connecting line whose suppression induces the minimum response time increase (solving this problem involves optimum routing determination for messages concerned).

In part two, we assume that access delays to the channels are exponentially distributed. This assumption is valid for an access protocol such as ALOHA, which is, of course, badly suited to local networks, and it would be useful to model, in terms of access time, more appropriate protocols (carrier sense). Service disciplines taking into account a higher priority for acknowledgment messages should also be studied. Diffusion approximation models may also be considered [8] as tools for examining these problems.

References

[1] E. Gelenbe, J. D. Geyl, O. Gibergues, and E. Horlait, "Réseau local à diffusion de paquets sur fibre optique: Le système XANTHOS," Lab. de Recherche en Informatique, Univ. Paris-Sud, Res. Rep. 63, Dec. 1979.

[2] J. Labetoulle, "Modélisation des systèmes informatiques et applications aux réseaux d'ordinateurs," Thèse Sc. Math., Univ. Paris Sud, Nov. 1978.

[3] W. J. Gordon and G. G. Newell, "Closed queueing systems servers," *Oper. Res.*, vol. 15, pp. 254-265, 1967.

[4] J. Buzen, "Computational algorithms for closed queueing networks with exponential servers," *Commun. Ass. Comput. Mach.*, vol. 16, pp. 527-531, 1973.

[5] F. Baskett, M. Chandy, R. Muntz, and J. Palacios, "Open, closed and mixed networks of queues with different classes of customers," *J. Ass. Comput. Mach.*, vol. 22, Apr. 1975.

[6] W. Chu, "Optimal file allocation in a multiple computer system," *IEEE Trans. Comput.*, Oct. 1969.

[7] P. J. Courtois, *Decomposability–Queueing and Computer Systems Applications* (ACM Monograph Series). New York: Academic, 1977.

[8] E. Gelenbe and I. Matrani, *Analysis and Synthesis of Computer Systems.* London: Academic, 1980.

Guy Bernard was born in La Guerche, France, in 1947. He received the Ecole Centrale des Arts et Manufactures degree in 1970 and the Doctorat de Troisieme Cycle degree in statistics in 1977.

Since 1974 he has been Assistant Professor in Statistics and Computer Science in the Université Paris-Nord, Villetaneuse, France. He joined the research group on modeling and performance evaluation of computer systems of the Laboratoire de Recherche en Informatique, Université Paris-Sud, Orsay, France, in 1978 and the Laboratoire d'Informatique des Systemes Experimentaux et leur Modelisation (ISEM), Université Paris-Sud, Orsay, France, in 1982. His present interests include protocol evaluation, interconnection of local area networks and performance evaluation of computer communication systems.

A Causal Model for Analyzing Distributed Concurrency Control Algorithms

BHARAT BHARGAVA, MEMBER, IEEE, AND CECIL T. HUA, MEMBER, IEEE

Abstract—An event order based model for specifying and analyzing concurrency control algorithms for distributed database systems has been presented. An expanded notion of history that includes the database access events as well as synchronization events is used to study the correctness, degree of concurrency, and other aspects of the algorithms such as deadlocks and reliability. The algorithms are mapped into serializable classes that have been defined based on the order of synchronization events such as lock points, commit point, arrival of a transaction, etc,.

Index Terms—Causal graph, concurrency control, correctness, deadlock, degree of concurrency, distributed system, event order, history, reliability, serializability, time stamp.

I. Introduction

SEVERAL algorithms for concurrency control or synchronization in distributed database management systems have been proposed in the literature and have been surveyed in [3], [5]. Specific proofs of correctness for individual algorithms have appeared in [2], [18], [25]. This paper presents a general approach for analyzing algorithms. This analysis is used to study several desirable properties of the algorithms such as proof of correctness, the allowable degree of concurrency, deadlock freedom, and robustness. A simple hypothetical concurrency control algorithm has been used to illustrate the approach. In [12], the majority consensus approach [25] has been analyzed as an example. In [4], examples of the analysis of locking algorithms and other types of algorithms have been included. The notion of serializability [19] is used to establish the degree of concurrency and as the correctness criterion. The deadlock freedom refers to the absence of an infinite delay of either the transaction or the system processing. The robustness refers to the capability of an algorithm in handling certain hardware/software failures and/or lost messages.

The basic approach used in this paper is as follows. 1) Identify all events that occur due to the processing of the transactions or the concurrency control algorithm. 2) Establish an

Manuscript received July 15, 1980; revised September 15, 1982.

B. Bhargava is with the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260.

C. T. Hua is with the Large Computer Products Division, Honeywell Information System, Inc., Phoenix, AZ 85005.

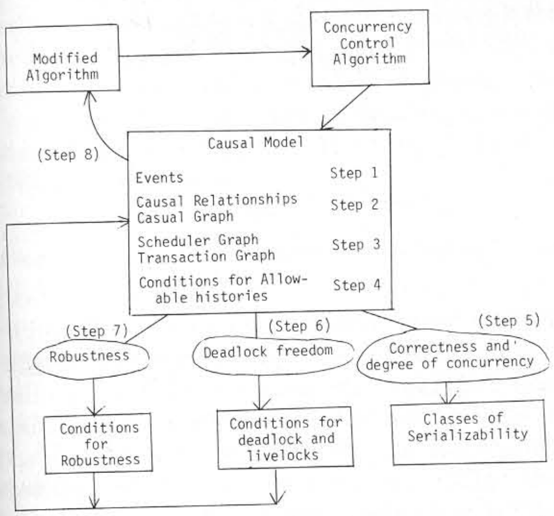

Modified Algorithm
Concurrency Control Algorithm
(Step 8)
Causal Model
Events
Step 1
Causal Relationships
Casual Graph
Step 2
Scheduler Graph
Transaction Graph
Step 3
Conditions for Allowable histories
Step 4
(Step 7)
(Step 6)
(Step 5)
Robustness
Deadlock freedom
Correctness and degree of concurrency
Conditions for Robustness
Conditions for deadlock and livelocks
Classes of Serializability

Fig. 1. Steps for analyzing and modifying concurrency control algorithms.

order (causal relationship) among all events. This gives a causal graph. 3) Derive subgraphs representing event ordering for the transaction and the concurrency control algorithm. The subgraphs are like a template representing generic processing for a transaction or a node. 4) Using the transaction and node subgraphs, determine the conditions for the allowable histories generated by the concurrency control algorithm. 5) Check if these conditions are satisfied by a subclass of all possible serializable histories. The classes of the serializability for distributed database systems have been established in [6]. If the conditions are satisfied, the degree of concurrency and correctness are verified; otherwise the algorithm produces nonrecognizable histories. 6) Study if there exists a sequence of events that lets the processing of a transaction or a concurrency controller proceed from an initial event to the terminal (final) event. If so deadlock freedom is guaranteed. 7) Check if the occurrence of some event transition is based on certain processing on another node or certain message transmissions. If so, the algorithm is not robust with respect to the failure of such processing or message transmission. 8) Identify events that are necessary for the deadlock freedom and robustness and add them to the original set of events. Establish relationships among these new events and other existing algorithms. This gives a new causal graph. From this improved causal graph obtain a new modified algorithm.

The above steps are also shown in Fig. 1. In this paper, only the first seven steps as discussed above have been presented. The last step is the topic of a future report.

In Section II, the causal model has been presented. In Section III, the mechanism of the analysis of concurrency control algorithms with an illustrative algorithm has been presented. The basic terminology has been presented in Appendix A. The classes of serializable histories for distributed database systems have been presented in [6], and the semantic information used in analyzing certain classes of algorithms has been included in [13]. Finally, a brief comparison of the causal model to other models such as Petri nets and path expressions has been included.

II. The Causal Model

In this section, a causal model for representing concurrency control algorithms is introduced. The model is based on the identification of events in an algorithm and the causal relationship among the events [4], [16]. The purpose is to map the specification of algorithms to the classes of serializable histories.

The first section describes a model for concurrency control algorithms. The model specifies the states and the events of concurrency control algorithms. The second section explains the causal rules which represent the causal relationship among the events. The third section describes the causal graph and node/transaction subgraphs to represent possible ordering of events. A hypothetic concurrency control algorithm is used for illustration.

A. A Model for Concurrency Control Algorithms

A *distributed concurrency control algorithm* Φ can be modeled as a collection of *local schedulers* $\{\Phi_n\}$ such that Φ_n is the local scheduler running at node n. The local schedulers represent the distributed components of the overall concurrency control algorithm.

Each local scheduler acts like a finite state machine, and has independent state and event specifications as well as local data structures. All local schedulers are not necessarily homogeneous. For example, a synchronization algorithm with centralized control may have one local scheduler running on a central (controlling) node and all other local schedulers as the slave processes. In such a case, the finite state machine for the central scheduler is considerably more complex than others. Since each node is assumed to have only one local scheduler, the term "node" may also be used in the following sections to refer to the scheduler at that node.

Each local scheduler Φ_n can be characterized by a quintuple $<S_n, E_n, M_n, D_n, L_n>$ where S_n is a set of states, E_n is a set of events, M_n is a set of message types, D_n is a set of local data structures, and L_n is a set of local operations on M_n and D_n.

A *state* of a local scheduler represents a finite period of local computation. Unlike the "state" of a variable, the state here refers to a stage in the execution of the algorithm rather than the status value of some variable. In a state, an algorithm can perform any number of operations with one exception: sending and receiving messages must be modeled as state transitions. Examples of legal operations in a state include operations on local data structures, message preparation, predicate evaluation over local data and/or message contents, etc.

An *event* is an activity that causes a scheduler to change its state. For example, as required by this model, sending and receiving messages constitute events. An event involving no message exchange models a transition between processing stages of an algorithm. In this case, the event signifies the completion of the processing in the previous state and the start of the next state.

Since each event is associated with a state transition of a local scheduler and vice versa, the set of events for a local scheduler can be viewed as the state transition function of the scheduler. An event is a member of the set $S_n \times S_n$. Also,

```
               e
S1 --------------> S2
                       f
S3 ---------MESS -----------> S4
   MESS/f     g
S5 --------------------S6
```

Fig. 2. The schematic representation of states and events.

messages can be identified by remote events since the act of sending a message must be modeled as an event.

In a schematic representation of an algorithm, events are represented by directed edges connecting states. Message sending will be shown as the message id embedded in the edge representing the sending event. Message reception will be denoted by tagging the message id or the sending event just beside the state from which the receiving event starts. For example, in Fig. 2, event *e* causes the scheduler to change from state *S*1 to state *S*2. Message MESS is sent when event *f* takes place (from state *S*3 to state *S*4). The receiving node changes state (event *g*) from state *S*5 to state *S*6 upon receiving the message MESS. Since MESS is sent in event *f*, the tag beside state *S*5 can also be *f* instead of MESS.

The *message types* M_n specify message id's and the message formats.

The *local data structures* D_n contain all local variables accessible only by Φ_n. The database at a node is considered as a part of the local data structure since it is accessible only by a local scheduler.

The *local operations* of Φ_n, L_n, are defined over M_n and D_n. Computational details are specified in L_n.

A concurrency control algorithm achieves synchronization by coordinating updates on local nodes, and the process of coordination is done through proper message exchanges among nodes. These message exchanges are events. The database accesses are associated with particular events in the algorithm. Hence, an output history of an algorithm can be viewed as a sequence of scheduler events embedded between the events associated with access operations. The read and write operations are associated to special events of a scheduler. The sequence of system events, thus, represents a detailed description of the synchronization process by a concurrency control algorithm. This is the notion of *expanded history*, i.e., history of atomic operations plus embedded scheduler events. Each event can be associated to a transaction and/or to a node. The node and transaction projections of an expanded history can be defined. This expanded notion of "history" of a system will be used in the following discussions.

The notation $e(i,j)$ is used to designate an event *e* for transaction *i* at node *j*. We would avoid the subscripts whenever possible and represent $e(i,j)$ by *e*. The subscript will be necessary when the identification of the transaction and the node for an event is important in the context of other events.

The notion of events and states is further illustrated by the state transition diagram of a hypothetic concurrency control algorithm in Fig. 3. The sending of a message is represented by embedding the message id in the transition arrow; each transition is also marked with corresponding events, ei, $0 \leqslant i \leqslant 5$.

The algorithm works as follows. A local scheduler waits

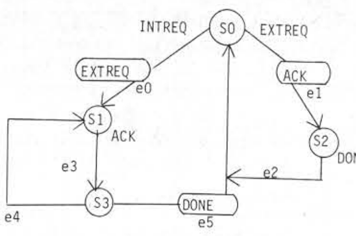


Fig. 3. The state transition diagram of a hypothetical algorithm.

for incoming transactions in the state *S*0 (the idle state). Upon receiving the request from a transaction (INTREQ), the local scheduler broadcasts a message EXTREQ to every other node (event *e*0) to obtain locks. The scheduler must initialize the counter and lock X_n. The scheduler then waits for acknowledgment (ACK) from all other nodes before proceeding to execute the transaction. This is implemented by local operations A_n^+ on the local variable A_n. Event *e*4 represents the waiting loop for acknowledgments. A DONE message will be broadcasted to inform schedulers at other nodes that the transaction has completed. If a scheduler is in the idle state and receives an EXTREQ message, it will send back an ACK message to acknowledge the request for locking and hold the lock for that transaction until a DONE message is received.

The local variables used by the local scheduler in this hypothetic algorithm include a simple counter for registering the acknowledgment messages and a lock variable for the data item *Y* (one data item assumed). Let $\Phi_n = \langle S_n, E_n, M_n, D_n, L_n \rangle$ be the local scheduler. Then

S_n: states = $\{S0, S1, S2, S3\}$;
E_n: events = $\{e0, e1, e2, e3, e4, e5\}$;
M_n: message types = {INTREQ, EXTREQ, ACK, DONE};
D_n: local data structures =
  A_n: a counter for ACK responses;
  X_n: a variable indicating whether the lock is held currently by some transaction;
  Y_n: the database at node *n*;
L_n: local operations =
  A_n^+: increase the counter by 1;
  X_n^+: increase X_n by 1 indicating the locking;
  A_n^-: initialize the counter to zero;
  X_n^-: unlock by resetting X_n to zero;
  $W_i^n[Y]$: local accesses to Y_n by transaction *i*.

The causal model requires reasonable specification of the operational aspects of an algorithm before the analysis of the algorithm.

B. Causal Rules

From the basic computational model, it can be observed that events of a concurrency control algorithm do not occur randomly. There are certain relationships among them. These relationships can be specified by causal rules.

1) Definitions of Causal Rules: A *causal rule U* is a quintuple $\langle u, @, v, L, P \rangle$ where u, v are events, and @ is one of the *causal relationships* $\{\rightarrow, \Rightarrow, \ggg\}$.

The causal relationship $\rightarrow$ specifies the ordering of nonmessage-related events. $<u,\rightarrow,v,L,P>$ is a causal rule if there are states $x,y,z \in S_n$ such that $u=<x,y>$, $v=<y,z>$,[1] the local operation L is performed in state y, P is a Boolean condition for v to occur, and no messages are involved in activating v. In other words, the algorithm of Φ_n must have the following state transition.

$$x \xrightarrow{u} y \xrightarrow{v} z.$$

Intuitively, event u precedes event v on node n, and Φ_n executes event v following the occurrence of u without waiting for messages. Between these two events, the local operation L will take place. The predicate P must be defined over D_n and M_n (it can test the contents of previous messages). If $<u,\rightarrow,v,L,P>$ is a causal rule for some L and P, then event u is said to *precede* event v (or v *follows* u).

The other two causal relationships, $\Rightarrow$ and $\Rrightarrow$, describe the ordering between message-related events. $<u,\Rightarrow,v,L',P>$ and $<w,\Rrightarrow,v,L'',Q>$ are causal rules iff there are states $x,y,z \in S_n$, $t,s \in S_k$, $k \neq n$, such that $u=<x,y>$, $v=<y,z>$, $w=<t,s>$, w is the event that node k sends a message to node n, and node n responds with event v at state y. L' is the local operation of node n in state y, which is independent of the message sent by the remote event w. L'' is the local operation of node n in state y that can be performed only after the message is received. P is the predicate that node n chooses to wait for messages after the event u, and Q is the condition that the message of event w will be recognized by node n. Schematically, nodes n and k must have the following state transitions.

node k: $t \xrightarrow{w} s$

node n: $x \underset{u}{\longrightarrow} y \underset{v}{\overset{w}{\longrightarrow}} z.$

It denotes that node n, while waiting at state y, receives a message sent by event w of node k, and then executes event v in response. Event w is said to *cause* event v if $<w,\Rrightarrow,v,L,Q>$ is a causal rule for some L and Q. Note that the causal rules $<u,\Rightarrow,v,L',P>$ and $<w,\Rrightarrow,v,L'',Q>$ are related to each other; no causal rules $<u,\Rightarrow,v,L',P>$ should exist without the corresponding $<w,\Rrightarrow,v,L'',Q>$ causal rules.

Either L or P in the above causal rules can be null. A null L signifies that no local operation is associated with the events involved, and a null P indicates that the causal relationships are unconditional, i.e., independent of local data values or messages.

The predicate P of a causal rule $<u,\Rrightarrow,v,L,P>$ can also be used to specify the sites receiving the message generated in event u. Hence, either message broadcasting or daisy-chain transmission can be described by a proper predicate P. The detailed semantics, such as the exact node id to which a message is sent, are in general not critical to the causal relationship $\Rrightarrow$.

The following notation is used in this paper to designate a causal rule $<u,@,v,L,P>$:

$u @ v + L$ if P.

[1]Recall that an event is a member of $S_n \times S_n$.

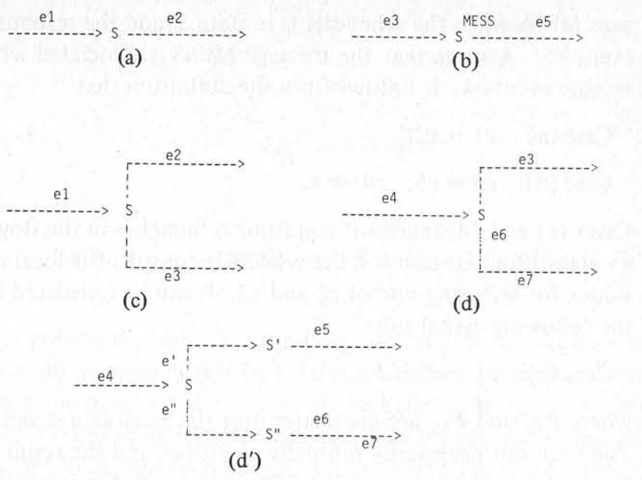


Fig. 4. Sample flow diagrams of an algorithm.

For example, the following causal rules are derived from the hypothetic algorithm in Fig. 3. In the listing, i,m designates transactions, and j,k designates nodes. $W_i^j[Y]$ represents the database access by transaction i to item Y at node j. Let INIT(i) be the originating node of transaction i and N be the set of nodes.

$$e0(i,j) \Rrightarrow e1(i,k) + X_k^+, \ \forall k \neq j;$$
$$e1(i,j) \Rrightarrow e3(i,k) + A_k^+, \ k = \text{INIT}(i), \ k \neq j;$$
$$e3(i,j) \rightarrow e4(i,j) \text{ if } A_j < |N| - 1;$$
$$e3(i,j) \rightarrow e5(i,j) + W_i^j[Y] + X_j^- \text{ if } A_j = |N| - 1;$$
$$e5(i,j) \Rrightarrow e2(i,k) + W_i^k[Y] + X_k^-, \ k \neq j;$$
$$e0(i,j) \Rightarrow e3(i,j) + A_j^- + X_j^-;$$
$$e4(i,j) \Rightarrow e3(i,j);$$
$$e1(i,j) \Rightarrow e2(i,j);$$
$$e5(i,j) \Rightarrow e0(m,j), \ m \neq i;$$
$$e5(i,j) \Rightarrow e1(m,j), \ m \neq i;$$
$$e2(i,j) \Rightarrow e0(m,j), \ m \neq i;$$
$$e2(i,j) \Rightarrow e1(m,j), \ m \neq i.$$

In the rule $e0(i,j) \Rrightarrow e1(i,k)$, the predicate "$\forall k \neq j$" indicates the broadcasting of messages, whereas in the rule $e1(i,j) \Rrightarrow e3(i,k)$, k is uniquely specified. The local access to the database is associated with the event $e5$; this is indicated by the rule $e3(i,j) \rightarrow e5(i,j)$. The update by a remote transaction is associated with the event $e2$, which is initiated by a message from a remote $e5$ event. Notice that in the rule $e0(i,j) \Rightarrow e3(i,j)$, the local operations of initializing A_j and X_j are included since they are independent of the response message. The last four rules indicate that the event $e0$ or $e1$ after the event $e5$ or $e2$ refers to a different transaction.

2) Inference of Causal Rules: To be an effective model for distributed concurrency control algorithms, causal rules should be easily derivable from common descriptions of an algorithm. An informal description of the mapping procedure follows.

Consider the examples in Fig. 4. They are sample flow diagrams of an algorithm. Case (a) represents the processing of a scheduler in state S; the scheduler activates $e2$ after entering state S via event $e1$. Case (b) represents the receipt of a message MESS when the scheduler is in state S, and the response is event $e5$. Assume that the message MESS is associated with a remote event $e4$. It follows from the definition that

Case (a): $e1 \rightarrow e2;$

Case (b): $e3 \Rightarrow e5;\ e4 \twoheadrightarrow e_5.$

Cases (c) and (d) represent conditional branches in the flow of an algorithm. In case (c), the scheduler consults the local data values for selecting one of $e2$ and $e3$. It can be translated into the following causal rules:

Case (c): $e1 \rightarrow e2 \text{ if } P_{12};\ e1 \rightarrow e3 \text{ if } P_{13};$

where P_{12} and P_{13} are predicates over the local data structure. They are not necessarily mutually exclusive, and the result can be nondeterministic execution. Case (d) stands for conditional processing depending on the presence or absence of messages. State S will choose $e7$ if a message is received, otherwise it will continue with $e5$. The definition of the relationships does not prohibit such behavior, which can be represented by

Case (d): $e4 \rightarrow e5 \text{ if } P;\ e4 \Rightarrow e7;\ e6 \twoheadrightarrow e7;$

where P is a predicate indicating the absence of message $e6$. However, case (d) can be alternatively represented as case (d′), where

Case (d′): $e4 \rightarrow e' \text{ if } P;\ e4 \rightarrow e'' \text{ if } \neg P;$
$e' \rightarrow e5;\ e'' \Rightarrow e7;\ e6 \twoheadrightarrow e7.$

The decomposition of state S of case (d) into the states S, S', and S'', of case (d′) may also reduce the complexity of deriving semantic conditions for causal rules.

So far, only the procedure for identifying the type of causal relationships between events has been discussed. To specify the local operation and the predicate part of a causal rule, the semantics within a state must be examined, and this step is important because the modeling power of causal rules depends on the semantic interpretation.

Local operations can usually be identified by collecting the semantics of each statement or the primitive in a particular state. Predicates are derived from the conditions on the control path through which the state activates an event. Specifically, for the causal rule $<u, \rightarrow, v, L, P>$, L is the local operation performed in the intermediate state between u and v. P is the predicate such that the state will perform local operations and finally lead to the activation of v. For the causal rules $<u, \Rightarrow, v, L', P>$ and $<w, \twoheadrightarrow, v, L'', Q>$, L' comprises the local operations between u and v that are independent of the message sent by w; L'' comprises the local operations after the message is received and recognized. Predicate P denotes the condition that after event u the state will choose a control path that leads to waiting for messages. Q is the condition that the state recognizes the message sent by w and activates event v.

These semantic specifications may not be easy to derive for certain operations and control flow. One solution is to refine the definition of the state and identify events which are on a lower level of abstraction. This process is similar to the process shown for case (d) where the original state S has been refined further to include extra states S' and S'' in case (d′). With a finer level of states and events, semantic inference is easier. The semantics of concurrency control algorithms is usually not difficult to specify as exemplified by the above hypothetical algorithm and as shown in [12].

The set of causal rules is not large. If the local schedulers are homogeneous, their causal rules are identical. In this case, the only difference between the specifications of two schedulers is in identifying the receiving node of a message for the relationship $\twoheadrightarrow$. However, the semantics of selecting the receiving node is fixed and can be represented by suitable quantifiers to the predicate. For example, broadcasting a message can be represented by the following causal rule:

$$u(i,j) \twoheadrightarrow v(i,k)\ \forall k \neq j, j, k \in N,$$

instead of repeating the rule for different nodes (N is the set of nodes). Hence, only one set of relationships for the scheduler needs be specified and others can be "folded" onto the set.

For heterogeneous schedulers, the number of specifications is also relatively small. For instance, only two graphs are necessary for the centralized control algorithm, one for the central scheduler and the other representing all other nodes.

C. Causal Graph

A *causal graph* $G = <V, E>$ for a set of causal rules of an algorithm is a labeled diagraph with vertices $V = \{ e \mid \text{events} \}$ and edges $E = \{ <e, f> \mid$ there exists a causal relationship @, local operation L, and predicate P such that $<e, @, f, L, P>$ is a causal rule$\}$. The vertices and the edges are labeled with their corresponding events and causal relationships. The edges in a causal graph are referred to as $\rightarrow$ edges, $\Rightarrow$ edges, or $\twoheadrightarrow$ edges according to their labels. The paths in a causal graph represent possible event sequences in the execution history.

The *node subgraph* consists of those paths containing only $\rightarrow$ and $\Rightarrow$ edges. These paths represent the possible execution history of each local scheduler at one node. A path in the node subgraph is called a *node path*.

The *transaction subgraph* consists of those paths containing $\twoheadrightarrow$ edges. A path in the transaction subgraph is called a *transaction path*. Events on transaction paths of the transaction subgraph are events for a single transaction. Transaction paths represent the message switching behavior of the schedulers concerning transaction synchronization. The derivation procedures of these subgraphs are explained below.

For concurrency control algorithms, there is one initial or wait state. In the wait state, say $S0$, a scheduler waits for a new transaction. All the events ending at $S0$ will have $\Rightarrow$ edges to all the events starting from $S0$. This implies that it is possible to have many cycles formed by these $\Rightarrow$ edges in the causal graph. The events returning to state $S0$ are not critical in the processing of the transactions. For example, new requests from a user are processed by starting the scheduler from the initial state $S0$. Thus, a scheduler returning to $S0$ without completing the processing of an older transaction must be able to resume the processing some time later. This is often done through local data structures such as a waiting list. As far as the local scheduler is concerned, the current processing of the older transaction at this node is completed. Events entering the idle state $S0$ represent the termination of a node's current processing of a transaction and are called *terminal node events*. Similarly the events starting from $S0$ are called *initial node events*.

To find the node subgraph from a causal graph, first remove from the causal graph all ↠ edges, and then remove all ⇒ edges from the terminal node events to the initial node events. The remaining → and ⇒ edges are meaningful for processing at a single node. The ↠ edges can be removed because they relate events at different nodes. The ⇒ edges from terminal node events to initial node events can be removed because of the arguments about the initial state $S0$. They are not related to the transaction processing at a node. Their presence is implied when the terminal and the initial node events are identified.

To find the transaction subgraph, it is necessary to identify the initial and the terminal events for transaction processing. An *initial transaction event* neither follows any event nor is caused by any event. It represents spontaneous activities such as user requests. An event $e(i,j)$ is a *terminal transaction event* if no further processing for transaction i at node j is needed. For concurrency control algorithms, terminal transaction events signify the final decision of acceptance or rejection on a transaction.

The following steps are used to find the transaction subgraph.

1) Remove all ⇒ edges from the causal graph.

2) For any event which is not a transaction terminal and has no outgoing edges, include all outgoing ⇒ edges that are related to the processing of the same transaction.

3) For any event with incoming ⇒ edges (generated by the previous step), also include all outgoing ⇒ edges that are related to the processing of the same transaction.

The intuition behind step 1) is based on the assumption that messages are necessary for concurrency control in the distributed environment. When a local scheduler is waiting for a message, it is the remote event of sending the message that carries most of the control information. In other words, the control flow for processing a transaction can be viewed as moving from node to node along with message transmission. Note that by "advancing" it does not necessarily mean that transactions are forwarded from node to node or remote procedures are invoked one by one. For example, the broadcasting of locking requests represents that the control flow for processing the transaction has forked simultaneously at all other remote nodes; the control flow will not return to the initial node until a response message is received. Hence, the ⇒ edge between the message broadcasting event to the message receiving event does not carry any semantic significance for the transaction processing. The ⇒ edge, however, does carry important ordering information for the node processing. It is therefore included in the node subgraph as explained previously in the procedure for generating node paths. The → edges are not removed because they represent the order between local events (events involving no messages). Both node processing and transaction processing have to follow the order depicted by the → edges.

The intuition behind step 2) is that nonterminal transaction events without outgoing → or ↠ edges (called *halting transaction events*) represent a temporary halt in the control flow of processing transactions. A typical example is a pending transaction waiting for the release of locks held by, say, a higher priority transaction. No messages can be generated and no further local processing is possible for that transaction. Hence the local scheduler may put the transaction in a local waiting list and resume normal processing. The transaction will be awakened by later messages. Hence, there are only outgoing ⇒ edges from this event. In this case, the flow of processing for this transaction will be represented by these ⇒ edges, and such edges should be included in the transaction subgraph. Complication arises in selecting which outgoing ⇒ edges are to be included. Because a local scheduler usually resumes normal processing after a halting transaction event, outgoing ⇒ edges from the event may lead to events that are not related to the halted transaction. In this case, semantics expressed by the predicate part of the causal rules for those ⇒ edges have to be examined. Only the ⇒ edges leading to those events related to the processing of the same halted transaction should be included.

Step 3) follows the same intuition as in step 2). It deals with the case that an awakened transaction may be asked to wait again. The awakening message may generate further messages that are associated with other transactions; this is represented by the outgoing ↠ edges from the event of receiving the awakening message. However, if the awakened transaction should be put to wait again, its processing flow cannot follow the outgoing ↠ edges, which is for messages of other transactions. Hence, the outgoing ⇒ edges are included to indicate the fact that the halted transaction must be processed locally. As in step 2), semantics have to be examined to determine which edge is related to processing the same transaction.

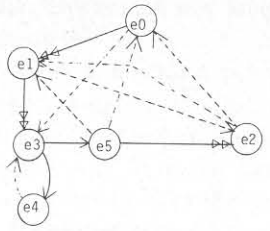


Fig. 5. The casual graph for the hypothetical algorithm.

The causal graph of the hypothetic algorithm of Fig. 3 is shown in Fig. 5. To simplify the drawing, the ⇒ edges are represented by dotted arrows. The transaction subgraph is shown in Fig. 6 and the node subgraph is shown in Fig. 7. The initial transaction event is $e0$, and the terminal transaction events are $e5$ and $e2$. The initial node events are $e0$ and $e1$, and the terminal node events are the same as the terminal transaction events. The terminal events are represented by a triangle rather than a circle.

The node subgraph is derived by removing all ↠ edges and all ⇒ edges from the terminal node events to the initial node events. The transaction subgraph is derived by first removing all ⇒ edges, then including the ⇒ edge between $e4$ and $e3$. The ⇒ edge from $e4$ to $e3$ is included because $e4$ is a nonterminal transaction event without outgoing → or ↠ edges and the ⇒

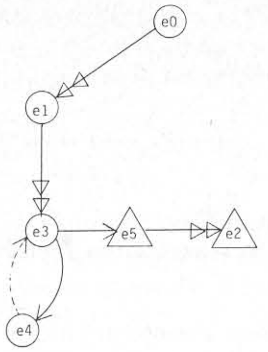


Fig. 6. The transaction subgraph for the hypothetic algorithm.

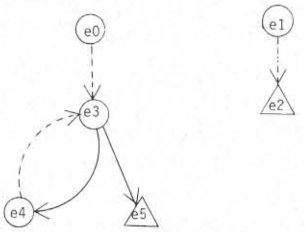


Fig. 7. The mode subgraph for the hypothetic algorithm.

edge is related to the same transaction (see the corresponding causal rule in Section II-B1).

D. Summary of the Causal Model

The causal model proposed in this paper is intended for verifying concurrency control algorithms of distributed database systems. An algorithm is modeled as a set of local schedulers; each scheduler is associated with a node in a distributed database system. The scheduler is first represented by a state transition diagram, where a state represents a finite period of local processing. Messages are modeled as state changes, and each state change constitutes an event. Then the causal relationships between these events are described by causal rules. A graphical representation of the causal rules, the causal graph, is constructed to assist in determining the possible event ordering. By considering events related to a single node and events related to a single transaction, two subgraphs can be derived from the causal graph. The paths in these two subgraphs and the semantic conditions related to each edge will be used in the final verification process.

III. Analysis of Concurrency Control Algorithms Using the Causal Model

In our model for distributed database systems, the notion of history is used to model system behavior. The concurrent read/write activities of the transactions are modeled as a sequence of atomic operations. The events related to a particular scheduler or a transaction can be described by subsequences called node projections and transaction projections. This sequence (either the overall history or the projections) represents the order of those atomic operations in real time. The basic terminology and formal definitions of transaction, history, and serializability are included in Appendix A.

The performance of a concurrency control algorithm can be measured by several parameters, e.g., average response time, average throughput, number of messages required to correctly synchronize, the degree of concurrency allowed, etc. Some of these measures are dependent on the types of transactions arriving in the system while some others are based on the particular characteristics of the system parameters, e.g., a secondary storage access I/O cost, CPU processing cost, data structures employed to broadcast messages, etc.

The degree of concurrency allowed by a concurrency control algorithm is a measure that is independent of the particular system parameters and has been used by [14], [19]. A synchronization algorithm is viewed as a scheduler which observes the sequence of operations requested by different transactions. If a particular sequence (or part of the history) is recognized by the scheduler to be acceptable, the scheduler allows further processing. If the scheduler cannot recognize a particular history, it will change the execution sequence of some operations so as to map the current history into one that is accepted by the scheduler. This may require delaying the operations of some transactions. If an algorithm recognizes more histories to be serializable, to a lesser extent, it will interfere with them. Thus, by studying the class of serializable histories recognized by the algorithms, we can compare their degree of concurrency. This measure is a qualitative comparison of algorithms rather than a quantitative comparison.

The problem of testing any arbitrary history for serializability has been shown to be NP-complete [19]. Several subsets of the serializable histories, however, have been identified for fast serializability test. These subsets called *classes* are defined by histories acceptable to a scheduler. We have extended the definitions of these classes for histories generated in a distributed database system. In [6], five classes are defined: global two-phase locking (G2PL), local two-phase locking (L2PL), distributed conflict preserving (DCP), distributed serializable in time-stamp order (DSTO), and distributed strictly serializable (DSS).

To prove the correctness of an algorithm, the recognizable histories of the algorithm are tested against these classes. If the histories of an algorithm are contained within a known class, then the correctness of the algorithm is guaranteed and the relative power of this algorithm with respect to the algorithms whose histories are contained in other classes can also be decided [14].

This section is further organized as follows. In the first subsection, the link between the order of events in a history and the causal rules is shown. In the second subsection, the mechanisms to check the semantic conditions in the causal rules are discussed. The steps for verifying an algorithm for the serializability of its output histories are described in the third subsection. Comments on the usage of causal graphs for investigation of other aspects of the algorithm, such as deadlock possibility and reliability, are included in the fourth subsection.

A. Event Ordering and Causal Rules

In the causal model, event sequences are represented by paths in the causal graph. A path in the causal graph defines the partial order for the events of the path in the history. The synchronization process of the schedulers can be traced by traversing the causal graph. An event A will precede another event B in the history if a path from A to B can be found in the causal graph.

We introduce two sets of events $Q(e)$ and $R(e)$ that are reachable from an event e in the causal graph. Formally they are defined as follows.

Let $Q(e) = \{ f \mid f$ is an event and there exists local operation L, predicate P such that $<e, \Rrightarrow, f, L, P>$ is a causal rule$\}$, and $R(e) = \{ f \mid f$ is an event and there exists local operation L, predicate P such that $<e, \Rightarrow, f, L, P>$ or $<e, \rightarrow, f, L, P>$ is a causal rule$\}$.

The predicate part of each causal rule represents the condition that the specified causal relation between the two events will appear in the execution history of the system. For an event $e(i,m)$ in the history, the local scheduler at node m will execute an event $f(k,m), i,k \in T$, as the very next event if 1) there is either $\Rightarrow$ or $\rightarrow$ causal relation between $e(i,m)$ and $f(k,m)$, and 2) the predicate for that causal relation has been satisfied. In other words, $f \in R(e)$. For a message-sending event $e(i,j)$, some remote nodes will subsequently execute an event $f(k,m), i,k \in T, j,m \in N, j \neq m$ if 1) there is $\Rrightarrow$ causal relation between $e(i,j)$ and $f(k,m)$, and 2) the predicate for that causal relation has been satisfied. Hence, $f \in Q(e)$. If e is a halting transaction event, i.e., an event with no outgoing $\rightarrow$ or $\Rrightarrow$ edges in the transaction paths, the next event for the halted transaction will be one of the events reachable from e via $\Rightarrow$ edges. Hence, $f \in R(e)$. In all cases, the event $f(k,m)$ can be viewed as *selected* by the predicate from the corresponding set–$R(e)$ for node processing or $R(e) \cup Q(e)$ for transaction processing. The following proposition summarizes the above discussion and relates the partial ordering of scheduler events in an execution history to the causal graph.

Proposition 3.1:

i) $\pi^i(f(k,j)) = \pi^j(e(i,j)) + 1$ for some $i, k \in T, j \in N, e, f \in E_j$ iff $f(k,j)$ is selected from $R(e)$.

ii) If $e(i,j)$ is an event for transaction i and $f(i,k)$ is selected from $R(e) \cup Q(e)$ for the same transaction $i, j, k \in N$, then $\pi_i(f(i,k)) > \pi_i(e(i,j))$.

iii) An event $f \in R(e)$ iff f is adjacent to e in the node subgraph.

iv) For events e, f of the same transaction, $f \in R(e) \cup Q(e)$ iff f is adjacent to e in the transaction subgraph.

The proof for Proposition 3.1 is intuitive and straightforward. Proposition 3.1i) states that any event which occurred immediately after event e in a node history must be one of the events in $R(e)$ since otherwise the local scheduler Φ_n could not possibly have advanced from e to f. Proposition 3.1ii) states that the next event of the same transaction selected from $R(e) \cup Q(e)$ should occur after e. Proposition 3.1iii) relates the set $R(e)$ to the node subgraph. The set $R(e)$ can be derived from the node subgraph since all $\Rightarrow$ and $\rightarrow$ edges are in the node subgraph. Proposition 3.1iv) relates the events of the same transaction in $R(e) \cup Q(e)$ to the transaction subgraph.

The permutation order of a node projection is related to the node subgraph in Proposition 3.1 by i) and iii). The set of possible next events for an event e at the same node are identified as those in the set $R(e)$. The exact selection of the next event depends on the semantics and the predicate part of the causal rule. If the selection is not unique, then nondeterministic choice is possible.

If some of the events in $R(e)$ can never be selected, they become useless events because they can never be activated. The algorithm can be redesigned to remove such events, and the resulting algorithm will have the same behavior as the original one. If no events in $R(e)$ can be selected, then the scheduler will be deadlocked after event e unless e is the last event of the history. Hence, it follows from Proposition 3.1i) that there exists at least one event from $R(e)$ for any event e in a history such that $\pi^j(f) = \pi^j(e) + 1$ unless e is the last event or a deadlock has occurred at node j. We assume that an algorithm is free from useless events before it is analyzed.

The permutation order of a transaction projection is related to the transaction subgraph in Proposition 3.1 by ii) and iv). One event f in the set $R(e)$ or $Q(e)$ must occur after e if both e and f are events of the same transaction. The event f may not occur immediately after e in the transaction projection because, for example, the event e may be a message broadcasting event and f is one of the responses. Then, the event f by a particular scheduler may not immediately follow e since this f may not be the first response to e.

If an event e is a terminal transaction event, then the events in $R(e)$ must be associated with another transaction (from the definition of terminal transaction events). If a nonterminal transaction event e cannot select from $R(e)$ or $Q(e)$ a next event for the transaction, then the transaction is deadlocked. Hence, for a nonterminal and nondeadlocked transaction event e, there exists at least one event f from either $R(e)$ or $Q(e)$ such that f is for the same transaction and $\pi_i(f) > \pi_i(e)$. Proposition 3.1iii) and 3.1iv) indicates that the selected events can be determined from examining the node and the transaction paths.

From the above proposition, the following theorem can be proven.

Theorem 3.1: Let $e(i,j), f(k,m)$ $i,k \in T, j,m \in N$, be events in a history corresponding to events e, f of the causal graph. If $f(k,m)$ is selected from $R(e) \cup Q(e)$ then $\pi(f(k,m)) > \pi(e(i,j))$.

Proof: From the definitions of the node and the transaction projections, $\pi(e) < \pi(f)$ iff $\pi^j(e) < \pi^j(f)$, and $\pi(e) < \pi(f)$ iff $\pi_i(e) < \pi_i(f)$ for events e, f. If $f(k,m)$ is selected from $R(e) \cup Q(e)$, it is related to either the node processing or the transaction processing. If $f(k,m)$ is selected from $R(e)$ (node processing), then $m=j$ and $\pi^j(f(k,j)) = \pi^j(e(i,j)) + 1 > \pi^j(e(i,j))$ from Proposition 3.1i). If $f(k,m)$ is selected from $Q(e)$ (transaction processing), then $k=i$ and $\pi_i(f(i,m)) > \pi_i(e(i,j))$ from Proposition 3.1ii). It follows from the above mentioned properties of π_i and π^j that $\pi(f(k,m)) > \pi(e(i,j))$. ∎

Theorem 3.1 says that any event from $R(e) \cup Q(e)$ selected by some predicate for an event e must follow e in the history. More than one event can be selected by different predicates, but all selected events must follow Theorem 3.1.

B. Checking the Semantic Conditions of Causal Graphs

In a causal graph, the edges reflect the ordering defined by the causal rules; the semantic information is captured by local operations and the predicate associated with the corresponding causal rule. The local operations and the predicate are defined over local data structures. Hence, it is necessary to examine the value of local data structures for checking the semantic conditions for the paths in the causal graphs. The following theorem tells us about the conditions for effectively checking the semantic conditions.

Theorem 3.2: The semantic conditions necessary for a path from one event to another in a causal graph can be effectively decided if the causal rules contain only first-order predicates and primitive recursive functions.

Proof: The proof is presented by considering two cases for the paths in the causal graph as discussed in the following.

Case 1–A Single Edge in the Causal Graph: Let D_n be the local data structure at node n, and let $(D_n)^u$ be the value of D_n just after the occurrence of an event u. For the causal rule

$$u \rightarrow v + L \text{ if } P$$

and the corresponding edge in the causal graph, the data value after v, $(D_n)^v$, can be characterized as

$$(D_n)^u\, P\, \{L\}.$$

It means that L is applied to $(D_n)^u$ if the predicate P is true over $(D_n)^u$. The above expression can be viewed as a small program, which evaluates the predicate P on the given input $(D_n)^u$ and performs the operation L. Given $(D_n)^u$ and simple P, L such as first-order predicates for P and primitive recursive functions for L, $(D_n)^v$ can be effectively determined. Once $(D_n)^v$ can be decided, the semantic conditions for any events which are reachable from v via $\rightarrow$ edges can also be determined. Let v be followed by some other events like in

$$v \rightarrow e + L_1 \text{ if } P_1;$$

$$v \rightarrow f + L_2 \text{ if } P_2;$$

then P_1 and/or P_2 can be effectively decided, and

$$(D_n)^e = (D_n)^v\, P_1\, \{L_1\} = (D_n)^u\, P\, \{L\}\, P_1\, \{L_1\};$$

$$(D_n)^f = (D_n)^v\, P_2\, \{L_2\} = (D_n)^u\, P\, \{L\}\, P_2\, \{L_2\}.$$

In this manner, given the initial value of D_n, the semantic conditions for any $\rightarrow$ edges in the causal graph can be checked.

Similarly, the semantics of $\Rightarrow$ and $\ggg$ relations can be represented as follows:

$$u \Rightarrow v + L' \text{ if } P';$$

$$w \ggg v + L'' \text{ if } P''.$$

Let n be the node at which the events u and v occur, then $(D_n)^v$ can be characterized as

$$(D_n)^u\, P'\, \{L'\}\, P'' \wedge (w \text{ occurred})\, \{L''\}.$$

The data value of D_n after u, $(D_n)^u$, is tested for P', then L' is applied to D_n. L'' will not be applied unless the condition $P'' \wedge (w \text{ occurred})$ is satisfied. The first part of the above notation for $(D_n)^v$, $P'\{L'\}$, represents the semantic interpretation of the causal rule

$$u \Rightarrow v + L' \text{ if } P';$$

where P' is independent of the message content from the event w. The second part has a predicate "w occurred," which represents the semantics for the occurrence of the event w. How to check this predicate is the major task in checking semantic conditions of causal graphs.

Let m be the node where the event w occurred. There are two aspects in representing the semantics of "w occurred."

1) The data value $(D_m)^w$. This will characterize the local operations associated with the event w at node m. Predicates on $(D_m)^w$ can be determined by the same inference rules as described here.

2) The temporal order of the event w. The temporal relationship between the event w and other events can be inferred as follows. If there are causal rules

$$e \rightarrow w$$

$$f \rightarrow w$$

then "w occurred" implies that "e or f occurred." Exactly which of the events e and f occurred depends on the semantic conditions of the two causal rules. To infer these conditions, the same procedure described in this section can be used. If there are causal rules

$$e \Rightarrow w$$

$$f \ggg w$$

then "w occurred" implies that "e and f occurred." The semantic conditions for "e and f occurred" can again be inferred recursively using the procedure in this section.

Case 2–A Loop in the Causal Graph: A loop in the causal graph represents possible repetitions of a set of events, and an important semantic condition for a loop is the *exit condition* of that loop. The exit condition refers to any semantic condition that can stop the cyclic repetitions of the events on a loop. The cyclic processing is stopped either by a terminal event or by the invocation of some other events which are not on the loop. A loop containing terminal events will not cause endless repetitions because the terminal events signify the completion of the processing; the edges coming out of the terminal events are either referring to different transactions or not semantically significant (see the discussions on generating node/transaction subgraphs). Hence, in general, the exit conditions are the predicates that enable an event which is not on the loop.

For example, consider the following causal rule

$$u \rightarrow v + L \text{ if } P$$

where u is involved in a loop which does not involve v. Assume that n is the node at which the events u and v occurred. An exit condition for that loop is established when P is satisfied by $(D_n)^u$. Another situation for an exit condition is represented by the following causal rules

$$u \Rightarrow v + L' \text{ if } P';$$

$$w \ggg v + L'' \text{ if } P''.$$

Let the events u and v occur at node n, the event w at node m. If u is involved in a loop which does not involve v, then an exit condition for the loop is established when "w occurred" and P'' is true over the data value

$$(D_n)^u \; P' \; \{L'\}.$$

(Compare this exit condition to the second part of the notation designating the value $(D_n)^v$). Note that if in the above case w is also on a loop in which v is not involved, then the above exit condition for the loop involving u is also a possible exit condition for the loop involving w. The loop involving w is broken when "w occurred" and P'' allows the responding event v to occur.

From the previous discussions on inferring the semantic conditions about a single edge, the exit conditions of the loops can also be effectively decided. ∎

Example: To illustrate the inference procedure described above, consider the following causal rules for the hypothetical algorithm.

$$e0(i,j) \Rrightarrow e1(i,k) + X_k^+, \; \forall k \neq j;$$
$$e1(i,j) \Rrightarrow e3(i,k) + A_k^+, \;\; k = \text{INIT}(i), \;\; k \neq j;$$
$$e3(i,j) \rightarrow e4(i,j) \text{ if } A_j < |N| - 1;$$
$$e3(i,j) \rightarrow e5(i,j) + W_i^j[Y] + X_j^- \text{ if } A_j = |N| - 1;$$
$$e0(i,j) \Rightarrow e3(i,j) + A_j^- + X_j^+;$$
$$e4(i,j) \Rightarrow e3(i,j).$$

From the third and the fourth causal rules, $(D_j)^{e3}$ will be tested against two predicates to determine which of the events $e4$ and $e5$ will follow. The two predicates, "if $A_j < |N| - 1$" and "if $A_j = |N| - 1$," can exclusively choose one of $e4$ and $e5$. $(D_j)^{e4}$ is identical to $(D_j)^{e3}$ since no local operations are involved in the third causal rule. $(D_j)^{e5}$ will be

$$(D_j)^{e3} \; \text{“}A_j = |N| - 1\text{”} \; \{W_i^j[Y] + X_j^-\}$$

where the update of transaction i will be performed and X_j will be initiated to zero. Note that the predicate "$A_j = |N| - 1$" will be true for both $(D_j)^{e3}$ and $(D_j)^{e5}$ since no operations on A_j are involved in the fourth causal rule. The acknowledgment count A_j will be initialized in the fifth causal rule.

The first and the second causal rules represent the message switching behavior for a transaction request. If $j = \text{INIT}(i)$, the initial node of transaction i, then the second rule can be restated as

$$e1(i,k) \Rrightarrow e3(i,j) + A_j^+, \;\; k \neq j$$

where k is some remote node having received the message by $e0(i,j)$ (the first rule). If node j has local data value $(D_j)^{e0}$, then the second and the fifth rule will produce

$$(D_j)^{e3} = (D_j)^{e0} \; \{A_j^- + X_j^-\} \; \text{“}e1(i,k) \text{ occurred”} \; \{A_j^+\}.$$

If node j has local data value $(D_j)^{e4}$, then the second and the sixth causal rules will produce

$$(D_j)^{e3} = (D_j)^{e4} \; \text{“}e1(i,k) \text{ occurred”} \; \{A_j^+\}.$$

The predicate "$e1(i,k)$ occurred" can be characterized by the data value $(D_k)^{e1}$ and the temporal ordering of events which leads to the occurrence of $e1(i,k)$. From the causal rules (the complete list of causal rules for the hypothetical algorithm), "$e1(i,k)$ occurred" implies that "$e0(i,j), j = \text{INIT}(i)$, occurred."

In the transaction subgraph of the hypothetical algorithm (Fig. 6), there is only one cycle formed by $e3$ and $e4$. The exit condition is specified in the fourth causal rule. Although the exit condition is "$A_j = |N| - 1$," the semantic condition for the sixth causal rule has to be considered since the loop involves this $\Rightarrow$ edge. From the above discussion, the increment to A_j will occur only after $e1(i,k)$ occurred, and $e1(i,k)$ occurred only because $e0(i, \text{INIT}(i))$ occurred. From the quantifiers on k in the first and the second rule, only one $e1(i,k)$ at any node $k \neq \text{INIT}(i)$ can respond to $e0(i, \text{INIT}(i))$. Hence, the exit condition "$A_j = |N| - 1$" is equivalent to "$e1(i,k), \forall k \neq j$, occurred." Since the database update $W_i^j[Y]$ is associated with the occurrence of the event $e5(i,j)$, from the exit condition, the database update will not occur until "$e1(i,k), \forall k \neq j$, occurred." This demonstrates how the event ordering semantics can be inferred from the exit condition and the local data values.

Important semantics for each class have been identified in [13].

C. Verification Strategies

In this section, the procedure for verifying the serializability of the output histories of an algorithm is informally presented. The basic idea is to view the causal graph and the subgraphs as specifications for the order of system events. Using Theorem 3.2 and Proposition 3.1, all possible permutations of system events can be derived from the node/transaction paths. Hence an examination of the paths and the order of events specified by them can decide whether all implied histories are serializable.

Three possible strategies are discussed below. The first strategy is for verifying algorithms based on locking. The second strategy is for verifying algorithms based on time stamps. The third strategy is for verifying algorithms which output DCP histories.

The procedure for finding whether a given set of causal rules of an algorithm will generate a known class of serializable histories can be characterized as follows. First, examine the algorithm for obvious synchronization mechanisms such as locks, time stamps, etc. If the algorithm uses locks, the strategy for verifying locking algorithms can be applied. If time stamps are used, the strategy for time-stamp-based algorithms is applicable. It is recommended that the verification process should start from a smaller class (G2PL or DSTO). The larger classes should be attempted if the algorithm cannot be shown to generate the smaller class. In this way, the characteristics of the histories which are allowed by the algorithm but not in the smaller class can usually be observed during the verification procedure, which can assist in finding sample histories for showing the hierarchical relationship between the algorithm's output and other classes. If none of the strategies for DSTO/DSS and G2PL/L2PL is applicable, the strategy for the class DCP should be applied. However, the class DCP lacks a general strategy and the verification procedure proposed here is based on heuristics.

1) Strategy for Verifying Locking Algorithms: Algorithms based on locking are expected to generate either G2PL or L2PL histories. Let the event α_i represent the first database access for transaction i, and let the event ω_i represent the last database access (read or write) to an entity never accessed before by the transaction i or a write access to an entity never written before by the transaction. Intuitively, the ω_i signifies the committment of a transaction. For more details, refer to [6].

Definition G2PL: A history h is in the *global two-phase locking* (G2PL) class iff there exists a set of global lock points $\{L_i \mid i \in T\}$ such that for transactions i and j

i) $\pi(\alpha_i) \leq L_i \leq \pi(\omega_i)\ \forall i \in T$.

ii) If σ_i and σ_j conflict, and $\pi(\sigma_i) < \pi(\sigma_j)$ then a) $L_i < L_j$, and b) $\pi(\sigma_i) < L_j$.

Definition L2PL: A history is in the *local two-phase locking* (L2PL) class iff there exists a set of local lock points $\{L_i^j \mid i \in T, j \in N\}$ such that for transactions i and j

i) $\forall i \in T\ L_i^k \leq \pi^k(\sigma_i)$ if $\pi(\omega_i) \leq \pi(\sigma_i)$, and $\pi^k(\alpha_i) \leq L_i^k$ if α_i is on node k.

ii) If σ_i and σ_j conflict on node k, and $\pi^k(\sigma_i) < \pi^k(\sigma_j)$ then a) $L_i^k < L_j^k$, and b) $\pi^k(\sigma_i) < L_j^k$,

iii) $L_i^k < L_j^k \Leftrightarrow L_i^m < L_j^m\ \forall k, m \in N$.

Note that mutual exclustion, the conventional class of synchronization in the operating systems, is just a special case of the class G2PL. For mutually exclusive accesses, consider the case $L_i = \alpha_i$. The conflicting accesses will be forced to execute serially. In other words, accesses are mutually exclusive since $\pi(\sigma_i) < L_j = \alpha_j$ for conflicting transactions i and j.

The verification strategy for locking algorithm has the following steps.

a) Identify in the transaction subgraph the event(s) which represent the lock points.

b) Identify in the node subgraph all events where the read/write atomic operations can occur. These events are called *access events*. Consider all possible pairs of these access events. For each pair call the first event of each pair σ_i and the second σ_j.

c) Find all the necessary events on the node paths from σ_i to σ_j. "Necessary" means that these events are on all the paths from σ_i to σ_j in the node subgraph.[2]

d) For every event e identified in step c), find in both transaction and node paths the corresponding lock point event(s) which either precede or follow the event e.

e) By Proposition 3.1 and Theorem 3.2, infer the order between those events identified in c) and d).

f) Test whether the order in part ii) of the definitions G2PL or L2PL is satisfied.

g) If all the event pairs meet the required order, then the algorithm only produces histories in the class G2PL or L2PL.

Step b) considers only node paths since σ_i and σ_j in part ii) of the definitions that are on the same node. Step c) traces the node paths to identify the events which can decide the order between σ_i and σ_j. Such identified events will be the pivot points in the paths for inferring the order. Step d) uses the transaction paths to determine the order between the lock point event and the access event since they both belong to the same transaction. If necessary, node paths will also be used.

[2]It is understood that there are implied $\Rightarrow$ edges between the terminal node events and the initial node events. These edges are usually not important semantically, but are considered here because they are useful in determining the events between σ_i and σ_j.

Step e) is the major step. It has to infer all possible ordering for identified events. Step f) and g) test the conditions for G2PL or L2PL and lead to conclusions.

In step e), the inference process may be quite involved since cycles are possible. Cycles in the causal graph and the subgraphs represent events that are executed repeatedly in the actual history. The number of repetitions must be shown to be finite to exclude the presence of an infinite loop. The cycles may also cause inconsistent ordering. For example, if both the α event and the access event σ are in a cycle, then both orderings $\pi(\alpha) < \pi(\sigma)$ and $\pi(\sigma) < \pi(\alpha)$ are possible. Careful examination of the semantics as described in the previous sections will be necessary to decide the proper order.

In step a) the events representing the global or the local lock points of a transaction must be identified. It may not be possible to identify this event in a straightforward manner. However, some useful heuristics are listed below.

1) This event must be an event with local locking operations since the lock point is reached when a transaction acquires all the necessary locks.

2) This event should precede an access event due to the fact that a lock point must precede the ω_i access event and that accesses can only be done after the lock has been granted.

3) This event may be the converging point of $\ggg$ edges in the transaction paths for G2PL algorithms. Since synchronization by global locking often involves broadcasting request and acknowledgment messages, this event may be activated by the receipt of messages.

2) Stragegy for Verifying Time-Stamp-Based Algorithms: For classes based on the locking approach such as G2PL, there are identifiable events and specific ordering rules. For classes such as DSTO, only the final serialization order is specified. The serialization order is defined by time stamps. The definitions of two classes DSTO and DSS that are based on time stamps are listed below.

Definition DSTO: The class DSTO contains all histories that are *distributed serializable in the time-stamp order*. A history $h = \langle D, T, \Sigma, \pi \rangle$ is in DSTO iff there exists a serial history $g = \langle D, T, \Sigma, \rho \rangle$ such that

i) $h^j \equiv g^j\ \forall j \in N$, and

ii) $\forall i, j \in T, i \neq j, \pi(\alpha_i) < \pi(\alpha_j)$ implies $\rho(\alpha_i) < \rho(\alpha_j)$.

Definition DSS: The class DSS contains all histories that are *distributed strictly serializable*. A history $h = \langle D, T, \Sigma, \pi \rangle$ is in DSS iff there is a serial history $g = \langle D, T, \Sigma, \rho \rangle$ such that

i) $h^j \equiv g^j\ \forall j \in N$, and

ii) $\forall i, j \in T, i \neq j, \pi(\omega_i) < \pi(\alpha_j)$ implies $\rho(\omega_i) < \rho(\alpha_j)$.

The histories in these classes relay on the order of the α event and/or the ω event to decide the final serialization order. Time stamps can be viewed as a means to register the order of α events for transactions.

The verification strategy for time-stamp-based algorithm has the following steps.

a) Identify the event(s) defining the final serialization order. In the case of DSTO, it is the event α for each transaction; for the class DSS, the ω_i event must also be considered.

b) Infer the order of access events for two conflicting transactions at each node and its relation to the order of the event(s) that define the serialization order.

c) Prove that the equivalent effect of accesses by conflicting transactions conforms to the order of the α/ω events.

Step c) relies on the semantic interpretation and the ordering information obtained from step b). Step b) is a general statement about a process that involves examining detailed semantics of each causal rule. Normally the process will trace the node/transaction paths in both directions from the event(s) α/ω to determine other access events and their relationship to conflicting accesses.

3) Strategy for Verifying Algorithms Producing DCP Histories: The histories in the class DCP have acyclic dynamic conflict graphs (DCG). The definition for DCG is given below.

Definition DCG: A dynamic conflict graph (DCG) for a history $h = <D, T, \Sigma, \pi>$ is a digraph $<V, E>$: V is the set of vertices which is the same as T, the set of transactions. E is the set of edges, where $<i, j>$ is an edge if and only if there exist atomic operations σ_i for transaction i, σ_j for transaction j such that σ_i, σ_j conflict and $\pi(\sigma_i) < \pi(\sigma_j)$.

From the definition, a DCG of a history is a graph structure that reflects the order among conflicting database accesses of several transactions. An acyclic DCG implies mean consistent ordering between the accesses from any two conflicting transactions. An acyclic DCG guarantees that the history is serializable. This provides a test for whether a history is in class DCP or not.

To maintain acyclic DCG distributively in an algorithm, however, is not easy. One technique is to associate the order of database accesses to the order of a special event. Since the order of any particular event in processing each transaction is necessarily acyclic, i.e., the special event for one transaction cannot possibly precede and follow the special event for another transaction simultaneously due to the natural law of time, associating the order of accesses to a special event guarantees acyclic DCG. The technique, of course, can only permit a subset of the DCP histories. As far as the author knows, no general algorithms which implement the whole DCP class have been proposed. The verification strategy below is geared toward this technique, but extensions of the verification strategy for general DCP algorithms are possible.

The verification strategy is as follows.

a) Identify the special event that decides the order of conflicting database accesses.

b) Infer the relationship between the temporal order of the events representing conflicting database accesses and the temporal order of the special events.

In step b) the inference procedure can use the techniques for inferring the relationship between the access events and the lock point events in the strategy for locking algorithms. Step a) may involve heuristics to determine the special event. A good guess is when the first write access of a transaction is accepted since that instance will make the effect of a transaction visible to other transactions.

4) Example: To illustrate how to use the strategy for locking, we now analyze the hypothetic algorithm for mutual exclusion. In [12], we have analyzed the majority consensus algorithm [25]. Recall that mutual exclusion is a special case of G2PL; the global lock point is the first access event, α. In the algorithm, database access activities are associated with two events, $e5$ and $e2$. The event $e5$ represents the global lock point since it is the first access activity, the α event, during transaction processing.

Lemma 3.1: The hypothetic algorithm only produces mutually exclusive accesses.

Proof: To prove the lemma, first, the event $e5$ is identified as the α event of this algorithm since all $e2$ events must follow $e5$ ($e2$ is only reachable from $e5$ in the transaction subgraph). Then, the event $e5$ of this algorithm is identified as the global lock point defined in Definition G2PL. The proof will follow the strategy for verifying locking algorithms. Consider two conflicting transactions i, j and $\pi(\sigma_i) < \pi(\sigma_j)$ for accesses σ_i, σ_j at node k. Then their accesses to the database on node k are either $e5$ or $e2$ in this algorithm. There are four cases.

1) Both $e5$. It is obvious that the condition of Definition G2PL is satisfied.

2) $\sigma_i = e5(i, k)$ and $\sigma_j = e2(j, k)$. $\pi(e5(i, k)) < \pi(e2(j, k))$. Since $e2$ is reachable only from $e1$ in the node subgraph (Fig. 7), the local scheduler at node k must execute $e1(j, k)$ immediately before $e2(j, k)$ (Proposition 3.1). Hence, $\pi(e5(i, k)) < \pi(e1(j, k)) < \pi(e2(j, k))$. In the transaction subgraph in Fig. 6, there is a path from $e1$ to $e5$; hence $\pi(e1(j, k)) < \pi(e5(j, m))$ for some node m (Theorem 3.2). This is the only possible ordering between $e1(j, k)$ and $e5(j, m)$ since $e1$ and $e5$ are not on a cycle. Therefore $\pi(e5(i, k)) < \pi(e5(j, m))$, which satisfies Definition G2PL.

3) $\sigma_i = e2(i, k)$ and $\sigma_j = e5(j, k)$. Again, from the transaction subgraph (Fig. 6), $\pi(e5(i, m)) < \pi(e2(i, k))$ for some node m. Hence, $\pi(e5(i, m)) < \pi(e5(j, k))$.

4) Both $e2$. Since $e2$ is the only member in $R(e1)$ from the node subgraph (Fig. 7), $\pi_k(e2(i, k)) = \pi_k(e1(i, k)) + 1$ by Proposition 3.1 (i). Hence $\pi(e1(i, k)) < \pi(e2(i, k)) < \pi(e1(j, k)) < \pi(e2(j, k))$ due to the fact that $\pi(e2(i, k)) < \pi(e2(j, k))$ and the properties of π_k. Since $\pi(e5(i, m)) < \pi(e2(i, k))$ and $\pi(e1(j, k)) < \pi(e5(j, n))$ for some node m, n (the requirement from the transaction subgraph), $\pi(e5(i, m)) < \pi(e2(i, k)) < \pi(e5(j, n))$. ■

Since $e5$ satisfies the condition of global lock point and it is also the α event (see the transaction subgraph), this algorithm will produce mutually exclusive accesses.

D. Other Aspects of Verification

Two operational aspects of an algorithm are the deadlock possibility and the reliability. The use of the causal model for the analysis of these aspects is presented in the following.

1) Deadlock Possibility: Both deadlock and livelock will delay a transaction indefinitely. The term "deadlock" will be used to refer to the indefinite delay of processing for the sake of simplicity.

The emphasis here is to analyze whether a deadlock is possible for an algorithm. The following proposition is based upon the observation in Section III-A.

Proposition 3.2: A concurrency control algorithm is deadlock free if and only if

a) the processing of a transaction starting from initial transaction events can reach the terminal transaction events in a finite number of events in the transaction projection;

b) every local scheduler starting from initial node events can reach terminal node events in a finite number of events in the node projection.

The justification for this proposition is intuitive.

Since event ordering in the node (transaction) projection is closely related to the node (transaction) path, the following list of conditions are used to verify the deadlock possibility of an algorithm.

1) Every nonterminal event must be on a path leading to terminal node/transaction events.

2) For every nonterminal event u, there is at least one event v in $R(u) \cup Q(u)$ such that v can be selected as the next event after u.

3) Events on the cycles in the transaction/node subgraphs can only be selected finitely many times for the same transaction.

Condition 1) guarantees that there exists a possible sequence of events which can take a local scheduler or a transaction to termination after the occurrence of a nonterminal event. Condition 2) requires that no events can lead to a dead-end situation without any possible next event. Condition 3) limits the number of times a transaction can execute the events on a cycle.

Theorem 3.3: Conditions 1), 2), and 3) are satisfied if and only if a concurrency control algorithm has the properties a) and b) in Proposition 3.2.

Proof:

a) The sufficient condition. Recall the assumption that an algorithm does not have "useless" events—events which will never be selected. Then it is clear that the processing of an algorithm will not be guaranteed to reach terminal events if condition 3) is not satisfied. If an event cannot select a next event [violation of condition 2)], the algorithm is deadlocked once its processing reaches this event. If an event is not on any path which may lead to terminal events, the processing of an algorithm will not be able to reach terminal events after this event no matter what next event it selects. Hence condition 1) is also necessary for the properties in Proposition 3.2.

b) The necessary condition. There are two aspects: the processing must end in terminal events and the processing must reach there via a finite number of events. Assume that some transaction (node) processing stops at a nonterminal event. It directly violates condition 2). Assume that some transaction (node) processing has incurred an infinite number of events. Since there are only a finite number of transactions and a finite number of events in the transaction (node) subgraphs, there must be a cycle such that the processing can repeatedly select events from the events on the cycle. This is a violation of condition 3). ∎

Since many deadlock or livelock situations can be examined only through semantic information, the complete analysis of the deadlock possibility cannot be performed by testing merely the ordering of system events.

2) Reliability: A complete analysis of the reliability aspects of a distributed concurrency control algorithm is beyond the scope of this paper. In what follows is an informal description of how to utilize the causal graph and event ordering to infer some characteristics of the reliability of an algorithm.

One way to interpret the causal rules, or the edges in the causal graph, is to view them as ordering specifications. This is the approach used in Section III-A. Another interpretation, however, is to view them as dependency specifications. If a causal relation exists between events u and v, i.e., $<u, @, v, L, P>$ is a causal rule for some $@$, L, and P, then the occurrence of v is dependent on the occurrence of u. For example, if $u \twoheadrightarrow v$, then v depends on the occurrence of u. Further, it implies that v also depends on the reliable transmission of the message by event u. Hence, studying the effect generated by assuming that one of the causal rules failed to occur can reveal the reliability of an algorithm, especially the dependency of an algorithm on certain events.

A possible scenario of the analysis is as follows.

1) Assume that some causal rule is not fulfilled, which indicates a failure situation.

2) Infer how many events are going to be affected: are they altered or prohibited?

3) Examine the corresponding effects on the node and the transaction projections, respectively.

The types of failures that can be modeled by step 1) are node failures (e.g., $u \rightarrow v$ is nullified), and lost messages (e.g., $u \twoheadrightarrow v$ is nullified). This type of analysis is at best qualitative in nature but should be adequate in identifying possible consequences of failures.

3) Example: To illustrate the above concepts, the hypothetic algorithm is analyzed for its deadlock possibility and reliability. To test whether a node will deadlock, the paths between initial node events and the terminal node events must be shown not to block the local scheduler [Proposition 3.2a)].

From the node subgraph (Fig. 7), event $e0$ is followed only by $e3$ and the only condition for $e3$ to occur is the occurrence of event $e1$. Also, the terminal node event $e5$ can be reached if and only if all other nodes have performed $e1$ (from the transaction subgraph in Fig. 6). Hence, a node cannot be guaranteed to complete processing of a transaction unless all other nodes execute $e1$. It can happen if the two local schedulers simultaneously execute the event $e0$. Neither of them can execute $e1$ before executing $e5$, which can be executed only if the other executes $e1$. Thus, deadlock is possible in this algorithm.

The deadlock possibility can also be shown in another way. In the node subgraph (Fig. 7), there is only one path from $e1$ to the terminal event $e2$. Hence, only when a DONE message is received a node can finish the processing of a remote lock request. Any node which performs $e1$ cannot terminate unless the transaction is finished. However, the completion of a transaction depends on the global consensus; two nodes which simultaneously perform the event $e1$ for different transactions will create a deadlock because no transactions can get the global consensus.

The reliability of this algorithm can be determined by examining the node paths and the $\twoheadrightarrow$ causal rules for the events on a node path. There are only three types of messages in this algorithm; they are represented by $e0$, $e1$, and $e5$. When an $e0$ message (EXTREQ) from a node to another is lost, i.e., $e0 \twoheadrightarrow e1$ is nullified, no corresponding $e1$ is possible since $e0$ is the only cause for $e1$. No global consensus is possible for the transaction and the transaction is deadlocked. A lost $e1$ (ACK) will have the same effect. Since $e5$ (DONE) is the only ause for $e2$, which is the terminal event for the node path l $\Rightarrow e2$, a node will not be able to finish if an $e5$ message is ost. Hence, any lost message causes deadlock, and this fact s indicated by the structure of the transaction subgraph Fig. 6): Any loss of messages in one of the events $e0$, $e1$, and 5 effectively breaks all the paths from the transaction initial vent ($e0$) to the transaction terminal events.

Any node failure will cause the algorithm to halt. As indiated by the node subgraph in Fig. 7, any local scheduler tarting from event $e0$ can reach the terminal node event $e5$ nly if $e1$ is responded to by all other nodes. If one of the odes failed to respond, the node that initiates $e0$ will be kept aiting, and no other transactions can proceed. On the other and, if the node initiating $e0$ fails to complete $e5$, all other odes cannot proceed either since the only possible way to tart $e1$ is after the node completes $e5$.

The algorithm is faulty because it allows deadlock to occur, nd is not reliable since deadlock will occur when the messages re lost or a node fails.

IV. Comparison to Other Models

We compare the causal model to three other modeling mechnisms: Petri nets, L systems, and path expressions. The omparisons are based on the descriptive capability for synhronization schedulers. Neither the exhaustive list of the odeling mechanisms nor the descriptive capability of the hree for other modeling purposes is included. The comparison s presented for discussion and to show the specific advantage f our approach rather than to make a general claim on its erits. For more details, refer to [13].

A. Petri Nets

Petri nets, surveyed in [20], are abstract formal models for nformation and control in systems exhibiting concurrent and synchronous behavior. They consist of places where one or more tokens can reside and the places are connected via transiions. A transition is enabled (fired) when all of its input places ave tokens. The firing of a transition causes tokens to be moved from their input places to their output places. The modeling of distributed processing is done by associating logcal interpretations to places, tokens, and firing rules. For xample, places can be thought of as different programs, such s producers and consumers, where tokens are resources to be roduced and consumed. Firing transitions can be used to odel the process of producing and consuming the resources. By properly connecting places, transitions into a net, and lacing the initial marking of tokens, the producer–consumer roblem can be represented. Petri nets can be used to model wo aspects of systems: events and conditions, and their elationships.

To prove the correctness of schedulers, we have to express he scheduler in the nets. To model a synchronization algoithm, the capability of representing value-dependent semantics s required. The Petri nets in their original form are not quite ble to do this without increased complexity. The predicate/ransition nets represent an extension of the Petri nets and are btained by adding predicates and semantics interpretation. detailed comparison of a distributed concurrency control algorithm using the causal model and predicate/transition nets [10], [26] has been done in [13]. From this study, we observe that the transformation from parallel processes to the modeling Petri nets is nontrivial. Even if a net could be constructed, some problems about the net such as the reachability problem, i.e., whether a target marking can be reachable from a given marking, have an exponential lower bound of complexity. Other problems such as the equivalence of reachable marking sets are even undecidable. The complexity bounds as well as the large number of places and transitions needed to model indicate that, although some analysis questions may be decidable using Petri nets, in a general case the cost of deciding may make such analysis unfeasible.

Another limitation of nets seems to be the difficulty in specifying the correctness condition for concurrency control such as serializability. It is easy to associate the condition of mutual exclusion with the exclusive presence of tokens in one place but the specification of the serializability condition is not obvious since it is necessary to know the ordering of events in the history. The conditions over place markings prescribe a set of configuration of tokens in places. These conditions seem to be memoryless in nature, i.e., how the tokens arrive at the prescribed configuration is not specified.

The causal model can easily represent value-dependent semantics through local data structures and local operations in the causal rules. The very general form of predicates for causal relations enables us to express the semantic interpretation of an algorithm. The purpose of the causal graph model is to concentrate on the global picture of event ordering. Events and their causal relations are also natural extensions from the descriptions of algorithmic state transitions, which increase the descriptive power of our model.

Another advantage of expressing read/write requests as events is that the transformation from serializability conditions to event ordering information is straightforward. Not only can problems such as mutual exclusion be modeled by the causal graph model, more complex synchronization policy like the two-phase locking protocol can also be described as event ordering constraints.

B. Linguistic Models

Ellis introduced the modeling of synchronization schedulers in distributed databases by L systems [8]. L systems, originally studied in [22], are similar to phrase structure grammers but with no terminal symbols and with simultaneous replacement rules selected from an arbitrary finite set of tables. Ellis expresses the synchronization algorithm in terms of evaluation nets, a modified form of Petri nets, and then translates the evaluation net semantics into a set of L system tables. The rules mimic the operation of each node's state transition and the changes of local variables by manipulating a string of symbols assembled from the state symbols of each node and the local variables. The message switching is modeled by letting one node examine/change other nodes' states and variables, i.e., the L system grammer is context sensitive. The algorithm's properties such as mutual update exclusion, and deadlock avoidance capability are expressed as membership problems of the language generated by the L grammer.

Analyzing L system models for algorithms is relatively easy because the membership problem of a context sensitive grammer can be automated. However, the translation from evaluation nets to L system grammers lacks a well-defined methodology, and the context sensitivity of the L grammars makes the rule tables extremely complex if the message switching is of arbitrary configuration.

The expression of high level synchronization policies is also a problem with L system models. In Ellis' example, the analysis is done by testing the membership of a string of local variables and local scheduler's state variables. The algorithm is shown "incorrect" by showing that a string of variables with two different nodes simultaneously having the same lock and the update is a member of the language. However, an algorithm is not necessarily correct if the above strings are absent because the serializability is not simply the state of local variables or the local program counters. To show an algorithm correct with L system models, we need to prove that members of the language are produced in correct sequences of derivation steps.

C. Path Expressions

Path expressions [7], [17], [23] follow the event ordering notion and specify the behavior of concurrent program operations. A path expression is a formal definition of some ordering constraint on the way in which occurrences of events in a system are to relate. The set of all path expressions for a system is an abstract description of the system and can be viewed as a set of grammar rules for generating strings representing system histories. From these path expressions, characteristics such as deadlock avoidance capability can be analyzed by expressing them in path expressions, then verifying whether the description of the algorithm implies such characteristics or not.

For example, the synchronization aspect for a version of the producer/consumer problem can be described in path expressions [7] in forms like

path {read}, {write} **end**

path write **end**.

The first path means that if the read procedure begins to execute, all reading requests are accepted. Same is the case with writes. A read and a write may not overlap, but a read can overlap with other reads. The second path ensures that the actions of writing are mutually exclusive. Thus, executions of write are synchronized with respect to read in the first path expression, the synchronization among writes is expressed in the second path.

Path expressions were originally proposed as synchronization tools rather than as verification models. Sometimes the description can be too abstract to leave any indication for implementation. The paths describe the desired ordering of operations but give little indication for implementation as we can see from the example. Although procedures for translating path expressions into P's and V's are given in [7], no translations to message switching algorithms are known.

Path expressions can be used as the verification tool by viewing them as conditions that an algorithm has to satisfy. In other words, path expressions can be viewed as behavior descriptions of the operations in an algorithm. The problem of proving that a set of path expressions for describing a system really represents the system, is difficult [23]. Contrary to the causal graph model, the originally proposed path expressions do not contain semantic as well as history information about the system's behavior. This makes the process of deducing/transforming path expressions from/to an algorithm difficult. An attempt of adding predicates to path expressions [1] is similar to our construct of causal rule predicates, but its emphasis is still on using path expressions as programming specification for synchronization. Andler presented an implementation of predicate path expressions through shared monitors for abstract object types; the applicability of such construct in distributed environment needs further research work.

Appendix A
Basic Terminology

A *distributed database management system* (DDBMS) is a database system distributed among a set of *nodes* N connected by telecommunication links. Each node has its own independent computing resources.

The database is modeled by a set of *logical database entities* which may have one or more *physical copies of data value*. The database entities are accessed by unique names; how this naming is maintained is insignificant to this paper. The database may be either completely or partially replicated, or it may be partitioned on different nodes.

A distributed database is *consistent* if it satisfies some predefined assertions about the intrinsic characteristics of the data values. For a replicated distributed database, it is necessary for the physical copies of the same database entity on different nodes to remain identical.

The user actions on a distributed database consist of a sequence of atomic operations. An *atomic operation* is represented by $\sigma_i = A_i^j[x]$ where i is a unique identification for a transaction, j is a unique identification for a node, A is either R or W representing read or write operation, and x is one or more logical database entities. As far as the DDBMS is concerned, these read/write operations constitute indivisible (or atomic) operations to the database [11]. The atomic operations are grouped into logical units called *transactions* that will preserve the database consistency if executed alone. A transaction can be viewed as a quantum change for the database from one consistent state to another; however, the consistency assertions may be temporarily violated during the execution of a transaction but must be satisfied when there are no incomplete transactions or the system is quiescent. The purpose of the concurrency control is to guarantee that the concurrent execution of a set of transactions does not result in an inconsistent database state.

The *transaction set* T represents all user transactions, and the *atomic operation set* Σ contains all the atomic operations specified by the transaction set. A transaction has to read only one copy of a replicated data entity but has to update all copies.

Two atomic operations σ_i, σ_j *conflict* if 1) they belong to different transactions, 2) both access the same database entity at the same node, and 3) at least one of them is a write operation. In particular, conflicting atomic operations σ_i and σ_j have 1) *WR-conflict* if σ_i is a write operation and σ_j is a read operation, 2) *RW-conflict* if σ_i is a read operation and σ_j is a write operation, and 3) *WW-conflict* if both σ_i and σ_j are write operations.

There are two special atomic operations in a transaction that are important. The *last new atomic operation* ω_i of transaction i is its last atomic operation such that the access is to a new database entity or the access is at a higher level[3] than before for a previously accessed entity. Every atomic operation after ω_i either accesses some used entity or repeats a lower level access. The *earliest new atomic operation* α_i for a transaction i is the first atomic operation which starts accessing new entities. Since each atomic operation accesses some database entities, α_i is simply the first atomic operation in a transaction.

For example, ω_i of the following transaction

$$R_i^1[x]\, W_i^2[y]\, W_i^2[z]\, W_i^3[y]\, W_i^3[z]$$

is $W_i^2[z]$ since z is the last new entity being accessed. The ω_i of the following transaction

$$R_i^1[x]\, W_i^2[y]\, W_i^2[z]\, W_i^3[y]\, W_i^3[z]\, W_i^1[x] R_i^2[z] R_i^1[x]$$

is $W_i^1[x]$ since it is the latest higher level access to any entity (x in this case).

The concurrent activities of a distributed database system can be modeled as a sequence of all atomic operations. This sequence is called the *history* of the system, and is represented by a quadruple $h = \langle D, T, \Sigma, \pi \rangle$ where D is a distributed database, T is the transaction set, Σ is the atomic operation set, and π is a *permutation function* which gives the permutation indices for atomic operation σ in $h (\sigma \in \Sigma)$. For example, if a history h is the following sequence

$$\alpha \beta \gamma \cdots \omega$$

then $\pi(\alpha) = 1$, $\pi(\beta) = 2, \cdots, \pi(\omega) = |\Sigma|$. A *serial history* is one in which each transaction runs to completion before the next one starts. In other words, in a serial history the atomic operations of different transactions are not interleaved.

Although the system's activities can be modeled as a string of atomic operations, the activity at one node is potentially independent of those at other nodes. Each node records its own history. To capture this notion of local activities, *the node projection* $h^j = \langle h, \Sigma^j, \pi^j \rangle$ of a history h is defined as the subsequence of h containing only those operations pertaining to node j where $\Sigma^j = \{\sigma \mid \sigma \in \Sigma$ and σ is performed at node $j\}$ is a subset of Σ, and π^j is the permutation function for h^j, i.e., $\pi^j(\sigma_1) < \pi^j(\sigma_2)$ iff $\pi(\sigma_1) < \pi(\sigma_2)$ for $\sigma_1, \sigma_2 \in \Sigma^j$. The order of the atomic operations in h is retained in π^j.

The activities of a transaction in a distributed database system can be modeled by a sequence of operations on the database related to this transaction. This sequence is called *the transaction projection* $h_i = \langle h, \Sigma_i, \pi_i \rangle$ where $\Sigma_i = \{\sigma \mid \sigma \in \Sigma$ and σ belongs to transaction $i\}$ is a subset of Σ, and π_i is the permutation function for h_i, i.e., $\pi_i(\sigma_1) < \pi_i(\sigma_2)$ iff $\pi(\sigma_1) < \pi(\sigma_2)$ for every $\sigma_1, \sigma_2 \in \Sigma_i$.

[3]For the transaction model used here, the read operation is considered lower level access when compared to the write operation.

From the above definitions, it is clear that a serial history h has the form

$$h_{i_1} h_{i_2} h_{i_3} \cdots h_{i_t} \text{ for } i_k \in T,\; k = 1, \cdots, t$$

where $t = |T|$ and $i_1 i_2 i_3 \cdots i_t$ is a permutation of transaction id's (h_{i_1}, say, is the transaction projection for transaction i_1 from the serial history h). Note that each node projection of a serial history is essentially a sequential execution of transactions following the same permutation order $i_1 i_2 \cdots i_t$ of the serial history.

Each operation in a history transforms one database state into another one. Two histories are *equivalent* or indistinguishable if they transform a given initial state to the same final database state. The notation $\equiv$ denotes the equivalence relation between histories. A history h is *serializable* iff there exists a serial history g such that $h^j \equiv g^j$ for every node j.

If every transaction when executed alone preserves the database consistency, then each node projection of a serializable history will also preserve the consistency. Since a serial history produces node projections with the same serial transaction order, a serializable history necessarily generates a consistent database. An algorithm is considered correct if all its allowed histories are serializable.

The use of serializability as a correctness criterion is popular among researchers [2], [9], [19], [21]. Although nonserializable histories can be consistent when semantic information is available [15], [24], we still consider serializability to be the correctness criterion. It has been shown in [14] that concurrency control algorithms with only syntactic information can at best produce serializable histories.

References

[1] S. Andler, "Predicate path expressions: A high-level synchronization mechanism," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-79-134, Aug. 1979.

[2] P. A. Bernstein, D. W. Shipman, and W. S. Wong, "Formal aspects of serializability in database concurrency control," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 203–216, May 1979.

[3] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surveys*, vol. 13, pp. 185–222, June 1981.

[4] B. Bhargava and C. T. Hua, "The causal graph model for distributed database synchronization algorithms," in *IEEE Proc. 4th Int. Comput. Software Applications Conf. (COMPSAC)*, Chicago, IL, Oct. 1980, pp. 444–452.

[5] B. Bhargava, "Concurrency control and reliability in distributed database system," in *Software Engineering Handbook*. Princeton, NJ: Van Nostrand Reinhold, 1983.

[6] C. Hua and B. Bhargava, "Classes of serializable histories and synchronization algorithms in distributed database systems," in *Proc. IEEE Int. Conf. Distributed Comput. Syst.*, Miami, FL, Oct. 18–22, 1982.

[7] R. H. Campbell and A. N. Habermann, "The specification of process synchronization by path expressions," in *Lecture Notes in Computer Science*, vol. 16, 1974, pp. 89–102.

[8] C. A. Ellis, "Consistency and correctness of duplicate database systems," in *Proc. 6th ACM Symp. Operating Syst.*, Nov. 1977.

[9] K. P. Eswaran, J. N. Gray, R. Lorie, and A. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 624–633, Nov. 1976.

[10] H. J. Genrich and K. Lautenbach, "The analysis of distributed

systems by means of predicate/transition nets," in *Proc. Int. Symp. Semantics Concurrent Computation (Lecture Notes in Computer Science 70)*. New York: Springer-Verlag, 1979, pp. 123–146.

[11] J. N. Gray, "A transaction model," IBM Res. Div., San Jose Laboratory, CA, IBM Res. Rep. RJ2895, Feb. 1980.

[12] C. T. Hua and B. Bhargava, "Analysis of the majority consensus algorithm for correctness and performance using the event ordering approach," in *IEEE Proc. Int. Conf. Inform. and Commun.*, San Diego, CA, Apr. 1983.

[13] C. T. Hua, "Verification of concurrency control algorithms for distributed database systems," Ph.D. dissertation, Dep. Comput. Sci., Univ. Pittsburgh, PA, Sept. 1981.

[14] H. T. Kung and C. H. Papadimitriou, "An optimality theory of concurrency control for databases," in *Proc. ACM SIGMOD 1979*, May 1979, pp. 116–126.

[15] L. Lamport, "Towards a theory of correctness for multi-user data base systems," Massachusetts Computer Associates, Inc., Tech. Rep. CA-7610-0712, Oct. 1976.

[16] L. Lamport, "Time clocks, and the ordering of events in a distributed system," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 558–564, July 1978.

[17] P. E. Lauer and R. H. Campbell, "Formal semantics for a class of high level primitives for coordinating concurrent processes," *Acta Informatica*, vol. 5, pp. 297–332, 1975.

[18] D. A. Menasce, G. Popek, and R. Muntz, "A locking protocol for resource coordination in distributed databases," *Trans. Database Syst.*, vol. 5, pp. 103–138, June 1980.

[19] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. Ass. Comput. Mach.*, vol. 26, pp. 631–653, Oct. 1979.

[20] J. L. Peterson, "Petri nets," *Comput. Surveys*, vol. 9, pp. 223–252, ACM, Sept. 1977.

[21] D. L. Rosenkrantz, R. E. Stearns, and P. M. Lewis II, "System level concurrency control for distributed database systems," *ACM Trans. Database Syst.*, vol. 3, pp. 178–198, June 1978.

[22] G. Rozenberg, *L Systems*. New York: Springer-Verlag, 1974.

[23] M. W. Shields, "Adequate path expressions," in *Proc. Int. Symp. Semantics Concurrent Computation (Lecture Notes in Computer Science 70)*. New York: Springer-Verlag, 1979, pp. 249–265.

[24] A. Silberschatz and Z. Kedem, "Consistency in hierarchical database systems," *J. Ass. Comput. Mach.*, vol. 27, pp. 72–80, Jan. 1980.

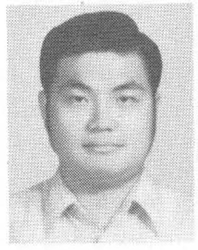
[25] R. H. Thomas, "A majority consensus approach to concurrency control," *ACM Trans. Database Syst.*, vol. 4, pp. 180–209, June 1979.

[26] K. Voss, "Using predicate/transition nets to model and analyze distributed database systems," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 539–544, Nov. 1980.

Bharat Bhargava (M'78) received the Ph.D. degree from Purdue University, West Lafayette, IN, in 1974.

He worked on the design and implementation of an ambulatory care information system at Indiana University Medical School from 1973-1976. He currently teaches in the Department of Computer Science at University of Pittsburgh, PA. His current research interest is in the area of concurrency control and reliability in distributed database management systems. He is working towards the design of robust software structures for an enroute air traffic control system under a research contract with Federal Aviation Agency.

Dr. Bhargava is a member of CODASYL Systems Committee. He serves on the Education Activities Board and the Publication Boards of the IEEE Computer Society. He was the Chairperson of the 1981 and 1982 IEEE Computer Society's Symposium on Reliability in Distributed Software and Database Systems. He is a member of the Program Committee of the 1982 International Conference on Distributed Computing Systems. He is Vice-Chairman of the IEEE Computer Society's technical committee on Distributed Computing.



Cecil T. Hua (S'80–M'81) received the B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1975, and the M.S. and Ph.D. degrees in computer science from the University of Pittsburgh, Pittsburg, PA, in 1977 and 1981, respectively.

He is presently with the Large Computer Products Division, Honeywell Information System, Inc., Phoenix, AZ, where he is responsible for the architectural design and the development of cooperated transaction processing in a distributed system. His areas of interest include distributed operating/database systems, office automation systems, and local area networks.

Dr. Hua is a member of the Association for Computing Machinery.