,
 $\text{lete}(FVG) = (FVG, F_{20}, \dots, F_{2m}, G_{20}, \dots, G_{2n}, I)$,
 $\text{comp}(F^*) = (I, F^*; F_{10}, \dots, F^*; F_{1m})$,
 $\text{lete}(F^*) = (F^*, F_{20}; F^*, \dots, F_{2m}; F^*)$,
 $\text{comp}(F^\infty) = (I, F^*; F_{10}, \dots, F^*; F_{1m})$,
 $\text{lete}(F^\infty) = (F^\infty, F_{20}; F^\infty, \dots, F_{2m}; F^\infty)$.

Let $F//G$ or $G//F$ be a subexpression in a sequent, where G does not contain //. Let an expression $F_1; F_2; \dots; F_m$ be the canonical form of F. Let F_i be the leftmost indivisible expression. Denoting $F_1; \dots; F_{i-1}$ or $F_{i+1}; \dots; F_m$ by H_1 or H_2 respectively, F is of the form $H_1; F_i; H_2$. When $\text{comp}(G) = (G_{10}, \dots, G_{1m})$ and $\text{lete}(G) = (G_{20}, \dots, G_{2m})$, the indicated $F//G$ is replaced by
 $(H_1//G_{10}); F_i; (H_2//G_{20}) \vee (H_1//G_{12}); F_i; (H_2//G_{22}) \vee \dots \vee (H_1//G_{1m}); F_i; (H_2//G_{2m})$. The case of $G//F$ is similar. In the actual programming, the sets $\text{comp}(G)$ and $\text{lete}(G)$ are not constructed, but it is processed in procedural way. (cf. 3.(6), (8))

2.3.5 Execution

(i) Execution of assignment statement

By G' we denote the result obtained by executing $x := E$ for G. Then G' is obtained as follows:
 Let $G_1; G_2; \dots; G_m$ be the canonical form of an expression G, and G_j the leftmost expression which contains an assignment statement to x. Then $G'_1, G'_2, \dots, G'_{j-1}$ are the results obtained by replacing E for x in $G_1, G_2, \dots, G_{j-1}$ respectively. If G_j is of the form $x := K$, then G'_j is $x := K'$, where K' is the result obtained by replacing E for x in K. If K does not contain x, then G'_j is G_j .

Let F be an arbitrary expression and $F_1; F_2; \dots; F_m$ its canonical form. And let F_i be the leftmost assignment statement of the form $x := E$ where E does not contain x, and let F_j ($i < j$) be the next leftmost expression which contains an assignment statement to x. Then the result obtained by execution of F_i is $F_1; \dots; F_i; F'_{i+1}; \dots; F'_j; F_{j+1}, \dots, F_m$.

(ii) Elimination of assignment statement

Let F and $F_1; F_2; \dots; F_m$ be the same as the previ-

ous one. And let F_i be the leftmost assignment statement to x . If we have F_j such that F_j is an assignment statement to x and $F_{i+1}; \dots; F_{j-1}$ does not contain the variable x , then F is replaced by $F_1; \dots; F_{i-1}; F_{i+1}; \dots; F_j; \dots; F_m$.

3. ALGORITHM

Given an arbitrary sequent $E \longrightarrow E_1, \dots, E_m$ and a set of assumptions, the prover constructs the proof-tree to the sequent in the following order.

(1) Whether the given sequent is an axiom or not, is checked. If the left-hand side of the sequent is $\overline{x}$, it is an axiom. If an expression in the left-hand side occurs in the right-hand side, it is also an axiom. When we have no remaining sequent, the proof is completed, and the obtained proof-tree is printed out. If we have a remaining sequent, the prover has the last one of them as the given one, and goes to (1). If it is not an axiom, then the prover goes to (2).

(2) The prover executes assignment statements contained in $E, E_1, \dots, E_m$, if any, from left to right and the control goes to (1). Otherwise it goes to (3).

(3) If a rule of inference 'inherent replacement (2.3.3 (ii))' is applicable to the given sequent, then the prover executes it, and the control goes to (1). Otherwise, it goes to (4).

(4) If a rule of inference 'replacement by assumption (2.3.3 (i))' is applicable to the given sequent, then the prover executes it, and the control goes to (1). Otherwise, it goes to (5).

(5) If a rule of inference ' $\vee$ -left' is applicable to the given sequent, then the prover executes it. Let the given one be $F(BVC) \longrightarrow \Delta$, where the indicated BVC is not in any inside of $\neg, *$ or $\forall$. Then for the next step, $F(B) \longrightarrow \Delta$ is the given sequent and $F(C) \longrightarrow \Delta$ is the last remaining one. And the control goes to (1). Otherwise, it goes to (6).

(6) If a rule of inference 'interruption' is applicable to the left-hand side of the given sequent, then the prover executes it. Let the given one be $F(G//H) \longrightarrow \Delta$, where the indicated $G//H$ is not in any inside of $\neg, *$ or $\overline{x}$. And let the result be $F(A_1 \vee \dots \vee A_n) \longrightarrow \Delta$. Then the prover constructs the following n sequents:

$F(A_1) \longrightarrow \Delta$ (the given sequent for the next step),
 $F(A_2) \longrightarrow \Delta$ (the last remaining sequent),
 $F(A_3) \longrightarrow \Delta, \dots, F(A_n) \longrightarrow \Delta$,

and the control goes to (1). If an interruption is not applicable to the left-hand side, then the control goes to (7).

(7) If a rule of inference ' $\vee$ -right' is applicable to the sequent, then the prover executes it. And the control goes to (1). Otherwise goes to (8).

(8) If a rule of inference 'interruption' is applicable to the right-hand side of the given sequent, then prover executes it. Let the given one be $\Gamma \longrightarrow \Delta_1, F(G//H), \Delta_2$, where the indicated $G//H$ is not in any inside of $\neg, *$ or $\forall$. And let the result be $\Gamma \longrightarrow \Delta_1, F(A_1 \vee \dots \vee A_n), \Delta_2$. Then the prover constructs the sequent $\Gamma \longrightarrow \Delta_1, F(A_1), \dots, F(A_n), \Delta_2$, and the control goes to (1). If an interruption is not applicable to the right-hand side, then it goes to (9).

(9) We fail the proof. Then we can select one of the following two modes. One, we stop the prover and print out the proof-tree obtained so far. Two, we return the control to the last remaining sequent, and continue to construct the proof-tree. If we have no remaining sequent, the proof-tree obtained so far is print out.

In our prover, we can change the order of (2), ..., (8) by presenting the table of the order of execution in advance.

4. EXAMPLES

4.1 An example of mutually exclusive but deadlocked program

E.W.Dijkstra gave an example of a program which satisfies the mutual exclusion but causes the "deadlock"[1]. His program is the following:

```
begin integer C1, C2; C1:=1; C2:=1;
  parbegin
    process 1: begin A1: C1:=0;
                  L1: if C2=0 then goto L1;
                  critical section 1; C1:=1;
                  remainder of cycle 1; goto A1
                end;
    process 2: begin A2: C2:=0;
                  L2: if C1=0 then goto L2;
                  critical section 2; C2:=1;
```

remainder of cycle 2; goto A₂
end

parent
end

Let P₁ or P₂ be C₁:=0; (C₂≠0 ∨ (C₂=0)[∞]); CS₁; C₁:=1
or C₂:=0; (C₁≠0 ∨ (C₁=0)[∞]); CS₂; C₂:=1, respectively.
Then the program except for the last "goto A₁"
or "goto A₂" can be expressed by the following:

C₁:=1; C₂:=1; (P₁//P₂),

provided that the program does not halt. By Δ, we denote the following:

CS₁; C₁:=1; CS₂; C₂:=1, CS₂; C₂:=1; CS₁; C₁:=1;
C₁:=0; C₂:=0; (I[∞]//I[∞]), C₂:=0; C₁:=0; (I[∞]//I[∞]).

Then the sequent which we want to prove is
C₁:=1; C₂:=1; (P₁//P₂) → Δ,

with assumptions 0=0, 1=1, 1≠0, 0≠1.

Because it is not an axiom and the replacement of assumptions can not be applied, the prover applies V-left, and we have

(1) C₁:=1; C₂:=1; ((C₁:=0; C₂≠0; CS₁; C₁:=1)
// (C₂:=0; (C₁≠0 ∨ (C₁=0)[∞]); CS₂; C₂:=1)) → Δ

and

(2) C₁:=1; C₂:=1; ((C₁:=0; (C₂=0)[∞]; CS₁; C₁:=1)
// (C₂:=0; (C₁≠0 ∨ (C₁=0)[∞]); CS₂; C₂:=1)) → Δ

From (1), applying V-left, we have

(3) C₁:=1; C₂:=1; ((C₁:=0; C₂≠0; CS₁; C₁:=1)
// (C₂:=0; C₁≠0; CS₂; C₂:=1)) → Δ,

and

C₁:=1; C₂:=1; ((C₁:=0; C₂≠0; CS₁; C₁:=1)
// (C₂:=0; (C₁=0)[∞]; CS₂; C₂:=1)) → Δ.

The prover applies interruption to the left-hand side of (3) for C₁=0, so we have

(4) C₁:=1; C₂:=1; (I//I); C₁:=0; ((C₂≠0; CS₁; C₁:=1)
// (C₂:=0; C₁≠0; CS₂; C₂:=1)) → Δ,

...,

C₁:=1; C₂:=1; (I//((C₂:=0; C₁≠0; CS₂; C₂:=1)); C₁:=0;
((C₂≠0; CS₁; C₁:=1) // I) → Δ.

Applying to (4) elimination of assignment(C₁:=1)
and replacement by assumption(G//I=G and G;I=G), we have

(5) C₂:=1; C₁:=0; ((C≠0; CS₁; C₁:=1)
// (C₂:=0; C₁≠0; CS₂; C₂:=1)) → Δ.

By the application of interruption to the left-hand side of (5) for C₂≠0, we have

(6) C₂:=1; C₁:=0; (I//I); C₂≠0; ((CS₁; C₁:=1)
// (C₂:=0; C₁≠0; CS₂; C₂:=1)) → Δ,

...,

C₁:=1; C₂:=1; (I//((C₁:=0; C₂:=0; C₁≠0; CS₂; C₂:=1));
C₂≠0; ((CS₁; C₁:=1) // I) → Δ.

Applying to (6) replacement by assumption(G//I=G and G;I=G), execution of assignment((C≠0)=(1≠0)) and replacement by assumptions((1≠0)=I and G;I=G), we have

(7) C₂:=1; C₁:=0; ((CS₁; C₁:=1)
// (C₂:=0; C₁≠0; CS₂; C₂:=1)) → Δ.

Applying interruption to the left-hand side of (7) for C₁=1, we have

(8) C₂:=1; C₁:=0; (CS₁//I); C₁:=1;
(I//((C₂:=0; C₁≠0; CS₂; C₂:=1)) → Δ,

(9) C₂:=1; C₁:=0; (CS₁//C₂:=0); C₁:=1;
(I//((C₁≠0; CS₂; C₂:=1)) → Δ,

....

Applying execution and elimination of assignment, replacement of assumption to (8), we have

(10) CS₁; C₁:=1; CS₂; C₂:=1 → Δ.

This is an axiom. Then the prover stores the sequents (8) and (10) in the disk storage and returns to (9). Repeating these operations, at length, the control returns to the decomposition of the sequent (which is one of the results of V-left to (2))

(11) C₁:=1; C₂:=1; ((C₁:=0; (C₂=0)[∞]; CS₁; C₁:=1)
// (C₂:=0; (C₁=0)[∞]; CS₂; C₂:=1)) → Δ.

Applying replacement by assumptions(K[∞];H=K[∞])
(11) becomes

(12) C₁:=1; C₂:=1; ((C₁:=0; (C₂=0)[∞])
// (C₂:=0; (C₁=0)[∞])) → Δ.

By interruption for C₁=0, we have

C₁:=1; C₂:=1; (I//I); C₁:=0;
((C₂=0)[∞] // ((C₂:=0; (C₁=0)[∞])) → Δ,

(13) C₁:=1; C₂:=1; (I//C₂:=0); C₁:=0;
((C₂=0)[∞] // (C₁=0)[∞]) → Δ,

(14) C₁:=1; C₂:=1; (I//((C₂:=0; (C₁=0)^{*})); C₁:=0;
((C₂=0)[∞] // (C₁=0); (C₁=0)[∞]) → Δ,

(15) C₁:=1; C₂:=1; (I//((C₂:=0; (C₁=0)^{*}; C₁=0));
C₁:=0; ((C₂=0)[∞] // (C₁=0)[∞]) → Δ.

Applying execution and elimination of assignment and replace by assumption, to (13), we have

(16) C₂:=0; C₁:=0; (I[∞] // I[∞]) → Δ.

This is an axiom and represents the deadlock. From (14), we have the same sequent as (16). Applying execution of assignment to (15), we have

$C_2:=0; (l=0)*; l:=0; C_1:=0;$

$((C_2=0)^\infty // (C_1=0)^\infty) \rightarrow \Delta.$

By replacement of assumptions, we have

$\Gamma \rightarrow \Delta,$

which is an axiom.

In this problem, the number of leaves of the proof-tree is 38. And the time for proving is 112 seconds.

4.2 Dijkstra's solution for two processes

E.W.Dijkstra gave the famous solution [2] p.58. In order to prove the correctness of the solution, it is sufficient to show the following sequent in our system. In the following, $h=1$ or $h=2$.

$C_1:=1; C_2:=1; t:=h;$

$$\left(\begin{array}{c} C_1:=0; \\ \left(\begin{array}{c} C_2=0 \\ t=1 \\ C_2 \neq 0 \end{array} * V \left(\begin{array}{c} C_2=0 \\ t \neq 1 \\ C_1:=1 \\ t \neq 2 \\ C_1:=0 \\ C_2 \neq 0 \end{array} * V \left(\begin{array}{c} C_2=0 \\ t \neq 1 \\ C_1:=1 \\ t \neq 2 \\ C_1:=0 \\ C_2=0 \\ t \neq 1 \\ C_1:=1 \\ (t=2)^\infty \end{array} * V \left(\begin{array}{c} C_2=0 \\ t \neq 1 \\ C_1:=1 \\ t \neq 2 \\ C_1:=0 \\ (C_2=0)^\infty \end{array} * V \left(\begin{array}{c} C_2=0 \\ t \neq 1 \\ C_1:=1 \\ t \neq 2 \\ C_1:=0 \end{array} \right) \right) \right) \right) \right) \\ CS_1; t:=2; C_1:=1 \end{array} \right)$$

//

$$\left(\begin{array}{c} C_2:=0; \\ \left(\begin{array}{c} C_1=0 \\ t=2 \\ C_1 \neq 0 \end{array} * V \left(\begin{array}{c} C_1=0 \\ t \neq 2 \\ C_2:=1 \\ t \neq 1 \\ C_2:=0 \\ C_1 \neq 0 \end{array} * V \left(\begin{array}{c} C_1=0 \\ t \neq 2 \\ C_2:=1 \\ t \neq 1 \\ C_2:=0 \\ C_1=0 \\ t \neq 2 \\ C_2:=1 \\ (t=1)^\infty \end{array} * V \left(\begin{array}{c} C_1=0 \\ t \neq 2 \\ C_2:=1 \\ t \neq 1 \\ C_2:=0 \\ (C_1=0)^\infty \end{array} * V \left(\begin{array}{c} C_1=0 \\ t \neq 2 \\ C_2:=1 \\ t \neq 1 \\ C_2:=0 \end{array} \right) \right) \right) \right) \right) \\ CS_2; t:=1; C_2:=1 \end{array} \right)$$

$\rightarrow CS_1; C_1:=1; CS_2; t:=1; C_2:=1,$

$CS_2; C_2:=1; CS_1; t:=2; C_1:=1.$

In this proof, the number of leaves was 690 and it took 3074 seconds long in which 2976 garbage collections were included.

5. CONCLUSION

Our prover is very useful for verifying logical properties such as the mutual exclusion or deadlock free problem. But at the present, this prover is basic and the construction of a proof-tree is straightforward. So it has many branches and leaves. In the next step, we intend to supply it with the following facilities: (1) analyses of syntactic patterns, for example, application of rules of inference with attention to a fixed variable, (2) restricted mathematical induction (cf. [7]) for processing infinitely long expression F^* , and (3) man-machine interaction.

REFERENCES

- [1] Dijkstra, E.W., "Co-operating sequential processes," Programming Languages, edited by F.Genuy, Academic Press, 1968, 43-112.
- [2] Gries, D., "An Exercise in Proving Parallel Programs Correct," CACM, vol.20, Dec. 1977, 921-930.
- [3] Hoare, C.A.R., "Towards a theory of parallel programming," Operating System Techniques, edited by C.A.R.Hoare and R.H.Perrott, Academic Press, 1972, 61-71.
- [4] Hoare, C.A.R., "Parallel Programming: An axiomatic approach," Computer Languages, vol.1, 1975, 151-160.
- [5] Knuth, D.E., "Additional Comments on a problem in concurrent programming control," CACM, vol.9, NO.5, May 1966, 321-322.
- [6] Nakanishi, M., KLISP Reference Manual (Revised Version), Keio Univ., 1970 (in Japanese).
- [7] Nakanishi, M., Nishimura, T. and Nagata, M., "Gentzen-type Formal System Representing Properties of Function and Its Implementation," Proc. of IJCAI 75, 1975, 57-64.
- [8] Nishimura, T., "Formalization of Concurrent Processes," Information Processing 77, 1977, 929-937.
- [9] Owicki, S. and Gries, D., "An axiomatic proof technique for parallel programs I," Acta Informatica, vol.19, No.5, May 1976, 279-285.
- [10] Owicki, S. and Gries, D., "Verifying properties of parallel programs: An axiomatic approach," CACM, vol.19, No.5, May 1976, 279-285.