A Wait-Free Probe Algorithm for Distributed Deadlock Detection and Resolution

Young Man Kim, Ten H. Lai, Neelam Soundararajan

Department of Computer and Information Science
2015 Neil Avenue Mall
Ohio State University
Columbus, Ohio 43210-1277

Email: lai, neelam@cis.ohio-state.edu

January 25, 1997

Abstract

Edge-chasing is the basis of many deadlock detection algorithms. This method detects a deadlock by propagating special messages called *probes* along dependency edges. When the initiator of a probe receives the probe back, it knows the existence of a deadlock. Once a deadlock is detected, a special message called *token* is sent to clean up those probes in the deadlock cycle which, if not removed, may later lead to phantom deadlock detections. Only after the token has traversed the entire deadlock cycle and returned to its initiator, the deadlock is resolved by aborting a so-called victim in the deadlock cycle. In a deadlock, all involved transactions are held waiting and all involved resources are locked up. It is thus desirable to resolve a deadlock as soon as it is detected, without waiting for the token message to go around the deadlock cycle. This paper proposes an algorithm that achieves this and thereby reduces the average deadlock persistence time by as much as two thirds.

Index Terms — Deadlock detection and resolution, distributed algorithm, correctness proof.

Contents

1 Introduction

Deadlock detection is an important problem in distributed database systems, and many solutions have been proposed in the research community [8, 18]. Schemes for deadlock detection could be centralized or distributed. One of the most popular methods for distributed deadlock detection is edge-chasing, which has been the basis of many deadlock detection algorithms (e.g., [1, 2, 3, 9, 13, 14, 15, 17, 19]). Originally proposed by Chandy, Misra and Hass [1, 2], an edge-chasing algorithm detects a deadlock by propagating special messages called *probes* along dependency edges. When the initiator of a probe receives the probe back, it knows the existence of a deadlock. Sinha and Natarajan [19] proposed the use of priorities to reduce the number of probe messages. They presented a basic algorithm for deadlock detection and resolution for the single-request model and an extended version for the multiple-request model. These algorithms were not implemented, nor was their correctness formally established. Choudhary et. al. [3] pointed out some errors in the basic algorithm of [19], and they proposed some modifications for correction. The resulting algorithm was actually implemented and extensively tested through simulation. However, when the algorithm was published in [3], one line was inadvertently omitted from the pseudocode [4, 16]. Kshemkalyani and Singhal [9] proposed some modifications to this algorithm, and offered a correctness proof.

These algorithms [3, 9, 19] all include a scheme for deadlock resolution. After a deadlock is detected, a special message called *token* is sent to visit every node in the detected deadlock. Its mission is to clean up those probes in the deadlock cycle that, if not removed, would become obsolete and lead to possible phantom deadlock detections, in which the algorithm claims to have detected a deadlock while there is actually none. Only after the token has traversed the entire deadlock cycle and returned to its initiator, the deadlock is resolved by aborting a so-called victim in the deadlock cycle.

The simulation study of [3] observed an average deadlock size of ten. The need to wait for a token to traverse a detected deadlock cycle considerably prolongs the average deadlock persistence time (i.e., the time from the formation of a deadlock to its resolution) and increases the average response time of a request. When a deadlock occurs, all involved transactions are held waiting, and all involved resources are locked up. It is desirable to resolve the deadlock as soon as it is detected, and not to stay idle and wait for a token message to go around the deadlock cycle. The algorithm in [14] achieves immediate abortion by allowing phantom deadlock detection and thus unnecessary abortion of victims. (The main contribution of [14] is however not in achieving immediate abortion, but in providing an efficient algorithm for the extended model where a node may have multiple outgoing edges.)

In this paper, we propose an edge-chasing algorithm that resolves a detected deadlock right away, without waiting for the token to return back to its initiator. This reduces the average deadlock persistence time of previous edge-chasing algorithms by as much as two thirds. Our algorithm has the two properties normally required of a deadlock detection algorithm: the *safety property* that ensures that no phantom deadlock will be detected, and the *progress property* that guarantees that every deadlock will eventually be detected and resolved.

Our algorithm is based on a simple strategy of setting up a barrier to prevent obsolete probes from propagation. While simple in concept, the resulting algorithm is considerably harder to prove correct than those that wait for the token to return. However, a proof of correctness is a must, in view that so many published distributed algorithms without a correctness proof turn out incorrect, as has been observed in [8, 18]. Elmagarmid et. al. [5] have employed a CSP axiomatic proof system [20] to prove the correctness of their algorithm. This approach is very formal, but apt to become tedious. In this paper, we adopt the assertional proof style of [10]. It is about the same level of rigorousness as the correctness proof of [9].

This paper is organized as follows. Section 2 describes a distributed database system and an abstract model of it. Section 3 presents our no-wait algorithm for deadlock detection and resolution, which is then proved correct in Section 4, with a brief performance analysis in Section 5.

2 Model of a Resource-Sharing Distributed Database System

2.1 Distributed Database System

A *distributed database system* consists of a collection of nodes (or sites) connected by a communication network. Each node has many *data items*, each managed by a *data manager* which has the exclusive right to operate on the data item. It is not necessary for a data manager to manage only one data item. However, for ease of understanding the algorithm, we assume a one-one correspondence between data managers and data items.

Transactions use the two-phase locking protocol [6] to access data items. To lock a data item, a transaction sends a *data-lock request* to the manager of that data item. A data manager honors a request if the data item is free (not locked by any other transaction); otherwise it places the request in a queue. A transaction whose request on a data item has been granted is called the *holder* of that data item; a transaction waiting in the request queue is called a *requester*. When the holder of a data item unlocks the data item (with a *data-release* message to the data manager), the data manager grants a request in the waiting queue (by sending a *data-grant* message to the transaction of that request). As in [3, 9], a data item can be locked by at most one transaction.

A transaction can be in one of two states: *active* or *wait*. Once a transaction sends a lock request to a data manager, it enters the wait state. A transaction changes its state from wait to active when its request is granted. A transaction with a pending request cannot make another request (on the same or another data item) until the present one is granted. Thus, while a transaction may be the holder of more than one data item, it can be the requester of at most one data item. When a transaction finishes its task, it releases all data items which it has locked, by sending *data-release* messages to data managers.

2.2 Model for Distributed Database System

For our purpose, we model a distributed database system as a set of *permanent* and *temporary* processes; each permanent process represents a data manager, and each temporary process a transaction. The set of permanent processes is fixed; its membership does not change over time. The set of temporary processes varies as new transactions enter the system and old ones leave. Each process is assigned a unique integer as identifier.

Assumption 1 *The identifier of each permanent process is larger than that of each temporary process.*

Processes communicate with two types of messages: the *connect* message "C" and the *disconnect* message "D". A "C" message introduces a dependency relationship between a temporary process and a permanent process, while a "D" message terminates an existing relationship. A dependency relationship is represented by a directed edge between two processes. An edge from a temporary process (transaction) u to a permanent process (data manager) v represents a lock request from u to v; while an edge from v to u represents a granted request (or a lock) by u. For any process x, there is at most one outgoing dependency edge, and the other end of the edge, if it exists, is denoted by OUT_x. On the other hand, x may have more than one incoming dependency edge; the other ends of these edges are recorded as a set IN_x.

The behavior of each process in the system is modeled by six actions, denoted as $A_i, 1 \leq i \leq 6$. The first three actions A_1-A_3 can be activated only by a temporary process and the fourth action A_4 may be taken only by a permanent process. The remaining two actions apply to both types of processes. The message channel from process u to process v is denoted by $c[u,v]$. The following explains the function of each action.

1. A_1: **Data Request.** If a temporary process v has no outgoing edge (i.e., $OUT_v = null$), it may select a permanent process w and tries to establish a dependency edge to w by sending a "C" message to w. In doing so, it sets OUT_v to w. (This models the action that a transaction issues a data-lock request to a data manager.)

2. A_2: **Data Release.** If a temporary process v has no outgoing edge, it may select an existing incoming edge, say from process u, and disconnect it by sending a "D" message to u and, at the same time, deleting u from IN_v. (This models that the holder of a data item releases the data.)

3. A_3: **Process Termination.** If a temporary process does not have any dependency edge incident upon it, it is free to terminate itself. A terminated process no longer exists in the system. (This models the completion of a transaction.)

4. A_4: **Data Grant.** If a permanent process v has no outgoing edge, it may select an incoming edge, say from u, and reverse the direction of the edge by sending to u a combined "C+D" message. Both IN_v and OUT_v are updated accordingly. (A data-grant message is modeled as two messages of existing types, thereby eliminating the necessity of introducing another type of message.)

5. A_5: **Incoming Edge Connection.** If a process v receives a "C" message from a process u, then u is added to IN_v.

6. A_6: **Outgoing Edge Disconnection.** If process v receives a "D" message from process OUT_v, then v disconnects its outgoing edge by nullifying OUT_v.

When a sequence of processes connected by dependency edges forms cycle, a deadlock occurs. This is formally defined in the following.

2.3 Problem Definition

Definition 1 Let $X = (x_1, \ldots, x_k)$ be a sequence of processes which are all distinct except for $x_1 = x_k$, with x_1 being smallest among all the x_i's. X is said to be a *deadlock* if and only if the following predicate is true:

$$\begin{aligned} deadlock(X) \;\; \equiv \;\; & \forall i, 1 \le i \le k-1 : \{(OUT_{x_i} = x_{i+1}) \wedge (x_i \text{ is not in the middle of any action}) \\ & \wedge(\text{there is no "}D\text{" message in channel } c[x_{i+1}, x_i])\} \end{aligned}$$

Definition 2 The problem of *deadlock detection and resolution* is to design an algorithm that, when working together with the underlying distributed database system, satisfies the following conditions.

Progress Condition: If there exists a deadlock X in the system, the deadlock is eventually detected by some process.

Safety Condition: If a process v claims to have detected a deadlock X, then $deadlock(X)$ is true.

Resolution Condition: Once a deadlock X is detected, a temporary process on the deadlock cycle takes some action to make $deadlock(X)$ false.

2.4 Probe-Based Algorithms

The probe-based or edge-chasing algorithms [1, 2, 3, 9, 13, 14, 15, 17, 19] for deadlock detection are based on a simple idea: each process creates a probe that propagates along dependency edges; should a probe return to its initiator, a deadlock is detected. For instance, in Fig. 1(a), if the probe p_d initiated by process d travels along (d, e, f, a, b, c) and returns back to d, the deadlock cycle (d, e, f, a, b, c, d) is detected.

For technical reasons, when a probe passes through a node, the node must keep a record of this probe, so that later on if the outgoing edge of this node changes, a copy of the probe in question can propagate along the new edge. For instance, in Fig. 1(a), if the probe p_w initiated by w has traveled to f, then each of the nodes w, x, d, e, f has a copy of p_w. If the value of OUT_x changes, then the p_w at x is ready to move to the new node pointed to by OUT_x. Each copy of p_w is active in the sense that it can propagate across a new outgoing edge.

Once a deadlock is detected, a widely used method for resolving the deadlock is to abort some process involved in the deadlock. In the above example, suppose d is the victim to be aborted. In previous algorithms [3, 9, 19], before d gets aborted, it has to send a special message called *token* to visit every node in the detected deadlock. Only after the token returns back to d, the process gets aborted. The purpose of this token is to clean up those probes in the deadlock cycle which have passed through d. These probes, called *obsolete* probes, must be removed or they may later lead to detection of phantom deadlocks. For instance, suppose d gets aborted without removing the p_w at f. After the abortion, edge (c, d), (x, d) and (d, e) are disconnected. (See Fig. 1(b).) Should c make the connection (c, w), and p_w continues to propagate to a, b, c, and finally to w, then process w will mistakenly think it has detected a deadlock and accordingly abort itself — the "deadlock" that w thinks it has detected is called a *phantom deadlock* since it actually does not exist.

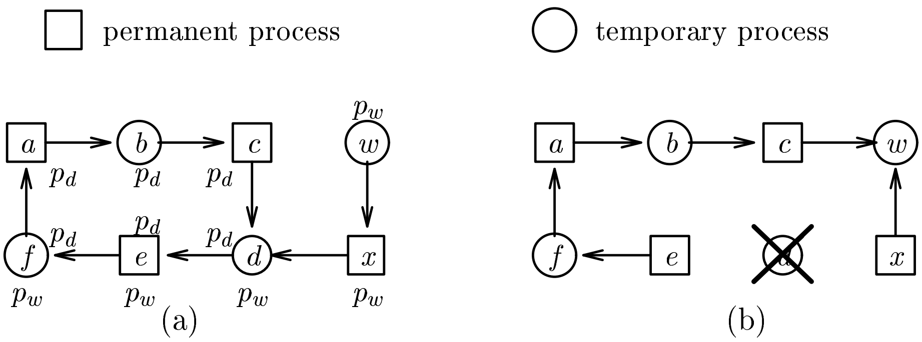

Figure 1: (a) Deadlock (d, e, f, a, b, c, d); (b) d gets aborted.

3 Algorithm for Deadlock Detection and Resolution

This section describes the proposed algorithm for deadlock detection and resolution. We give an overview of the algorithm in section 3.1; describe the algorithm in detail in 3.2; and demonstrate the algorithm with an example in 3.3.

3.1 Overview

Our algorithm differs from previous ones mainly in its not waiting for the token to come back. For instance, in the example described in section 2.4, after d sends the token to e, our algorithm will abort d immediately without waiting for the token to clean up the obsolete probes. We employ a new strategy to avoid phantom deadlock detection: when d is aborted, set a "barrier" at c to prevent any probe (such as p_w) from moving beyond c. This way, w will not detect any phantom deadlock. In order to make this strategy work, there are many technical problems that need to be solved. We give an overview of our algorithm below, and formally describe it in the next subsection.

When a dependency edge (v, w) from a temporary process v to a permanent process w is established, a probe p_v is created at w. Although p_v is actually created by w, it is said to be *initiated* by v, and v is called the *initiator* of p_v. (Note that this has the same effect as p_v being created by v and forwarded to w. Only temporary processes may initiate probes.) If w has an outgoing edge, then a copy of p_v is sent to the next process. If edge (v, w) is involved in a deadlock cycle, then probe p_v will eventually return to v and the deadlock is detected. As in [19], in order to reduce the number of probe messages, a probe p is propagated to a process x only if (the identifier) of x is larger than (that of) the initiator of p. With this scheme, many probes will not be forwarded. In particular, if there is a deadlock, only the probe initiated by the smallest process in the deadlock cycle will manage to return to its initiator.

A probe p arriving at a process v is stored in a local variable QUE_v (which is a list or a set) as a 4-tuple $(initiator, sender, class, state)$, where *initiator* is the initiator of p; *sender* is the process that sent p to v; *class* is a binary value indicating whether p is a *primary* probe or a *secondary* (its meaning will be explained later); *state* records the propagation status of p and has two possible values, *waiting* or *sent*. If a copy of p has been sent to the next process, *state* is set to *sent*; otherwise, it is *waiting*. These parameters are referred to as $initiator(p)$, $sender(p)$, and so forth. When a probe p is forwarded to another process, only the value $initiator(p)$ is sent along. We denote a probe message by $P(x)$ where x is the initiator of the probe.

If a probe initiated by a process x returns back to x, then x detects a deadlock and goes ahead to abort itself. In the process of abortion, x disconnects all incoming edges (i.e., unlocks all data items it has held), and sends a token to the next process in the deadlock. A token is denoted by $T(x, S)$, where x is the initiator of the token, and S is a set of process identifiers. The job of the token is to eliminate all the probes in the deadlock cycle whose initiators are included in S.

To implement the barrier strategy mentioned in the beginning of this section, a new variable BAR_v is introduced. BAR_v is a set of *barriers*. Each barrier is created by a process under abortion and is denoted by the identifier of the process itself. The function of BAR_v is this: as long as there is a barrier in BAR_v, process v does not pass any probe in QUE_v to any other process.

When a node v sends a "D" message to disconnect an incoming edge, say, (w, v), a subset of BAR_v is passed to w. So, the "D" message is modified to "$D(S)$", where S is the subset of barriers to be passed to w.

Consider the example in Fig. 1. When process d gets aborted, it creates a token and a barrier. (So there is a one-one correspondence between tokens and barriers.) The token is sent to e and it travels around the deadlock cycle $(e, f, a, \ldots)$. The barrier is sent to c in a "$D(S)$" message. If later on edge (b, c) gets disconnected, that barrier will be passed to b. So, like the token, the barrier also travels around the deadlock cycle, but in the opposite direction $(c, b, a, \ldots)$. When the token and the barrier meet at some node, they eliminate each other. Should they cross each other in an edge (x, y), then the token is removed by y and the barrier by x.

3.2 Formal Description

In this section, the six actions modeling the resource-sharing distributed system in section 2.2 are slightly modified to include some statements dealing with probes. Three new actions A_7-A_9 that form the nucleus of our algorithm for deadlock detection and resolution are added.

1. **[Connecting an edge]** If a temporary process v has no outgoing edge (i.e., $OUT_v = null$), it is free to make a connection to a permanent process by sending to the latter a "C" message.

A_1^v ::= **if** $OUT_v = null$ **then**

  $OUT_v :=$ select$(P \cup \{null\})$, where P is the set of all permanent processes;

  if $OUT_v \neq null$ **then** send a "C" message to OUT_v;

2. **[Disconnecting an edge]** If a temporary process v has no outgoing edge, it is free to disconnect an incoming edge, say (u, v), by sending to u a "$D(S)$" message. In doing so, all probes at v that came to v through channel (u, v) are deleted from the probe queue of v.

A_2^v ::= **if** $OUT_v = null$ **and** $IN_v \neq \emptyset$ **then**

  $u :=$ select$(IN_v \cup \{null\})$;

  if $u \neq null$ **then**

    disconnect edge (u, v) by calling procedure DISC$(u, v, 0)$;

    delete from QUE_v all probes p that have $sender(p) = u$;

3. **[Terminating itself]** If a temporary process v has no outgoing nor incoming edge, it is free to terminate. A terminated process no longer exists in the system.

A_3^v ::= **if** $OUT_v = null$ **and** $IN_v = \emptyset$ **then**

  terminate itself;

4. **[Reversing an edge]** If a permanent process v has no outgoing edge, it is free to reverse the direction of an incoming edge, say (u, v), by sending a "C" and a "D" message to u. In addition, all probes p such that $sender(p) = u$ are deleted from QUE_v. These probes must be deleted because edge (u, v) no longer exists.

A_4^v ::= **if** $OUT_v = null$ **and** $IN_v \neq \emptyset$ **then**

$u := \text{select}(IN_v);$

$\text{DISC}(u, v, 1);$ /* send a combined "$C + D$" message to u */

$OUT_v := u;$

delete all probes of the form $(\cdot, u, \cdot, \cdot)$ from QUE_v,

where a dot "$\cdot$" means "don't care" or "any value".

Comments: A combined "$C+D$" message is simply a "C" and a "D" message bounded together. The receiver of such a message will react as if it received two separate messages. One may regard a "$C + D$" as a new type of message that triggers both actions A_5 and A_6.

5. **[Receiving a connection message]** If process v receives a "C" message from process u, then u is added to IN_v. Furthermore, if $u < v$ then a probe is created.

A_5^v ::= **if** v receives a "C" message from u **then**

$IN_v := IN_v \cup \{u\};$

if $(u < v)$ **then** insert a probe $(u, u, \cdot, waiting)$ in QUE_v;

Comment: In the inner if-statement, although the probe is actually created by v, it is marked as being initiated by and coming from u. With this, a process will be able to detect a deadlock on receiving a probe initiated by itself.

6. **[Receiving a disconnection message]** If process v receives a "$D(Z)$" message from process OUT_v, then v disconnects its outgoing edge. All probes in QUE_v change their status to waiting, and the barriers contained in Z is used to update BAR_v. (Note that a "$D(\cdot)$" message received from a process apart from OUT_v will be ignored.)

A_6^v ::= **if** v receives a "$D(Z)$" message from OUT_v **then**

$OUT_v := null;$

$BAR_v := BAR_v \cup \{z \in Z : (z, \cdot, \cdot, \cdot) \in QUE_v\};$

for all probes $p \in QUE_v$ **do** $status(p) := waiting;$

Comments: Instead of adding all barriers in Z to BAR_v (which would result in an incorrect algorithm), we add only those which are initiators of some probes in QUE_v. If a $z \in Z$ does not have a corresponding probe in QUE_v, that is because such a probe has been removed by a token (see A_9 below). This may happen should the token and the "$D(Z)$" message cross each other in the channel between process v and process OUT_v.

7. **[Probe propagation]** If process v has an outgoing edge and holds no barrier (in BAR_v), then send to process OUT_v all probes in QUE_v which are "ready".

A_7^v ::= **if** $OUT_v \neq null$ **and** $BAR_v = \emptyset$ **then**

  for every $p \in QUE_v$ with ready$(p) = true$ **do**

    send a probe message "$P(initiator(p))$" to process OUT_v;

    $status(p) := sent$;

  where

    ready(p) $\equiv$ (p is primary and waiting) **and** ($initiator(p) \leq OUT_v$).

Comments: Several comments are in order. First, probes are allowed to propagate to the next process (OUT_v) only if no barrier is set up at v (i.e., $BAR_v = \emptyset$. Without this provision, it will be possible that a probe continues to explore, detects another deadlock involving an already broken probe-path, and thus further breaks the probe path in resolution. If this happens, the token associated with the original broken probe-path may not be able to traverse the entire broken path. Second, only those primary probes which have not been forwarded and whose initiators are no higher than OUT_v in identifier, are forwarded.

8. **[Receiving a probe]** If process v receives a probe "$P(x)$" from process u, then three cases may arise. First, if u is not connected to v, v ignores the probe. Second, if u is connected to v and $x \neq v$, then a probe is inserted in QUE_v. Third, if u is connected to v and $x = v$, then this probe was originally initiated by v and it has now come back, indicating the detection of a deadlock. In this case, v aborts itself.

A_8^v ::= **if** v receives a "$P(x)$" message from u **then**

case

  a) ($u \notin IN_v$): do nothing;

  b) $u \in IN_v$ **and** $x \neq v$:

    insert the probe $(x, u, \cdot, waiting)$ into QUE_v;

  c) $u \in IN_v$ **and** $x = v$: /* a deadlock is detected */

    insert $(v, u, primary, sent)$ into QUE_v;

$BAR_v := BAR_v \cup \{v\};$ /* set up a new barrier */

for every process y in IN_v **do**

DISC$(y, v, 0)$; /* disconnect incoming edges */

$S := \{initiator(p) : p = (\cdot, \cdot, \cdot, sent) \in QUE_v\};$

if $(OUT_v \neq null)$ **then** send a token "$T(v, S)$" to OUT_v;

kill process v;

Comments: 1) Note that in the third case, the probe is inserted into QUE_v. Recall that, conceptually, when the edge (v, OUT_v) was established, the probe was initiated by v and sent to OUT_v (so it should have been inserted into QUE_v at that time). But the probe was actually created by process OUT_v, without any record in QUE_v. So the probe is now inserted into QUE_v, so that it needs not be treated as a special case later on. 2) S indicates the probes whose paths are now broken. A token is sent out to hunt for these probes. 3) The token is sent out only if $OUT_v \neq null$ (otherwise, no destination for the token). OUT_v may be null because once the incoming edges of v were disconnected, it is not impossible that all edges on the deadlock cycle, including edge (v, OUT_v), were disconnected quickly before v is ready to sent out the token.

9. **[Receiving a token]** If process v receives a token "$T(x, S)$" from process u, then remove all out-of-date probes and no-longer-needed barriers from QUE_v and BAR_v, respectively. Furthermore, a set S' is constructed so that S' includes initiators of all removed probes that have been propagated to the next process. A token containing S' is sent to the next process.

$A_9^v ::=$ **if** v receives a token "$T(x, S)$" from u **then**

$S' := \{z \in S : (z, u, \cdot, sent) \in QUE_v\};$

if $(OUT_v \neq null)$ **and** $(S' \neq \emptyset)$ **then**

send a token "$T(x, S')$" to OUT_v;

$BAR_v := BAR_v - \{z \in S : (z, u, primary, \cdot) \in QUE_v\};$

delete from QUE_v all probes p such that $initiator(p) \in S$

and $sender(p) = u$

if $x = u$ **and** $u \in IN_v$ (i.e., v is the successor of the aborted process) **then**

$IN_v := IN_v - \{u\};$ /* delete the incoming edge */

if $x = u$ **and** $u = OUT_v$ **then**

$OUT_v := null$;

change the status of each probe in QUE_v to *waiting*;

Comments: The case "$x = u$ **and** $u = OUT_v$" may arise for the same reason as stated in comment 3 of item 8, except that in this case edge (v, OUT_v) is reversed instead of disconnected.

10. **[Deleting an edge]** To delete an edge coming from u, perform the following.

DISC(u, v, R) ::=

$IN_v := IN_v - \{u\}$;

$B := \{b \in BAR_v : (b, u, primary, \cdot) \in QUE_v\}$;

$BAR_v := BAR_v - B$;

if $R = 0$ **then**

send a "$D(B)$" message to u

else

send a combined "$C + D(B)$" message to u;

11. **[Inserting a probe]** When a probe p is being inserted into QUE_v, it is actually inserted only if no probe in QUE_v has the same initiator and sender as p. In that case, it is inserted as a primary if there is no other probe in QUE_v having the same initiator as p; a secondary, otherwise.

12. **[Deleting a probe]** Whenever a primary probe p is deleted from QUE_v, a probe in QUE_v with the same initiator as p, if any, is promoted to a primary probe.

3.3 Examples

1. In Fig. 2(a), a deadlock (a, b, c, d, e, f, a) has been formed. Assume that d has the smallest identifier among these processes (d must be a temporary process since all permanent processes have larger identifiers). When edge (d, e) was created (i.e., when d's "C" message arrived at e), a probe p_d was created at e, according to action A_5. This probe was marked as being initiated by d and was placed in QUE_v. Suppose that p_d, by actions A_7 and A_8, has propagated through (e, f, a, b, c) and arrived at d. Then the deadlock is detected by d in execution of A_8. Each process in the deadlock has a copy of p_d stored in a local QUE. Assume that probes p_h, p_l and p_m have

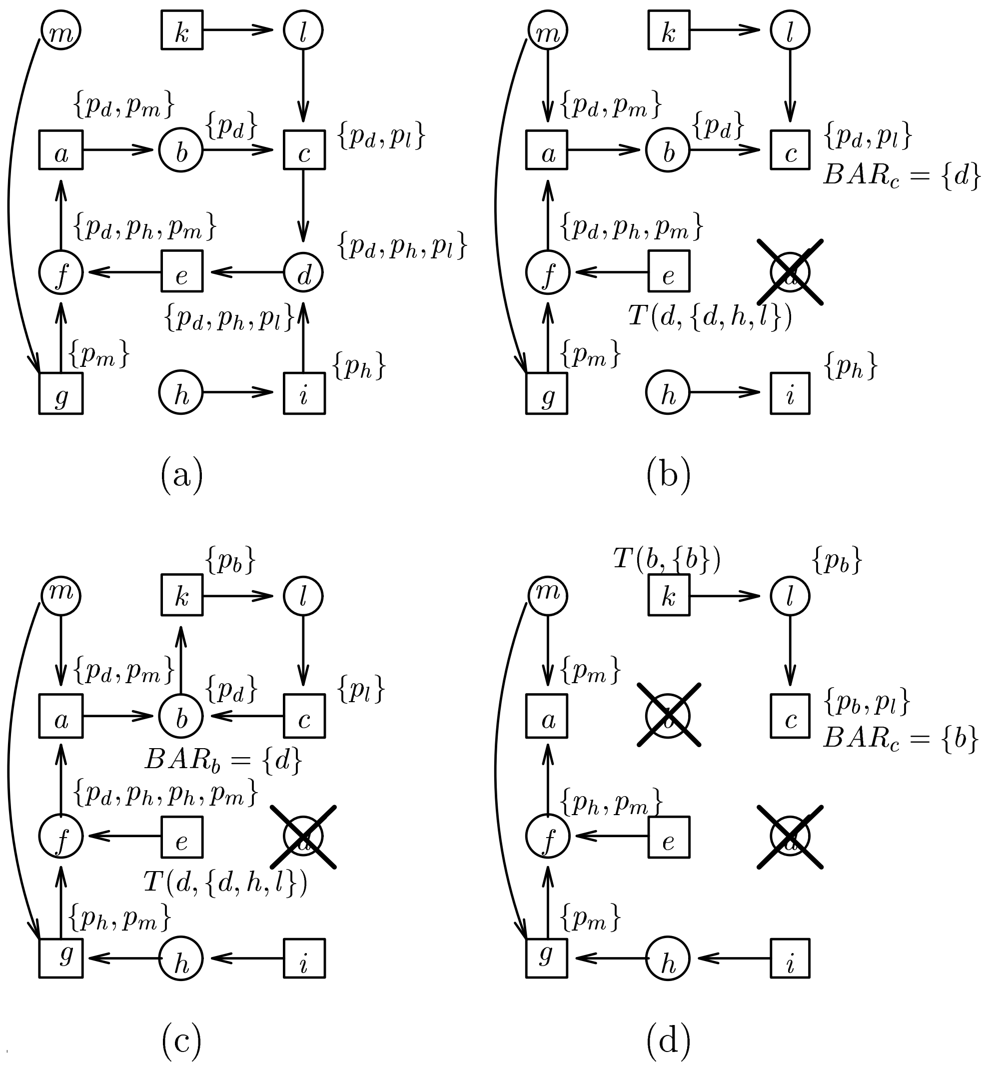

Figure 2: Examples

arrived at processes f, e and a, respectively, where p_x denotes the probe initiated by x. Then $BAR_f = \{p_d, p_h, p_m\}$ and $BAR_i = \{p_h\}$, and so forth.

2. Figure 2(b) shows that upon detecting the deadlock, d aborts itself by doing the following. (i) Create a barrier. (ii) Send a "$D(B)$" message to c and i so as to disconnect edges (c, d) and (i, d); the parameter B enclosed in the message for i is an empty set, while the B for c is $\{d\}$, where d denotes the barrier created by process d. (iii) Send a token $T(d, S)$ to e, with $S = \{d, h, l\}$, in order to eliminate all probes initiated by process d, h, or l that are still in the deadlock cycle.

3. Figure 2(c) results from (b) should the following events occur. (i) c reverses edge (b, c) with a "$C + D(B)$" message and passes barrier d to b (i.e., $B = \{d\}$ in the "$D(B)$" message). (ii) b quickly establishes edge (b, k) with a "C" message. (iii) k creates a probe p_b with b as the initiator. (iv) i reverses edge (h, i) with a "$C + D$" message. (v) h establishes edge (h, g) with a "C" message which causes g to create a probe p_h. (vi) p_h propagates to f where it is stored in QUE_f as a secondary, since there is already a p_h in QUE_f.

4. Figure 2(d) shows the configuration that results from the following events. Suppose that the probe p_b at process k moves to l, to c and arrives at b. Then a new deadlock (b, k, l, c, b) is detected and b gets aborted. In its process of abortion, b creates a new barrier so that $BAR_b = \{d, b\}$. Barrier d is passed to process a, and barrier b to process c. (So at this point, $BAR_b = \emptyset$, $BAR_a = \{d\}$, and $BAR_c = \{b\}$.) Meanwhile, assume that token $T(d, S)$ moves from e to f, with $S = \{d, h\}$ (note that l has been dropped from S). At f, the token removes p_d and the primary copy of p_h from QUE_f; as a result, the secondary p_h becomes a primary. Suppose that $T(d, S)$, with $S = \{d\}$, continues to move to a, and removes p_d from QUE_a and removes barrier d from BAR_a. BAR_a becomes empty, and $T(d, S)$ vanishes because S becomes empty (see A_9 of the algorithm). Now without any barrier at a, probe p_m can move to the next process if a has an outgoing edge.

5. In step 3, we let b make a request to k and create edge (b, k). What if b makes the request to e instead and establishes edge (b, e)? In this case, a deadlock (b, e, f, a, b) will be formed. Suppose b detects the deadlock and aborts itself. Then a new barrier b will be created (the barrier is denoted as b because it is created by process b). Both barriers b and d will be sent to a, and BAR_a will become $\{b, d\}$. This explains why a BAR may contain more than one element.

4 Correctness Proof

Assume that all processes and channels are failure-free, the buffer in a channel is unbounded, and all messages are delivered in a first-in-first-out fashion with arbitrary but finite delay. We prove the correctness of our algorithm by establishing its safety and progress properties as defined in Definition 2; our algorithm obviously has the resolution property.

As in [11], we adopt the common approach of defining an execution of a concurrent algorithm to be a sequence of *atomic actions* in which concurrent actions of separate processes are assumed to be interleaved in an arbitrary manner. We regard each action A_1-A_9 of our algorithm as an atomic action, noting that there is at most one incoming message involved in each action.

We establish some basic properties of our algorithm in section 4.1, prove the safety property in 4.2, and verify the progress condition in 4.3.

4.1 Basic Properties

A process may *terminate* as the result of a normal termination procedure (A_3) or it may *abort* itself to resolve a deadlock (A_8). Once a process is terminated or aborted, it does not receive or send any message, nor does it take any action. The values of its local variables and the contents of its incoming channels are no longer meaningful. Thus, for convenience, we make the following definition.

Definition 3 For any aborted or terminated process v, define

- $IN_v = BAR_v = QUE_v = \emptyset$, and $OUT_v = null$; and
- $c[x, v] = \emptyset$ for each incoming channel $c[x, v]$.

A probe may have multiple copies in the system. At a node, a probe is stored in QUE as a 4-tuple (*initiator*, *sender*, *class*, *state*); in a channel, it is a probe message containing the *initiator*. In either form, it represents a copy of a probe. In the following definition, p refers to a particular copy of a probe.

Definition 4 The notations *initiator*(p) and *sender*(p) for a probe p are abbreviated as $i(p)$ and $s(p)$, respectively.

Definition 5 For each probe p, the *location* of p, denoted by $\hat{p}$, is defined as follows.

$$\hat{p} = \begin{cases} \text{process } v, & \text{if } p \in QUE_v \\ \text{process } w, & \text{if } p \text{ is in a channel toward } w,\ c[\cdot, w]. \end{cases}$$

Also, let *channel*(p) denote the channel from $s(p)$ to $\hat{p}$.

For clarity, we list a number of simple facts about channels, edges, probes, tokens, and barriers. All these facts may be readily observed from the algorithm, and will often be applied in subsequent arguments without explicit mentioning.

Fact 1 Let u and v be two processes, p be a probe, and "$T(\cdot,\cdot)$" be a token. The following are true.

1. Channels, edges
   (a) Permanent processes communicate only with temporary processes, and vice versa. As a result, $c[u,v] = \emptyset$ if both processes are permanent or both are temporary. Besides, dependency edges exist only between temporary and permanent edges.
   (b) A permanent process never sends a single "C" or "$D(\cdot)$" message; they are always sent together as a combined message.
2. Probes
   (a) Only temporary processes may initiate probes.
   (b) If several probes in QUE_v have a common initiator, only one of them is primary and the others are secondary. Furthermore, their *senders* are all distinct.
   (c) A probe can be propagated only if it is a primary.
   (d) For each probe p in QUE_v, $s(p) \in IN_v$.
   (e) If $OUT_v = null$, then all probes in QUE_v are in the *waiting* status.
3. Tokens
   (a) If $T(v,\cdot)$ exists in the system, then v is an aborted process and $T(v,\cdot)$ is unique (i.e., there is no other token initiated by v).
   (b) If "$T(v,S)$" is in $c[v,x]$, it is the last message in the channel. If "$T(v,S)$" is in $c[u,x]$, where $u \neq v$, then $u \notin S$.
4. Barriers
   (a) If $z \in BAR_v$, then there exists a probe in QUE_v with $i(p) = z$.
5. Identifiers
   (a) For any probe p, $i(p) \leq \hat{p}$ and $s(p) \leq \hat{p}$.
   (b) For any token "$T(x,S)$", $y \leq x$ for each y in S.

For any two processes v and w, there may be a well-established edge from v to w; there may be not; there may be an edge under construction; or there may be an edge being disconnected.

We describe these different situations by assigning a status to edge (v, w) — *empty*, *growing*, *established*, or *shrinking*. Initially, edge (v, w) is *empty* (i.e., not connected). When process v sends a connection message to process w, edge (v, w) is *growing*. When w receives the connection message, (v, w) becomes *established*. Eventually, process w will disconnect edge (v, w) by sending a disconnection message to process v and (v, w) becomes *shrinking*. When v receives the disconnection message, (v, w) becomes *empty* again. We define these edge states formally and show their state transitions.

Definition 6 Let v and w be two distinct processes. Edge (v, w) is said to be

empty	if $OUT_v \neq w$, "C" $\notin c[v, w]$, and $v \notin IN_w$;
growing	if both v and w are alive, $OUT_v = w$, "C" $\in c[v, w]$, and $v \notin IN_w$;
established	if both v and w are alive, $OUT_v = w$, "C" $\notin c[v, w]$, and $v \in IN_w$;
shrinking-1	if both v and w are alive, $OUT_v = w$, "C" $\notin c[v, w]$, $v \notin IN_w$, and "$D(\cdot)$" $\in c[w, v]$;
shrinking-2	if v has been aborted, "$T(v, \cdot)$" $\in c[v, w]$, and either "C" $\in c[v, w]$ or $v \in IN_w$;
shrinking-3	if w has been aborted, $OUT_v = w$, and "$T(w, \cdot)$" $\in c[w, v]$.

Lemma 1 *Let v and w be any two processes. Edge (v, w) is either empty, growing, established, or shrinking, and it changes state according to the transition diagram in Figure 3.*

Before proving the lemma, we wish to comment that some transitions included in Fig. 3 actually cannot happen (e.g., the transition from growing to shrinking-2 or -3). They are included in the diagram because their impossibility has not been proved.

The above lemma and the following two corollaries will be proved simultaneously.

Corollary 1 *If edge (v, w) is empty, growing, established, or shrinking-3, then "$D(\cdot)$" $\notin c[w, v]$.*

Corollary 2 *If a channel contains a "C" or "D" message, then there is no other message of the same type in that channel.*

Proof. We prove the lemma and the two corollaries at the same time. Let v and w be two arbitrary processes, and let A be any action. Initially, edge (v, w) is empty and both $c[v, w]$ and $c[w, v]$ are empty, so the first part of the lemma, as well as the two corollaries, is true. Now, assume that before action A, edge (v, w) is either empty, growing, established, or shrinking, and both corollaries are true. We show that, after the execution of A, the edge is still in one of those

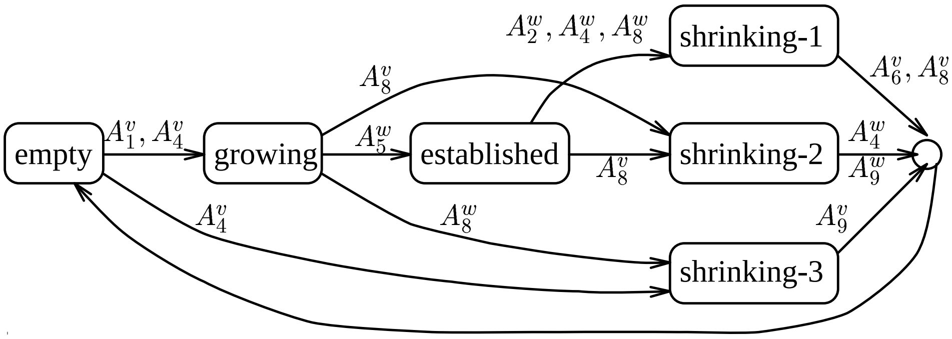

Figure 3: State transition diagram of edge (v, w)

states, and any change in state follows the transition diagram in Fig. 3. We will make sure that the two corollaries remain valid after the action, often without explicitly stating it.

Observe that an action taken by a process other than v or w will not change the state of edge (v, w), for such an action does not change the value of OUT_v, $c[v, w]$, IN_w, or $c[w, v]$. Consequently, only those actions at v and w need to be considered.

Case 1: (v, w) is empty before action A. First, let A be an action of v. The only actions that may change the edge's state are A_1^v and A_4^v, provided that w is the process selected by the select function . In the case of A_1^v, w is a permanent process; and after the action, (v, w) becomes growing, no new "D" message is introduced to channel $c[w, v]$, and the "C" message in $c[v, w]$ is still unique. Thus both the lemma and the two corollaries remain valid after the action. In the case of A_4^v, since $w \in IN_v$ before the action, edge (w, v), the opposite of (v, w), is either in the established state and hence w is alive at time $t^-(A)$, or it is in the shrinking-2 state and thus w has been aborted. Thus, after the action, (v, w) becomes either growing or shrinking-3 depending on whether w is alive or has been aborted. In both cases, the two corollaries are still true. Now, suppose A is an action of w. Since (v, w) is empty, no action at w that may occur under this condition, may change any of these conditions: $OUT_v \neq w$, "C" $\notin c[v, w]$, $v \notin IN_w$, and "$D(\cdot)$" $\notin c[w, v]$. Thus, (v, w) remains empty.

Case 2: (v, w) is growing before action A. First, suppose A is an action of v, which obviously is none of A_1–A_4 since $OUT_v \neq null$. A_6 is also impossible, for at time $t^-(A)$ there is no "$D(\cdot)$" message in $c[w, v]$ for v to receive. In the case of A_5, A_7, or A_9, the edge remains growing, as such an action is easily seen to not change any condition concerning (v, w)'s status. The only action of v that may possibly change (v, w)'s state is A_8 in which v aborts itself and edge (v, w) becomes shrinking-2. Next, consider the case of w executing the action. One readily sees that actions A_1, A_6 and A_7 at w have no effect on the state of (v, w), and that actions A_2 and A_4 cannot happen at w since $v \notin IN_w$. A_9 does not change the edge's state; this is because w is alive and so even if w passes the received token to v during the action, the edge's state does not change to shrinking-3. In the case of A_5, w receives the only "C" message in $c[v, w]$ and (v, w) becomes established. It remains to consider A_3 and A_8, in which w either terminates or aborts itself. In either case, w is a temporary process and v a permanent. By Fact 1(1b), channel $c[v, w]$ contains (at time $t^-(A)$) a "$D(\cdot)$" that follows the "C" message; and thus, at time $t^-(A)$, edge (w, v) is shrinking-1 and $OUT_w = v$. So A_3 cannot occur at w; and in the case of A_8, w aborts itself and adds a "$T(w, \cdot)$" to $c[w, v]$, changing edge (v, w)'s state to shrinking-3.

Case 3: (v, w) is established before action A. In this case both v and w are alive at time $t^-(A)$. First let A be an action of v. It cannot be A_1, A_2, A_3, or A_4, as $OUT_v \neq null$; and it cannot be A_6 either, since there is no "$D(\cdot)$" message in $c[w, v]$ for v to receive. Actions A_5, A_7 and A_9 each obviously have no effect on the edge's state. As for A_8, if v gets aborted in the action, then edge (v, w) becomes shrinking-2. Now suppose A is an action executed by w. If it is an A_2, A_4, or A_8, then v is removed from IN_w and a "$D(\cdot)$" message is added to $c[w, v]$, causing edge (v, w) to become shrinking-1. A_3 cannot happen because $IN_w \neq \emptyset$. Actions A_1, A_5, A_6 and A_7 each obviously have no effect on (v, w)'s state; and neither does A_9, since the token received by w cannot be initiated by the still alive v (i.e., the condition of the last **if** statement in A_9 is false).

Case 4: (v, w) is shrinking-1 before action A. First let A occur at v. It cannot be A_1, A_2, A_3, or A_4 due to the condition $OUT_v \neq null$. Neither A_5 nor A_7 may change the edge's state. Action A_6 obviously will result in an empty (v, w). Case (c) of A_8 will also result in an empty (v, w) (rather than a shrinking-2 because "C" $\notin c[v, w]$ and $v \notin IN_w$; also note that "$D(\notin)$"$c[w, v]$ because channel $c[w, v]$ is assumed to be empty once v gets aborted). Edge (v, w)'s state may change in A_9 only if this action is triggered by "$T(w, \cdot)$". Such a token, if it exists at time $t^-(A)$, must be the very last message in $c[w, v]$ (by Fact 1(3b)) and be preceded by a "$D(\cdot)$" (because of the shrinking-1 status of (v, w)). Thus, A cannot be an action triggered by "$T(w, \cdot)$". So A_9 has no effect on (v, w)'s state. Now, if A is an action of w, it is not hard to see that (v, w) remains shrinking-1, as no action at w may possibly change any of these conditions: $OUT_v = w$, "C" $\notin c[v, w]$, $v \notin IN_w$, and "$D(\cdot)$" $\in c[w, v]$.

Case 5: (v, w) is shrinking-2 before action A. In this case A must be an action of w since v has already been aborted. If "C" $\in c[v, w]$ at time $t^-(A)$ and the "C" message arrives at w in

$A = A_5$, then (v, w) remains shrinking-2. If $v \in IN_w$ at $t^-(A)$ and w reverses edge (v, w) in action $A = A_4$, then (v, w) becomes empty. (Recall that by convention channel $c[w, v]$ is always empty once v is aborted.) If w receives "$T(v, \cdot)$" in A_9^w, edge (v, w) becomes empty. It can be easily checked that A_1^w–A_3^w cannot happen and A_6^w-A_8^w will not change the edge's shrinking-2 status.

Case 6: (v, w) is shrinking-3 before action A. Since w has already been aborted, we only have to consider v's action. If A is an A_9 action in which v receives token "$T(w, \cdot)$", then OUT_v is set to *null* and edge (v, w) becomes empty. Actions A_1^w-A_4^w cannot happen, and A_5^w-A_8^w will not change the edge's shrinking-3 status. □

Corollary 3 *Every process has at most one outgoing edge which is not empty.*

Proof. From Definition 6 and Lemma 1, one readily sees that if two nonempty edges, say (v, w) and (v, x), coexist, they must both be shrinking-2, or one shrinking-2 and the other shrinking-3. The former case is impossible, since there is only one "$T(v, \cdot)$" and $c[v, w]$ and $c[v, x]$ cannot both contain it. The latter case is also impossible because the processes at the ends of an edge cannot both be aborted. □

When an edge (v, w) from a temporary to a permanent process becomes established, the permanent, w, creates a probe p whose initiator and sender are both set to v. The probe is inserted into QUE_w. If w has an outgoing edge, w propagates probe p by sending a copy of it to the next process, say x, which keeps the probe in QUE_x and later on may propagate it by sending a copy to another process, say y. Now, processes w, x, and y all have a probe p — w has the original, x has a copy of the original, and y has a copy of a copy. These concepts lead us to define the following.

Definition 7 The probe "$P(initiator(p))$" sent to $w = OUT_v$ in action A_7 is said to be a *direct copy* or a *child* of p, whether it is still in the form of a message traveling in channel $c[v, w]$ or has already been stored in QUE_w in the form of a 4-tuple.

Definition 8 A probe p is said to be *barren* if it is in a channel $c[x, y]$ such that $x \notin IN_y$ and there is no "C" message in $c[x, y]$ that precedes p.

A barren probe does not get propagated. When it arrives at the destination process, the process does nothing but discard the probe (see A_7). In the following definition, recall that $\hat{p}$ denotes the location of p (see Definition 5).

Definition 9 Let q be a child of p. If $q \in QUE_{\hat{q}}$, then channel $c[\hat{p}, \hat{q}]$ is said to be *clean* (with respect to the parent-child pair (p, q)) if it contains no "C" message, no probe with the same initiator as q's, and no token "$T(\cdot, S)$" with $i(q) \in S$. If q is still in $c[\hat{p}, \hat{q}]$, then the channel is said to be *clean* if there are no mentioned messages in $c[\hat{p}, \hat{q}]$ that follow q.

The following lemma explores the relationship between a parent probe and its non-barren child.

Lemma 2 *If q is a non-barren child of p, then p's status in $QUE_{\hat{p}}$ is "primary" and "sent" (i.e., $p = (\cdot, \cdot, primary, sent)$), $OUT_{\hat{p}} = \hat{q}$, and channel $c[\hat{p}, \hat{q}]$ is clean.*

Proof. Initially, no probe exists and the lemma is obviously true. Assuming the lemma is true before an action A, we show that it remains true after the action. Let q be any non-barren child of $p \in QUE_{\hat{p}}$ after the action. There are two possibilities.

Case 1: q is a new child which was just created and added to $c[\hat{p}, \hat{q}]$ in action A (which must be an A_7). The validity of the lemma follows directly from A_7.

Case 2: q was already a non-barren child before action A. First note that edge $(\hat{p}, \hat{q})$ was at time $t^-(A)$ either growing or established, since $\hat{p}$ and $\hat{q}$ were both alive and q was not barren (the latter implying "C" $\in c[\hat{p}, \hat{q}]$ or $\hat{p} \in IN_{\hat{q}}$). At time $t^-(A)$, since edge $(\hat{p}, \hat{q})$ was growing/established, $OUT_{\hat{p}} = \hat{q}$. In action A, $\hat{p}$ cannot send any "C" message (since $OUT_{\hat{p}} \neq null$ at time $t^-(A)$), or any probe with the same initiator as p's (because of p's *sent* status at time $t^-(A)$), or any token "$T(\cdot, S)$" with $i(p) \in S$ (as otherwise p would have been removed from $QUE_{\hat{p}}$ in A). So $c[\hat{p}, \hat{q}]$ remains clean after action A. The status of p may change from *sent* to *waiting* only in action A_6 in which $\hat{p}$ receives a "$D(\cdot)$" message from $OUT_{\hat{p}}$ and changes $OUT_{\hat{p}}$ to *null*. But since that is apparently not the case (for $OUT_{\hat{p}} \neq null$ at time $t^+(A)$), p's *sent* status does not change in action A. As for p's primary status, note that a primary probe remains primary until it is removed. So the lemma remains valid after action A. □

Case 2 of the above proof also establishes the following.

Corollary 4 *If q is a non-barren child of p, then edge $(\hat{p}, \hat{q})$ is either in the growing state or in the established state. If q has already been received by $\hat{q}$, then only the latter state is possible.*

Definition 10 A sequence $P = (p_1, \ldots, p_k)$ of probes currently in the system is said to be a *probe chain* if p_i is a child of p_{i-1} for all i, $1 < i \leq k$, and, furthermore, p_k is not barren. A probe chain is *maximal* if it is not a proper subsequence of any other chain.

Definition 11 If $P = (p_1, \ldots, p_k)$ is a probe chain, then $Proj(P)$ denotes the sequence of processes $(\hat{p}_1, \ldots, \hat{p}_k)$.

A probe chain splits into two chains should the location of some probe in the chain get aborted (in an A_8 action). For instance, if $\hat{p}_i$ gets aborted, then probe p_i disappears from the system, and $P = (p_1, \ldots, p_k)$ is split into $(p_1, \ldots, p_{i-1})$ and $(p_{i+1}, \ldots, p_k)$. (In extreme cases, one or both of the subchains may be empty.) When this occurs, process $\hat{p}_i$ sends out a token $T(\hat{p}_i, S)$ that will "chase" and remove probes $p_{i+1}, \ldots, p_k$ one by one. The notion of chasing is formally formulated in the following.

Definition 12 A probe chain $P = (p_1, \ldots, p_k)$ is said to be *chased* by a token "$T(\cdot, S)$" if the following conditions are all satisfied:

(a) "$T(\cdot, S)$" is in the channel via which p_1 came from, i.e., $c[s(p_1), \hat{p}_1]$ (this condition implies that p_1 is, or was, a child of some other probe);

(b) $i(p_1) \in S$;

(c) preceding "$T(\cdot, S)$" in the channel there is no "C" message, no probe with initiator equal to $i(p_1)$, and no any other token "$T(\cdot, S')$" with $i(p_1) \in S'$.

Definition 13 If v is a process that has been aborted, then for any process u (dead or alive), let $CH(u, v)$ denote the maximal probe chain that was initiated by u and is being chased by the token "$T(v, S)$" generated by v (if there is no such a chain, then $CH(u, v) = \epsilon$). If v is not an aborted process, let $CH(v, v)$ denote the maximal probe chain $(p_1, \ldots, p_k)$ that was initiated by v and has $sender(p_1) = v$ (again, $CH(v, v) = \epsilon$ if no such chain exists).

$CH(v, v)$ and $CH(u, v)$ are well defined in the sense that they each denote a unique probe chain. The following lemma establishes this fact.

Lemma 3 *$CH(u, v)$ and $CH(v, v)$ each denote a unique probe chain.*

Proof. Let v be an aborted process, and assume that two non-empty chains, $(p_1, \ldots, q_k)$ and $(q_1, \ldots, q_l)$, each satisfy the condition of $CH(u, v)$. Since there is only one token initiated by v (Fact 1 (3a)), the two chains are chased by the same token. Therefore, p_1 and q_1 are at the same process and have the same sender (i.e. $\hat{p}_1 = \hat{q}_1$ and $s(p_1) = s(q_1)$). By Fact 1(2b), p_1 and q_1 are the same probe, and it follows from the condition of $CH(u, v)$ and Lemma 2 that $(p_1, \ldots, q_k) = (q_1, \ldots, q_l)$.

Next, consider $CH(v, v)$ with v not aborted. Let $(p_1, \ldots, q_k)$ and $(q_1, \ldots, q_l)$ both be nonempty and satisfy the condition of $CH(v, v)$. We only have to show that p_1 and q_1 refer to the same probe; the rest of the argument will follow the same line as in the above case. It is clear from the algorithm that a process actually never sends a probe initiated by itself to other processes. Thus, probe p_1 must be a probe that was created at $\hat{p}_1$ when $\hat{p}_1$ received a "C" message from v. Thus, $p_1 \in QUE_{\hat{p}_1}$ and by Fact 1(2d), $v \in IN_{\hat{p}_1}$. Since v is not aborted and $\hat{p}_1$ is alive, the only state of edge $(v, \hat{p}_1)$ consistent with all of these conditions is *established* (see Definition 6). So $OUT_v = \hat{p}_1$. Similarly, $OUT_v = \hat{q}_1$, and thus $\hat{p}_1 = \hat{q}_1$. □

In the above proof, we actually have proved the following corollary.

Corollary 5 *If v is a terminated process, then $CH(v, v) = \epsilon$. If v is alive and $CH(v, v) \neq \epsilon$, then $OUT_v \neq null$ and $CH(v, v)$ starts at process OUT_v.*

The following lemma indicates that all probes except for the barren ones are grouped into CH chains.

Lemma 4 *If p is a non-barren probe initiated by a process u, then p belongs in some probe chain $CH(u, x)$, where either x is alive with $x = u$, or x is an aborted process (with $x = u$ or $x \neq u$).*

Proof. Initially, there is no probe in the system and the lemma is true. Now assume that the lemma is true before an action A. We show that the lemma remains true after the execution of A.

At time $t^+(A)$, let p be any non-barren probe originally initiated by u. There are two possibilities: that p is created during action A, and that p already exists at time $t^-(A)$. A probe may be created only in action A_5 (receiving a "C" message) or A_7 (propagating a probe), and u may be aborted, alive, or terminated at time $t^-(A)$. So for the first of the above two possibilities we consider the following cases.

Case 1: u is any process and p is a probe created within $A = A_7$ by some process v. At time $t^-(A)$, v has a probe q in its QUE, with $i(q) = u$; and in action A, v creates p as a child of q and sends it to OUT_w. Since the lemma is assumed to be true before action A, q is in some $CH(u, x) = (p_1, \ldots, p_k)$ at time $t^-(A)$. Since every p_i, $i < k$, is in the *sent* status (by Lemma 2) and cannot produce a new child, it must be that $q = p_k$. Since p is not barren, it belongs in $CH(u, x) = (p_1, \cdots, p_k, p)$ after action A.

Case 2: u has been aborted by time $t^-(A)$ and p is created in $A = A_5$ by some process v. In this case v receives a "C" message from u in action A. So at time $t^-(A)$, channel $c[u, v]$ contains a "C" message. The only possible state at time $t^-(A)$ for edge (u, v) is thus shrinking-2, and that means that channel $c[u, v]$ contains a token "$T(u, S)$" that was created when u was aborted. By Fact 1(3b), $u \in S$ and "$T(u, S)$" is behind "C" in the channel; and apart from "$T(u, S)$", there is no other token $T'(\cdot, S')$ with $u \in S'$ in the channel. From these discussions, it is clear that after the action, (p) is a maximal chain initiated by u and being chased by "$T(u, S)$". So, $CH(u, u) = (p)$, justifying the assertion that p belongs in some $CH(u, x)$.

Case 3: u is alive at time $t^-(A)$ and p is created in $A = A_5$ by some process v. In this action, v receives a "C" message from u and correspondingly creates probe p. It's obvious that after the action $CH(u, u) = (p)$.

Case 4: u has been terminated by time $t^-(A)$ and p is created in $A = A_5$ by some process v (which receives a "C" message from u). This case actually cannot happen, for Lemma 1 and Definition 6 together indicate that every outgoing edge of a terminated process is empty.

Now consider the case that p has been non-barren since $t^-(A)$. At time $t^-(A)$, p belongs to $CH(u, x)$ for some x, since the lemma is assumed to be true at that moment. If during the action no probe is deleted from $CH(u, x)$, then after the action p certainly is still in $CH(u, x)$. It is thus sufficient to consider only those actions in which at least one probe is removed from $CH(u, x)$: namely, actions A_2, A_4, A_8, and A_9. Let $CH(u, x) = (p_1, \ldots, p_k)$. Notice that A_2 and A_4 are prohibited at $\hat{p}_l (1 \leq l \leq k-1)$ since $OUT_{\hat{p}_l}$ is not null (Lemma 2). Since the channels along $Proj(CH(u, x))$ are clean, A_9 at $\hat{p}_l (2 \leq l \leq k)$ can not remove p_l. Moreover, if x is alive $(u = x)$, then since there is no token "$T(\cdot, S)$" with $u \in S$ in $c[u, \hat{p}_1]$ (by Fact 1(3a) & (3b)), A_9 at $\hat{p}_1$ cannot remove p_1, either. Thus, only the following cases are possible. (It has been assumed that p belongs in $CH(u, x) = (p_1, \ldots, p_k)$ at time $t^-(A)$.)

Case 5: $A = A_2^{\hat{p}_k}$ or $A_4^{\hat{p}_k}$, in which $\hat{p}_k$ sends a "$D(\cdot)$" message to disconnect edge $(\hat{p}_{k-1}, \hat{p}_k)$. In this case, either p_k becomes barren (if it is in the channel from $\hat{p}_{k-1}$ to $\hat{p}_k$) or it is removed from the QUE at $\hat{p}_k$ (and thus removed from the system). In both cases, $CH(u, x)$ reduces to $(p_1, \cdots, p_{k-1})$ after the action. Since p is assumed to be in the system and not barren, p is not p_k and thus is still in $CH(u, x)$.

Case 6: $A = A_8^{\hat{p}_l}$ $(1 \leq l \leq k)$, in which $\hat{p}_l$ aborts itself. First note that $\hat{p}_1, \ldots, \hat{p}_k$ are all different except that $\hat{p}_k$ may equal one of the others (since, by Lemma 2, each p_i, $i < k$, is primary). If $\hat{p}_k = \hat{p}_i$ for some $i \neq k$ and it is $\hat{p}_i$ that gets aborted (i.e., $l = i$), then after the action $CH(u, x) = (p_1, \ldots, p_{i-1})$ and $CH(u, \hat{p}_i) = (p_{i+1}, \ldots, p_{k-1})$, and p belongs in one of the two chains. (Note that $p \neq p_i$, since p_i no longer exists in the system after $\hat{p}_i$ gets aborted.) In all other cases, after $\hat{p}_l$ aborts itself, $CH(u, x) = (p_1, \ldots, p_{l-1})$ and $CH(u, \hat{p}_l) = (p_{l+1}, \cdots, p_k)$; and

p, not equal to $\hat{p}_l$, is in one of them.

Case 7: $A = A_9^{\hat{p}_1}$, in which the chasing token "$T(x,S)$" arrives at $\hat{p}_1$. In this case, p_1 is deleted (and discarded) from $QUE_{\hat{p}_1}$. That implies $k > 1$, since p was in $CH(u,x)$ at $t^-(A)$ and still exists at $t^+(A)$ after p_1 has been deleted in A. Lemma 2 indicates that, at time $t^-(A)$, channel $c[\hat{p}_1, \hat{p}_2]$ is clean and thus contains no "C" message, no probe initiated by u, and no token "$T(\cdot, S')$" with $u \in S'$. During action A_9, token "$T(x,S)$", with $u \in S$, is introduced in channel $c[\hat{p}_1, \hat{p}_2]$ to chase $(p_2, \cdots, p_k)$. Thus, $CH(u,x) = (p_2, \cdots, p_k)$ at time $t^+(A)$. (Note that an A_9 action at $\hat{p}_i$, $i > 1$, will not change $CH(u,x)$ since $c[\hat{p}_{i-1}, \hat{p}_i]$ is clean (by Lemma 2).) $\square$

4.2 Proof of Safety Condition

In this section, we prove the safety condition that our algorithm never detects phantom deadlocks. Suppose that a process v detects a deadlock chain by a returning probe at the head of a live chain $CH(v,v)$. The fact $Proj(CH(v,v))$ ends with process v implies that $Proj(CH(v,v))$ is a real deadlock. When process v aborts itself, a nonempty set of obsolete chains $\{CH(x,v) : \forall x\}$ is created. In the set, $CH(v,v)$ is the longest chain and $Proj(CH(x,v))$ is a prefix of $Proj(CH(v,v))$ for all x. Moreover, a disconnection message, "$D(S)$", is sent to the preceding process of v, say u, in deadlock cycle $Proj(CH(v,v))$. When the disconnection message arrives at process u, u inserts v into BAR_u to prevent any chain $CH(x,v)$ from propagating itself out of the broken deadlock cycle. On the other hand, a token message is sent by v to chase probe chain $CH(v,v)$ as well as $CH(x,v)$ and remove the out-of-date probes so that eventually $CH(v,v)$ and $CH(x,v)$ becomes empty and thus the set $\{CH(x,v) : \forall x\}$ becomes empty. The following definition formulates this fact.

Definition 14 Let v be an aborted process and $CH(v,v) = (p_1, \ldots, p_k)$. $CH(v,v)$ is said to be *regular* if either of the following two conditions is satisfies.

1. $CH(x,v) = \epsilon$ for all x; in particular, $CH(v,v) = \epsilon$.
2. $CH(v,v) \neq \epsilon$ and the following are satisfied.
   (a) p_k is a primary probe in $QUE_{\hat{p}_k}$.
   (b) If p_k is in the "waiting" status, then $v \in BAR_{\hat{p}_k}$.
   (c) If p_k is in the "sent" state, then $\hat{p}_k$'s outgoing edge is in the shrinking state, being disconnected by a "$D(S)$" message (i.e., "$D(S)$" $\in c[OUT_{\hat{p}_k}, \hat{p}_k]$) with $v \in S$.

We first establish some useful properties about a regular probe chain in the following two lemmas. The second lemma proves that if a probe chain $CH(x, x)$ is regular for every aborted process x, the next deadlock detection is always a true detection. Thus, the only remaining task will be showing that $CH(x, x)$ is regular for every aborted process x.

Lemma 5 *If $CH(v, v) = (p_1, \ldots, p_k) \neq \epsilon$ is regular, then the following are true.*
(a) $\hat{p}_i \neq \hat{p}_j$ whenever $i \neq j$.
(b) For any $j < k$, $Proj(CH(\hat{p}_j, \hat{p}_j))$ is a prefix of $(\hat{p}_{j+1}, \ldots, \hat{p}_k)$.
(c) For each process x, $Proj(CH(x, v))$ is a prefix of $Proj(CH(v, v))$.

Proof. Lemma 2 and Definition 14 together indicate that all p_i, $1 \leq i \leq k$, are primary, from which statement (a) follows immediately.

Now prove (b). Assume $CH(v, v) \neq \epsilon$, or it is done. By Corollary 5, $CH(\hat{p}_j, \hat{p}_j)$ is a probe chain starting at p_{j+1}. Let $CH(\hat{p}_j, \hat{p}_j) = (q_{j+1}, \ldots, q_l)$, where $\hat{q}_{j+1} = \hat{p}_{j+1}$. It follows from Lemma 2 and a simple induction that the two sequences $(\hat{p}_{j+1}, \ldots, \hat{p}_k)$ and $(\hat{q}_{j+1}, \ldots, \hat{q}_l)$ overlap with each other until the shorter one terminates: $\hat{p}_i = \hat{q}_i$ for $i = j+1, \ldots, \min(k, l)$. We show that $l \leq k$. Since $CH(v, v)$ is regular, p_k is in $QUE_{\hat{p}_k}$. If p_k is in the sent status, then edge $(\hat{p}_k, OUT_{\hat{p}_k})$ is shrinking and no probe chain, even $CH(\hat{p}_j, \hat{p}_j)$, can go beyond $\hat{p}_k$. Now suppose p_k is in the waiting status so that $i(p_k)$ is in $BAR_{\hat{p}_k}$. When $i(p_k)$ was added to $BAR_{\hat{p}_k}$, all probes in $QUE_{\hat{p}_k}$ (including q_k, if it was there) changed their status to waiting (see action A_6). Since then, all probe chains reaching $\hat{p}_k$ from $\hat{p}_{k-1}$ (including $(q_{j+1}, \ldots, q_l)$, if $l \geq k$)) have been stopped by $BAR_{\hat{p}_k}$ from propagation. Hence, $l \leq k$ and $CH(\hat{p}_j, \hat{p}_j)$ cannot go beyond $\hat{p}_k$.

Statement (c) can be proved similarly to (b). □

Lemma 6 *Assume that $CH(x, x)$ is regular for every aborted process x. If in execution of action A_8 a process v claims to have detected a deadlock, then at the time immediately before the action, $Proj(CH(v, v))$ forms a deadlock.*

Proof. Assume that process v claims to have detected a deadlock in execution of action A_8^v. In the action, v receives a probe p with $i(p) = v$ and $s(p) = u \in IN_v$. Consider the moment immediately before A_8^v is executed, when p is still in channel $c[u, v]$.

Since $u \in IN_v$, p is not barren. By Lemma 4, p is the last element of either $CH(v, v)$ or $CH(v, x)$ for some aborted process $x \neq v$. (p must be the last element because it has no child.) In the former case, it is evident from Corollary 4 that $Proj(CH(v, v))$ forms a deadlock. As for the latter case, we show that it cannot happen.

Assume the contrary that p be the last element of $CH(v,x)$ for some aborted $x \neq v$. Then $v = \hat{p}$ is the last element of $Proj(CH(v,x))$. By Fact 1(5b), $v \leq x$. However, because of $CH(x,x)$'s regular property, $Proj(CH(v,x))$ is a prefix of $Proj(CH(x,x))$ (by Lemma 5(c)), and hence $\hat{p}$ is an element of $Proj(CH(x,x))$. By Fact 1(5a), $x \leq \hat{p} = v$. So, $v = x$, contradicting the assumption that $v \neq x$. □

The following lemma proves the premise of Lemma 6, namely, that $CH(x,x)$ is regular for every aborted process x. From these two lemmas follows directly the safety condition of the algorithm.

Lemma 7 *$CH(v,v)$ is regular for every aborted process v.*

Proof. Initially, there is no aborted process in the system and the lemma is obviously true. Assume that the lemma is true up to time $t^-(A)$, where A is an arbitrary action. We show that the lemma remains valid after action A, i.e., at time $t^+(A)$.

Let v be any aborted process as of time $t^+(A)$. There are two possibilities: (i) v is alive at time $t^-(A)$ and gets aborted during action A, and (ii) v has already been aborted by time $t^-(A)$.

First, suppose that v gets aborted during action A. It must be that $A = A_8^v$, and that during the action v receives back a probe originally initiated by v itself and thus claims to have detected a deadlock. The deadlock is a real one as of time $t^-(A)$ and can be described as $Proj(CH(v,v))$ (by Lemma 6 and the assumption that Lemma 7 is true up to time $t^-(A)$). Let $CH(v,v) = (p_1, \ldots, p_k)$ at time $t^-(A)$, where $\hat{p}_1 = OUT_v$ and $\hat{p}_k = v$. After v gets aborted in action A, edge $(\hat{p}_{k-1}, \hat{p}_k)$ starts to shrink, with a "$D(S)$" message going from $\hat{p}_k = v$ to $\hat{p}_{k-1}$. $CH(v,v)$ becomes $(p_1, \ldots, p_{k-1})$, which is now being chased by the newly created token "$T(v,S)$". It is clear that p_{k-1} satisfies the three conditions in Definition 14 and so the updated $CH(v,v)$ is still regular.

Next, consider the case where v has already been aborted by time $t^-(A)$. We show that $CH(v,v)$ remains regular after action A. Let $CH(v,v) = (p_1, \ldots, p_k)$ at time $t^-(A)$. It is sufficient to consider only those actions which may change $CH(v,v)$'s structure:

- $A = A_2^{\hat{p}_k}$ ($\hat{p}_k$ sends a "$D(S)$" message, with $v \in S$, to disconnect edge $(\hat{p}_{k-1}, \hat{p}_k)$): In this case, p_k is discarded by $\hat{p}_k$ and $CH(v,v)$ reduces to $(p_1, \ldots, p_{k-1})$, which is evidently regular.
- $A = A_6^{p_k}$ (a "$D(S)$" message arrives at $\hat{p}_k$): After the action, $CH(v,v)$ is unchanged but p_k becomes waiting and $v \in S$ is added to $BAR_{\hat{p}_k}$. $CH(v,v)$ remains regular.
- $A = A_8^{\hat{p}_j}$, $1 \leq j \leq k$ ($\hat{p}_j$ aborts itself): At time $t^-(A)$, $Proj(CH(\hat{p}_j, \hat{p}_j))$ forms a deadlock (by Lemma 6 and the induction hypothesis of this lemma). This can happen only for $j = k$ (as

a consequence of Lemma 5(b)). Now, as part of the deadlock, $\hat{p}_k$ has a non-shrinking outgoing edge. Since $(p_1, \ldots, p_k)$ is regular, p_k must be in the *waiting* status and $v \in BAR_{\hat{p}_k}$. When $\hat{p}_k$ aborts itself, it sends a "$D(S)$" message to $\hat{p}_{k-1}$ (if $k > 1$) with $v \in S$. Thus, after the action, $CH(v, v) = (p_1, \ldots, p_{k-1})$ remains regular.

• $A = A_9^{\hat{p}_1}$ (the chasing token "$T(x, S)$" arrives at $\hat{p}_1$): After the action, $CH(v, v) = (p_2, \ldots, p_k)$, which is regular. □

The above proof also establishes the following.

Corollary 6 *Once v is aborted, $CH(v, v)$ does not grow (propagate) any more.*

Theorem 1 *If in execution of A_8^v a process v claims to have detected a deadlock, then $Proj(CH(v, v))$ forms a deadlock at time $t^-(A_8^v)$.*

Proof. It follows directly from Lemmas 6 and 7. □

4.3 Proof of Progress Condition

This section proves the progress condition that any real deadlock cycle is eventually detected. Our plan of proof is as follows.

Suppose $X = (x_1, \cdots, x_k)$ is a deadlock cycle, and $P = CH(x_1, x_1) = (p_2, \ldots, p_k)$ is the probe chain initiated by x_1. We describe the status of P by a 6-tuple $M(P)$ that indicates the length of P, whether the last probe of P is in a channel or in a QUE, etc. Then we show that the value of $M(P)$ always decreases unless deadlock cycle X is detected. Since the domain of possible values of $M(P)$ is finite and non-negative, deadlock cycle X is eventually detected.

The following lemma shows that if $X = (x_1, \ldots, x_n)$ is a deadlock cycle then the probe chain initiated by x_1 is located at $(x_2, x_3, \ldots)$.

Lemma 8 *If $X = (x_1, \ldots, x_n)$ is a deadlock cycle and $CH(x_1, x_1) = (p_2, \ldots, p_k)$, where x_1 is smallest among all x_i's, then $CH(x_1, x_1) \neq \epsilon$ and $\hat{p}_i = x_i, 2 \leq i \leq k$.*

Proof. By assumption, $x_1 \in IN_{x_2}$. When the last time x_1 was inserted into IN_{x_2}, apparently in an A_5 action at x_2, a new probe $p = (x_1, x_1, \cdot, \cdot)$ was added to QUE_{x_2}. Since then, x_2 has not received any token "$T(\cdot, S)$" such that $x_1 \in S$, or x_1 would have seen a probe initiated by itself and therefore gotten aborted. Thus, p is still in QUE_{x_2}. By Lemma 4, p must be in a chain $CH(x_1, v)$ for some process v. Since p is not a copy (child) of any other probe, it is the first element of $CH(x_1, v)$. Furthermore, $CH(x_1, v)$ does not have a chasing token, and so $CH(x_1, v)$

equals $CH(x_1, x_1)$, with $p = p_2$. This proves $CH(x_1, x_1) \neq \epsilon$ and $k \geq 2$. That $\hat{p}_i = x_i$ for $\leq i \leq k$ follows from these facts: 1) $\hat{p}_2 = x_2$, 2) $OUT_{x_i} = x_{i+1}$ (by Definition 1), and 3) $OUT_{\hat{p}_i} = \hat{p}_{i+1}$ for $2 \leq i < k$ (by Lemma 2). □

Suppose $X = (x_1, \cdots, x_k)$ is a deadlock cycle, and $P = CH(x_1, x_1) = (p_2, \ldots, p_k)$ is the probe chain initiated by x_1. We want to show that P keeps growing until it reaches x_1 and thus detects the deadlock. There are two possible reasons P may not grow: (i) p_k is not a primary probe, or (ii) $BAR_{x_k} \neq \emptyset$. By Fact 1(4a) and Lemma 4, each barrier u in BAR_{x_k} is associated with a probe chain $CH(u, u)$. Thus the set of all probe chains $CH(u, u)$ that have passed through x_k provides the needed information about BAR_{x_k}, where u is an aborted process. This set of probe chains is denoted as $Block(P)$, as formally defined in the following.

Definition 15 If $X = (x_1, \ldots, x_n)$ is a deadlock cycle and $P = CH(x_1, x_1) = (p_2, \ldots, p_k)$, then define the following. (Recall from Lemma 8 that $\hat{p}_i = x_i$, $2 \leq i \leq k$.)

- $Block(P) = \{CH(u, u) : u$ is an aborted process with $x_k \in Proj(CH(u, u))\}$.
- $block(P)$ = the probe chain in $Block(P)$ whose initiator is smallest among all probe chains in $Block(P)$; undefined if $Block(P) = \emptyset$.
- The *state* of P is a 6-tuple $M(P) = (m_1, m_2, m_3, m_4, m_5, m_6)$ where
  - $m_1 = n - k$, i.e., the distance between x_k and x_n in X.
  - $m_2 = 0$ if $p_k \in QUE_{x_k}$; 1 otherwise (i.e., if p_k is still in channel $c[x_{k-1}, x_k]$.
  - m_3 = the number of messages preceding p_k if $m_2 = 1$; 0, otherwise;
  - $m_4 = |Block(P)|$;
  - m_5 = the distance between the chasing token of $block(P)$ and x_k; 0 if $block(P)$ is undefined.
  - m_6 = the number of messages preceding the chasing token of $block(P)$; 0 if $block(P)$ is undefined.
- Let $\Gamma(P)$ denote the set of all possible values of $M(P)$, and let "$<$" denote the lexicographic order between any two elements in $\Gamma(P)$. □

Lemma 9 *Let $X = (x_1, \ldots, x_n)$ be a deadlock cycle with x_1 being smallest among all x_i's. If $P = CH(x_1, x_1) = (p_2, \ldots, p_k)$, then the following are true.*

(a) If p_k is in QUE_{x_k}, then p_k is in the waiting status.

(b) If p_k, the last element of P, does not change in an action (i.e., it remains as the last element of P), then $Block(P)$ does not get any new element.

(c) If $Block(P) = \emptyset$, then $BAR_{x_k} = \emptyset$ and p_k is a primary.

Proof. (a) Assume that p_k is in the "sent" status. Since P is maximal, p_k has no non-barren child and thus (x_k, x_{k+1}) is a shrinking edge. This contradicts the assumption that X be a deadlock cycle, for every edge in a deadlock cycle is an established one.

(b) By Corollary 6, $CH(u,u)$ does not grow after u is aborted. Thus, the only possibility for $Block(P)$ to get a new element, say $CH(v,v)$, is that before an action x_k is in $CH(v,v)$ and in the action v aborts itself, thus making $CH(v,v)$ eligible to be in $Block(P)$. But in this case, $Proj(CH(v,v))$ would have formed a deadlock (by Theorem 1) before the action. This is possible only if $Proj(CH(v,v)) = X$, since both deadlock cycles pass a common node. Due to Fact 1(5a), v has the smallest identifier in $Proj(CH(v,v))$. Thus, $v = x_1$ and $CH(v,v) = P$. In the action under consideration, the last element of $CH(v,v)$ arrives at v and gets discarded when v gets aborted. This proves (b).

(c) Let $Block(P) = \emptyset$. We first show $BAR_{x_k} = \emptyset$. Assume to the contrary that BAR_{x_k} contains some barrier initiated by u (an aborted process). Then, by Fact 1(4a), QUE_{x_k} contains a probe, p, initiated by u. This probe is not barren since it is in a QUE. Therefore, according to Lemma 4, p is an element of $CH(u,v)$ for some aborted process v (v may or may not equal u). So, by Lemma 8, x_k is an element of $Proj(CH(u,v))$. Since the latter is a subsequence of $Proj(CH(v,v))$ (by Lemma 7), x_k is an element of $Proj(CH(v,v))$ and so $CH(v,v) \in Block(P)$, in contradiction to the assumption that $Block(P) = \emptyset$. This proves the first part of (c).

Now, to show p_k to be a primary, again assume $Block(P) = \emptyset$. If p_k is not primary, then there is a primary probe q in QUE_{x_k} that has the same initiator as p_k (i.e., $i(q) = x_1$). By Lemma 4, q belongs to $CH(x_1, v)$ for some process v, where either $v = x_1$ or v has been aborted. Probe q is not an element of P, since p_k is already in P, which is known to have no two probes from the same QUE. Therefore, q is an element of $CH(x_1, v)$ for some aborted v. The same argument as above for $CH(u,v)$ leads to $CH(v,v) \in Block(P)$, which is again a contradiction. □

Now we show that the algorithm always works in the direction of decreasing $M(P)$ for $P = CH(x_1, x_1)$, where x_1 is the smallest process in a deadlock cycle $X = (x_1, \ldots, x_n)$.

Lemma 10 *Let $X = (x_1, \ldots, x_n)$ be a deadlock cycle such that x_1 is smallest among all processes in the cycle, and let $P = CH(x_1, x_1) = (p_2, \ldots, p_k)$. For any $z > (0,0,0,0,0,0)$ in $\Gamma(P)$,*

$$\{deadlock(X) \wedge M(P) = z\} \textbf{ ensures } \{(deadlock(X) \wedge M(P) < z) \vee (\neg deadlock(X))\}$$

Proof. It suffices to prove the following two statements.

(a) For any action A, if $\{deadlock(X) \wedge M(P) = z\}$ is true immediately before A, then $\{(deadlock(X) \wedge M(P) \leq z) \vee \neg deadlock(X)\}$ is true after the action.

(b) Whenever $\{deadlock(X) \wedge M(P) = z\}$ is true, there is an executable action A such that $\{(deadlock(X) \wedge M(P) < z) \vee \neg deadlock(X)\}$ is true after A.

To prove (a), let A be any action and $z = (z_1, z_2, z_3, z_4, z_5, z_6) > 0$ be any element in $\Gamma(P)$ such that $\{deadlock(X) \wedge M(P) = z\}$ is true before action A. Recall that $M(P) = (m_1, m_2, m_3, m_4, m_5, m_6)$. We first show that if $deadlock(X)$ remains true then (m_1, m_2) does not increase in A.

Case 1: $m_2 = 1$ at time $t^-(A)$. In this case, p_k is in channel $c[x_{k-1}, x_k]$. If the value of m_2 does not change in the action, then m_1 does not change either. Should m_2 reduce to 0 (i.e., p_k arrive at x_k) in action A, then there are two possibilities: (i) if $m_1 > 0$, then m_1 does not change; (ii) if $m_1 = 0$, then A is an A_8 in which the deadlock is detected and resolved and $deadlock(X)$ becomes false.

Case 2: Suppose $m_2 = 0$ at time $t^-(A)$. It's obvious that if m_2 increases to 1 (i.e., x_k sends a copy of p_k to x_{k+1}), then m_1 decreases by 1 (as the last element of $block(P)$ is now p_{k+1} rather than p_k). So it is sufficient to consider only the case where m_2 does not change during the action. Under the condition that m_2 remains intact, the value of m_1 may possibly increase only if p_k is removed, which may possibly occur only in action A_2, A_4, or A_9. A_2 and A_4 actually cannot happen, since $OUT_{x_k} \neq null$. In an A_9 action, p_k can be removed from QUE_{x_k} only if the received token $T(\cdot, S)$ comes from x_{k-1} with $x_1 \in S$. This is impossible, however, since channel $c[x_{k-1}, x_k]$ ($c[x_{k-1}, x_k]$) is clean (by Lemma 2). Thus, an A_9 action by x_k can not remove p_k.

We have shown that (m_1, m_2) never increases if X remains deadlocked. If (m_1, m_2) decreases, we are done. So assume that (m_1, m_2) does not change value during action A. In this case, m_3 cannot increase, and by Lemma 9(b), m_4 cannot increase, either. Thus, (m_1, m_2, m_3, m_4) does not increase. Now suppose (m_1, m_2, m_3, m_4) does not change value during action A (otherwise it decreases and we are done). It's obvious that (m_5, m_6) will not increase in A. Therefore, $M(P)$ never increases, and (a) is proved.

Now we prove (b). The case where $(m_2, m_3) > 0$ is trivial. So assume $(m_2, m_3) = (0, 0)$, and thus p_k is in the QUE of process x_k. Furthermore, assume $m_1 > 0$, or the deadlock has been detected and we are done.

Case 1: $m_4 = 0$. In this case, Lemma 9 indicates that $BAR_{x_k} = \emptyset$ and p_k is in the primary and waiting status. Since $OUT_{x_k} = x_{k+1} \neq null$ and $i(p_k) = x_1$ is smallest in X, an A_7 action by x_k will propagate p_k and reduce m_1.

Case 2: $m_4 > 0$. If $m_6 > 0$, it's obvious that m_6 will eventually become 0. If $m_6 = 0$, the arrival of the chasing token of *block*(P) will reduce the value of (m_4, m_5). □

The following theorem establishes the progress condition.

Theorem 2 (Progress condition) *If X is a deadlock cycle, then deadlock(X) eventually becomes false.*

Proof. By Lemma 10, either the value of $M(P)$ eventually becomes $(0,0,0,0,0,0)$, forcing *deadlock*(X) to become false; or *deadlock*(X) becomes false before $M(P)$ reduces to $(0,0,0,0,0,0)$. □

5 Performance

Our main concern of performance is the *deadlock persistence time*, DPT, which is defined in [18] as the period from the time a deadlock occurs to the time it is resolved. The shorter the DPT, the less the system degrades due to deadlocks. DPT consists of two components: the *deadlock detection time (DT)* and the *deadlock resolution time (RT)*. DT denotes the period from the time a deadlock occurs to the time it is detected. RT covers the period from the time the deadlock is detected to the time it is resolved. Thus, $DPT = DT + RT$. We measure the length of a time period by the number of messages that need to be sent sequentially during the period. In the case of DT, it equals the number of remaining hops of the probe that leads to the detection of the deadlock, counting from where the probe locates at the time when the deadlock occurs.

Let L be the average length of a deadlock cycle. Assume that at the time when a deadlock occurs, the probe initiated by the process with the smallest identifier in the deadlock cycle has the same probability to be at any process in the cycle. Then, for our algorithm, $DT = 0.5L$, $RT = 0$, and $DPT = 0.5L$. For previous algorithms [3, 9, 19], $DT = 0.5L$, $RT = L$, and $DPT = 1.5L$. Our algorithm is three times as fast as previous ones in detecting and resolving deadlocks.

6 Conclusion

An important performance criterion for deadlock detection and resolution is the deadlock persistence time – the time from the formation of a deadlock to its resolution. After detecting a deadlock, previous probe-based algorithms do not resolve the deadlock until all the obsolete probes in the deadlock cycle have been removed, which considerably prolongs the deadlock persistence time. We have proposed a scheme to remedy this drawback. Our algorithm resolves a deadlock right away upon detecting the deadlock, before the obsolete probes are removed. This reduces the average deadlock persistence time by as much as two thirds. We have only considered the model where each node has at most one outgoing dependency edge. If a node may have multiple outgoing edges, how to reduce the deadlock persistence time without introducing phantom deadlock detections is still an open problem.

References

[1] K.M. Chandy and J. Misra, "A distributed algorithm for detecting resource deadlocks in distributed systems," *Proc. Symp. on Principles of Distributed Computing*, Aug. 1982, pp. 157-164.

[2] K.M. Chandy, J. Misra, and L.M. Haas, "Distributed Deadlock Detection," *ACM Trans. on Computer Systems*, May 1983, pp. 144-156.

[3] A.N. Choudhary, W.H. Kohler, J.A. Stankovic, and D. Towsley, "A Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution," *IEEE Trans. on Software Eng.*, Vol. 15, No. 1, Jan. 1989, pp.10-17.

[4] A.N. Choudhary, W.H. Kohler, J.A. Stankovic, and D. Towsley, "Correction to 'A Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution'," *IEEE Trans. on Software Eng.*, Vol. 15, No. 12, Dec. 1989, p. 1644.

[5] A.K. Elmagarmid, N. Soundararajan, and M.T. Liu, "A Distributed Deadlock Detection and Resolution Algorithm and Its Correctness," *IEEE Trans. on Software Eng.*, Vol. 14, No. 10, Oct. 1988, pp. 1443-1452.

[6] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notion of Consistancy and Predicate Locks in a Database System," *Communications of the ACM*, Vol. 19, Nov. 1976, pp. 624-633.

[7] S.-T. Huang, "A distributed deadlock detection algorithm for CSP-like Communication," *ACM TOPLAS*, Vol. 12, No. 1, Jan. 1990, pp.102-122.

[8] E. Knapp, "Deadlock Detection in Distributed Database Systems," *ACM Computing Surveys*, Vol. 19, No. 4, Dec. 1987, pp. 303-327.

[9] A.D. Kshemkalyani and M. Singhal, "Invariant-Based Verification of a Distributed Deadlock Detection Algorithm," *IEEE Trans. on Software Eng.*, Vol. 17, No. 8, August 1991, 789-799.

[10] L. Lamport, "An Assertional Correctness Proof of a Distributed Algorithm," *Science of Computer Programming*, Vol. 2, 1982, pp. 175-206.

[11] L. Lamport, "A theorem on atomicity in distributed algorithms," *Distributed Computing,* vol. 4, pp. 59-68, 1990.

[12] D.E. Menasce and R.R. Muntz, "Locking and Deadlock Detection in Distributed Databases," *IEEE Trans. on Software Eng.*, Vol. 5, No. 3, May 1979, pp. 195-202.

[13] D.P. Mitchell and M.J. Merritt, "A distributed algorithm for deadlock detection and resolution," *Proc. of the ACM Symp. on Principles of Distributed Computing*, 1984, pp. 282-284.

[14] M. Roesler and W.A. Burkhard, "Resolution of Deadlocks in Object-Oriented Distributed Systems," *IEEE Trans. Computers*, Vol. 38, No. 8, Aug. 1989, pp. 1212-1224.

[15] B. Sanders and P.A. Heuberger, "A Distributed Deadlock Detection and Resolution with Probes," *proc. of 3rd Int. Workshop on Distributed Algorithms*, Sept. 1989.

[16] K. Shafer and M. Singhal, "A Correct Priority-based Probe Algorithm for Distributed Deadlock Detection and Resolution and Proof of Its Correctness," Technical Report OSU-CISRC-4/89-TR16, Computer and Information Science, Ohio State Univ., 1989.

[17] B. Shyam and D.M. Dhamdhere, "A New Priority Based Probe Algorithm for Distributed Deadlock Detection," IIT Bombay Technical Report, 1990.

[18] M. Singhal, "Deadlock Detection in Distributed Systems," *IEEE Computer*, Nov. 1989.

[19] M.K. Sinha and N. Natarajan, "A priority based distributed deadlock detection algorithm," *IEEE Trans. Software Eng.*, Vol. 11, No. 1, Jan. 1985, pp. 67-80.

[20] N. Soundararajan, "Axiomatic semantics of CSP," *ACM TOPLAS*, Vol. 6, 1984, pp. 647-662.