Detecting Termination in Static and Dynamic Systems

Xinfeng Ye,
Department of Computer Science,
University of Auckland,
New Zealand.

John A. Keane,
Department of Computation,
UMIST,
Manchester, UK.

Abstract

Distributed termination detection concerns detecting the termination of a distributed computation spread across a set of processors. Most solutions to the problem are not intended for dynamic systems where processes can be created and destroyed during the computation. In this paper, a termination detection algorithm which can be applied to both static and dynamic systems is proposed. The scheme can be applied to any kind of connection topology. The number of control messages is lower than some previous approaches.

1 Introduction

In distributed systems where processes communicate by passing messages, the detection of the termination of a distributed computation is non-trivial because no process has complete knowledge of the global state. Many solutions have been proposed [2, 3, 4]. Apart from [2], most solutions are not intended for dynamic systems where processes can be created and destroyed during the computation.

This paper introduces a termination detection scheme which works for both static and dynamic systems. The scheme is based on message counting and can be applied to any connection topology. The communication complexity of the scheme is much lower than [2] which has the same system model.

2 System Model

A computation consists of a set of processes residing on different processors. Processes communicate by exchanging messages. Communication is asynchronous, and messages are delivered with arbitrary finite delay. Message delivery does not have the FIFO property. There are two message types: one, termed basic messages, is used by the processes to exchange information during computation; the other, termed control messages or probes, is used by the termination detection algorithm.

A computation is static if the set of processes which form the computation is fixed. The computation is dynamic if processes can be created and (or) destroyed during the computation. During computation, a process is either live or dead. A dead process is a process which has been destroyed. It cannot send or receive messages, nor can it create new processes. A live process is either active or passive. A process is active on its creation. An active process can (a) send and receive basic messages, and (b) create new processes. An active process becomes passive or dead when it finishes its computation. A passive process can be reactivated to become active when it receives a basic message.

No process can send messages to processes which have not (yet) been created. This assumption is reasonable since in practice the identity of a process is not known until the process is created. Messages sent to dead processes are regarded as undeliverable and discarded.

A computation has a set of initial processes. These processes are active when the computation starts. They activate and/or create other processes. It is assumed that the initial processes know the identity of each other. This assumption can be realised either by running a network learning algorithm before the computation starts or by assigning the identity information to each initial process when the initial processes are distributed to the system. It is not assumed that non-initial processes know the identities of other processes.

A computation terminates if (a) all processes have become passive or dead, and (b) there are no basic messages in transit which can reactivate passive processes. Given these conditions no processes can become active. Since dead processes cannot be reactivated, termination is not affected by basic messages sent to dead processes.

In order to detect termination, each process maintains some data to record the communication activities and status of the process. For each process p, a process p executes the termination detection algorithm on behalf of p. p and p share the data used for termination detection. The use of a separate set of processes to execute the termination detection algorithm is due to the existence of dead processes. Dead processes are destroyed, thus they cannot participate in executing the detection algorithm. Hence, a process is needed to act on behalf of the dead pro-
cess. As a result, computation can be regarded as consisting of two sets of processes: one, termed compute processes (CPs), is responsible for carrying out computation; the other, termed detector processes (DPs), detects termination. The processes in the two sets have a one-to-one correspondence. No DP can be destroyed until termination is declared. A pair of corresponding compute and detector processes reside on the same processor and communicate through shared data.

For simplicity, in the following discussion a pair of corresponding compute and detector processes is sometimes termed a process. It should be noted that when a process handles a basic message (or a probe), it means that the CP (or the corresponding DP) handles the basic message (or the probe).

The detection algorithm is initiated by the corresponding DPs of the initial processes. These DPs are termed initiators. They are also the initiators of the probes which they create during the execution of the detection algorithm.

3 The Termination Detection Scheme

The scheme is based on message counting to determine whether there are basic messages in transit. During computation, each process records the number of basic messages sent and received. Since only passive processes can be reactivated, if all basic messages sent to passive processes have been received, then no processes can be reactivated.

The initiators are responsible for detecting termination. When the initial processes complete their computation, the initiators send out probes to the processes where the initial processes have sent basic messages. The probes check (a) whether the basic messages have been received, and (b) whether the processes have completed their computation.

The active processes will hold the probes. When a passive process receives a probe, the process can determine whether there are basic messages on their way to the process from the value carried by the probe. If there are such messages, the process will be reactivated, thus, the process will keep hold of the probe. Otherwise, the process has two alternatives:

1. If the process has sent basic messages whilst active, then the probe is split (if necessary), and sent to the processes to which basic messages have been sent.

2. If the process has not sent any basic message since the last probe left (if any), the process sends the probe to the probe's initiator.

A dead process has the same alternatives when it receives a probe. The scheme guarantees that all probes are returned to their initiators eventually if and only if the computation has terminated. When all initiators receive all probes they have initiated, they declare the termination of the computation.

In order to allow the initiators to determine whether the probes initiated by them have all been returned, each probe carries a weight. The sum of the weights of the probes initiated by an initiator is fixed\(^1\). The fixed value is known by the initiator. Each initiator accumulates the weight of the probes returned to it. From the accumulated weight, an initiator can determine whether all its probes have been returned.

3.1 Data Structures

Probes are used to exchange information. Each probe is a tuple \((\text{initiator}, \text{weight}, \text{number})\). \text{initiator} identifies the initiator of the probe. \text{weight} denotes the weight carried by the probe. When a probe is passed between two processes, say from \(i\) to \(j\), \text{number} records the number of basic messages which have been sent from \(i\) to \(j\) since the last time that \(i\) sends a probe (if any) to \(j\). The three components of a probe, say \(pb\), are denoted as \(pb[\text{initiator}], pb[\text{weight}]\) and \(pb[\text{number}]\) respectively.

Each process has variables \text{out} (a set) and \text{in} (an integer). Each element in \text{out} of a process, say \(p\), is a pair \((\text{pid}, \text{num})\) where \(\text{num}\) is the number of the basic messages sent from \(p\) to process \(pid\). \text{in} of a process, say \(p\), records the difference between the number of basic messages received by \(p\) and the number sent to \(p\) from other processes. From the received probes, \(p\) determines the number of basic messages which have been sent to it. When a process is created, \text{out} and \text{in} are initialised to \text{empty} and \text{zero} respectively.

\text{dead} and \text{passive} are Boolean-variables that record the status of the CPs. \text{dead} is set to \text{true} when a CP finishes and is destroyed. \text{passive} is set to \text{true} when a CP finishes and becomes passive. Compute and detector processes share the variables \text{dead}, \text{passive}, \text{out} and \text{in}.

A process may receive probes from different initiators. Each process has a variable \text{engager} indicating the identity of the initiator whose probe is being held by the process. Each process also has a variable \text{engaged} indicating whether the process is holding a probe.

A process might receive more than one probe created by the same initiator. Such probes will be merged. Therefore, a variable, \text{weight}, is used to accumulate the weight carried by the probes initiated by the same initiator.

Each initiator has a variable \text{initiated} indicating whether the initiator has ever created any probes. \text{initiated} is initialised to \text{false}. An initiator might create several rounds of probes. Thus, \text{e-weight} is used to record the sum of the weight of the probes which have been created by the initiator. The sum of the weight of each round of the probes is equal to the sum of weights of all the previous rounds of probes. This value is denoted as \text{f-weight}.

Each initiator maintains a set \text{completed}. The identifier of an initiator is added to the \text{completed} set of all the initiators if all the probes created by the initiator have been returned to the initiator. In the following, \text{init} is used to denote the set which records the identifiers of all the initiators in the computation.

\(^1\)When a probe is split, the sum of the weights of the new probes is equal to the weight of the probe which is split.
3.2 The Rules of the Scheme

The following rules are used by the CPs. They maintain the status variables, dead and passive, and record the sending and receiving of basic messages.

1. when send a basic message to process i or create a new process i:
   - out = out ∪ \{(i, 1)\}
   - if ¬∃(id, num) : out. id = i then
     - out = out ∪ \{(id, num + 1)\}
   - endif

2. when receive a basic message or is created:
   - in = in + 1

3. when become dead: dead ← true

4. when become passive: passive ← true

5. when is activated/reactivated or is created:
   - passive ← false

When a process sends a basic message, out is updated. If the process has not recorded the sending of a basic message to process i, a new item for i is added to out (line 3). Otherwise, the number of basic messages is incremented and the old item (id, num) is removed before the new item (id, num + 1) is added to out (line 5).

The elements in out guide how the probes are passed (described later). To make the probes reach all the processes, when a new process is created it is assumed that the creator of the new process has sent a basic message to the new process (line 3). Also, the new process is regarded as having received a basic message from its creator on its creation (line 7).

The following rules are used by the DP's to handle the probes. pid represents the identifier of the process executing the rules. The rules ensure that (a) the sum of the weight of the probes is equal to the values that are known to the initiators of the probes, and (b) the probes are sent back to their initiators when the processes become passive or dead.

11. when receive a probe probe:
   - in = in + probe[number]
   - if engaged then
     - if (pid = i) or (pid ∈ init ∧ initiated) then
       - weight ← probe[weight]
     - end
   - end
   - engaged ← true
   - if (pid ∈ init) ∧ ¬(initiated) /
     - send (probe[initiator], probe[weight], n) as a probe to process probe[initiator]
   - end
   - engaged ← false
   - if (pid ∈ init) and ¬engaged then
     - if probe[initiator] = engaged then
       - weight ← weight + probe[weight]
     - else probe[initiator] ≠ engaged then
       - send (probe[initiator], probe[weight], n) as a probe to process probe[initiator]
     - end
   - end
   - if ∃(i, num) : out. i = probe[initiator] then
     - out = out ∪ \{(i, num)\}
   - else
     - out = out ∪ \{(id, num + 1)\}
   - endif

29. where n =
   \[
   \begin{cases}
   \text{num} & \text{if } \exists(i, num) : \text{out. } i = \text{probe[initiator]} \\
   0 & \text{if } \neg\exists(i, num) : \text{out. } i = \text{probe[initiator]}
   \end{cases}
   \]
30. endif
31. endif
32. when ((passive ∧ in ≥ 0) ∨ dead) ∧ engaged:
   - if out = \emptyset then
     - if engaged = pid then
       - send probe (pid, weight, num) to process i for each (i, num) in out
     - else
       - send probe (engager, weight, num) to process i for each (i, num) in out
     - endif
   - endif

37. e-weight ← e-weight + f-weight
38. else /
39. send probe (engager, weight, 0) to process engager
40. engaged ← false
41. endif
42. endif
43. else /
44. out = \emptyset
45. endif
46. if engaged = pid then
47. send probe (engager, weight, 0) to process engager
48. engaged ← false
49. endif

probe[number] records the number of basic messages sent to the process which receives the probe. Thus, when a process receives a probe, in is decreased by probe[number] (line 12).

When a process receives a probe, if the process is not holding a probe (line 13), the process updates the status variables to indicate that the probe has been accepted (line 16 and 17). In the meantime, weight is assigned value according to the probe (line 15). However, if the process which receives the probe is an initiator and it has never generated any probe, the received probe is sent back to the probe's initiator (line 19). This is because the process will initiate probes later; thus, there is no need for the process to hold a probe. If probe probe is sent back to its initiator, probe[number] is set equal to the number of the basic messages sent to the probe's initiator, and out is updated accordingly (line 20 and 21).

If a process receives a probe, say probe, whose initiator is different from the probe already held by the process (lines 23 and 26), then probe is sent back to its initiator (line 27). This is because there is no need for more than one initiator to check the status of the same process repeatedly. If a process receives probes created by the same initiator (lines 23 and 24), the probes are merged by adding their weight together (line 25).

Probes are used to check whether processes have become passive or dead. Thus, a probe should not leave an active process until the process becomes passive or dead. For a passive process which is holding a probe, the probe sends probes to other processes when in ≥ 0 holds (line 32). This is because,
according to the definition of $m$ in §3.1, if $m < 0$
holds, then the process will receive some basic mes-
sages. As a result, the passive process will be reac-
tivated, thus, there is no need to send out the probe
now. For a dead process which is holding a probe, the
process sends out the probe regardless of the value of $m$
(line 32), since a dead process cannot be reactivated.

If $out = \emptyset$ holds, the process sent basic messages
whilst it was active (lines 33 and 1). Thus, probes are
sent to the processes where these messages have
been sent to check if they have been received. There
are two cases to consider:

1. The process is an initiator waiting for the return
of the probes initiated by itself (line 34).

   If none of the previously initiated probes have
   been returned to the initiator, then no probes can
   be split. Thus, instead of waiting for the return
   of a probe, the process sends out a new round
   of probes and sends them out (line 35). Since
   the sum of the weight of the newly generated
   probes is $f$-weight (line 36), the total weight
   of the probes initiated by the initiator is in-
   creased by $f$-weight (line 37).

2. The process is not an initiator (line 38).

   The probe is split and sent to the appropriate
   processes (line 39). Thus, $engaged$ is set to false
   (line 41) to indicate that the process no longer
   holds any probe.

In both cases, $out$ must be set to empty after the
probes are sent out (line 43); so that, if the process
sends out probes again in the future, probes will not
be sent to check the receipt of the same basic mes-

If $out = \emptyset$ holds (line 44), no basic messages have
been sent to other processes whilst the process is
active. If the process is the initiator of the probe,
then nothing need be done; otherwise, the probe is
returned to its initiator (lines 45 and 46).

The following rules are used by the initiators. They
govern probe generation and termination testing.

$p$ is the identifier of the process executing the rule.

50. when ((passive $\land$ $m \geq 0$) $\lor$ dead) $\land$ $\neg$initiated:

51. initiated $\leftarrow$ $true$

52. if $out \neq \emptyset$ then

53. send probe $(p, weight, num)$ to

54. process $i$ for each $(i, num)$ in $out$

55. $e$-weight $\leftarrow$ $f$-weight

56. $out$ $\leftarrow$ $\emptyset$

57. $engager$ $\leftarrow$ $pid$

58. $engaged$ $\leftarrow$ $true$

59. else /* $out = \emptyset$ */

60. $weight$ $\leftarrow$ $e$-weight

61. endif

62. when $(engager$ $= pid)$ $\land$ $engaged$

63. send $finished(i)$ to each $i$ in init

64. $engaged$ $\leftarrow$ false

65. when receive $finished(i)$:

66. when completed $= init$ declare termination

If an initiator has never generated a probe, the
initiator generates probes when it becomes dead or
when it becomes passive and $m \geq 0$ (line 50). This
is because either it will not process the basic messages
or no expected basic messages are on their way to
reactivate the process. The generated probes are
sent to the processes where the basic messages have
been sent (line 53). The initiator is informed that
the sum of the weight of the probes is $f$-weight (lines
54 and 55). After the probes are sent, $out$ is set to
empty (line 56) for the same reason as at line 43.
$engager$ and $engaged$ are set to indicate that the
initiator is involved in detection processing (lines 57
and 58). If no basic messages have been sent during
the computation (line 59), then no probes need to be
sent out. This is equivalent to the initiator having
received all the probes initiated by it (line 60).

If an initiator is waiting to receive its probes and
$e$-weight $=$ weight holds (line 62), it means that
the initiator has received all its probes. Therefore,
it has completed the detection processing initiated
by itself; and it can inform all the other initiators
(line 63). Since the execution has been completed,
$engaged$ is set to $false$ (line 64).

When a $finished(i)$ message is received from $i$, it
means that $i$ has completed its detection processing.
Thus, $i$ is added to completed (line 65). It should be
noted that, even if an initiator, say $p$, has completed
the detection processing initiated by itself, $p$ can still
be involved in the detection processing initiated by
other initiators (line 14). Although each process will
be reached by the probes of at least one initiator,
the probes of an initiator might not reach all the
processes in the system. Thus, to guarantee that
all the processes have been checked, termination can
only be declared when all initiators have completed
their detection processing (line 66).

3.3 Complexity

The complexity of a termination detection algo-

rithm concerns the upper bound on the number of
control messages sent in the worst case.

1. Probes are only sent to the processes where ba-
sic messages have been sent (lines 35, 39 and
53). This would generate at most $m$ probes
where $m$ is the number of basic messages which
are sent during the computation.

2. If a process has not sent any basic message to
other processes since the last probe was sent or
there is no need to hold a probe, the process will
send the probe back to the initiator of the probe
(lines 46, 19 and 27). Since a probe can only
reach a process if the process receives at least
one basic message, at most $m$ probes are sent
from processes to the initiators of the probes.

3. Each initiator sends messages to all other ini-

tiators when it completes its detection process.
This generates at most \(k(k-1)\) messages where \(k\) is the number of initiators.

The upper bound is the sum of the above three cases. That is, \(2m + k(k-1)\).

### 3.4 Correctness Proof

According to [1], a scheme is correct if (a) it does not deadlock, and (b) it declares termination if and only if the computation has terminated. The theorems in this section show that the scheme in this paper satisfies these criteria.

**Lemma:** For any compute process (CP) \(p\), when \(p\) changes from active to passive or dead, its corresponding detector process (DP) \(\hat{p}\) must be reached by at least one probe.

**Proof:** Let \(\hat{S}\) denote the set of DPs which can be reached by at least one probe when their corresponding CPs change from active to passive or dead. Let \(\hat{T}\) denote the set of DPs which cannot be reached by any probe when their corresponding CPs change from active to passive or dead. \(S\) and \(T\) denote the sets of the corresponding CPs of \(\hat{S}\) and \(\hat{T}\).

According to line 50, initially \(\hat{S}\) includes all the initiators which have not generated any probes. Therefore, \(\hat{S}\) is not empty initially. Also from line 50, if \(\hat{T}\) includes some initiators, then these initiators must have initiated probes. From lines 35, 39 and 53, probes are sent to the processes where basic messages have been sent. Therefore, the processes in \(\hat{T}\) must not receive any basic message from the processes in \(\hat{S}\). Otherwise, the processes in \(\hat{T}\) will receive probes from the processes in \(\hat{S}\). Thus, basic messages received by processes in \(\hat{T}\) must be sent by processes in \(\hat{S}\).

When computation begins, only initial processes are active. The other CPs, including the initial processes which have become passive, can only be activated or reactivated through receiving basic messages. Since \(\hat{T}\) does not include any initial processes which have always been active (line 50), all the processes in \(\hat{T}\) can only be activated by receiving basic messages.

**Assumption 1:** Assume that process \(p\) is the first one in \(\hat{T}\) to be activated.

Since the processes in \(\hat{T}\) do not receive basic messages from the processes in \(\hat{S}\), \(p\) must be activated by receiving a basic message from a process \(q\) in \(\hat{T}\). However, since only active processes can send basic messages, \(q\) must be activated before \(p\). This contradicts **Assumption 1**.

Therefore, the processes in \(\hat{T}\) cannot be activated without receiving a basic message from the processes in \(\hat{S}\). Hence, some of the processes in \(\hat{T}\) must be activated by receiving basic messages from the processes in \(\hat{S}\); and, according to lines 35, 39 and 53, the corresponding DPs of these processes in \(\hat{T}\) will be reached by some probes. Therefore, these processes in \(\hat{T}\) should be in \(\hat{S}\) and their corresponding DPs should be in \(\hat{S}\).

A similar argument can be applied to the processes left in \(\hat{T}\) (\(\hat{T}\)) until all of them are included in \(\hat{S}\) (\(\hat{S}\)).

Hence, \(\hat{T}\) (\(\hat{T}\)) is an empty set and \(\hat{S}\) (\(\hat{S}\)) includes all the processes in the computation. Therefore, the lemma holds.

**Theorem 1:** All probes will be returned to their initiators eventually.

**Proof:** The computation will terminate eventually, i.e., all processes will become either passive or dead. From lines 32 and 50, dead processes do not hold any probes, because the probes are always passed on.

If a passive process holds a probe, according to lines 32 and 50, the probe is passed on when \(m \geq 0\). Thus, if a process retains a probe, then \(m < 0\) must hold. From \(\S\), \(m = rec - p\text{-number}\) where \(rec\) is the number of basic messages received by the process, and \(p\text{-number}\) is the number of basic messages sent to the process and that have been recorded in the probes received by the process. As the transmission delay is finite, all basic messages will reach their destinations. Therefore, \(rec \geq p\text{-number}\) will hold eventually. In other words, \(m \geq 0\) will hold eventually. Hence, it is impossible that \(m < 0\) holds for a process forever. Thus, no passive processes can hold the probes forever.

According to the **Lemma**, a process receives at least one probe after the process becomes passive or dead. From the above, the received probe will be passed on. From lines 43 and 56, \(out\) is set to empty when the probe leaves. \(out\) remains empty while the process does not send out basic messages. Thus, \(out\) remains empty when all the processes become passive or dead. If \(out = \emptyset\), probes are sent to their initiators (line 46). Hence, all the probes are returned to their initiators. Thus, theorem 1 holds. 

According to the proof of theorem 1, the following can be obtained:

**Corollary 1:** When the computation terminates, all the probes will be returned to their initiators eventually.

**Theorem 2:** If the computation terminates, \(completed = init\) holds for all the initiators eventually.

**Proof:** According to lines 37 and 55, the sum of the weight of the probes which are labelled as being initiated by \(p\) is always equal to \(e\text{-weight}\). Thus, from lines 25 and 60, \(e\text{-weight} = \text{weight}\) must hold when all the probes initiated by \(p\) are returned to \(p\).

Hence, from lines 62, 63 and 65, \(\overline{p}\) will be included in the \(completed\) set of all the initiators.

From **Corollary 1**, when the computation terminates, all the probes will be returned to their initiators. Thus, like \(\overline{p}\) all other initiators will also be included in the \(completed\) set. As a result, \(completed = init\) holds. Hence, the theorem holds.

From the proof of theorem 2, it can be seen that an initiator is included in the \(completed\) set if all the probes generated by the initiator have been returned to the initiator. Hence, the following can be obtained:

**Corollary 2:** \(completed = init\) holds for an initiator if and only if all the probes have been returned to their initiators.
The scheme terminates when \textit{completed} = \textit{init} holds for an initiator (line 66). Hence, from Theorem 1 and Corollary 2, the scheme will terminate, i.e., the scheme does not deadlock. Thus, the first correctness criterion is satisfied.

**Theorem 3**: If \textit{completed} = \textit{init} holds for an initiator, then (a) \textit{in} = 0 \land \textit{out} = \emptyset holds for every passive process, (b) \textit{out} = \emptyset holds for all the dead processes, and (c) all the processes are passive or dead.

**Proof**: Proof by contradiction.

- **Step 1**: Assumption 1: \textit{out} \neq \emptyset holds for process \textit{p}.
  
  Since only active processes can send basic messages, \textit{out} can only become non-empty when \textit{p} is active. Thus, according to the Lemma, \( \hat{p} \) (i.e., the corresponding DP of \textit{p}) must be reached by at least one probe after \textit{p} becomes passive or dead. Since \textit{completed} = \textit{init} holds, according to Corollary 2, all the probes have been returned to their initiators. Thus, \( \hat{p} \) must have sent out the probe. According to lines 43 and 56, \textit{out} is set to \emptyset when a probe leaves a process. Thus, assumption 1 is wrong.

- **Step 2**: Assumption 2: There is an active process \textit{p}.
  
  According to the Lemma, the corresponding DP of \textit{p} will be reached by at least one probe when \textit{p} becomes passive or dead. Since \textit{completed} = \textit{init} holds, from Corollary 2, all probes have been returned to their initiators. Thus, \textit{completed} = \textit{init} holds, all the initiators have completed their detection processing (lines 63 and 65). Thus, no new probes can be created. Hence, no probes can be sent to \textit{p}. This contradicts the Lemma, thus, assumption 2 is wrong.

- **Step 3**: Assumption 3: \textit{in} \neq \emptyset holds for a passive process, say \textit{p}.
  
  There are two cases to consider.

  - **Case 1**: \textit{in} < 0 holds for \textit{p}.
    
    From the definition of \textit{in} in §3.1, if \textit{in} < 0, \textit{p} will receive some basic messages, thus \textit{p} will be reactivated. Hence, the situation will be the same as assumption 2. Also, assumption 2 is wrong. Hence, if \textit{in} \neq \emptyset holds, \textit{in} < 0 must not hold.

  - **Case 2**: \textit{in} > 0 holds for \textit{p}.
    
    From step 1, it can be seen that \textit{out} = \emptyset holds for all processes, i.e., all processes which have sent basic messages to \textit{p} must have recorded the number of basic messages being sent to \textit{p} in the probes sent to \textit{p} (lines 35 and 43, etc.). From the definition of \textit{in}, \textit{in} \leq 0 holds when all probes sent to \textit{p} have been received by \textit{p}. Thus, if \textit{in} > 0 holds, some probes must be on their way to \textit{p}. However, since \textit{completed} = \textit{init} holds, from Corollary 2, there are no probes in transit. Hence, \textit{in} > 0 cannot hold.

    From the above discussion, it can be seen that assumption 3 is also wrong. Thus, theorem 3 holds. \( \square \)

From case 1 of step 3 in the proof, the following corollary can be obtained.

**Corollary 3**: If \textit{completed} = \textit{init} holds for an initiator, then there are no basic messages which can activate the processes in the network.

From line 66, when \textit{completed} = \textit{init} holds for an initiator, the scheme declares termination. Thus, Theorem 3 and Corollary 3 mean that the scheme only declares termination when the computation has really terminated. Hence, Theorem 2, Theorem 3 and Corollary 3 mean that the scheme declares termination if and only if the computation has terminated. Thus, the second correctness criterion is satisfied.

4 Conclusions

The scheme in this paper uses message counting to cope with non-FIFO message transmission. The upper bound of the control messages is the same as in [3]. In practice the number of probes sent by the scheme in this paper should be lower because in [3] if process \textit{i} sends a basic message to \textit{j}, then two probes are exchanged between \textit{i} and \textit{j}. The scheme in this paper only requires one probe to be sent from \textit{i} to \textit{j}. In [3], during the detection execution, a tree is formed. A process in the tree cannot start a new round of test signals until it has received echoes from all subtrees rooted at the process. Thus, concurrency is reduced. In contrast, the scheme here has no dependencies between the sending and receiving of probes.

The communication complexity of the scheme in [2] is \( O(mn) \), while the scheme in this paper has an upper bound \( 2m + k(k - 1) \) (\( m \) is the number of basic messages, \( n \) the number of processes, and \( k \) the number of initiators). In practice, \( k \) is normally far less than \( n \), and \( m \) should be far greater than \( n \). Thus, the complexity of the scheme here is lower than in [2]. The size of the probes in this paper is reasonably small and fixed. In contrast, the size of the control messages in [2] is proportional to the number of processes in a computation.

In this paper, for each compute process a detector process is set up to perform detection, thus, permitting a dynamic computation. For a dynamic computation, the compute process can be destroyed and its resources reclaimed. As compute processes generally use much more resource than detector processes, the flexibility of the system is increased.

References


