Formal verification of distributed deadlock detection algorithms

Brian Matt Johnston

University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
Johnston, Brian Matt, "Formal verification of distributed deadlock detection algorithms" (1993). UNLV Retrospective Theses & Dissertations. 293.
https://digitalscholarship.unlv.edu/rtds/293

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Formal verification of distributed deadlock detection algorithms

Johnston, Brian Matt, M.S.
University of Nevada, Las Vegas, 1993
FORMAL VERIFICATION OF DISTRIBUTED DEADLOCK DETECTION ALGORITHMS

by

Brian Matt Johnston

A thesis submitted in partial fulfillment of the requirement for the degree of

Master of Science
in
Computer Science

Department of Computer Science
University of Nevada, Las Vegas
May, 1993
The Thesis of Brian Matt Johnston for the degree of Master of Science in Computer Science is approved.

Chairperson, Ajoy Kumar Datta, Ph. D

Examinining Committee Member, Kazem Taghva, Ph. D

Examining Committee Member, Laxmi Gewali, Ph. D

Graduate Faculty Representative, Ashok Iyer, Ph. D

Dean of the Graduate College, Ronald W. Smith, Ph. D

University of Nevada, Las Vegas
May 1993
ABSTRACT

The problem of distributed deadlock detection has undergone extensive study. Formal verification of deadlock detection algorithms in distributed systems is an area of research that has largely been ignored. Instead, most proposed distributed deadlock detection algorithms have used informal or intuitive arguments, simulation or just neglect the entire aspect of verification of correctness.

As a consequence, many of these algorithms have been shown incorrect. This research will abstract the notion of deadlock in terms of a temporal logic of actions and discuss the invariant and eventuality properties. The contributions of this research are the development of a distributed deadlock detection algorithm and the formal verification of this algorithm.
ACKNOWLEDGMENTS

I owe a special thank you to my advisor, Dr. Ajoy Kumar Datta for his continually support, inspiration and challenging discussions throughout the course of this thesis. Without his careful guidance, this thesis would not have been possible. Thank you for providing the encouragement necessary to persevere in my endeavors. I am also indebted to my committee members, Dr. Kazem Taghva, Dr. Laxmi Gewali and Dr. Ashok Iyer for serving on my committee and review of my thesis. I extend my gratitude to the Department of Computer Science for providing me with a teaching assistantship, which has allowed to pursue my research.

To Kathy Eyster and Jon Thumim at System Computing Services, I owe you my sanity. Thank you for all your help in preparing my thesis for the graduate college. Your knowledge of WordPerfect is unsurpassed. Indeed, I owe the entire staff at System Computing Services a special thank you.

For my family and friends, thank you for enduring my moods over the last few months. Without your love and support this thesis would not have been possible.
LIST OF FIGURES

2.1 Legitimate States of a transaction .................................................. 7
2.2 Difference between Wait_for(Ti) and Held_by(Ti) .......................... 9
2.3 Actions of a transaction ................................................................. 10
2.4 Example before deadlock .............................................................. 13
2.5 Example after deadlock ............................................................... 13
CHAPTER 1

INTRODUCTION

A collection of autonomous processes spatially separated, that communicate through some communications network are commonly referred to as a distributed computer system. Processes that operate in this environment share no common memory nor do they have global clock. These processes share information regarding the operation of the distributed system by passing messages. Distributed computer systems make no assumptions about the particular hardware contained in the system. Therefore, a wide variety of architectures can be used.

Processes in a distributed system have the ability to share resources and information over the communication network. Any process in a distributed system can request a resource at any other location in the system without any knowledge of the status of that particular resource. So, it is possible for that resource to be unavailable for two reasons: either the site or communication network to that resource has failed or that resource is currently locked by another process. This research will address the latter topic only. Since processes in these distributed systems have the ability to lock other processes from certain resources, the possibility of deadlock can arise. A group
of processes is considered to be deadlocked when no process can ever be satisfied unless a drastic action within the system is taken. In general, distributed systems provide the ability for sharing of resources (eg. printers or specialized hardware), divide the tasks of information processing and distributing a large database over many locations. [21] This creates greater resiliency in the information processing system. However, the major drawback is the possibility of deadlock.

Characterization of Deadlock Detection Algorithms

Distributed deadlock detection algorithms can be classified into four major classes of algorithms: path-pushing, edge-chasing, diffusing computations and global state detection. These approaches are all concerned with detecting cycles in graph. However, the methods by which detection is accomplished are greatly different. [3, 5, 9, 10, 15, 20, 22, 23]

The first class of deadlock detection algorithms, path-pushing, relies on the construction of some version of the global wait-for graph at each site. These algorithms rely on informing neighboring sites about the state of a local site. The information is continually updated and forwarded to the neighbors.

Another group of algorithms are the edge-chasing algorithms. These algorithms depend on the forwarding of probe messages along the edges of the graph to detect the cycle. If a cycle exists in a given graph then the probe message will return to the initiator of the probe. Therefore, a cycle can be detected.

The next type of algorithms to be considered is the diffusing computation
algorithms. The concepts behind these algorithms is the use of a computation superimposed on the operation of the database itself. These computations grow and shrink by sending query and reply messages. If the computation terminates then a deadlock exists in the system. [7]

The final classification of deadlock detection algorithms is the global state detection. This broad class of algorithms depend on determining a consistent global state of the system without suspending the computation of the database itself. By constructing local histories and defining a partial ordering of the system, deadlocks can be detected.

This research will focus solely on the edge-chasing class of algorithms. All deadlock detection algorithms have two criteria on which they are determined to be correct. The first criterion is all deadlocks are detected within a finite time. The other aspect of correctness no phantom deadlock will be reported. Phantom deadlocks occur when the algorithm believes there is cycle in the graph of the system when in reality no cycle actually exists.
CHAPTER 2

DEADLOCK DETECTION IN DISTRIBUTED SYSTEM

The focus of this chapter is to gain insight into how deadlocks occur in a distributed database system. A distributed database system is a database system in which the information contained does not reside at one centralized site. These systems function transparently to the user of the database with respect to transaction processing. In this chapter concurrency control and transaction processing will be formalized and a distributed deadlock detection algorithm will be developed.

2.1 Concurrency Control

In the distributed database model each site has a data manager responsible for maintaining the portion of the database at that particular site. The data manager at each site has three responsibilities: submitting requests to other data managers, processing message requests from other data managers and concurrency control for its subset of the database. When any transaction needs access to a nonlocal portion of the database, the local data manager must submit a request, via message passing, to
the data manager which controls that particular piece of the database. Upon receiving a message, a given data manager will process the message by either granting or denying access to the data object needed. The concurrency control mechanism used in most databases is the two-phase locking protocol. [2, 8, 11]

2.2 Transaction Processing

Transactions have three properties: failure atomicity, permanence and serializability. Failure atomicity ensures that if any given transaction fails prior to completion, all actions taken by that transaction will be undone. Permanence of transactions guarantee that if a transaction completes successfully, the results of this operation will never be lost. The serializability property of transactions ensures that concurrent operations of these transactions will not produce database inconsistencies (this is ensured by the two-phase locking protocol). These transaction require access to the database to read, modify and update data. So, transactions must make requests to their data manager to access the database. When a transaction makes a lock request for a given data object there are two possible outcomes. If that particular object is in use by another transaction, the requesting transaction will then block and stop execution until that object becomes available. Suppose the data object available, then the transaction making the request will be granted an exclusive lock on that object. Subsequently, any other transactions needing that object will be forced to wait until that transaction completes.

An individual transaction's processing can be viewed through a state diagram.
Transactions can move through five legitimate states: running, pending, waiting, halted and deadlocked. When a transaction has all needed data objects or needs no access to any data objects, it will be actively processing. After processing completely, a transaction will move to its final state which is halted. However, if this transaction needs part of the database for completion, it will make a request to its data manager and move into the pending state. Pending implies that a request has been made and no response has come back. Once a decision has been made, the transaction will either be granted the lock request and return to the running state or blocked by another transaction and forced into the wait state. If the system is
abstracted in terms of a transaction wait-for graph, a deadlock occurs when a cycle forms among a subset of the waiting transactions.

2.3 Related Work

Current work in the area of distributed deadlock detection has focused primarily on the development of algorithms. Most of the algorithms proposed fail to address the area of formal verification. Most algorithms present informal arguments of correctness of the algorithm. Consequently, they are prone to errors. [13]

For example, Sinha and Natarajan have developed an edge-chasing algorithm for detecting deadlocks. They offered an informal description of the correctness of this algorithm. The authors concluded that the algorithm met the criteria for correctness. In 1989, this algorithm was modified to correct for the errors in the original paper. Again, this modification was informally shown to be correct. However, in 1990 the modified algorithm was shown to have errors. The new modification of this algorithm was formally shown to be correct using temporal logic. [6, 17, 18, 25]

2.4 Proposed Algorithm

This proposed algorithm falls within the category of edge-chasing algorithms. Each site in the network carries a unique site identifier called Site_ID. Within the network a site maintains a certain portion of the database. Each site owns some data objects and maintains a few transactions. Each data object is identified by a unique identifier
denoted by $Data_{obj}$. Every data object controlled by a site

![Diagram of transactions]

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Wait_for</th>
<th>Held_by</th>
<th>Request_Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td>$nil$</td>
<td>$nil$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>$T_1$</td>
<td>$T_0$</td>
<td>$T_0$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$T_0$</td>
<td>$T_1$</td>
<td>$nil$</td>
</tr>
</tbody>
</table>

Figure 2.2 Difference between $Wait_{for}(Ti)$ and $Held_{by}(Ti)$ has a variable associated with it called $Locked_{by}$. The variable $Locked_{by}$ determines the current state of the data object. If the data object is not locked by any transaction, $Locked_{by}$ will store $nil$, otherwise it stores the identification of the locking transaction.

Each transaction has a unique site identifier denoted by $T_{ID}$. A transaction can use data objects within its own site or make explicit requests for a data object in another site. As each site has a unique $Site_{ID}$, and every transaction within a site has a unique $T_{ID}$, the $T_{ID}$ can be considered to be unique throughout the network (see Figure 2.2).
Assumption 1: A transaction can have at most one outstanding lock request.

In case a transaction needs more than one data object, the second data object can be requested only after the first data object has been granted.

Each transaction $T_i$ at site $S_i$ has the following data structure: a variable called $Wait_{for}(T_i)$, a variable called $Held_{by}(T_i)$, and a queue of requesting transactions $Request_{Q}(T_i)$. If the current transaction is not waiting for any other transaction then $Wait_{for}(T_i)$ is set to nil, else, it denotes which transaction is at the head of the locked data object. $Held_{by}(T_i)$ is set to nil if the current transaction is executing, otherwise, it stores the transaction that is holding the data object required by the current transaction. $Request_{Q}(T_i)$ is a tuple $(T_j, D_j)$, where $T_j$ is the requesting transaction and $D_j$ is the particular data object held by $T_j$.

The difference between $Wait_{for}(T_i)$ and $Held_{by}(T_i)$ can be well understood with the help of an example.

![Figure 2.3 Actions of a transaction](image-url)

As shown in Figure 2.3, Transaction T2 is waiting for a data object held by
transaction T1, which is further waiting on transaction T0. Thus Held_by(T2) is T1, while Held_by(T1) is T0. As described above, Wait_for(T1) and Wait_for(T2) are equal to T0.

Suppose a transaction Tj makes a lock request for a data object Dj. If Dj is free then Dj is granted to Tj and Locked_by(Dj) is set to Tj. If Dj is not free then Dj sends a not granted message to Tj along with the transaction identifier locking Dj. Tj joins the Request_Q(Tj) and sets its Wait_for equal to Wait_for(Tj). Now Tj initiates an update message to modify all the Wait_for variables which are affected by the changes in Locked_by variable of the data objects. Update message is a recursive function call that will continue updating all elements of every Request_Q in the chain.

When a transaction Tj receives the update message it checks if its Wait_for value is the same as the new Wait_for value. If it is not the same then the value is modified. Now, a check for deadlock is performed. If a deadlock is not detected then the update message is forwarded, else deadlock is declared and deadlock resolution is initiated.

The transaction detecting the deadlock is chosen as the transaction to be aborted. This transaction sends a clear message to the transaction holding its requested data object. It also allocates every data object it held to the first requester in its Request_Q and enqueues remaining requesters to the new transaction.

The transaction receiving the clear message purges the tuple in its Request_Q having the aborting transaction as an element.
{Transaction Ti makes a lock request for data object Dj}

Begin
    send lock_request(Ti) to Dj;
    wait for granted / not granted;
    if granted then
        begin
            Locked_by(Dj) := Ti;
            Held_by(Ti) := {empty set};
        end
    else {Suppose Dj is being used by transaction Tj}
        begin
            Held_by(Ti) := Tj;
            Enqueue(Ti, Request_Q(Tj));
            if Wait_for(Tj) = nil then
                Wait_for(Ti) := Tj
            else
                Wait_for(Ti) := Wait_for(Tj);
            update(Wait_for(Ti), Request_Q(Ti));
        end;
End;

{Data object Dj receiving a lock_request(Ti)}

Begin
    if Locked_by(Dj)=nil then
        send granted
    else begin
        send not granted to Ti;
        send Locked_by(Dj) to Ti;
    end;
End;

{Transaction Tj receiving an update message}

Begin
    if Wait_for(Tj) \neq Wait_for(Ti) then
        Wait_for(Tj) := Wait_for(Ti);
    if Request_Q(Tj) - Wait_for(Tj) = nil then
        update(Wait_for(Ti), Request_Q(Tj))
    else
        DECLARE DEADLOCK;
End;
2.5 Example

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Wait_for</th>
<th>Held_by</th>
<th>Request_Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₀</td>
<td>nil</td>
<td>nil</td>
<td>T₁</td>
</tr>
<tr>
<td>T₁</td>
<td>T₀</td>
<td>T₀</td>
<td>T₂</td>
</tr>
<tr>
<td>T₂</td>
<td>T₀</td>
<td>T₁</td>
<td>T₃</td>
</tr>
<tr>
<td>T₃</td>
<td>T₀</td>
<td>T₂</td>
<td>T₄, T₆</td>
</tr>
<tr>
<td>T₄</td>
<td>T₀</td>
<td>T₃</td>
<td>T₅</td>
</tr>
<tr>
<td>T₅</td>
<td>T₀</td>
<td>T₄</td>
<td>nil</td>
</tr>
<tr>
<td>T₆</td>
<td>T₀</td>
<td>T₃</td>
<td>nil</td>
</tr>
</tbody>
</table>

Figure 2.4 Example before Deadlock

Consider a distributed database with seven transactions as shown in Figure 2.4. The state of each transaction is also shown in the figure. However, it does not necessarily imply that each transaction resides in the same site.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Wait_for</th>
<th>Held_by</th>
<th>Request_Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₀</td>
<td>T₀</td>
<td>T₃</td>
<td>T₂</td>
</tr>
<tr>
<td>T₁</td>
<td>T₀</td>
<td>T₀</td>
<td>T₂</td>
</tr>
<tr>
<td>T₂</td>
<td>T₀</td>
<td>T₁</td>
<td>T₃</td>
</tr>
<tr>
<td>T₃</td>
<td>T₀</td>
<td>T₂</td>
<td>T₄, T₆, T₀</td>
</tr>
<tr>
<td>T₄</td>
<td>T₀</td>
<td>T₃</td>
<td>T₅</td>
</tr>
<tr>
<td>T₅</td>
<td>T₀</td>
<td>T₄</td>
<td>nil</td>
</tr>
<tr>
<td>T₆</td>
<td>T₀</td>
<td>T₃</td>
<td>nil</td>
</tr>
</tbody>
</table>

Figure 2.6 Example after Deadlock

shows the state of the system before the occurrence of deadlock.
When transaction $T_0$ makes a request to transaction $T_3$, a cycle is created and the state of the above system changes, as shown in Figure 2.6. $T_0$ joins the Request$_Q$ of $T_3$. To will update its Wait_for to reflect the current state and will propagate the update message to all elements in its own Request$_Q$. This continues until $T_3$ discovers that Wait_for($T_3$) intersected with Request$_Q$(T$_3$) is not nil. Now, $T_3$ declares deadlock.
CHAPTER 3

TEMPORAL LOGIC

A logic whose truth and falsity that depend on its placement in time is referred to as temporal logic. Most forms of propositional logic are static. Therefore, if a proposition is true it will always be true. Temporal logic falls within the class of modal logics which have been developed to study modes of truth. Temporal logic provides operators that allow one to reason how the truth of a given proposition can vary throughout time. Pnueli proposed the use of temporal logic in the verification of programs in 1977. In this chapter, temporal logic will be developed, explained and the edge-chasing deadlock detection algorithm from the previous chapter will be verified correct. This will be done by detailing precisely how temporal logic works within the framework of first-order logic. Once the axioms and operators of temporal logic have been devised, properties of this logic will be detailed. [4, 12, 14, 16, 19]

3.1 What is Temporal Logic

In temporal logic the defining structure is a totally ordered set with three basic assumptions regarding this set:
The temporal logic that is being considered in this research is merely first order logic extended by the inclusion of temporal operators. The fundamental concept underlying temporal logic is that any given proposition may yield different truth values at different points in time.

Now, some logical operators for temporal logic can be defined. $\Diamond A$ says that proposition $A$, hold at the time point immediately after the reference point. $\Diamond$ is the next time operator. $\square A$ implies that proposition $A$ holds at all time points after the reference point. $\square$ is defined as the henceforth operator. $\Diamond$ is the eventually operator. Informally, $\Diamond A$ says that there is a point in time after the reference point such that $A$ holds. The final operator is until. $A$ until $B$ informally is $A$ holds at all following time points up to a time point at which $B$ holds. [1, 26]

### 3.2 Properties of Temporal Logic

This distributed system is abstracted by defining actions. An action is any possible change in the system state. Actions represent the relation between the old states and new states of the system. Actions are always considered to be atomic. Any change of state in this system require message passing or processing. So, the actions of this system are simply the events which cause a transition from one state to another.

Let $G=(V,E)$ be the transaction wait-for graph of this system, where
V = \{T_{ID} \mid \text{T_{ID} not halted}\} and E = \{(T_{IDi}, T_{IDj}) \mid \text{Transaction i is waiting on Transaction j}\}. Let G_i be a connected component of the graph. All G_i are trees until deadlock situation occurs in which case a cycle will exist. A cycle is formed when a transaction, T_i, must wait for a transaction, T_j, that is already waiting direct or indirectly on T_i. Let such a graph be denoted C(G_i). Further define \(|C(G_i)|\) to be the length of the cycle.

Let state(T_{ID}) be a predicate defined on any given transaction in the system in the following manner: state(T_{ID}) = \{state of the given transaction at a reference point in time\}.

Now, six temporal logic of actions can be defined.

**Action 1:** Lock request is made by T_i
\[
\text{state}(T_i) = \text{running} \quad \bigcirc (\text{state}(T_i)) = \text{pending}
\]

**Action 2:** Receive message lock request granted
\[
\text{state}(T_i) = \text{pending} \quad \bigcirc (\text{state}(T_i)) = \text{running}
\]

**Action 3:** Receive message lock request not granted
\[
\text{state}(T_i) = \text{pending} \quad \bigcirc (\text{state}(T_i)) = \text{waiting}
\]

**Action 4:** Transaction enter Phase 2 of locking protocol
\[
\text{state}(T_i) = \text{running} \quad \Diamond (\text{state}(T_i)) = \text{halted}
\]

**Action 5:** Transaction T_i releases data object needed by T_i
\[
\text{state}(T_i) = \text{waiting} \quad \Diamond (\text{state}(T_i)) = \text{running}
\]

**Action 6:** Transaction T_i receives an update message
\[
\text{state}(T_i) = \text{pending} \quad \text{wait until state change}
\]

\[
\text{state}(T_i) = \text{waiting} \quad \text{Propagate update message or Declare deadlock}
\]

In addition to these six actions, the criteria for correctness will be established through the invariant and eventuality properties.

Invariant or safety properties of a distributed system is defined in the following
manner: A distributed program does nothing wrong. Invariant properties are initially true and stable. A stable property of system is one that once it holds, it will hold forever. The first criterion for correctness of distributed deadlock detection algorithms is: No phantom deadlocks are detected.

Eventuality or liveness properties ensure that a distributed program does what it is suppose to do. In deadlock detection, the liveness property dictates that all deadlocks are detected in a finite amount of time.

3.3 Verification of Proposed Algorithm

The goal of the verification will be to show that the proposed algorithm satisfies both criteria for correctness. The first aspect to be considered is the safety property. This is the property that guarantees that the program does nothing wrong.

In distributed deadlock detection, no phantom deadlocks represents a safety property. To determine that the proposed algorithm provides this safety property, no phantom deadlocks will be shown to be invariant.

Suppose a given property \( P \) must be shown to be invariant, the following conditions must be met: \( P \) must hold initially and \( P \) holds regardless of the actions of the system during execution. The second condition is known as stability.
3.4 Proof of Safety Property

At time=0 all transactions in the system will have their variables initialized to nil. At this point in time no transaction will receive any messages. So no transaction will be able to declare deadlock. Therefore, initially no phantom deadlocks will be found.

Suppose this distributed system has only one transaction. So, in this case no true deadlock can exist. For deadlock to be declared by the algorithm in this system, the system must successfully execute actions 1 and 3 to be in a position to declare a deadlock, phantom or not. If the transaction needs a particular data object, then a lock request will be made. This is the first action of the system. When the transaction moves to the pending state, the transaction is in a busy wait state pending the notification of the result of its request. At some point in time the result will be returned. This message generates one of two possible actions in the system. A granted message is action 2. This will move the transaction back to the running state, and no deadlock can be declared. Action 3 indicates a not granted message was returned to the requesting transaction. A not granted message implies that another transaction has locked that particular data object. However this contradicts the assumption that only one transaction is in the system. Similarly, action 5 can not be applied to a one transaction system. Action 6 assumes the receipt of an update message from another transaction, however, it was assumed that only one transaction was in the system. When action 4 occurs in this system a transaction is entering phase two of the locking protocol; so no phantom deadlocks will be detected. Now, the base step is valid and
no phantom deadlocks will be declared.

Suppose this is a stable property for up to an n transaction system. Since the property is assumed to be stable for up to n transactions, there will be at most n-1 edges in any waiting chain of transactions. If they were n edges then a cycle would exist.

When action 1 is executed in the system, a transaction moves to the pending state and will wait for a message to be received. At this point the other n transactions will be running, pending or waiting. Since, this transaction is in the pending state, it must wait before forwarding any update messages it receives. Consequently, no deadlock will be declared. The transaction which has made the request for the resource will at some point receive notification regarding its request. If the transaction receives a granted message it will transition back to the running state and there will be less than n edges in the wait_for graph. So, by assumption no deadlock will be declared. Suppose that the transaction receives a not granted message. So, it will move to the waiting state as the result of action 3. The wait for graph will have precisely n edges in this instance (if there were more a cycle would exist). This transaction, as the result of this action, will begin propagating an update message. Suppose that every transaction in this system has a value for wait_for. That would imply a deadlock exists. So in this system, there exists a transaction whose wait_for variable must be nil. Similarly, there exists a transaction in the system whose Request_Q is nil. Therefore, the update message will reach a transaction that has a nil Request_Q and the message will stop. So, no deadlock will be declared as a
consequence of action 3. Suppose $n+1$ transactions are waiting and $n$ edges exist in the graph, when action 5 occurs a transaction will move back to the running state. With only $n-1$ edges now in the graph the inductive assumption holds. Action 6 considers what will happen upon the receipt of an update message. A transaction which receives an update message will propagate that message to its Request_Q.

Action 6 has a similar argument as Action 3. So, no phantom deadlocks are declared.

3.5 Proof of Liveness Property

Suppose in this system a deadlock will occur in a finite amount of time. Consider a distributed system with two transactions. So, $|C(G)|$ is precisely 2 when the system is deadlocked. There are two possible cases for a deadlock to develop in this system. The first instance can occur when both transactions are in the running state and simultaneously request resources that the other transaction has a locked.

Let the system be at time $T_i$ when such a request occurs. So, action 1 will occur in the system for both transactions. So at $T_{i+1}$, both transactions will have sent their requests. Without loss of generality, assume that at $T_j$, where $j > i+1$, that both transactions receive a message indicating that the resource was not granted. So, each transaction will execute Action 3 at $T_{j+1}$. Both will be in the waiting state and a cycle will exist in this transaction wait-for graph. Each transaction must distribute the information regarding their state change to waiting. This is accomplished by propagating the update message at $T_{j+2}$. In a finite amount of time, one or both transactions will receive the message from the other regarding this state change (denoted by $T_k$, where $k > j+2$). So, the difference between the Request_Q and
Wait_for variables will not be nil (if they were nil, one transaction would be in the running state). Therefore, a deadlock would be declared at time \( T_{k+1} \).

The other case can occur when a transaction is in the waiting state and the other transaction needs a resource from it to complete. At \( T_i \), the running transaction makes a request for a resource and moves to the pending state. In a finite time this transaction will receive a not granted message from the other transaction. At that point, the transaction will move to the waiting state, forming the deadlock in the system. This transaction must then propagate the information regarding the state change to the other transaction. Once the other transaction receives the update message, it will declare deadlock because its Request_Q and Wait_for difference is not nil. Therefore, in the base step deadlock is declared.

Now, assume a system with \( n \) transactions will declare deadlock for all \( |G(G)| \leq n \). Consider a system with \( n+1 \) transactions. Again, there are two possible cases, simultaneous requests that result in deadlock or two requests that are separated in time that generate a deadlock.

Let \( |G(G)| = n-1 \) and \( T_i \) be the time in the system. When simultaneous requests are made the \( n \) and \( n+1 \) edges are added to the transaction wait-for graph. Thus, forming the cycle. When this occurs two transactions begin propagating update messages in the system. At \( T_{i+n+1} \), the two transactions making the request to form the cycle will receive an update message and will execute Action 6. So, both will discover the deadlock.

Suppose \( n \) transactions are waiting for the \( n+1 \) transaction to complete.
However, that transaction makes a request that generates a cycle when $|C(G)| = n+1$. The system has deadlock. When that transaction makes that particular request and eventually receives the not granted message it will be propagating the update message through the system. At $T_i + n + 1$ the transaction which formed the cycle will receive an update message. This transaction will discover that its Request_Q and Wait_for difference are not nil and declare a deadlock.

Therefore, the proposed algorithm detects all deadlocks in a finite time. So, both criteria for correctness have been verified and the proposed algorithm is correct.
CHAPTER 4

CONCLUSION

This thesis addresses one of the most important topics in distributed deadlock detection and distributed algorithms in general, formal verification. The techniques of formal verification through temporal logic can be applied to a wide variety of distributed algorithms.

Chapter 2 was devoted to development of the fundamentals of a distributed database. The area of concurrency control, through a data manager, was detailed. A discussion of the two phase locking protocol was also given. Finally, these design principles coalesced in the proposed edge-chasing distributed deadlock detection algorithm.

The emphasis of this thesis was to describe temporal logic and use that structure to formally verify the proposed algorithm. By first defining the six legitimate actions of the system and the three properties which exist in temporal logic, first order logic was extended to include temporal properties. Working with safety and liveness properties within the system and temporal logic, it was possible to demonstrate the
correctness of the proposed algorithm.

Most distributed deadlock detection algorithms developed, do not address the formal verification of correctness. Consequently, the major advantage to formal verification, is the ability to develop and verify correct distributed deadlock detection algorithms.
BIBLIOGRAPHY


[18] A. Kshemkalyani and M. Singhal, Invariant-based verification of distributed deadlock detection Algorithm


