A Systematic Approach to Parallel Program Verification

Tadao TAKAOKA

Department of Computer Science
Ibaraki University
Hitachi, Ibaraki 316, JAPAN
E-mail: takaoka@cis.ibaraki.ac.jp
Phone: +81 294 38 5130
Fax: +81 294 34 3254

August, 1995

Abstract

In this paper we investigate parallel program verification with directed graphs and assertion matrices. The parallel computational model is that with shared variables and each comprising process runs asynchronously. A program graph is a direct product of the flowcharts of comprising processes. The vertices of the graph correspond to global control points of the given parallel program, and edges correspond to an execution of one statement of one process, whereby the control moves in the process. We attach assertions to the vertices in the graph, and statements to edges which are either assignment statements or branching operations. If we can verify the consistencies of the assertions over edges, we say the set of assertions are induced by the parallel program and the pre-condition. It is usually difficult to find these assertions.

When we only have two processes, the set of assertions becomes an assertion matrix. We show that the assertion matrix can be decomposed into an independent part and dependent part uniquely. Using this property, we can prove the independent part first and then the dependent part using the information so far obtained, that is, incrementally.

Finally a new concept of eventual correctness is introduced, which states the partial correctness of the parallel program and its eventual termination.

Keywords: Parallel program verification, assertional method, logical independence, incremental verification, eventual correctness

1 Introduction

Parallel programs are hard to design. Parallel program verification is even harder. Since the 1970's, many efforts have been made to establish easy methods for verification. If we wish to verify a parallel program with n concurrent processes, we have to take into consideration the state space consisting of the values of program variables and current control points of the processes, whose size is very large, exponential in n.

To avoid this complexity, Hoare [4] and Milner [11] gave parallel computational models in which the comprising processes cooperate with each other only through communication. On the other hand, we sometimes face a situation where processes cooperate through shared memory. One such example is a parallel implementation of on-the-fly garbage collection algorithm [2].

In this paper we consider a parallel computational model in which the comprising processes share global variables and proceed asynchronously mainly based on the Ada tasks. Also we consider synchronization mechanisms in our model. Manna and Pnueli [10] gave a proof technique based on possible execution traces in the state space including program variables and control points. In their framework the lengths of traces have no upper bound related to the size of the given parallel program. The state space method is also employed in the analysis of deadlocks by Petri nets by Murata [12], in which a direct product of Petri nets is drawn visually to analyze control flows of a parallel program.

Our approach is based on the assertional method, which was originated for sequential program verification by Hoare [5] and Floyd [3]. There are a number of authors who used the assertional method for parallel program verification. They include Ashcroft [1], Owicki and Gries [13], Lamport [9], etc. Generally speaking, we can not verify a parallel program by checking the consistencies of assertions on program variables in the comprising processes locally, since an assertion in one process is affected by where other processes are currently being executed. Ashcroft employs a global assertion which incorporates all control points of all processes. Owicki and Gries employ auxiliary variables to indicate the current control points of the processes. Lamport uses what are called at-predicates, which are essentially predicates of program counters of the processes.

In this paper we use a more visual method based on directed graphs. We introduce the concept of program graph, which is a direct product of flowcharts of the comprising processes. This graph-theoretic approach was used in Keller [8], Joseph, et al. [7], Paczkowski [14], Takaoka [16], etc. In this method, assertions are given to the vertices of the graph and the consistencies of assertions over the edges are verified. The number of vertices, and consequently, assertions, of the directed graph is very large. If we have two processes with m and n control points, for example, we have to give mn assertions. Some efforts to reduce the number of these assertions are made in the above mentioned papers. We propose in this paper the concepts of assertion matrices and incremental verification for parallel program verification with two processes. The (i, j)-element of the assertion matrix is an assertion for the vertex corresponding to control points i and j in the two processes. We show that a given assertion matrix is decomposed into an independent part and dependent part uniquely. Using this decomposed form, we can verify the independent part first, which is rather easy, and then the dependent part, that is, incrementally. Intuitively speaking, the independent part is the set of assertions that hold in each process independently. Using a few examples, we show how this new technique reduces our efforts for parallel program verification. We also give a logical basis for the new technique. Finally we give a new concept of eventual correctness which

is partial correctness plus eventual termination.

2 Programming constructs and labels

The basic syntax of our parallel programs is the same as that in [13]. A parallel program S consists of processes shown as follows:

$$S \equiv [S_1 \parallel \ldots \parallel S_n].$$

In this construct, $S_1, \cdots, S_n$ are called processes, which are executed concurrently. Each S_i mainly follows the syntax and semantics of PASCAL or Ada and is executed sequentially. We introduce labels in each S_i in such a way that a label is put at the control point before each statement and after the last statement. Control points are identified by these labels. The parallel computational model adopted here is that S_i's share global variables and are executed asynchronously if there are no synchronization statements.

We use the statement **await** B for indirect synchronization and the pair **accept**(k) and **call**(k) for direct synchronization primitives, where k is to identify those primitives. If the **await** B statement is in process S_i and $B =$ **false**, it waits at this point until B is made **true** by some other process. This is viewed as synchronization using shared memory. If **accept**(k) and **call**(k) are in processes S_i and S_j, the two processes synchronize directly using program counters. This is a simplified version of rendezvous in Ada, that is, a buffered blocking wait.

Let α be the label of control point before boolean condition B and β and γ are those at the exits of B with **true** and **false** respectively. Then we define the connective between α and β by B and that between α and γ by $\neg B$. Let α and β be the labels of control points before and after assignment statement $x := e$. Then the connective between α and β is defined by $x := e$.

Let α and β be the labels of control points before and after **accept**(k) in process S_i and γ and δ be those before and after **call**(k) in process S_j. Then the connective between (α, γ) and (β, δ) is defined by **accept**$(k) \cdot$ **call**(k) which is abbreviated by **syn**(k).

Definition 1 Let $L_1, \cdots, L_n$ be the sets of control points of processes $S_1, \cdots, S_n$ in a parallel program $S \equiv [S_1 \| \cdots \| S_n]$. The program graph $G = (V, E, C)$ for S is a directed graph with the set of vertices, V, that of edges, E, and the mapping C from E to connectives in S, where

$$\begin{aligned} V &= L_1 \times \cdots \times L_n \text{ (direct product)} \\ E &= \{(\boldsymbol{\xi}, \boldsymbol{\xi}') \mid \text{There is a connective from } \boldsymbol{\xi} \text{ to } \boldsymbol{\xi}'\} \\ C(\boldsymbol{\xi}, \boldsymbol{\xi}') &\text{ is the connective from } \boldsymbol{\xi} \text{ to } \boldsymbol{\xi}'. \end{aligned}$$

There is a connective from $(\xi_1, \cdots, \xi_i, \cdots, \xi_n)$ to $(\xi_1, \cdots, \xi_i', \cdots, \xi_n)$ if there is a connective from ξ_i to ξ_i' in S_i. There also is a connective **syn**(k) from $(\xi_1, \cdots, \xi_i, \cdots, \xi_j, \cdots, \xi_n)$ to $(\xi_1, \cdots, \xi_i', \cdots, \xi_j', \cdots, \xi_n)$ if **syn**(k) exists between (ξ_i, ξ_j) to (ξ_i', ξ_j').

Connectives are attached to edges in G as labels. The execution of a parallel program corresponds to traversing over the program graph. Our atomicity assumption for assignment statements is that the evaluation of right-hand side and the storing operation are separate ones. If we have data dependency in two assignment statements $x := e_1$, and $y := e_2$ in two processes, we can expand them as $z_1 = e_1;\ x := z_1$ and $z_2 := e_2;\ y := z_2$.

Alternatively for short cut we can have diagonal edges to execute these statements; two diagonal edges corresponding to the two statements if x and y are the same variables, and one diagonal edge corresponding to the parallel execution of the two statements if x and y are different. If there is no data dependency, the parallel execution is equivalently divided into two edges, that is, interleaving. A similar definition of parallel execution of more assignment statements can be made. We mainly focus on parallel programs with two processes.

Definition 2 The valid triples for connectives are defined as follows:

Assignment	$\{P_e^x\}x := e\{P\}$
Parallel assignment	$\{P_{e_1,e_2}^{x,y}\}x := e_1$ **par** $y := e_2\{P\}$
Condition B	$\{P\}B\{P \wedge B\}$
Condition $\neg B$	$\{P\}\neg B\{P \wedge \neg B\}$
Synchronization	$\{P\}\mathbf{syn}(k)\{P\}$.

In the above $P_{e_1,e_2}^{x,y}$ means the simultaneous substitution of e_1 and e_2 for x and y in P, and $x := e_1$ **par** $y := e_2$ means the parallel execution of $x := e_1$ and $y := e_2$.

In addition, we use frequently the following consequence rule for valid triples with connective C.

$$\frac{P \supset P_1, \{P_1\}C\{P_2\}, P_2 \supset Q}{\{P\}C\{Q\}}.$$

Definition 3 The proof augmented graph (proof graph, for short) for $\{P\}S\{Q\}$, where S is a parallel program $S \equiv [\alpha_1\colon\ S_1;\ \beta_1\colon\ \|\ \cdots\ \|\ \alpha_n\colon\ S_n;\ \beta_n:]$, is a program graph of S with assertions $P_{\boldsymbol{\xi}}$ attached to each vertex $\boldsymbol{\xi}$ in such a way that

$$\begin{aligned} &P \equiv P_{\boldsymbol{\alpha}},\ \text{where } \boldsymbol{\alpha} = (\alpha_1, \cdots, \alpha_n)\\ &Q \equiv P_{\boldsymbol{\beta}},\ \text{where } \boldsymbol{\beta} = (\beta_1, \cdots, \beta_n)\\ &\{P_{\boldsymbol{\xi}}\}C(\boldsymbol{\xi},\ \boldsymbol{\xi}')\}\{P(\boldsymbol{\xi}')\}\ \text{for all } (\boldsymbol{\xi},\ \boldsymbol{\xi}') \in E. \end{aligned}$$

If we have a proof graph for $\{P\}S\{Q\}$, we can say we have established the partial correctness of $\{P\}S\{Q\}$.

Example 1 The proof graph for

$\{\mathbf{true}\}$
$[1\colon\ x := 1;\ 2\colon\ \|\ 1\colon\ y := 2;\ 2:]$
$\{x = 1 \wedge y = 2\}$

is given in Fig. 1.

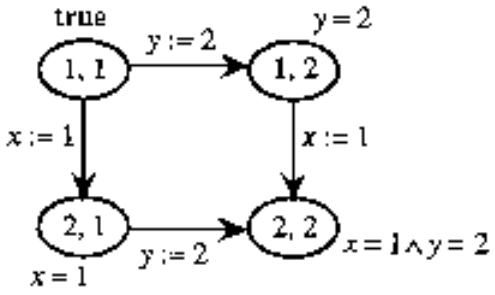

Fig. 1. Proof graph for Example 1

Example 2 The proof graph for

$$\{x = 1 \wedge y = 2\}$$
$$[1{:}\ x := y;\ 2{:}\ \ \|\ \ 1{:}\ y := x;\ 2{:}]$$
$$\{(x = 1 \wedge y = 1) \vee (x = 2 \wedge y = 2) \vee (x = 2 \wedge y = 1)\}$$

is given in Fig. 2.

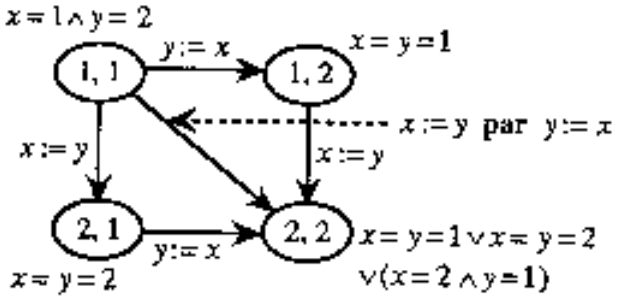

Fig. 2. Proof graph for Example 2

The three "or"ed conditions at $(2,2)$ correspond to the three incoming edges. From this point on, we assume that an assignment statement is atomic for simplicity.

Example 3 A proof graph for a parallel program

$$[\,S_1 \equiv \begin{array}{l} 1 : x := 1; \\ 2 : \mathbf{accept}(1); \\ 3 : \end{array} \quad \| \quad S_2 \equiv \begin{array}{l} 1 : \mathbf{call}(1); \\ 2 : x := 2; \\ 3 : \end{array}$$
$$]$$

is given in Fig. 3.

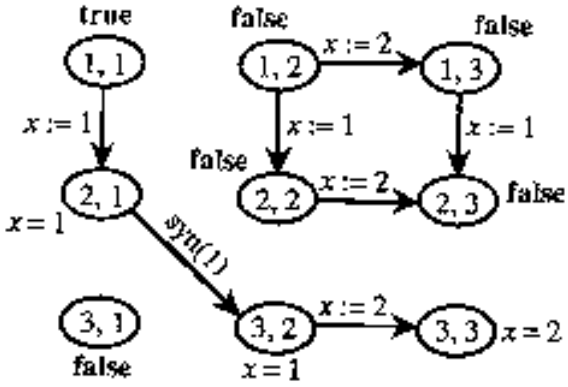

Fig. 3. Proof graph for Example 3

Example 4 A proof graph for a parallel program

$$\begin{array}{l} S \equiv \\ [S_1 \equiv 1 : \ \mathbf{while}\ x > 0\ \mathbf{do\ begin} \\ \qquad\qquad 2 : \ x := x - 1; \\ \qquad\qquad\ \ \mathbf{end}; \\ \qquad\quad 3 : \\ \qquad \| \\ \ \ S_2 \equiv 1 : \ x := 0; \\ \qquad\quad 2 : \\ ] \end{array}$$

is given in Fig. 4.

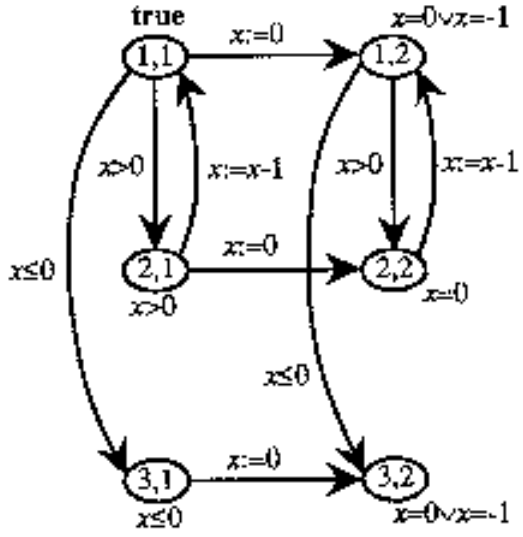

Fig. 4. Proof graph for Example 4

3 Logical background

We use only integer values for numerical values in our program. Since assertions are attached to vertices in the program graph, they depend only on program variables. For assertions P and Q, if $P \supset Q$, we say P is stronger than or equivalent to Q. If $P \supset Q$ and $P \not\equiv Q$, we say P is stronger than Q.

Let the labels of control points in S_i be given by $\{1, 2, \cdots, m_i\}$ and the initial vertex in the program graph be given by $\boldsymbol{\xi}_0 = (1, 1, \cdots, 1)$. A path is a sequence of edges.

Definition 4 Let assertion $P_{\boldsymbol{\xi}}$ be given to each vector $\boldsymbol{\xi}$. The set of assertions $\{P_{\boldsymbol{\xi}} | \boldsymbol{\xi} \in V\}$, which is denoted by $(P_{\boldsymbol{\xi}})$, is said to be induced by program configuration (P_0, S), where P_0 is the precondition of a parallel program S, if

$$P_0 \supset P_{\boldsymbol{\xi}_0}$$
$$\{P_{\boldsymbol{\xi}}\} C(\boldsymbol{\xi}, \boldsymbol{\xi}') \{P_{\boldsymbol{\xi}'}\} \text{ for all } (\boldsymbol{\xi}, \boldsymbol{\xi}') \in E.$$

Note that if $n = 1$, $(P_{\boldsymbol{\xi}})$ is an assertion vector and if $n = 2$, $(P_{\boldsymbol{\xi}})$ is an assertion matrix.

Theorem 1 *Let $(P_{\boldsymbol{\xi}})$ and $(Q_{\boldsymbol{\xi}})$ be sets of assertions induced by (P_0, S). Then sets of assertions $(P_{\boldsymbol{\xi}} \wedge Q_{\boldsymbol{\xi}})$ and $(P_{\boldsymbol{\xi}} \vee Q_{\boldsymbol{\xi}})$ are induced by (P_0, S).*

Proof. It follows directly from the "and" and "or " rules in [6]. □

We extend the "and" and "or" rules to countably many assertions.

Definition 5 Let the set of paths from $\boldsymbol{\xi}_0$ to $\boldsymbol{\xi}$ be denoted by $PATH(\boldsymbol{\xi})$. Let the strongest post-condition Q such that $\{P\}C\{Q\}$ be denoted by $sp(P, C)$ and called the strongest post-condition for P and C. For the precondition P_0 for S and a path $\sigma = (\boldsymbol{\xi}_0, \boldsymbol{\xi}_1)(\boldsymbol{\xi}_1, \boldsymbol{\xi}_2) \cdots (\boldsymbol{\xi}_{k-1}, \boldsymbol{\xi}_k)$, the assertion Q_σ is defined by

$$\begin{aligned} R_0 &\equiv P_0 \\ R_i &\equiv sp(R_{i-1}, C(\boldsymbol{\xi}_{i-1}, \boldsymbol{\xi}_i)), \ (i = 1, \cdots, k) \\ Q_\sigma &\equiv R_k. \end{aligned}$$

Note that $Q_\lambda \equiv P_0$ where λ is the null path. The assertion $Q_{\boldsymbol{\xi}}$ is then defined by

$$Q_{\boldsymbol{\xi}} \equiv \bigvee_{\sigma \in PATH(\boldsymbol{\xi})} Q_\sigma.$$

Theorem 2 *The set of assertions $(Q_{\boldsymbol{\xi}})$ defined above is induced by (P_0, S) and is the strongest set of assertions among those induced by (P_0, S).*

Proof. Since $PATH(\boldsymbol{\xi}_0)$ has the null path, we have $P_0 \supset Q_{\boldsymbol{\xi}_0}$. Next we observe that for any $(\boldsymbol{\xi}, \boldsymbol{\xi}') \in E$, $PATH(\boldsymbol{\xi}')$ includes all paths $\sigma \cdot (\boldsymbol{\xi}, \boldsymbol{\xi}')$ such that $\sigma \in PATH(\boldsymbol{\xi})$. Therefore we have $\{Q_{\boldsymbol{\xi}}\} C(\boldsymbol{\xi}, \boldsymbol{\xi}') \{Q_{\boldsymbol{\xi}'}\}$ from the extended "or" rule.

To prove that $(Q_{\boldsymbol{\xi}})$ is the strongest, we prove by induction on σ that $Q_\sigma \supset P_{\boldsymbol{\xi}}$ for any path $\sigma \in PATH(\boldsymbol{\xi})$ and any set of assertions $(P_{\boldsymbol{\xi}})$ induced by (P_0, S). For the basis of $\sigma = \lambda$, we have $Q_\lambda \equiv P_0 \supset P_{\boldsymbol{\xi}_0}$ by definition. For the induction step, assume that $Q_\sigma \supset P_{\boldsymbol{\xi}}$ and take any path $\sigma' = \sigma \cdot (\boldsymbol{\xi}, \boldsymbol{\xi}')$. Since $\{P_{\boldsymbol{\xi}}\} C(\boldsymbol{\xi}, \boldsymbol{\xi}') \{P_{\boldsymbol{\xi}'}\}$, we have $\{Q_\sigma\} C(\boldsymbol{\xi}, \boldsymbol{\xi}') \{P_{\boldsymbol{\xi}'}\}$, from the consequence rule. Since $Q_{\sigma'}$ is the strongest post-condition for Q_σ and $C(\boldsymbol{\xi}, \boldsymbol{\xi})$, we have $Q_{\sigma'} \supset P_{\boldsymbol{\xi}'}$. □

Theorem 3 *Let $(P_{\boldsymbol{\xi}})$ be a set of assertions induced by (P_0, S). For all $\boldsymbol{\xi}$ such that $(\boldsymbol{\xi}, \boldsymbol{\eta}) \in E$ assume that $\{P_{\boldsymbol{\xi}}\}C(\boldsymbol{\xi}, \boldsymbol{\eta})\{Q_{\boldsymbol{\eta}}\}$. Let the set of assertions $(Q_{\boldsymbol{\xi}})$ be defined, for other vertices than $\boldsymbol{\eta}$, by*

$$Q_{\boldsymbol{\xi}} = \mathbf{true}, \text{ for } \boldsymbol{\xi} \neq \boldsymbol{\eta}.$$

Then the set of assertions $(P_{\boldsymbol{\xi}} \wedge Q_{\boldsymbol{\xi}})$ is induced by (P_0, S).

Proof. From the "and" rule we have that for all $\boldsymbol{\xi}$ such that $(\boldsymbol{\xi}, \boldsymbol{\eta}) \in E$,

$$\frac{\{P_{\boldsymbol{\xi}}\}C(\boldsymbol{\xi}, \boldsymbol{\eta})\{P_{\boldsymbol{\eta}}\}, \{P_{\boldsymbol{\xi}}\}C(\boldsymbol{\xi}, \boldsymbol{\eta})\{Q_{\boldsymbol{\eta}}\}}{\{P_{\boldsymbol{\xi}} \wedge \mathbf{true}\}C(\boldsymbol{\xi}, \boldsymbol{\eta})\{P_{\boldsymbol{\eta}} \wedge Q_{\boldsymbol{\eta}}\}}.$$

We also have that for all $\boldsymbol{\gamma}$ such that $(\boldsymbol{\eta}, \boldsymbol{\gamma}) \in E$,

$$\frac{\{P_{\boldsymbol{\eta}}\}\{C(\boldsymbol{\eta}, \boldsymbol{\gamma})\{P_{\boldsymbol{\gamma}}\}, \{Q_{\boldsymbol{\eta}}\}C(\boldsymbol{\eta}, \boldsymbol{\gamma})\{\mathbf{true}\}}{\{P_{\boldsymbol{\eta}} \wedge Q_{\boldsymbol{\eta}}\}C(\boldsymbol{\eta}, \boldsymbol{\gamma})\{P_{\boldsymbol{\gamma}} \wedge \mathbf{true}\}}.$$

□

This theorem shows that we can verify the given program incrementally, that is, we start with a rather weak and simple set of assertions $(P_{\boldsymbol{\xi}})$ and strengthen it gradually by finding $Q_{\boldsymbol{\eta}}$'s described above. This concept of incremental verification becomes effective particularly if it is combined with the concept of logical independence defined in the next section.

4 Logical independence

We introduce the concepts in linear algebra, such as vector and matrices, into program verification. We only treat the case that the number of processes is one or two in this section. If we have only one process, the set of assertions $(P_{\boldsymbol{\xi}})$ becomes an assertion vector. If we have two processes, $(P_{\boldsymbol{\xi}})$ becomes an assertion matrix. For two assertions P and Q, if $P \supset Q$, we write $P \leq Q$. If $P \leq Q$ and $P \not\equiv Q$, we write $P < Q$. Similar notations are used for assertion vectors and assertion matrices using their elements. The "and" and "or" operations on assertion vectors and matrices are defined by using their elements. In the following, we treat a parallel program $S = [S_1 \parallel S_2]$ such that the processes S_1 and S_2 have m labels $\{1, 2, \cdots, m\}$ and n labels $\{1, 2, \cdots, n\}$ respectively.

Definition 6 The outer product $\boldsymbol{p} \times \boldsymbol{q}$ of two assertion vectors $\boldsymbol{p} = (P_1, \cdots, P_m)$ and $\boldsymbol{q} = (Q_1, \cdots, Q_n)$ is defined by the (m, n) assertion matrix whose (i, j) element is $P_i \wedge Q_j$. If an (m, n) assertion matrix P is expressed as $P = \boldsymbol{p} \times \boldsymbol{q} \wedge M$ for some (m, n) assertion matrix M, this expression is called a decomposition of P, or we say P is decomposed into $\boldsymbol{p} \times \boldsymbol{q} \wedge M$. In this form, $\boldsymbol{p} \times \boldsymbol{q}$ is called the independent part and M is called the dependent part. When P is induced by (P_0, S), the meaning of this form is that P_i holds at label i in S_1 regardless of where the control of S_2 is, and vice versa.

Theorem 4 *Any assertion matrix P is decomposed into the form $\boldsymbol{p} \times \boldsymbol{q} \wedge M$ such that $\boldsymbol{p} \times \boldsymbol{q}$ is the strongest assertion matrix and M is the weakest assertion matrix satisfying $P = \boldsymbol{p} \times \boldsymbol{q} \wedge M$. This form is called the normal form of P.*

Proof. A decomposition of P trivially exists since we can choose as $\boldsymbol{p} = \boldsymbol{t}_m$, $\boldsymbol{p} = \boldsymbol{t}_n$ and $M = P$ where $\boldsymbol{t}_k$ is the k-dimensional assertion vector all of whose elements are **true**. Assume that $\boldsymbol{p} \times \boldsymbol{q}$ and $\boldsymbol{p}' \times \boldsymbol{q}'$ are two strongest assertion matrices such that $\boldsymbol{p} \times \boldsymbol{q} \neq \boldsymbol{p}' \times \boldsymbol{q}'$ in $P = \boldsymbol{p} \times \boldsymbol{q} \wedge M$ and $P = \boldsymbol{p}' \times \boldsymbol{q}' \wedge M'$. Then we have

$$\begin{aligned} P &= \boldsymbol{p} \times \boldsymbol{q} \wedge M \wedge \boldsymbol{p}' \times \boldsymbol{q}' \wedge M' \\ &= (\boldsymbol{p} \wedge \boldsymbol{p}') \times (\boldsymbol{q} \wedge \boldsymbol{q}') \wedge M'', \end{aligned}$$

where $M'' = M \wedge M'$. Note that $(\boldsymbol{p} \wedge \boldsymbol{p}') \times (\boldsymbol{q} \wedge \boldsymbol{q}') < \boldsymbol{p} \times \boldsymbol{q}$ and $(\boldsymbol{p} \wedge \boldsymbol{p}') \times (\boldsymbol{q} \wedge \boldsymbol{q}') < \boldsymbol{p}' \times \boldsymbol{q}'$, which contradicts the fact that $\boldsymbol{p} \times \boldsymbol{q}$ and $\boldsymbol{p}' \times \boldsymbol{q}'$ are strongest.

Now let $P = \boldsymbol{p} \times \boldsymbol{q} \wedge M$ and $P = \boldsymbol{p} \times \boldsymbol{q} \wedge M'$ be two decompositions where $\boldsymbol{p} \times \boldsymbol{q}$ is the strongest matrix whose unique existence is shown above and M and M' are the weakest matrices such that $M \neq M'$ in the decompositions. Then $P = \boldsymbol{p} \times \boldsymbol{q} \wedge (M \vee M')$ is a decomposition and $M \vee M' > M$ and $M \vee M' > M'$, which is a contradiction. □

Theorem 5 *Assume assertion matrix P be induced by (P_0, S). Let assertion vectors $\boldsymbol{p} = (P_1, \cdots, P_m)$ and $\boldsymbol{q} = (Q_1, \cdots, Q_n)$ be defined by*

$$P_i = \bigvee_{j=1}^{n} P_{ij}, Q_j = \bigvee_{i=1}^{m} P_{ij},$$

where $P = (P_{ij})$. Then $\boldsymbol{p} \times \boldsymbol{q}$ is the unique strongest part in the normal form $P = \boldsymbol{p} \times \boldsymbol{q} \wedge M$.

Proof. Assume there is another decomposition form $P = \boldsymbol{p}' \times \boldsymbol{q}' \wedge M'$ where $\boldsymbol{p}' = (P_1', \cdots, P_m')$, $\boldsymbol{q} = (Q_1', \cdots, Q_n')$ and M' is an (m, n) matrix (M_{ij}'). Then by definition,

$$\begin{aligned} P_i &= \bigvee_{j=1}^{n} (P_i' \wedge Q_j' \wedge M_{ij}') \\ &= P_i' \wedge (\bigvee_{i=1}^{n} (Q_j \wedge M_{ij}')), \end{aligned}$$

from which we see $P_i \leq P_i'$ $(i = 1, \cdots, m)$. Similarly we have $Q_j \leq Q_j'$ $(j = 1, \cdots, n)$. Hence $\boldsymbol{p} \times \boldsymbol{q} \leq \boldsymbol{p}' \times \boldsymbol{q}'$. Theorem 4 guarantees the uniqueness of the strongest part $\boldsymbol{p} \times \boldsymbol{q}$ and the weakest part M. □

Example 5 The normal form of the assertion matrix in Example 4 is given as follows:

$$\begin{bmatrix} \textbf{true} & x = 0 \vee x = -1 \\ x > 0 & x = 0 \\ x \leq 0 & x = 0 \vee x = -1 \end{bmatrix} = \begin{bmatrix} \textbf{true} \\ x \geq 0 \\ x \leq 0 \end{bmatrix} \times [\textbf{true} \quad x = 0 \vee x = -1]$$
$$\wedge \begin{bmatrix} \textbf{true} & \textbf{true} \\ x > 0 & \textbf{true} \\ \textbf{true} & \textbf{true} \end{bmatrix}.$$

Definition 7 Let a parallel program $S = [S_1 \parallel S_2]$ and a program configuration (P_0, S) be given. An assertion vector $\boldsymbol{p} = (P_1, \cdots, P_m)$ and is said to be locally induced by (P_0, S_1) if $P_0 \supset P_1$ and $\{P_\xi\} C(\xi, \xi') \{P_{\xi'}\}$ for any connective $C(\xi, \xi')$ in S_1. The local induction of $\boldsymbol{q}$ by (P_0, S_2) is similarly defined. The vector $\boldsymbol{p}$ is said to be unaffected

by S_2 if $\{P_\xi\}C(\eta,\eta')\{P_\xi\}$ for any label ξ in S_1 and any connective $C(\eta,\eta')$ in S_2. The unaffectedness of $\boldsymbol{q}$ by S_1 is similarly defined. If $\boldsymbol{p}$ and $\boldsymbol{q}$ are locally induced by (P_0, S_1) and (P_0, S_2), and unaffected by S_1 and S_2, they are said to be logically independent with respect to (P_0, S). The concept of unaffectedness is slightly stronger and easier to use than interference-freedom of Owicki and Gries [13] .

Theorem 6 *If $\boldsymbol{p}$ and $\boldsymbol{q}$ are logically independent with respect to (P_0, S), $\boldsymbol{p}\times\boldsymbol{q}$ is induced by (P_0, S).*

Proof. We see $P_0 \supset P_1 \wedge Q_1$. From the "and" rule, we have

$$\frac{\{P_\xi\}C(\xi,\xi')\{P_{\xi'}\}, \{P_\eta\}C(\xi,\xi')\{P_\eta\}}{\{P_\xi \wedge P_\eta\}C(\xi,\xi')\{P_{\xi'} \wedge P_\eta\}}.$$

Similarly we have $\{P_\xi \wedge P_\eta\}C(\eta,\eta')\{P_\xi \wedge P_{\eta'}\}$. □

Theorem 7 *The assertion vectors defined in Theorem 5 are locally induced by (P_0, S_1) and (P_0, S_2) respectively.*

Proof. It follows directly from the "or" rule. □

The assertion matrices $\boldsymbol{p} \times \boldsymbol{q}$ in Theorem 5 is not necessarily induced by (P_0, S). When we do not know any assertion matrix P induced by (P_0, S) that satisfies some condition desired by our program verification purpose, how can we build such an assertion matrix? Our strategy is to find logically independent vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ and set $P \leftarrow \boldsymbol{p} \times \boldsymbol{q}$, which is induced by (P_0, S). If P does not satisfy the desired condition, we attach some $M = (M_{ij})$ to set $P \leftarrow P \wedge M$ where all but one element of M are **true**, and prove that P is induced by (P_0, S) based on Theorem 3. If P is not strong enough, we repeat this process. This process is incremental because to prove consistencies of newly attached M_{ij} in P, we use the information so far accumulated in P. The procedure is summarized in the following.

Incremental Verification Procedure.

(1) Initialization. Find logically independent vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ and set $P \leftarrow \boldsymbol{p} \times \boldsymbol{q}$.

(2) While P does not satisfy the given property do

> Find some $M = (M_{ij})$
> Set $P \leftarrow P \wedge M$
> Prove that P is induced by (P_0, S)
> Repeat (2).

In this method, we only deal with $m + n + \#(M)$ assertions where $\#(M)$ is the number of non-**true** assertions in the final M. This number $m + n + \#(M)$ is much smaller than mn in many applications, whereby we can reduce verification effort.

Example 6 The Peterson-Fisher mutual exclusion algorithm proven in [10] is given below with a simpler proof.

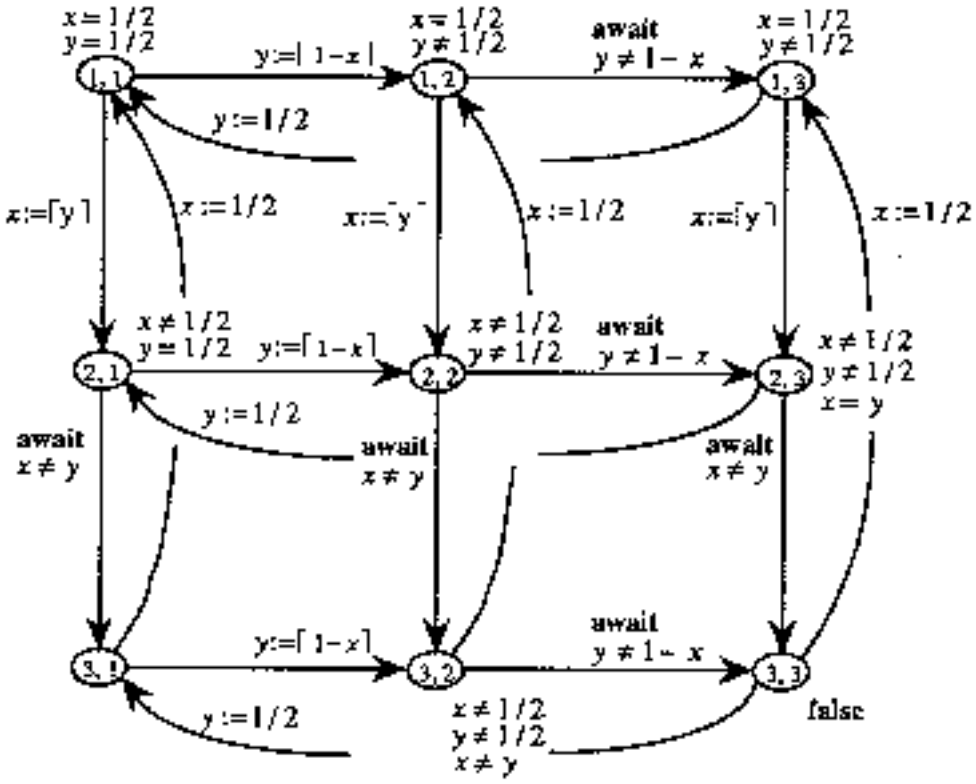

Fig. 5. Proof graph for Example 6

$S \equiv \{x = y = 1/2 \equiv P_0\}$

$[\, S_1 \equiv$ **loop**
$\quad \{\text{local task 1}\}$
$\quad 1 : x := \lceil y \rceil;$
$\quad 2 :$ **await** $x \neq y;$
$\quad \{\text{critical section 1}\}$
$\quad 3 : x := 1/2;$
$\quad$ **end loop**

$\parallel \quad S_2 \equiv$ **loop**
$\quad \{\text{local task 2}\}$
$\quad 1 : y := \lceil 1 - x \rceil;$
$\quad 2 :$ **await** $y \neq 1 - x;$
$\quad \{\text{critical section 2}\}$
$\quad 3 : y := 1/2;$
$\quad$ **end loop**

$\,]$.

The symbol $\lceil\ \rceil$ means the rounding-up operation. Local tasks and critical sections are commented out for simplicity. The proof graph is given in Fig. 5. Note that conjunctions are put in the assertions with "$\wedge$" signs omitted. Also we assume the global assertion that x and y take only three values 0, 1 and 1/2 applies everywhere in the program.

We choose $\boldsymbol{p}$ and $\boldsymbol{q}$ as follows:

$$\begin{aligned} \boldsymbol{p} &= (x = 1/2 \quad x \neq 1/2 \quad x \neq 1/2 \wedge x \neq y) \\ \boldsymbol{q} &= (y = 1/2 \quad y \neq 1/2 \quad y \neq 1/2 \wedge y \neq 1 - x). \end{aligned}$$

It is easy to see that $\boldsymbol{p}$ and $\boldsymbol{q}$ are locally induced by (P_0, S_1) and (P_0, S_2). To see unaffectedness, we examine assignment statements $y := \lceil 1-x \rceil$ and $y := 1/2$ for assertion

$x \neq 1/2 \wedge x \neq y$ in $\boldsymbol{p}$ and assignment statements $x := \lceil y \rceil$ and $x := 1/2$ for assertion $y \neq 1/2 \wedge y \neq 1 - x$ in $\boldsymbol{q}$. It is easy to see that these statements keep the corresponding assertions invariant. Thus $\boldsymbol{p}$ and $\boldsymbol{q}$ are logically independent, meaning that $\boldsymbol{p} \times \boldsymbol{q}$ given below is induced by (P_0, S).

$$\boldsymbol{p} \times \boldsymbol{q} = \begin{bmatrix} x = 1/2 & x = 1/2 & x = 1/2 \\ y = 1/2 & y \neq 1/2 & y \neq 1/2 \\ & & y \neq 1 - x \\ x \neq 1/2 & x \neq 1/2 & x \neq 1/2 \\ y = 1/2 & y \neq 1/2 & y \neq 1/2 \\ & & y \neq 1 - x \\ x \neq 1/2 & x \neq 1/2 & \textbf{false} \\ y = 1/2 & y \neq 1/2 & \\ x \neq y & x \neq y & \end{bmatrix}$$

Since we have a desired condition of **false** at $(3,3)$, meaning that $(3,3)$ is mutually excluded, $P = \boldsymbol{p} \times \boldsymbol{q}$ is the sought assertion matrix, which is given in a normal form. The $(2,3)$ element of P is equivalent to $x \neq 1/2 \wedge y \neq 1/2 \wedge x = y$.

Example 7 The Peterson algorithm for mutual exclusion [15] is given below.

$S \equiv \{thinking_1 = yes \wedge thinking_2 = yes : P_0\}$

$[S_1 \equiv$ **loop**	$\parallel S_2 \equiv$ **loop**
{local task 1}	{local task 2}
$1 : thinking_1 := no;$	$1 : thinking_2 := no;$
$2 : turn := 2;$	$2 : turn := 1;$
$3 :$ **await** $thinking_2 = yes \vee turn = 1;$	$3 :$ **await** $thinking_1 = yes \vee turn = 2;$
{critical section 1}	{critical section 2}
$4 : thinking_1 := yes;$	$4 : thinking_2 := yes;$
end loop	**end loop**

].

Its program graph is given below.
We abbreviate $thinking_1$ and $thinking_2$ as t_1 and t_2. The vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ are chosen as follows:

$$\begin{aligned} \boldsymbol{p} &= (t_1 = yes \quad t_1 = no \quad t_1 = no \quad t_1 = no) \\ \boldsymbol{q} &= (t_2 = yes \quad t_2 = no \quad t_2 = no \quad t_2 = no). \end{aligned}$$

Then clearly $\boldsymbol{p}$ and $\boldsymbol{q}$ are logically independent, since t_1 and t_2 are changed locally in S_1 and S_2 respectively. We set $P = \boldsymbol{p} \times \boldsymbol{q}$ as the starting matrix shown in the following.

$$P = \begin{bmatrix} t_1 = yes & t_1 = yes & t_1 = yes & t_1 = yes \\ t_2 = yes & t_2 = no & t_2 = no & t_2 = no \\ t_1 = no & t_1 = no & t_1 = no & t_1 = no \\ t_2 = yes & t_2 = no & t_2 = no & t_2 = no \\ t_1 = no & t_1 = no & t_1 = no & t_1 = no \\ t_2 = yes & t_2 = no & t_2 = no & t_2 = no \\ t_1 = no & t_1 = no & t_1 = no & t_1 = no \\ t_2 = yes & t_2 = no & t_2 = no & t_2 = no \end{bmatrix}$$

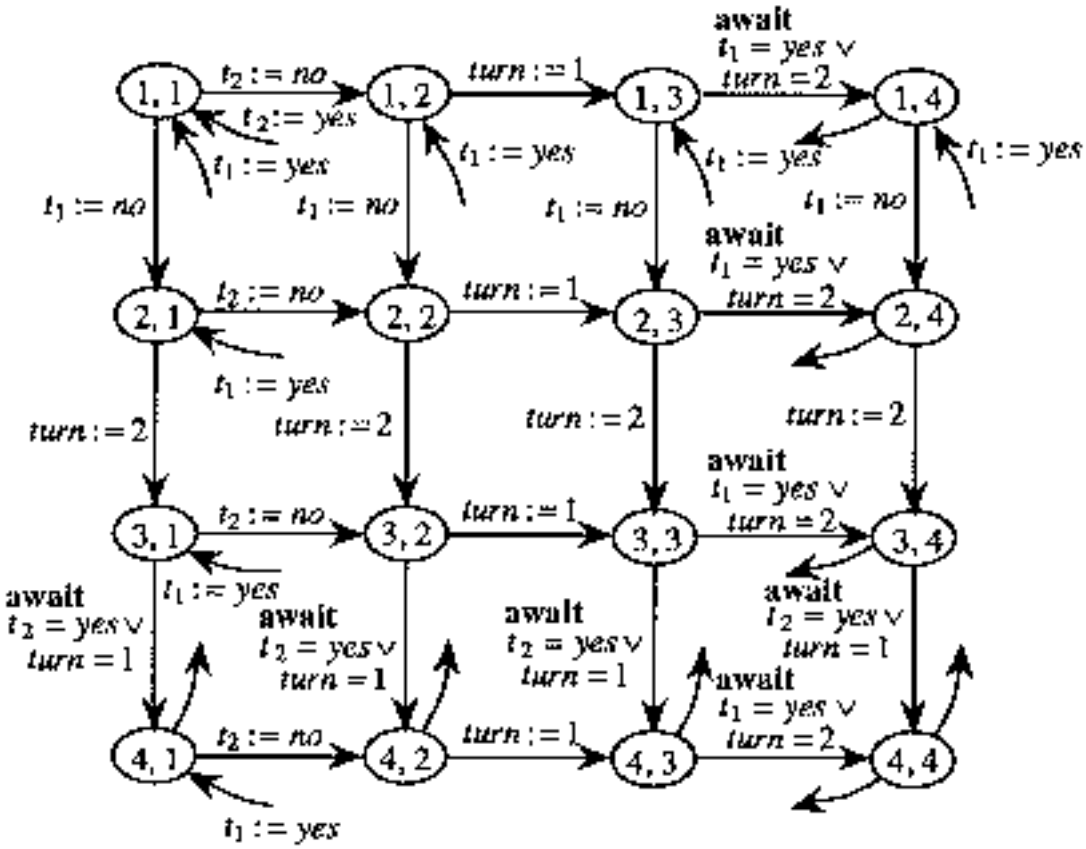

Fig. 6. Program graph

First we attach M given below and set $P \leftarrow P \wedge M$.

$$M = \begin{bmatrix} \mathbf{true} & \mathbf{true} & \mathbf{true} & \mathbf{true} \\ \mathbf{true} & \mathbf{true} & \mathbf{true} & \mathbf{true} \\ \mathbf{true} & \mathbf{true} & \mathbf{true} & \mathbf{true} \\ \mathbf{true} & \mathbf{true} & turn = 1 & \mathbf{true} \end{bmatrix}$$

The consistencies over edges $(4,2) \rightarrow (4,3)$ and $(3,3) \rightarrow (4,3)$ are verified. Next we choose M whose elements are all true except for $M_{34} = (turn = 2)$ and set $P \leftarrow P \wedge M$. The new P is similarly proved to be induced by (P_0, S). Finally, we choose M such that M_{ij} are **true** except for $M_{44} = \mathbf{false}$. Since the accumulated information tells us that $P_{43} = (t_1 = no \wedge t_2 = no \wedge turn = 1)$ and $P_{34} = (t_1 = no \wedge t_2 = no \wedge turn = 2)$, we can verify the consistencies over edges $(4,3) \rightarrow (4,4)$ and $(3,4) \rightarrow (4,4)$. Thus we have

$$\begin{aligned} P \;\; = \;\; & \begin{bmatrix} t_1 = yes \\ t_1 = no \\ t_1 = no \\ t_1 = no \end{bmatrix} \times \begin{bmatrix} t_2 = yes & t_2 = no & t_2 = no & t_2 = no \end{bmatrix} \\ \wedge \;\; & \begin{bmatrix} \mathbf{true} & \mathbf{true} & \mathbf{true} & \mathbf{true} \\ \mathbf{true} & \mathbf{true} & \mathbf{true} & \mathbf{true} \\ \mathbf{true} & \mathbf{true} & \mathbf{true} & turn = 2 \\ \mathbf{true} & \mathbf{true} & turn = 1 & \mathbf{false} \end{bmatrix}. \end{aligned}$$

Taking disjunctions over rows and columns of P yields $\boldsymbol{p}$ and $\boldsymbol{q}$, and hence the above form of P is the normal form. Although P is not the strongest assertion matrix induced by (P_0, S), P is strong enough to prove the desired condition of **false** at $(4,4)$, which means mutual exclusion. In this example, $\#(M) = 3$, and we worked on seven assertions.

5 Composite infinite loops

Infinite loops in parallel programs are harder to detect than in sequential programs. We give an obvious sufficient condition for infinite loops. The path from $\boldsymbol{\xi}_0$ to $\boldsymbol{\xi}$ is said to be admissible with respect to $\boldsymbol{a}_0$ if we can actually start from $\boldsymbol{x} = \boldsymbol{a}_0$ and traverse the path, performing the labels on the edges. If a label is a Boolean condition, performing the condition means that it is induced by the instances of the variables at the vertex. Let the resulting condition at $\boldsymbol{\xi}$ be $\boldsymbol{x} = \boldsymbol{a}$. If there is another admissible path from $\boldsymbol{\xi}$ to $\boldsymbol{\xi}$ with respect to $\boldsymbol{a}$ and the resulting condition is $\boldsymbol{x} = \boldsymbol{a}$, then there is an infinite loop from $\boldsymbol{\xi}$ to $\boldsymbol{\xi}$. If the infinite loop involves two or more processes, it is called a composite infinite loop or a racing.

Example 8

$\{\textbf{true}\}$
$S \equiv$ [1 : **while** $x > 0$ **do begin** $\|$ 1 : **while** $x < 10$ **do begin**
2 : $x := x - 1$; 2 : $x := x + 1$;
end **end**
3 : 3 :
]
$\{x = 0 \vee x = 10\}$.

The proof graph is given in Fig. 6. With initial condition $x = 5$, for example, we traverse the path (1,1) $\rightarrow$ (2,1) $\rightarrow$ (2,2) $\rightarrow$(1,2) $\rightarrow$ (1,1) and we have $x = 5$. In this program we can prove the partial correctness of $\{\textbf{true}\}S\{x = 0 \vee x = 10\}$ despite the infinite loop, whereas if a sequential program T has an infinite loop with precondition P we have $\{P\}T\{\textbf{false}\}$. Note that each process in this parallel program terminates if executed separately. We can prove that the computer comes to (3,3) with probability one under the regularity assumption given next.

Definition 8 If it is possible to follow edges $e_1, \cdots, e_k$ at vertex $\boldsymbol{\xi}$ after evaluation of $P_{\boldsymbol{\xi}}(\boldsymbol{x})$ with $\boldsymbol{x} = \boldsymbol{a}$, they are said to be compatible at $\boldsymbol{\xi}$ with respect to $\boldsymbol{a}$. Let the probability that such e_i is followed in this situation be $p(e_i)$. Note that $p(e_1)+\cdots+p(e_k) = 1$. If $p(e_i) > 0$ $(i = 1, \cdots, k)$ for compatible edges at $\boldsymbol{\xi}$ and these probabilities are independent from vertex to vertex and among visits to $\boldsymbol{\xi}$, then the computation of the given parallel program is said to be regular.

Note that the concept of regular computation is stronger than that of fair computation.

Theorem 8 *In a parallel program $S = [S_1 \parallel S_2]$ with pre-condition P_0, we assume that there is no synchronization primitives in S_1 or S_2. Let $P = (P_{\boldsymbol{\xi}})$ be the strongest assertion matrix induced by (P_0, S). If S_1 and S_2 terminate when they are run separately from any $\boldsymbol{\xi}$ with any $\boldsymbol{x}$ satisfying $P_{\boldsymbol{\xi}}(\boldsymbol{x})$ at $\boldsymbol{\xi}$, and the reachable state of $\boldsymbol{x}$ is finite, then the parallel program S terminates with precondition P_0 with probability one under regular computation.*

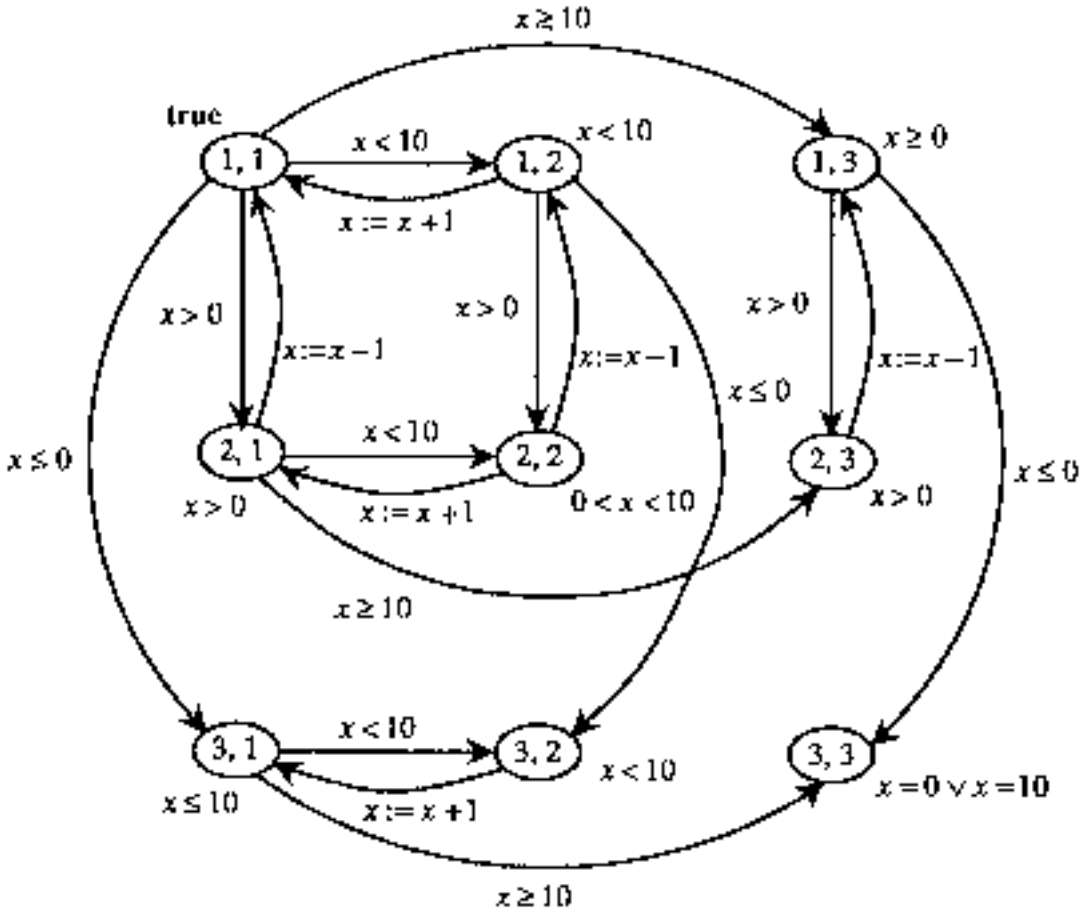

Fig. 7. Proof graph for Example 8

Proof. Without loss of generality, suppose there is a composite infinite loop ℓ. If we execute only S_1 in ℓ, we assume we have a loop ℓ_1 in S_1 and similarly we have ℓ_2 in S_2 by executing only S_2 in ℓ. Both S_1 and S_2 go out of loop after finite numbers of repetitions, say k_1 and k_2. We can set these finite k_1 and k_2 because the reachable state of $\boldsymbol{x}$ is finite and we can take the maximum numbers of repetitions starting at $\boldsymbol{\xi}$ with $\boldsymbol{x}$ satisfying $P_{\boldsymbol{\xi}}(\boldsymbol{x})$ for all possible $\boldsymbol{\xi}$ and $\boldsymbol{x}$. The probability that k_1 successive repetitions of S_1 or k_2 successive repetitions of S_2 do not occur, tends to zero in the long run of ℓ under regular computation. □

Example 7 clearly satisfies the conditions of Theorem 8. Note that the racing will stop much earlier than the above pessimistic observation. Racing that stops with probability one is not hazardous in a sense. Regular computation requires randomized scheduling of processes, which may be time consuming. Special hardware for scheduling may reduce the overhead time for scheduling. Based on the above observations, we have a new concept of correctness "eventual correctness" in the following.

Definition 9 For a parallel program, assume that the partial correctness $\{P\}S\{Q\}$ is proved. If we can prove that S terminates with probability one under regular computation, we say $\{P\}S\{Q\}$ is eventually correct.

Note that we have

$$\text{totally correct} \ < \ \text{eventually correct} < \ \text{partially correct},$$

where $a < b$ means a is stronger than b.

Excercise. Show the normal form of the assertion matrix of Example 8.

6 Concluding remarks

We only dealt with the partial and eventual correctness of parallel programs. There are many more technical issues in parallel programs such as semaphores, deadlocks, progressive property under a fairness assumption, etc. The technique in this paper with assertion matrices and directed graphs is useful for the analysis of these problems as well. To put the new concept of eventual correctness in practice, we need an efficient and reliable random number generator, which is still difficult and shares the same fate with randomization techniques in other areas of computer science. It is still worth while to investigate the properties of the new concept assuming the existence of good randomization techniques. The visual technique with directed graphs will also be suitable for interactive parallel program verification, where a human verifier can work on a directed graph on the display.

References

[1] Ashcroft, E.: Proving assertions about parallel programs, *JCSS*, Vol. 10, pp. 110–135 (1975).

[2] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S. and Steffens, E. F.: On-the-fly Garbage collection: An Exercise in Cooperation, *CACM*, Vol. 21, No. 11, pp. 966–975 (1978).

[3] Floyd, R. W.: Assigning meanings to programs, *Proc. Symp. on Appl. Math.*, Vol. 19, Amer. Math. Soc. pp. 19–32 (1967).

[4] Hoare, C. A. R.: An axiomatic semantics for computer programming, *CACM*, Vol. 12, No. 10, pp. 576–583 (1969).

[5] Hoare, C. A. R.: Communicating sequential processes, *CACM*, Vol. 21, No. 8, pp. 666–677 (1978).

[6] Igarashi, S., London, R. L. and Luckham, D. C.: Automatic program verification. I. A logical basis and its implementation, *Acta Inf.*, Vol. 4, No. 2, pp. 145–182 (1975).

[7] Joseph, T. A., Räuchle, T. and Toueg, S.: State machines and assertions: an integrated approach to modeling and verification of distributed systems, *Science of Computer Programming*, Vol. 7, No. 1, pp, 1–22 (1986).

[8] Keller, R. M.: Formal verification of parallel programs, *CACM*, Vol. 19, No. 7, pp. 371–384 (1976).

[9] Lamport, L.: Control predicates are better than dummy variables for reasoning about program control, *TOPLAS*, Vol. 10, No. 2, pp. 267–281 (1988).

[10] Manna, Z. and Pnueli, A.: Adequate proof principles for invariance and liveness properties of condurrent programs, *Science of Computer Programming*, Vol. 4, No. 3, pp. 257–289 (1984).

[11] Milner, R.: A calculus of communicating systems, *LNCS*, Vol. 92, Springer-Verlag (1980).

[12] Murata, T.: Petri nets: properties, analysis and applications, *Proc. IEEE*, Vol. 77, No. 4, pp. 541–580 (1989).

[13] Owicki, S. and Gries, D.: An axiomatic proof technique for parallel programs, *Acta Info.*, Vol. 6, No. 4, pp. 319–340 (1976).

[14] Paczkowski, P.: Ignoring nonessential interleavings in assertional reasoning on concurrent programs, Mathematical Foundations of Computer Science 1993, *LNCS*, Vol. 711, Springer-Verlag, pp. 598–607 (1993).

[15] Peterson, G. L.: Myths about the mutual exclusion, *Inf. Proc. Lett.*, Vol. 12, No. 3, pp. 115–116 (1981).

[16] Takaoka, T.: Parallel program verification with directed graphs, *Proc. 1994 ACM Symp. Appl. Comp.*, pp. 462–466 (1994).