Invariant-Based Verification of a Distributed Deadlock Detection Algorithm

Ajay D. Kshemkalyani, *Student Member, IEEE*, and Mukesh Singhal, *Member, IEEE*

Abstract— **Most previous proposals for distributed deadlock detection are incorrect because they have used informal/intuitive arguments to prove the correctness of their algorithms. Informal and intuitive arguments are prone to errors because of the highly complex nature of distributed deadlock detection/resolution algorithms. In this paper we first correct the priority-based probe algorithm for distributed deadlock detection and resolution of Choudhary *et al.* [3] and then formally prove that the modified algorithm is correct (i.e., it does detect all deadlocks and does not report phantom deadlocks). Our proof technique is novel because we first abstract the properties of the deadlock detection and resolution algorithm by invariants, and then show that the invariants imply the desired correctness of the algorithm. This is the first attempt at a formal proof of the correctness of a distributed deadlock detection and resolution algorithm.**

Index Terms— **Correctness proof, distributed algorithms, distributed databases, distributed deadlock detection, invariant.**

I. Introduction

Proof of the correctness of distributed deadlock detection algorithms is very difficult because of the highly complex operation of these algorithms. This is the primary reason why most previous proposals for distributed deadlock detection have either: (i) ignored the correctness proof [16], (ii) have given informal/intuitive argument of correctness [6], [9], [10], or (iii) have used a simulation technique to show correctness [3]. Informal arguments are prone to errors, and correctness verification by simulation is also prone to errors because it is not complete (there is no guarantee that the simulation will pass through all possible states of the algorithm and traverse all paths). That is why so many distributed deadlock detection algorithms have been shown to be incorrect [7], [15].

It is only recently that attention has been paid to a rigorous correctness proof of distributed deadlock detection algorithms (e.g., [4], [8], [11], [13], [14]). However, due to the highly complex operation of distributed deadlock detection/resolution algorithms and lack of an underlying theory for deadlock detection/resolution, most of these correctness proofs have used either *ad hoc* methods (e.g., [4], [11]) or operational arguments (e.g., [13], [14]) for proof. This paper is geared towards more systematic and elegant proofs of distributed deadlock detection algorithms.

In this paper we give a correctness proof of a priority-based probe algorithm for distributed deadlock detection and resolution [3]. A priority-based deadlock detection and resolution algorithm assigns priorities to transactions, and uses these priorities to reduce the number of messages exchanged to detect a deadlock. We have chosen a priority-based probe algorithm due the following reasons: (i) recently, there has been considerable interest in priority-based probe algorithms for distributed deadlock detection, and they represent a wide class of deadlock detection algorithms (e.g., [1], [3], [11]–[14], [16], [17]), (ii) probe-based algorithms are elegant in that they do not require construction of transaction-wait-for graph (as in [10]), and (iii) they are efficient because a deadlock detection is initiated only if a higher priority transaction is blocked by a lower priority transaction (all deadlock detections initiated by lower priority transactions are suppressed).

We have chosen Choudhary *et al.*'s deadlock detection/resolution algorithm [3] for correctness proof because it is a classical example of the propagation of errors in distributed deadlock detection and resolution algorithms (and it also reflects how intuitive arguments for correctness can be very misleading). Note that in [3], Choudhary *et al.* showed that the priority-based probe algorithm for deadlock detection/resolution of Sinha and Natarajan [16] is incorrect and fails to detect all deadlocks, and may report false deadlocks. Choudhary *et al.* presented an algorithm [3] which they claim rectifies the problems of the Sinha and Natarajan algorithm [16]. However, problems have been reported in Choudhary *et al.*'s algorithm [2], [13], [14], and it fails to detect all deadlocks and may report false deadlocks.

In this paper we first eliminate deficiencies of the priority-based probe algorithm for distributed deadlock detection and resolution of Choudhary *et al.* [3]. We identify the correctness requirements of a probe-based deadlock detection algorithm, viz., necessary and sufficient conditions to be satisfied to detect all deadlocks and to not report false deadlocks. We then formally prove the correctness of the modified algorithm; i.e., show that it does detect all deadlocks and does not report false deadlocks. Our proof technique is novel because we first abstract the properties of the deadlock detection and resolution algorithm by invariants, and then use these invariants to show that the necessary and sufficient correctness conditions are satisfied.

This is the first attempt at a formal proof of the correctness of a distributed deadlock detection and resolution algorithm.

The rest of the paper is organized as follows: In the next section we describe a model of the database used by the algorithm. In Section III we correct Choudhary *et al.*'s priority-based probe algorithm for distributed deadlock detection and

Manuscript received April 7, 1989; revised March 28, 1991. Recommended by E. Gelenbe.

The authors are with the Department of Computer and Information Science, The Ohio State University, 2036 Neil Avenue Mall, Columbus, OH 43210.

IEEE Log Number 9101140.

Fig. 1. Wait-for dependencies between transactions and a data manager.

resolution and present a modified algorithm. Section IV contains a proof of the correctness for the modified algorithm. Finally, Section V contains concluding remarks.

II. A Model of Distributed Database Systems

A distributed database system is a collection of sites connected by a communication network. There is no global memory, so sites communicate by passing messages. The communication network is assumed to be reliable (i.e., messages are delivered error-free without ever being lost) and messages arrive in the order sent. A site may contain many data items of the database. However, to simplify the presentation we assume that a data item X_i has a unique data manager DM_i which has the exclusive right to operate or grant locks on it. Each data manager DM_i maintains a queue, called the Request_Q (and denoted by $Request\_Q_i$), of transactions which are yet to be granted locks on the data item.

Transactions use two-phase locking for concurrency control. A transaction may access a data item only after it has obtained a lock on that data item. If a transaction wishes to access the data item X_i, it sends a lock request to DM_i. (The transaction is then called a *requester* of X_i.) DM_i immediately grants this lock request if no other transaction currently holds a lock on X_i. If X_i is currently locked by a transaction (called the *holder* of X_i), then the *requester* is placed in the Request_Q of DM_i. When the *holder* of X_i releases it, DM_i grants the lock request of one of the transactions in its Request_Q, if the Request_Q is not empty. This second transaction will be referred to as the $newholder$.

Fig. 1 depicts a situation where the transaction T_i is waiting for data item X_k (managed by DM_k) which is currently held by transaction T_j. (The notation in Fig. 1 is slightly different from the notation in the original algorithm [16]. However, we feel that explicit indication of the data item under contention enhances the understanding of the algorithm.) Each data item can only be accessed by one transaction at a time (i.e., accesses are exclusive). As discussed by Sinha and Natarajan [16], this restriction can be removed to allow for shared access.

A transaction T_i maintains a queue, called a *Probe_Q* (and denoted by $Probe\_Q_i$), which contains probes received by it. This queue contains information relating to transactions that are waiting on it either directly or transitively. Every transaction can be in one of two states: *active* or *waiting*. A transaction goes from active to wait state when it issues a lock request to access a data item. A transaction goes from wait to active state when it receives a grant of its lock request.

Each transaction is assigned a distinct priority. For two transactions T_i and T_j, Priority(T_i)>Priority(T_j), simply written as $T_i > T_j$, iff $i < j$. In probe-based algorithms a deadlock is detected by circulating a *probe* through the deadlock cycle.

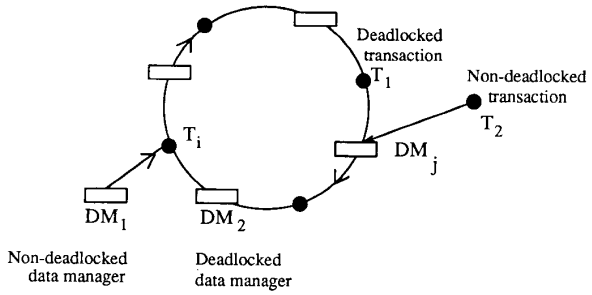

Fig. 2. Nondeadlocked and deadlocked data managers and transactions.

A probe is an ordered pair (*initiator, junior*), where $initiator$ denotes a transaction that has faced an *antagonistic conflict* and $junior$ represents the lowest priority transaction that the probe has traversed. (An antagonistic conflict occurs between two transactions, T_i and T_j, if T_i requests a lock on a data item for which T_j is the $holder$ and $T_i > T_j$.) Priority-based algorithms are more efficient because a probe is initiated (and propagated) only if an antagonistic conflict occurs.

When a transaction is involved in a deadlock, only one of the data managers where it is a holder is part of the deadlock cycle. We call such a data manager the *deadlocked* data manager of that transaction. Other data managers are called *nondeadlocked* data managers of that transaction. For example, in Fig. 2 for transaction T_i, DM_2 is the deadlocked data manager and DM_1 is the nondeadlocked data manager. Similar to this definition, we can have *deadlocked* and *nondeadlocked* transactions for a data manager. For example, in Fig. 2, for data manager DM_j, T_1 is a deadlocked transaction and T_2 is a nondeadlocked transaction.

III. A Modified Priority-Based Deadlock Detection Algorithm

In this section we present a priority-based deadlock detection and resolution algorithm which is a corrected version of Choudhary *et al.*'s algorithm [3]. We first give a brief description of Choudhary *et al.*'s algorithm, then point out problems with it and present remedies, and then finally present a probe-based deadlock detection and resolution algorithm which eliminates the deficiencies of their algorithm [3].

Choudhary *et al.*'s algorithm has two phases: the deadlock detection phase and deadlock resolution phase. In the deadlock detection phase, a deadlock detection is initiated only if a higher priority transaction is blocked by a lower priority transaction. Deadlock initiation consists of the generation of a probe message and its propagation in the wait-for path or cycle. A deadlock is detected when the probe initiated on behalf of a transaction returns to it. (In a deadlock cycle this happens only with the highest priority transaction, because a transaction suppresses all the probes initiated on the behalf of lower priority transactions.) The second phase in which the deadlock is resolved is entered only after a transaction has detected a deadlock. The detector of a deadlock resolves the deadlock by aborting the lowest priority transaction (called *victim*) in the

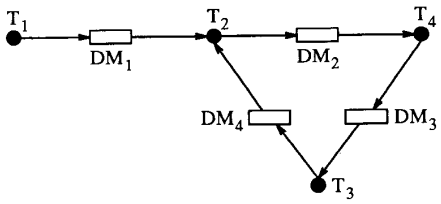

Fig. 3. An example.

cycle. Note that all the probes sent on behalf of the victim or propagated by it must be cleaned from the system. This is accomplished by propagating a "clean" message through the cycle and reinitiating the valid probes that got cleaned (by the clean message).

Although Choudhary *et al.*'s algorithm has fixed most of the problems of the Sinha and Natarajan algorithm, it has the deficiencies listed below.

A. Insufficient Regeneration of Probes

In Choudhary *et al.*'s algorithm, after the Probe_Q of a transaction has been purged by a clean message, it is not rebuilt properly. The problem is that a transaction rebuilds its Probe_Q only with the help of the data manager which is part of the resolved deadlock, and does not rebuild the Probe_Q with the help of its nondeadlocked data managers. Consequently, it fails to regenerate probes which were initiated or propagated by nondeadlocked data managers. The loss of these probes may result in a failure to detect a later deadlock involving transactions waiting directly or transitively on a transaction in the currently resolved deadlock cycle, as pointed out in [13] and [14].

For example, in Fig. 3, suppose that deadlock has been resolved by aborting T_4 after T_4 has circulated a clean message in the cycle. In this example, after the probe (T_1, T_2) has been purged from the Probe_Q of T_2, the probe (T_1, T_2) never gets re-sent to T_2. This is because after transaction T_2 in the deadlock cycle has purged its Probe_Q in response to the clean message, it does not rebuild its Probe_Q with the help of the data manager DM_1, where it is a holder. Probe (T_1, T_2) never gets replaced in the Probe_Q of T_2 after it has been purged by the clean message from T_4. This will result in an undetected deadlock if later T_2 or any of its successors waits on T_1.

Remedy: For correctness, a transaction must rebuild its Probe_Q with the help of not only the data manager in the resolved deadlock cycle, but also with the help of all other (nondeadlocked) data managers for which it is a holder [13]. For example, in Fig. 3, after the Probe_Q of transaction T_2 has been purged, T_2 should also use the information at data manager DM_1 (besides the information at data manager DM_4) to rebuild its Probe_Q.

B. Transmission of Invalid Probes by a Victim

In Choudhary *et al.*'s algorithm, a *victim* enters the abort phase only after it receives the clean message it initiated. It is only after entering the abort phase that a victim starts discarding any message that it receives. However, it handles all the messages normally before entering the abort phase. That is, it may propagate some of the probes after it has sent a clean message, but before receiving the clean message! Note that this will result in improper cleaning of probes and may cause detection of false deadlocks. (Shyam and Dhamdhere have shown that this algorithm indeed detects false deadlocks [14].)

For example, in Fig. 3, suppose T_2 is the victim chosen to resolve the deadlock. Consider the following sequence of actions: (i) T_2 sends a "clean" message to DM_2 and waits for it to return before aborting. (ii) In the meanwhile, it receives probe (T_1, T_2) from DM_1 and forwards the probe to DM_2. (iii) T_2 now receives its own "clean" message and aborts. (iv) T_4 becomes active (when T_3 releases DM_3) and starts waiting on T_1 through data manager DM_5 (not shown in Fig. 3). (v) T_4 receives the probe (T_1, T_2) from DM_2 and forwards it to DM_5 (where T_1 is the *holder*). (vi) DM_5 declares the false deadlock cycle (T_1, T_2, T_4).

Remedy: This problem can be taken care of by putting the following restriction: "**A victim discards all the messages which it receives after it has sent out a 'clean' message.**" This will correct the problem because no probe can get past a victim after it has initiated a clean message for cleaning the probes which denote nonexistent dependencies. For example, in Fig. 3, after T_2 sends out a "clean" message (step (i)), it does not forward the probe (T_1, T_2) it receives.

C. A Remark

Since a transaction uses a two-phase locking algorithm, the information in its Probe_Q remains pertinent until the transaction commits or aborts. After a transaction has entered the second phase of locking (i.e., releasing of locks has begun), it can no longer be involved in a deadlock cycle because it will not request any more locks. Therefore as a minor change to Choudhary *et al.*'s algorithm, *a transaction will discard its Probe_Q when it enters the second phase of locking.* Note that keeping Probe_Q even after entering the second phase does not cause any problem. This change is primarily made to simplify the definition of irrelevant probes later and consequently the proof of correctness. Without this change the proof would be more complicated.

Next, we present a modified deadlock detection and resolution algorithm which eliminates the deficiencies of Choudhary *et al.* s' algorithm [3]. (All the changes to the algorithm are underlined.)

D. The Modified Deadlock Detection/Resolution Algorithm

Deadlock detection occurs in three basic steps: data manager initiation of probes, sending and receiving of probes by transactions, and reception of probes by data managers.

1) Data Manager Initiation of a Probe: A data manager initiates, propagates, or reinitiates a probe in the following situations:

1) When a data item is locked by a transaction, if a lock request arrives from a transaction of higher priority, the data manager *initiates* the probe $(requester, holder)$

and sends it to the *holder*. (This is a case where the $requester$ faces an antagonistic conflict.)

2) A transaction releases a data item during its second phase in the two-phase locking protocol, or when it aborts. When the current *holder* releases a data item, the data manager performs the actions: (a) schedules a waiting lock request, called *newholder*, and removes *newholder* from its Request_Q. (Any policy can be used to select a *newholder*.) (b) Initiates the probe $(requester, newholder)$ for each lock request in its Request_Q such that $requester > newholder$, and sends it to the *newholder*.
3) When the current *holder* releases a data item, the data manager requests all transactions in its Request_Q to retransmit their complete Probe_Qs to itself. It then forwards each probe (*initiator, junior*) to the $newholder$, such that $initiator > newholder$. (This allows probes which would have otherwise been lost with the previous holder to be propagated to the $newholder$.)

2) Receiving and Sending of Probes by a Transaction: Recall that the *Probe_Q* of a transaction contains information about transactions that are waiting on it either directly or transitively. Since a transaction uses a two-phase locking protocol, the information in its Probe_Q remains valid until the transaction commits or aborts.

After a transaction enters the second phase of locking it does not need its Probe_Q any more. A probe received in the second phase is ignored. Otherwise, if the transaction is in the first phase, it sends a copy of its Probe_Q to its data manager where its waiting in the following three cases:

1) When a transaction T receives the probe (*initiator,junior*), it performs the following:

```
if (junior > T)
then
    junior := T;
    save probe in the Probe_Q;
if (T is in wait state)
then
    transmit a copy of the saved
    probe to the data manager
    where it is waiting;
```

2) When a transaction issues a lock request to a data manager and waits for the lock request, it transmits a copy of its Probe_Q to that data manager.
3) If a transaction is waiting and receives a request for its Probe_Q from the manager where it is waiting, it sends a copy of all the probes in its Probe_Q to that data manager.

3) Reception of a Probe by a Data Manager: When a data manager receives the probe ($initiator$, $junior$), it performs the following actions:

```
if (holder > initiator)
then discard the probe;
else if (holder < initiator)
    then propagate probe to the holder
    else declare deadlock and initiate
        deadlock resolution;
```

When a deadlock is detected, the detecting data manager has the identity of two transactions in the cycle, $initiator$ and $junior$. $Junior$ is chosen by the data manager as the $victim$ (i.e., the transaction to be aborted to resolve the deadlock)

E. Deadlock Resolution

To resolve a deadlock, not only must the victim be aborted, but all Probe_Q's of transactions that received probes sent on behalf of the victim or propagated by it must be updated. This is accomplished by propagating a "clean" message through the cycle and rebuilding the Probe_Q's that got cleaned. Resolution can be explained in four basic steps: roll-back of the victim, handling of a clean message by a data manager, handling of a clean message by a transaction, and handling of a rebroadcast message by a data manager.

1) Roll-back of the Victim: To abort the $victim$, the data manager that detected the deadlock must send an abort_signal, abort ($junior$, $initiator$) to the $victim$. Before the $victim$ is aborted, it is necessary to update all Probe_Q's of transactions in the cycle. Hence on receiving an abort_signal, the $victim$ takes the following two actions:

1) It initiates a message, clean ($victim$, $initiator$), and sends it to the data manager where it is waiting. It discards any message that it receives from now on.
2) The $victim$ enters the abort phase only when it receives the clean message it originated. Once it enters the abort phase, the $victim$ releases all the locks it holds, withdraws its pending request by having its id removed from the Request_Q of the data manager where it is waiting, and aborts.

2) Handling of a Clean Message by a Data Manager: When a data manager receives a clean ($victim$, $initiator$) message, it does the following:

1) Propagates the clean message to its $holder$.
2) Reinitiates probes for each $requester$ in its Request_Q, such that $requester > holder$. (This reinitiation of probes is necessary as the $holder$'s Probe_Q was cleaned by the clean message.)
3) It requests each nondeadlocked transaction in its Request_Q to retransmit its Probe_Q.

Each transaction in its Request_Q, except the one from which it received the clean message, is a nondeadlocked transaction. Nondeadlocked data managers and nondeadlocked transactions can be identified by adding an extra field indicating the last data manager or transaction that forwarded the clean message, to the clean message, or from the information derivable from the communication protocol. For simplicity, we assume such information is available.

3) Handling of a Clean Message by a Transaction: A correction is needed in this part of Choudhary *et al.* 's algorithm so that after a transaction has purged its Probe_Q, it rebuilds its Probe_Q properly (i.e., with the help of all data managers where it is a holder). In order to do this, after purging its Probe_Q, a transaction requests all of its nondeadlocked data managers (by sending them a rebroadcast message) to reinitiate and propagate probes on behalf of all the transactions

in its Request_Q. The transaction then uses this additional information to help rebuild its Probe_Q.

When a transaction T receives a clean $(junior, initiator)$ message, it acts as follows:

```
purge every probe from its Probe_Q;
send a Rebroadcast Request to all nondeadlocked
data managers at which T currently holds a lock;
if(T is waiting)
then
    if (T = junior)
    then
        enter the abort phase and
        release all locks
    else propagate the clean message to the
        data manager where T is waiting
else discard the clean message
```

All data managers at which a transaction currently holds a lock, except the data manager from which the clean message is received, are nondeadlocked data managers. Nondeadlocked data managers can be identified as explained in Section III-E.2..

4) Handling of a Rebroadcast Message by a Data Manager: When a data manager receives a rebroadcast message, it acts as follows:

```
if (Request_Q not empty)
then
    request each transaction in Request_Q
    to retransmit Probe_Q and reinitiate
    probes for each requester in Request_Q
    such that requester > holder
else discard the rebroadcast message.
```

Thus in response to a rebroadcast message, a data manager reinitiates all those probes which were initiated or propagated by it on behalf of the transactions in its Request_Q before the *holder* purged its Probe_Q. Note that all these probes genuinely belong to the Probe_Q of the *holder*, because their initiator transactions are still waiting. However, the data manager may reinitiate/retransmit some duplicate probes, because not all of the probes it had first initiated/retransmitted may have reached the *holder* when its Probe_Q was purged. We assume that all duplicate probes are discarded.

IV. Proof of Correctness

A. Preliminaries

We first abstract the state of the system by a wait-for graph referred to as the TWFG (transaction wait-for graph), where we associate each data manager DM_i with transaction T_j such that $T_j = holder(DM_i)$. A dependency from T_j to T_k through a data manager (where T_j is the *requester* and T_k is the *holder*) is represented by an edge from T_j to T_k in the TWFG. This makes data managers transparent.[1]

[1] This abstraction is sufficient for our proofs even though some information is lost. Consider T_j and T_k waiting on T_i. When T_i releases a data item which is allocated to T_j, we have two distinct possibilities: (i) T_k continues waiting on T_i for a different data item, and (ii) T_k now waits on T_j, because T_k was waiting for the same data item as T_j was on T_i.

In the single-request model, the TWFG of the system is a forest of *isolated* trees. Whenever the root of a tree begins waiting on one of its descendants, the "tree" is deadlocked. At any instant, a "tree" can contain only a single deadlock. When a "tree" is deadlocked, all the transactions in the tree are blocked. This state persists until the deadlock is resolved, at which instant the tree structure is re-established. All concurrently existing deadlocks involve distinct sets of data items and transactions.

We give a formal description of the TWFG changes in the system using a graph model. All the nodes in the graph (i.e., transactions in the system) have a unique identifier. Let $R(i)$, a TWFG tree rooted at node i, be the set containing all the nodes in the tree. Only the root of a tree is active and can block on other nodes. An edge (i, j) forms when node i blocks on node j. When the root node i of a tree $R(i)$ blocks on a node in $R(i)$, the tree $R(i)$ transforms into a nontree denoted by $R_c(i)$. An edge (i, j) is removed when node j replies to node i or node i aborts. The predicate $abort(i)$ is used to indicate that node i has aborted after its last deadlock. The TWFG changes due to various actions in the system are described using state transitions in the Appendix.

A dependency path from i to j, denoted by $i \xrightarrow{*} j$, indicates that node i is waiting directly or transitively on node j. We now make the following two pertinent observations based on the system model:

Observation 1: If $i \xrightarrow{*} j$ and j does not lie in a deadlock, then no node on the path $i \xrightarrow{*} j$ lies in a deadlock.

Observation 2: If $i \xrightarrow{*} j$ and j does lie in a deadlock, then no node on the path $i \xrightarrow{*} j$ lies in *another* deadlock.

Since the existence of a dependency and a probe at a transaction manager/data manager varies with time, we use temporal logic operators [5] to get a convenient handle on the invariants for the algorithm. The operator $\bigcirc$ stands for "next step/instant assuming time is discrete." We use the "step" to mean a message hop. The operator $\Box$ stands for "henceforth."

We define a deadlock $C(R_c(k)) \subseteq R_c(k)$ to contain all the nodes in $R_c(k)$ that are reachable from themselves:

Definition 1: $C(R_c(k)) = \{i \mid i \in R_c(k) \bigwedge i \xrightarrow{*} i\}$.

Therefore $R_c(k) - \mathrm{C}(R_c(k))$ contains all the nodes of $R_c(k)$ that are not part of the deadlock cycle $\mathrm{C}(R_c(k))$. In the single request model, all the nodes reachable from a node in $C(R_c(k))$ belong to $C(R_c(k))$.

Remark 1: $\langle \forall i, \forall j : i, j \in C(R_c(k)) :: i \xrightarrow{*} k \xrightarrow{*} j \Longrightarrow (\Box(i \xrightarrow{*} k \xrightarrow{*} j)\ \underline{until}\ abort(k))\rangle$.

Next, we describe the features of a probe-based algorithm. A *probe-based* deadlock detection algorithm uses control messages (probes) of the form, probe $(initiator, \lambda)$, such that:

1) $initiator$ is the initiator of the probe message.
2) A probe carries only bounded information that is independent of the size of the system or the path traced by the probe.
3) A probe $(initiator, \lambda)$ at node j computes a function $\lambda = f_p(\langle nodes\ visited\ by\ probe \rangle)$.
4) A probe is only sent forward or backward along existing dependency edges in the TWFG.

5) A copy of a received probe that is not discarded is forwarded as soon as possible.
6) A node i detects a deadlock when it receives a probe where *initiator* = i.

Theorem 1: The necessary and sufficient condition for detecting all deadlocks is :

$$(\mathbf{C1})\langle \exists i : i \in C(R_c(k)) :: C(R_c(k)) \Longrightarrow \bigcirc^{|C(R_c(k))|} (probe\ (i, f_p(i \xrightarrow{*} i))\ at\ node\ i)\rangle.$$

Proof: (*Necessity:*) By Remark 1, $C(R_c(k))$ continues to exist unless some node in it aborts. A node in $C(R_c(k))$ can abort only after $C(R_c(k))$ is detected. If for no node i in the cycle its probe traverses the entire cycle in $|C(R_c(k))|$ steps of its formation, then the deadlock containing the path $i \xrightarrow{*} i$ will not be declared because: (i) a node i detects a deadlock only under feature (6) of a probe-based algorithm, (ii) all probe information will have been forwarded within $|C(R_c(k))|$ steps of deadlock formation by feature (5) above, and (iii) the particular deadlock C($R_c(k)$) is detected only if the probe function $\lambda = f_p(i \xrightarrow{*} i)$ (feature (3) of a probe-based algorithm).

(*Sufficiency:*) If $(C1)$ holds, then some node i *will* detect deadlock C($R_c(k)$) (which contains the path $i \xrightarrow{*} i$) within $\bigcirc^{|C(R_c(k))|}$ steps of its formation (follows from features (5) and (6) of a probe-based algorithm). □

Theorem 2: The necessary and sufficient conditions for not detecting false deadlocks are:

$$(\mathbf{C2})\langle(\forall i) : i \in R(j) \vee i \in R_c(j) - C(R_c(j)) :: probe(initiator, *)\ at\ node\ i \Longrightarrow initiator \neq i\ \rangle, \text{ and}$$

$$(\mathbf{C2'})\langle(\forall i) : i \in C(R_c(j)) :: probe(i, \lambda)\ at\ node\ i \Longrightarrow (i \xrightarrow{*} i \wedge f_p(i \xrightarrow{*} i) = \lambda)\rangle$$

Proof: (*Necessity:*) If $(C2)$ does not hold, a nondeadlocked node i can detect deadlock when it receives a probe with *initiator* = i. If $(C2')$ does not hold, a deadlocked node i may declare itself a part of another (nonexistent) deadlock when it receives a probe (i, λ), where $f_p(i \xrightarrow{*} i) \neq \lambda$.

(*Sufficiency:*) Node i declares itself deadlocked *only* when it receives a probe with *initiator* = i. $(C2)$ guarantees that a nondeadlocked node never receives such a probe. $(C2')$ guarantees that a deadlocked node receives its own probe (i, λ) only if it is forwarded by the nodes in the deadlock $C(R_c(k))$, and hence $f_p(i \xrightarrow{*} i) = \lambda$. □

Next, we identify and prove two invariants (I1) and (I2) for the corrected priority-based probe algorithm presented in Section III. The invariants (I1) and (I2) will be used to prove that the algorithm meets $(C1)$ and $(C2 \wedge C2')$, respectively. To reason about the algorithm, we confine attention only to transaction nodes in the TWFG.

Definition 2: A path $T_i \xrightarrow{*} T_j$ is antagonistic, denoted by $T_i \xrightarrow{>,*} T_j$, if $(\forall T_k) : T_k$ *lies along* $T_i \xrightarrow{*} T_j :: (T_i > T_k \wedge T_i \geq T_j)$.

Invariant I1: $(\forall T_i)(\forall T_j) :: T_i \xrightarrow{>,*} T_j \Rightarrow (\exists n, \bigcirc^n((T_i, junior) \in Probe\_Q_j \vee \neg(T_i \xrightarrow{>,*} T_j)))$ [proved in Corollary 2].

The probe initiated for a blocked transaction T_i is propagated to all transactions T_j to which there is an antagonistic path from T_i within a finite number n of message hops or the path ceases to exist. This invariant is used to show that all deadlocks are detected (in Theorem 3).

Invariant I2: $(\forall T_i)(\forall T_j) :: (T_i, junior) \in Probe\_Q_j \Rightarrow T_i \xrightarrow{>,*} T_j$ [proved in Lemma 4].

A probe with initiator T_i exists in the Probe_Q of transaction T_j at any instant only if there is an antagonistic path from T_i to T_j at that instant. This invariant is used to show that no false deadlocks are detected (in Theorem 4).

In the proof of correctness, we show that the corrected algorithm eventually detects all deadlocks, and never detects a phantom (nonexistent) deadlock. The basic idea is to show that Invariants I1 and I2 are preserved by the algorithm, and that these invariants guarantee the correctness conditions $(C1)$ and $(C2) \wedge (C2')$.

B. Proof of "Detection of All Existing Deadlocks"

Let $C(R_c(k))$, henceforth referred to as C in the context of the algorithm, be a deadlock cycle, and T_h be the highest priority transaction in the cycle. Observe that the probe $(T_h, junior)$ initiated by the data manager, where T_h is waiting, is the only one that could possibly circulate around C, because probes initiated on behalf of other transactions in C will face at least one antagonistic conflict (and will be blocked). Thus the probe which is initiated on behalf of T_h in the deadlock cycle C should not be prevented from traversing the edges of TWFG. (Otherwise, a failure to detect existing deadlocks will result.)

We now show that Invariant I1 is indeed preserved by the algorithm. We enumerate the three rules that govern the propagation of probes:

Observation 3: When an active transaction receives an (antagonistic) probe, it stores the probe in its Probe_Q (see Section III-D.2.2).

Observation 4: When a transaction blocks, it forwards a copy of all the probes in its Probe_Q (see Section III-A.2).

Observation 5: When a blocked transaction receives an (antagonistic) probe, a copy of the probe is forwarded immediately to the data manager on which the transaction is waiting (see Section III-D.2.1).

Corollary 1: If probes are propagated for deadlock detection without performing any cleaning for deadlock resolution, then Invariant I1 holds.

Proof: For a path $T_i \xrightarrow{>,*} T_j$, the data manager on which T_i waits initiates the probe $(T_i, junior)$. The probe will be forwarded by each transaction along $T_i \xrightarrow{>,*} T_j$ (by Observations 3–5) and by each data manager along $T_i \xrightarrow{>,*} T_j$ (see Section III-D.3). The maximum number of message hops from the time the path forms is n, the number of transactions and data objects along the path. □

However, to resolve any deadlock C', the Probe_Q's of all transactions in C' are cleaned. Certain probes need to be

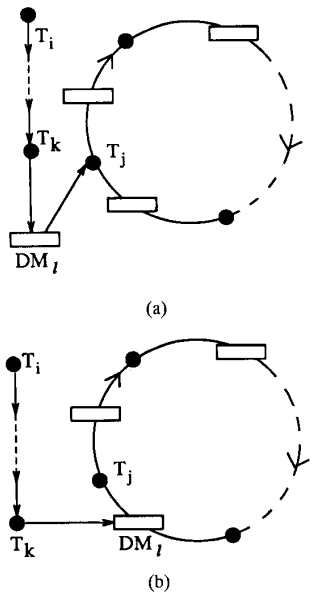

Fig. 4. Cases I(a) and I(b) in Lemma 1.

retransmitted to rebuild the cleaned Probe_Q's if the paths represented by the probes still exist after the victim of deadlock C' aborts. The biggest threat to the orderly flow of a probe $(T_i, junior)$ through the edges of TWFG is the cleaning of a Probe_Q containing the probe $(T_i, junior)$ at a transaction in a deadlock cycle C' which is being resolved. It must be shown that the probe $(T_i, junior)$ will propagate along all antagonistic paths from T_i, even though clean messages may be purging the Probe_Q's of transactions. (We are not concerned with a transaction that discards its Probe_Q after the first phase of the two-phase locking protocol (i.e., after lock-point), because it will not deadlock during the second phase).

Lemma 1: After T_j's Probe_Q has been cleaned due to the participation of T_j in a deadlock C', the algorithm ensures that all the probes which still represent an antagonistic path to T_j after the victim of C' aborts are forwarded to T_j.

Proof:

Suppose T_j received the probe $(T_i, junior)$ from transaction T_k. When T_j cleans its Probe_Q, the following two cases arise:

I T_k is not a part of the resolved deadlock C',
or
II T_k is a part of the resolved deadlock C'.

We will prove these cases separately. Case II breaks down into the following two subcases:

II.A T_k is the victim chosen to resolve the deadlock C', or
II.B T_k is not the victim chosen to resolve the deadlock C'.

Case I: Since the last time T_k forwarded the probe $(T_i, junior)$ to T_j, T_k has not received a clean message by Observation 2. So its Probe_Q contains the probe $(T_i, junior)$. Fig. 4 shows two possibilities for Case I: (a) T_k is connected to T_j through a nondeadlocked data manager DM_l. T_k will send this probe to T_j in response to the rebroadcast message from T_j to DM_l (see Sections III-E.3 and E.4). (b) T_k is connected to T_j through a deadlocked data manager DM_l. DM_l will request T_k to retransmit $Probe\_Q_k$ (see Section III-E.2.3). Therefore the probe will be retransmitted by T_k to T_j in both possibilities if path $T_i \xrightarrow{>,*} T_k \rightarrow T_j$ still exists.

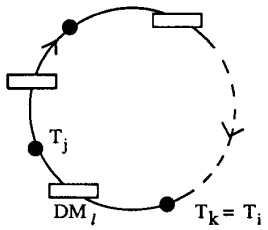

Fig. 5. Case II: B:1 in Lemma 1.

Case II.A: $T_i \xrightarrow{>,*} T_k \rightarrow T_j$ is *no longer* an (antagonistic) path when the victim T_k aborts.

If $i = k$, the victim $T_i < T_j$. T_i's probe would never have been forwarded to T_j (see Section III-D.1) and the data manager where T_i is waiting will never reinitiate the probe $(T_i, junior)$ (see Section III-E.2.2).

If $i \neq k$, then from the instant T_k (the victim) sends out the clean message until it gets back the clean message and aborts, it does not forward any probes (see Section III-E.1). Therefore T_j will *not* receive the probe $(T_i, junior)$ again unless another transaction completes a path from T_i to T_j and propagates the probe $(T_i, junior)$ to it.

The path $T_i \xrightarrow{>,*} T_j$ through T_k does not exist after the victim aborts, and the probe $(T_i, junior)$ is not forwarded to T_j by T_k.

Case II.B: Fig. 5 shows one scenario for this case. T_k is a direct predecessor of T_j in the deadlock cycle, and so $Probe\_Q_k$ is also cleaned.

(Case II: B:1) If $T_k = T_i$, as shown in Fig. 5, then DM_l on which T_k is waiting reinitiates probe $(T_i, junior)$ when it handles the clean message (see Section III-E.2.2).

(Case II: B:2) If $T_k \neq T_i$, then by Observations 3–5, if T_k again receives the probe $(T_i, junior)$ and $T_k \rightarrow T_j$ holds, the probe will be propagated to T_j. Therefore we need only show that T_k is again sent the probe $(T_i, junior)$ if the path $T_i \xrightarrow{>,*} T_k$ exists after the deadlock victim aborts. This is, however, merely a recursive application of this lemma. The recursion terminates when: (i) T_k does not lie on C' (Case I), (ii) T_k is the victim (Case II: A), or (iii) $T_k = T_i$ (Case II: B:1).

Therefore either by recursive application or direct reinitiation, T_j is again sent the probe $(T_i, junior)$ by T_k if the recursion termination case is (i) or (iii) above. In both (i) and (iii), the path $T_i \xrightarrow{>,*} T_j$ through T_k may exist after the victim of C' aborts. However, for (ii) above, the path $T_i \xrightarrow{>,*} T_j$ through T_k does *not* exist after the victim aborts and T_k does not forward the probe $(T_i, junior)$ to T_j. Thus $Probe\_Q_j$ is rebuilt per Lemma 1. □

Corollary 2: $(\forall T_i)(\forall T_j) :: T_i \xrightarrow{>,*} T_j \Rightarrow (\exists n, \bigcirc^n((T_i, junior) \in Probe\_Q_j \bigvee \neg(T_i \xrightarrow{>,*} T_j)))$. (Invariant I1)

Proof: By Lemma 1 and Corollary 1, the Probe_Q of T_j will contain the probe on behalf of T_i in a finite number of steps after the formation of $T_i \xrightarrow{>,*} T_j$ (since all message transmission delays are finite). □

Theorem 3: The algorithm detects all deadlocks.

Proof: We show that the algorithm satisfies $(C1)$, which is the necessary and sufficient condition to detect all deadlocks. We first restate $(C1)$:

$$(\mathbf{C1}) : \langle \exists i \ : i \in C(R_c(k)) :: C(R_c(k)) \Longrightarrow \bigcirc^{|C(R_c(k))|} (probe\ (i, f_p(i \xrightarrow{*} i)) \text{ at node i}) \rangle.$$

Let T_h be the highest-priority transaction in $C(R_c(k))$, and consider the antagonistic path $T_h \xrightarrow{>,*} T_h$. On applying invariant (I1) to such a path, within a finite number n of hops, the probe initiated on behalf of T_h will reach T_h (it will actually reach only the DM in $C(R_c(k))$, where T_h is the holder because of the distinction the algorithm makes between transaction and data managers), since $T_h \xrightarrow{>,*} T_h$ persists (Remark 1). By Observations 3–5 and the fact that the probe on behalf of T_h is not cleaned/discarded at any node in $C(R_c(k))$ after the cycle is formed and before the cycle is detected, we infer: (a) the number of message hops n is bounded by $|C(R_c(k))|$, and (b) the probe has never retraced its path and has been forwarded by each node along $h \xrightarrow{>,*} h$. Hence the probe function f_p has been computed over the path $h \xrightarrow{*} h$.

Thus the consequence of $(C1)$ is satisfied. □

C. Proof of "Absence of Phantom Deadlocks"

A *phantom deadlock* is a deadlock which does not exist in reality. To show that the algorithm does not detect phantom deadlocks, we must show that any deadlock detected by the algorithm indeed exists. To do this, we show that any deadlock detected by the algorithm is detected without the use of *irrelevant* probes, defined next.

Definition 3: An *irrelevant* probe $(T_i, junior)$ at T_j is a probe that has been propagated by or initiated on behalf of a transaction T_k which is now active or aborted. Alternatively, an *irrelevant* probe is one that violates Invariant I2. The probe is irrelevant to deadlock detection because the path of *waiting* transactions through which it traversed no longer exists.

Lemma 2: $(\forall DM_i)(\forall T_j) :: T_j \in Request\_Q_i \Rightarrow (T_j \rightarrow DM_i)$. (No Request_Q contains the id of a transaction that is not waiting on it.)

Proof: A transaction T_j is placed in $Request\_Q_i$ of a data manager DM_i only when it begins to wait on DM_i. We must show that T_j cannot become active or abort until after its id has been removed from $Request\_Q_i$. For T_j to become active, it must have become the *newholder* of X_i. When DM_i grants T_j's lock request, it removes T_j's entry from $Request\_Q_i$ (see Section III-D.1.2). Therefore before T_j becomes active, its id must have been removed from $Request\_Q_i$.

When transaction T_j aborts, it withdraws its pending request (see Section III-E.1.2). Withdrawal of a pending request causes DM_i to remove the corresponding entry from $Request\_Q_i$. Thus before T_j aborts, its entry is removed from $Request\_Q_i$. □

Lemma 3: No data manager initiates or reinitiates an irrelevant probe.

Proof: A data manager initiates a probe on behalf of a transaction only when it forces that transaction to wait (on an antagonistic conflict) (see Sections III-D.1.1 and D.1.2). A data manager reinitiates probes on behalf of only the transactions in its Request_Q (that face an antagonistic conflict) (see Sections III-E.2.2 and E.4). By Lemma 2, all of these transactions must be waiting on the holder of this data, and thus the probes are relevant. □

Lemma 4: $(\forall T_i)(\forall T_j) :: (T_i, junior) \in Probe\_Q_j \Rightarrow T_i \xrightarrow{>,*} T_j$ (the Probe_Q of a transaction never contains an irrelevant probe). (Invariant I2)

Proof: Consider a transaction T_j which is in the first phase of the two-phase locking protocol. (Otherwise, the proof is trivial because T_j discards its Probe_Q in the second phase.) From Lemma 3 we know that if a data manager initiates probe $(T_i, junior)$, the probe is relevant. A transaction manager/data manager propagates probes along its outward dependency. A probe $(T_i, junior)$ is placed in $Probe\_Q_j$ after T_j has received it from a data manager where it is the *holder*. The probe received indicates that T_i *was waiting* antagonistically on T_j either directly or transitively. In the latter case, each intermediate transaction *was waiting* on the subsequent one. Consider such an intermediate transaction T_k that forwarded probe $(T_i, junior)$ to T_j. We must show that if T_k aborts or becomes active after forwarding the probe $(T_i, junior)$ to T_j, the probe $(T_i, junior)$ representing the path $T_i \xrightarrow{>,*} T_j$ through T_k will never be in $Probe\_Q_j$ after T_k aborts or becomes active. Without loss of generality, let T_k be the first such transaction to abort/become active.

Case 1 (T_k aborts): T_k belongs to a deadlock C and all T_l such that $T_k \xrightarrow{*} T_l$ belong to C by Observations 1 and 2. T_j belongs to C because no transaction that has forwarded the probe from T_k to T_j has yet become active or aborted. For T_k to abort, it must first propagate a clean message through the cycle C (see Section III-E.1). When the transactions in the cycle (including T_j) receive this clean message, they purge their Probe_Q's (see Section III-E.3). When T_j rebuilds its Probe_Q, the Probe_Q will never contain probe $(T_i, junior)$ representing the path $T_i \xrightarrow{>,*} T_j$ through T_k by an application of Lemma 1 for which recursion terminates in Case II.A.

Case 2 (T_k becomes active): T_k must have become the *holder* of the data item for which it was waiting. This can happen only after the previous holder of the data item aborts or becomes active and releases the data item—both contradictions, because T_k is the first intermediate transaction to become active/abort after forwarding the probe $(T_i, junior)$ to T_j.

Thus a probe $(T_i, junior)$ at T_j indicates a path $T_i \xrightarrow{>,*} T_j$. □

Theorem 4: The algorithm does not detect any false deadlocks.

Proof: We show that the algorithm satisfies $(C2)$ and $(C2')$, which are the necessary and sufficient conditions to not detect any false deadlocks. We first restate $(C2)$ and $(C2')$:

$$(\boldsymbol{C2}) : \langle (\forall i) : i \in R(j) \vee i \in R_c(j) - C(R_c(j)) :: probe(initiator, *) \text{ at } i \implies initiator \neq i \rangle, \text{ and}$$

$$(\boldsymbol{C2'}) : \langle (\forall i) : i \in C(R_c(j)) :: probe(i, \lambda) \text{ at } i \implies (i \xrightarrow{*} i \wedge f_p(i \xrightarrow{*} i) = \lambda) \rangle$$

Consider node $i, i \in R(j) \vee i \in (R_c(j) - C(R_c(j)))$, which receives a probe $(initiator, junior)$. On applying invariant (I2) to T_i, we conclude the existence of $T_{initiator} \xrightarrow{>,*} T_i$ (consequence of (I2)). However, since T_i is nondeadlocked, $T_i \not\xrightarrow{*} T_i$. Thus $T_{initiator} \neq T_i$ and the consequence of $(C2)$ holds.

Consider node $i, i \in C(R_c(j))$, which receives a probe (i, λ). On applying (I2) to T_i, we conclude the existence of $T_i \xrightarrow{>,*} T_i$. From Observations 3–5 and noting that a probe never retraces its path, we conclude that for the probe (T_i, λ) at node T_i, $\lambda (= junior) = f_p(T_i \xrightarrow{>,*} T_i)$, the consequence of $(C2')$. □

V. Concluding Remarks

In this paper we have fixed two deficiencies of the deadlock detection/resolution algorithm of Choudhary *et al.* [3] (viz., inadequate generation of probes after cleaning of Probe_Q's and propagation of messages after sending out a "clean" message by a victim) and have presented a correct priority-based deadlock detection/resolution algorithm.

The paper identified the correctness conditions of a probe-based deadlock detection algorithm by proving the necessary and sufficient conditions to detect all deadlocks and to not detect false deadlocks. A major contribution of the paper is that we have given a formal proof of the correctness of the modified algorithm. (Note that many deadlock detection and resolution algorithms are incorrect because their authors have used informal or intuitive reasoning to prove their correctness.) A novel feature of the correctness proof is that it abstracts the operation of the algorithm by invariants, and uses the invariants to prove the desired properties of the algorithm. This is the first attempt at a formal proof of the correctness of a deadlock detection/resolution algorithm. Invariants are a powerful abstraction of the algorithm behavior and provide us with a good handle on the correctness proof, despite the highly complex nature of the algorithm. Invariants enable us to isolate the essence of the properties of an algorithm, and thus allow us to obtain elegant correctness proofs. Although we have illustrated how to use invariants in the correctness proof for a specific algorithm, the invariant-based technique can be used to prove the correctness of other deadlock detection/resolution algorithms, as well as other distributed algorithms.

This paper dealt with the single request model and ruled out spontaneous aborts of transactions. Thus deadlock was a stable property until detected and resolved. If either multiple requests or spontaneous aborts are permitted, then deadlock is no longer a stable property of a distributed system. When a site declares a deadlock there is no guarantee that the deadlock still exists, because a deadlocked process might have spontaneously aborted or might have been aborted to resolve an adjoining deadlock. (Note that in the multiple request model, a process can belong to several deadlocks concurrently.) Due to the lack of the current global state of the system, a site can never assert whether a detected deadlock currently exists in the system. Hence a big problem which is faced when multiple requests or spontaneous aborts are allowed is that the distinction between the detection of a false deadlock by an incorrect algorithm and the detection of a deadlock which is concurrently resolved becomes blurred [8]. However, the invariant-based technique has been found to be effective in the correctness proof of a deadlock detection/resolution algorithm in a multiple request, spontaneous abort model [8].

Appendix
State Transition Rules for TWFG

In this appendix we describe changes in the TWFG due to various actions in the system using state transitions. This helps us to understand the formation and resolution of a distributed deadlock.

Let $R(i)$ and $R_c(i)$ denote a tree and a nontree, respectively, as defined in Section IV-A. Let $\mathcal{T}$ be the set of all trees in the system. Let $\mathcal{T}_c$ be the set of all nontrees in the system. Define the state of the TWFG to be the tuple $\langle \mathcal{T}, \mathcal{T}_c \rangle$. The initial system state is $\langle \{\{i\} | i \text{ is a node}\}, \varnothing \rangle$. We enumerate five rules **(R1)–(R5)** that when executed atomically describe changes to the TWFG due to various actions. These rules are derived by noting that state transitions can be triggered only by roots of trees and aborting nodes. (It can be shown that the set of rules is consistent and complete.) State transitions occur whenever an edge forms from the root of a tree $R(i)$: (a) to a node in another tree $R(k)$, (b) to a node in another nontree $R_c(k)$, or (c) to a node in the same tree $R(i)$. In addition, a transition occurs whenever: (d) the root of $R(i)$ releases a data item, which is then assigned to another node which was waiting on node i. Lastly, a transition occurs when: (e) a node in a nontree aborts and releases the data items it held. Edge (i, j) forms when node i blocks on node j (specifically, when the data manager on which i blocks determines that j is the $holder$), and the edge is removed when node j replies to node i (specifically, when the data manager freed by j determines that i is the $newholder$) or node j aborts.

In the notation used for specifying the rules, "$\longrightarrow$" indicates a state transition. Since a state transition is specified by a sequence of actions and a rule may contain several state transitions, the final state f for each rule is indicated by $OUTPUT(f)$. In a sequence of actions, "$\leftarrow$" indicates an assignment, "$x; y$" indicates sequential execution of x and y, where x and y are actions, "$\rightarrow$" is used with a CSP guard, and "$*$" is the CSP repeat command.

$\boldsymbol{R1} : \langle \mathcal{T}, \mathcal{T}_c \rangle \longrightarrow$
$\langle \mathcal{T} \leftarrow \mathcal{T} - \{R(i)\} - \{R(k)\};\ R(k) \leftarrow R(k) \cup R(i);$
$OUTPUT(\mathcal{T} \cup \{R(k)\}),\ OUTPUT(\mathcal{T}_c) \rangle$
$(j \in R(k), k \neq i$ when edge (i, j) forms.)

Rule R1 specifies the growth of a tree $R(k)$ when the root of another tree $R(i)$ begins waiting on a node j in $R(k)$.

$$\begin{aligned}&\textbf{R2}: \langle \mathcal{T}, \mathcal{T}_c \rangle \longrightarrow \\ &\quad \langle OUTPUT(\mathcal{T} - \{R(i)\}), \\ &\quad R_c(i) \leftarrow R(i);\ OUTPUT(\mathcal{T}_c \cup \{R_c(i)\})\rangle \\ &(j \in R(k), k = i \text{ when edge } (i,j) \text{ forms.})\end{aligned}$$

Rule R2 specifies the conversion of a tree $R(i)$ into a nontree $R_c(i)$; this happens when node i begins waiting on a node within $R(i)$.

$$\begin{aligned}&\textbf{R3}: \langle \mathcal{T}, \mathcal{T}_c \rangle \longrightarrow \\ &\quad \langle OUTPUT(\mathcal{T} - \{R(i)\}), \\ &\quad \mathcal{T}_c \leftarrow \mathcal{T}_c - R_c(k);\ R_c(k) \leftarrow R_c(k) \cup R(i); \\ &\quad OUTPUT(\mathcal{T}_c \cup \{R_c(k)\})\rangle \\ &(j \in R_c(k) \text{ when edge } (i,j) \text{ forms.})\end{aligned}$$

Rule R3 specifies the growth of a nontree $R_c(k)$ when the root of a tree $R(i)$ begins waiting on a node j in $R_c(k)$

$$\begin{aligned}&\textbf{R4}: \langle \mathcal{T}, \mathcal{T}_c \rangle \longrightarrow \\ &\quad \langle \mathcal{T} \leftarrow \mathcal{T} - \{R(i)\} \\ &\quad R(i) \leftarrow R(i) - R(j) - \cup_{k \in Samedata(j\,R(i)\,)} R(k); \\ &\quad R(j) \leftarrow R(j) \cup_{k \in Samedata(j\,R(i)\,)} R(k); \\ &\quad OUTPUT(\mathcal{T} \cup \{R(i)\} \cup \{R(j)\}), \\ &\quad OUTPUT(\mathcal{T}_c)\rangle \\ &(j\,i \in R(i) \text{ edge } (j\,i) \text{ is removed.})\end{aligned}$$

Rule R4 specifies the transition when the root of a tree $R(i)$ releases a data item which is allocated to node j waiting on it. Function $Samedata(jj, arri)$ takes inputs j and $R(i)$ and returns the set $Samedata(jj, arri)$ of other nodes waiting on i for the same data item as j was. Then, all $kkinSamedata(jj, arri)$ begin waiting on j for the same data item, i.e., edges (k, j) are formed for all $kkinSamedata(jj, arri)$.

$$\begin{aligned}&\textbf{R5}: \langle \mathcal{T}, \mathcal{T}_c \rangle \longrightarrow \\ &\quad (\langle R(v) \leftarrow R_c(k); \mathcal{T} \leftarrow \mathcal{T} \cup \{R(v)\}, \\ &\quad \mathcal{T}_c \leftarrow \mathcal{T}_c - \{R_c(k)\}\rangle \longrightarrow \\ &\quad \langle output\ of\ rule(R4)\ where\ i = v\ and \\ &\quad j\ receives\ the\ deadlocked\ data\rangle \longrightarrow \\ &\quad * [\langle |R(v)| \neq 1 \rightarrow output\ of(R4)\ where\ i = v\rangle] \longrightarrow \\ &\quad \langle OUTPUT(\mathcal{T} - \{R(v)\}), OUTPUT(\mathcal{T}_c)\rangle) \\ &(\text{Deadlocked edge } (j, v) \text{ is removed when } v \text{ aborts.})\end{aligned}$$

Rule R5 specifies how a nontree breaks up into one or more trees when a node in the nontree aborts, using four state transitions described in the following steps: (i) Non-tree $R_c(k)$ involving the deadlock is converted to tree $R(v)$ when the victim $v \in R_c(k)$ withdraws its request, (ii) the victim v releases the deadlocked data item to j, by rule R4, (iii) the victim releases all nondeadlocked data items it holds by repeatedly using rule R4, and (iv) the victim is eliminated from the set of trees.

ACKNOWLEDGMENT

The authors would like to thank the anonymous referees for their invaluable comments on an earlier version of this paper.

REFERENCES

[1] K. M. Chandy, J, Misra, and L. M. Haas, "Distributed deadlock detection," *ACM Trans. Comput. Syst.*, pp. 144–156, May 1983.
[2] A. L. 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. Software Eng.*, vol. 15, p. 1544, Dec. 1989.
[3] A. L. Choudhary, W. H. Kohler, J. A. Stankovic, and D. Towsley, "A modified priority-based probe algorithm for distributed deadlock detection and resolution," *IEEE Trans. Software Eng.*, vol. 15, pp. 10–17, Jan. 1989.
[4] A. K. Elmagarmid, N. Soundararajan, and M. T. Liu, "A distributed deadlock detection and resolution algorithm and its correctness," *IEEE Trans. Software Eng.*, vol. 14, pp. 1443–1452, Oct. 1988.
[5] B. Hailpern and S. Owicki, "Verifying network protocols using temporal logic," in *Proc. NBS Symp. on Comput. Network Protocols*, 1980.
[6] G. S. Ho and C. V. Ramamoorthy, "Protocols for deadlock detection in distributed database systems," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 554–557, Nov. 1982.
[7] E. Knapp, "Deadlock detection in distributed database systems," *ACM Comput. Surveys*, pp. 303–328, Dec. 1987.
[8] A. D. Kshemkalyani and M. Singhal, "Characterization and correctness of distributed deadlocks," The Ohio State Univ., Columbus, Tech. Rep. CISRC-6/90-TR15, 1990.
[9] D. E. Menasce and R. R. Muntz, "Locking and deadlock detection in distributed databases," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 195–202, May 1979.
[10] R. Obermarck, "Distributed deadlock detection algorithm," *ACM Trans. Database Syst.*, pp. 187–210, June 1982.
[11] M. Roesler and W. A. Burkhard, "Resolution of deadlocks in object-oriented distributed systems," *IEEE Trans. Computers*, pp. 1212–1224, Aug. 1989.
[12] B. Sanders and P. A. Heuberger, "A distributed deadlock detection and resolution with probes," in *Proc. 3rd Int. Workshop on Distributed Algorithms* (Nice, France), Sept. 1989, pp. 207–218.
[13] K. Shafer and M. Singhal, "A correct priority-based probe algorithm for distributed deadlock detection and resolution and its correctness proof," The Ohio State Univ., Columbus, Tech. Rep. No. OSU-CISRC-4/89-TR16, 1989.
[14] B. Shyam and D. M. Dhamdhere, "A new priority-based probe algorithm for distributed deadlock detection," Indian Instit. Technol., Bombay, Tech. Rep. No. TR-015-90, 1990.
[15] M. Singhal, "Deadlock detection in distributed systems," *IEEE Computer*, pp. 37–48, Nov. 1989.
[16] M. K. Sinha and N. Natarajan, "A priority-based distributed deadlock detection algorithm," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 67–80, Jan. 1985.
[17] G. Vossen and S. S. Wang, "Towards efficient algorithms for deadlock detection and resolution in distributed systems," in *Proc. 5th Int. Conf. Data Eng.*, 1989, pp. 287–294.

Ajay D. Kshemkalyani (S'85) received the B. Tech. degree in computer science and engineering from the Indian Institute of Technology, Bombay, in 1987, and the M.S. degree in computer and information science from Ohio State University, Columbus, in 1988, where he is currently a candidate for the Ph.D. degree in computer and information science. His areas of research interest are in distributed and parallel computing, operating systems, databases, and computer architecture.

Mukesh Singhal (M'86) was born in India in 1959. He received the B.E. degree in electronics and communication engineering (with High Distinction) from the University of Roorkee, Roorkee, India, in 1980. He obtained the Ph.D. degree from the Department of Computer Science, University of Maryland, College Park, in 1986.

Since 1986 he has been on the faculty of the Department of Computer and Information Science, Ohio State University, Columbus, where he has been promoted to an Associate Professor. His current research interests are in distributed systems, distributed databases, and performance modeling. He has published in various IEEE Transactions and Journals. He is coauthoring a book entitled *Advanced Operating Systems* to be published (New York: McGraw–Hill) in 1992.

Dr. Singhal has served as a co-guest editor of a special issue of the IEEE COMPUTER ON DISTRIBUTED COMPUTING SYSTEMS.