A STATE MODEL FOR THE CONCURRENCY CONTROL PROBLEM IN DATABASE MANAGEMENT SYSTEMS

by

S. Lafortune and E. Wong

Memorandum No. UCB/ERL M85/27

18 April 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A STATE MODEL FOR THE CONCURRENCY CONTROL PROBLEM IN DATABASE MANAGEMENT SYSTEMS*

Stéphane Lafortune

Eugene Wong

Department of Electrical Engineering and Computer Sciences

and Electronics Research Laboratory

University of California

Berkeley, CA 94720.

ABSTRACT

The purpose of this paper is to propose a discrete event dynamical system model for concurrent actions by multiple users on one database. This model shows that concurrency control, which is the task of scheduling these users, is a case of control with partial state information. It is used to analyze and compare existing concurrency control techniques and to suggest new ones. In particular, the *Two-Phase Locking* and the *Declare-Before-Unlock* protocols are studied in detail.

April 17, 1985

* Research sponsored by the U.S. Army Research Office contract DAAG29-82-K-0091, the National Science Foundation grant ECS-8300463, and in the case of the first author, a scholarship from the Natural Sciences and Engineering Research Council of Canada.

1. INTRODUCTION

This paper considers the concurrency control problem in database management systems as a problem of controlling a dynamical system. Its purpose is to propose a state space model for the dynamical system consisting of concurrent actions by multiple users on one database, and to study how to control this system. The concurrency control problem consists in the task of scheduling these users, the action of each user on the database being described by a transaction. It is widely accepted that the appropriate criterion for concurrency control is serializability. Roughly speaking, this means that the same effect is produced on the database as if the transactions had executed in some serial order. Serializability is the control objective we adopt.

Despite the extensive work done thus far on concurrency control (see, e.g., [1-8, 11]), this problem had never been formulated in the framework of dynamical systems. This was the motivation for our work and in this paper we formulate a model that is used to characterize what can and what cannot be achieved by existing concurrency control techniques, and to suggest ways of improving these techniques. In particular, the well-known *Two-Phase Locking Protocol* introduced in [1], and the recently proposed *Declare-Before-Unlock Protocol* [8] are analyzed in detail. The model shows how the two features: *declare* and *must-precede graph* of [8] are useful in improving the performance of the widely used two-phase locking protocol.

In our framework, concurrency control is a problem of supervisory control for a discrete event dynamical system. In this sense, our model fits into the framework of [10], although, as we emphasize in section 3 and appendix A, the main issue in our control problem is one of partial state information rather than one of controllability.

This paper is organized as follows. In section 2, we introduce some terminology and make precise the notion of serializability. The model we propose is described in section 3. Section 4 gives some background we need about control by locking, probably the most popular concurrency control technique. In section 5, we compare various graphs that are

estimates of the state of the system and are used with control by locking. Their properties are characterized in terms of our model. Sections 6 and 7 are devoted to a detailed analysis of control by locking. With the help of our model, we study the properties and limitations of some important locking protocols. Finally, section 8 contains a discussion of our results, and in the appendices, we comment on the interpretation of the model in terms of supervisory control, present a new protocol, and discuss the issue of decentralized control when the database is distributed.

2. PRELIMINARIES

2.1 - Definitions and terminology

We begin with a simple representation of the concurrent actions by multiple users on one database. Suppose that $a, b, c, \ldots$ denote atomic units of data that we call *objects*. A *transaction* is a description of the actions of one user on the database. It consists of a finite sequence of actions, each action (say, read or write) touching a single object. The transactions are assumed to map a consistent state of the database to a new consistent one, i.e. they are all individually correct. We denote a transaction T_i by $T_i = \tau_i(o_1)\tau_i(o_2)\cdots\tau_i(o_{n_i})$ where $\tau_i(o_j)$ means that the j-th action of T_i is on object o_j. We will also use the notation (i, j, o_j) when we want j to appear explicitly (see section 3). For now, we do not distinguish between actions of different types, e.g., reading and writing. Nor do we assume that the objects for the same transaction are distinct. Extension to read and write actions is discussed in section 8.

The fact that many transactions are being executed simultaneously results in an interleaving of their actions, this interleaving respecting the ordering within each of them. A *complete execution*, denoted E^c, of a (finite) set of transactions is an interleaved sequence of all the actions from the transactions. An *execution*, denoted E, is a prefix[1] of a complete execution. Hence, the set of all executions contains the set of all complete

[1] Prefix means that 0,1 or more actions are removed, starting from the end (right, in our notation).

executions. An execution is said to be *serial* if there is no interleaving between the actions from the transactions.

Two executions E_1 and E_2 (of the same transactions) are said to be *equivalent* if for every object o, the subsequence of E_1 touching o is the same as the corresponding subsequence from E_2 [1]. An execution is said to be *serializable* [1-7] if it is equivalent to some serial execution.

Example 2.1 : Let T_1 and T_2 be two transactions:

$$T_1 = \tau_1(a)\tau_1(b). \quad T_2 = \tau_2(b)\tau_2(c).$$

Then examples of an execution and a complete execution are:

$$E_2 = \tau_1(a)\tau_2(b)$$
$$E^c_{1-2} = \tau_1(a)\tau_2(b)\tau_2(c)\tau_1(b).$$

Here, E_2 is serial and E^c_{1-2} is serializable because it is equivalent to the serial execution T_2T_1. To see this, observe that the subsequences touching the individual objects in E^c_{1-2} are

$$a : \tau_1(a) \quad b : \tau_2(b)\tau_1(b) \quad c : \tau_2(c).$$

and so are identical to those for T_2T_1. □

2.2 - Serializability

We now wish to give an equivalent characterization of the notion of serializability which we will use in the sequel. Consider an execution E. When there exist some object b and actions (i, k, o_k) and (j, m, o_m) in E with $o_k = o_m = b$, the tuple $((i, k), (j, m), b)$ is called a *conflicting pair* due to object b between the k-th action of T_i and the m-th action of T_j. In this case, we say that transaction T_i *precedes* T_j in E if (i, k, o_k) comes before (j, m, o_m) in E. It is well known that an execution E is serializable if and only if *precedes in E* is a partial ordering, i.e. if and only if all conflicting pairs in E are consistently ordered [1]. (By consistent we mean that the partial order is transitive and asymmetric.)

We can depict the situation by a *precedence graph* PG(E) as follows. Let the nodes of PG(E) represent the transactions. For a given execution E add a directed arc from T_i to T_j in PG(E) if there is some object b such that, in E, T_i acts on b immediately before T_j does so. Then the execution E is serializable if and only if PG(E) is cycle free. For future reference, we restate this result in the form of a theorem.

Theorem 2.1 [1] : An execution is serializable iff[2] its precedence graph has no cycles. □

Example 2.2 : The two transactions of Example 2.1 have only one conflicting pair: $((T_1, 2), (T_2, 1), b)$. Hence, any execution of them is serializable since the PG of it is either

$$T_1 \rightarrow T_2 \quad or \quad T_2 \rightarrow T_1$$

the first one corresponding to all executions equivalent to the serial execution T_1T_2, the second one to those equivalent to T_2T_1. □

The dynamics of this problem correspond to the generation of an execution from the actions of all concurrent transactions. The concurrency control problem consists in ensuring that any complete execution of a set of transactions is always serializable. In particular, incomplete executions have to be serializable. Non-serializable executions are not acceptable because they result in possible violations of the consistency of the database; the lost-update problem and the inconsistent retrieval problem [2] are examples of such undesirable situations. Observe that in general the set of objects touched by each transaction is not known beforehand.

This notion of serializability is often referred to as conflict-serializability (CSR) in the literature. When one distinguishes between read and write actions, the definition of CSR is in similar terms, but this time assuming that read - read pairs are non-conflicting (i.e. at least one of the two actions in a conflicting pair must be a write). In this case however, more general definitions of serializability are possible if one is only interested in com-

[2] If and only if.

paring the initial and final states of the database. Examples are state-serializability (SSR) [6] and view-serializability (VSR) [7], to use the terminology of [7]. The case of SSR was studied in detail in [6] where it is shown that, in contrast to CSR where only a graph has to be tested for acyclicity, testing whether an execution is SSR is an NP-complete problem. The same is true for VSR [7]. On the other hand, as argued in [7], it is not clear if the extra concurrency allowed by these more general definitions is really desirable in practice. In this paper, we consider only CSR. In any case, CSR, VSR and SSR are all equivalent when one does not distinguish between read and write actions.

3. STATE MODEL FOR CONCURRENCY CONTROL

3.1 - Model formulation

In this section, we describe the model we propose for concurrency control. Consider the dynamical system where N transactions are being executed concurrently. From an execution, one can tell whether or not this execution is serializable by looking at its current PG (from Theorem 2.1). But one cannot tell whether or not this execution can be completed to a serializable one unless one knowns for each transaction the objects that remain to be acted on. We want to define a state for this system that will contain all required information about current and eventual (i.e. for any completion of the execution) serializability. Therefore, the state needs to have more information than what is contained in the PG of section 2.2. This justifies the following construction for the state space model of this system.

Let Σ be the set of all actions from the N transactions T_1 to T_N. The elements of Σ are inputs to the system. They have the form of a triple $\sigma = (i, k, o_k)$ meaning that the k-th action of transaction T_i is on object o_k. Let CP denote the set of *all* conflicting pairs among the elements of Σ. We now define four sets of finite strings w of elements of Σ. No repetitions of any element of Σ are allowed in a string w.

$$\Sigma_c := \{\, w : w \textit{ is an execution} \,\}^3$$

Σ_{se} := { w : w *is a serializable execution* }

$\Sigma_{\overline{cse}}$:= { w : w *is a prefix of a complete serializable execution* }

Σ_{cse} := { w : w *is a complete serializable execution* }.

(By execution it is understood an execution of T_1 to T_N, as defined in section 2.1.) Clearly, these sets are strictly nested in general. The first three contain the empty string denoted $w = 1$.

The state of the system, denoted q, is represented as a graph composed of N nodes and of three types of (labeled) arcs: (i) dashed, not directed; (ii) dashed, directed; (iii) solid, directed. A state q is also called a *state graph*, abbreviated SG. It is constructed from an initial state and a state transition function that we now define.

Let the initial state, denoted q_0, be a graph with N nodes representing the N transactions and one undirected dashed arc for each conflicting pair in CP. Recall that the elements of CP are of the form $((i, k), (j, m), b)$. The arc is drawn between nodes i and j and has for label this whole tuple.

Given an input action $\sigma \in \Sigma$ and a state q, the state transition function, denoted $\phi_e(\sigma, q)$, is as follows. In response to $\sigma = (i, k, o_k)$, identify all arcs attached to node i whose labels contain σ (i.e. all conflicting pairs in which σ is involved). If there are no such arcs, set $\phi_e(\sigma, q) = q$. Otherwise, determine which of the following situations prevail for each of these arcs and do as indicated:

(i) the arc is dashed and not directed (meaning the other action in this conflicting pair has not occurred yet); in this case, direct the arc out of node i;

(ii) the arc is dashed and directed into node i (meaning the other action has occurred); in this case, replace the dashed arc by a solid one in the same direction;

(iii) in all other cases, ϕ_e is undefined.

The state space, denoted Q_e, is defined to be the set of all such state graphs (SG) which are generated by all possible executions of the N transactions, starting from the ini-

[3] := stands for "defined equal to."

tial state q_0. More precisely, define the transition function Φ on Σ_e in the following recursive way:

(i) $\Phi(1) := q_0$

(ii) for any $w' \in \Sigma_e$ and $\sigma \in \Sigma$ such that $w := w'\sigma \in \Sigma_e$,

$$\Phi(w) := \phi_e(\sigma, \Phi(w')).$$

(Observe that the assumptions in (ii) guarantee that ϕ_e in the above equation is always defined.)

Then the state space is just the range of Φ:

$$Q_e = \{q : \textit{there exists } w \in \Sigma_e \textit{ such that } q = \Phi(w)\},$$

and we similarly define Q_{se}, $Q_{\overline{cse}}$ and Q_{cse} by replacing Σ_e in the above definition by Σ_{se}, $\Sigma_{\overline{cse}}$ and Σ_{cse}, respectively. Q_e contains all state graphs that will ever be reached by the N transactions.

Remark 3.1 : When we will talk about nesting of state subsets, the same will be true for their corresponding inverse images under Φ^{-1}, which are sets of executions. □

The state transition function is hence the partial function $\phi_e : \Sigma \times Q_e \rightarrow Q_e$ corresponding to the above description. It is a partial function because $\phi_e(\sigma, q)$ is defined only when there exists $w \in \Phi^{-1}(q)$ such that $w\sigma \in \Sigma_e$. It is now clear that given an input action from Σ, the current SG and ϕ_e completely and uniquely determine the new SG.

Non-conflicting actions are not considered in this model because they have no effect on serializability and achievable concurrency. Without loss of generality, we can ignore them since they do not induce a transition of the state.

Observe that the mapping, under Φ, of the various sets of executions into the corresponding sets of states, is not injective. Recall the definition of equivalence in section 2.1. Then the following result is clear.

Lemma 3.1 : Two executions have the same state graph iff they are equivalent.

Proof: First observe that two equivalent executions are composed of the same actions. So in their state graphs the sets of directed dashed arcs and of solid arcs will be respectively identical. If the actions on each object appear in the same order in these executions, then the arcs in their SG's will have identical directions. Conversely, identical state graphs imply same sets of actions, and identical directions imply that conflicting actions occur in the same order and so the subsequences touching each object are the same. □

Therefore, as far as serializability is concerned, there is no loss of generality in assuming, de facto, a one-to-one relation between an execution and its corresponding state graph, since this graph contains all necessary information for this purpose. However, when analyzing a specific execution in terms of some concurrency control method (as we do in sections 6 and 7), one has to consider the whole state trajectory of this execution. For an execution $E = e_1 \cdots e_n$ where the e_i's are individual actions, this trajectory is defined to be $Traj(E) := \{q_0, \cdots, q_n\}$ where $q_{i+1} = \phi_e(e_{i+1}, q_i)$.

Lemma 3.2 : $Traj(E)$ completely and uniquely determines E.

Proof: From the definition of ϕ_e it is clear that given q_i and q_{i+1}, both in Q_e, there is a unique $\sigma \in \Sigma$ such that $\phi_e(\sigma, q_i) = q_{i+1}$. (Look at the labels on the arcs that became directed or solid to identify the appropriate action σ.) Therefore, the entire trajectory determines a unique execution which has to be E (otherwise it would not be the trajectory of E). □

Our state space model can be viewed in the terminology of automata theory [9] as a deterministic automaton or generator [10]:

$$G = (Q_e, \Sigma, \phi_e, q_0, Q_m),$$

where Q_m can be taken as Q_{cse}, the set of acceptable final states which correspond to the complete serializable executions. (Observe that in our model the states $q \in Q_e$ are themselves graphs. Also, our generator G is accessible by construction.) The sets Σ_e, Σ_{sc}, $\Sigma_{\overline{cse}}$ and Σ_{cse} are *languages*, and in particular Σ_e is the language generated by the uncontrolled generator G. We comment further on this relation with [10] in appendix A.

Example 3.1 : Consider the two transactions

$$T_1 = \tau_1(a)\tau_1(b) \qquad T_3 = \tau_3(b)\tau_3(a)$$

and the complete execution

$$E^c_{1-3} = \tau_1(a)\tau_3(b)\tau_3(a)\tau_1(b).$$

E^c_{1-3} has two conflicting pairs: $((T_1, 1), (T_3, 2), a)$ and $((T_1, 2), (T_3, 1), b)$. $Traj(E^c_{1-3})$ is drawn in Figure 3.1. □

3.2 - Characterization of the state space

The nesting of the various subsets of the state space defined above is represented in Figure 3.2. For convenience, we denote $Q_{e-se} = Q_e - Q_{se}$ and $Q_{se-\overline{cse}} := Q_{se} - Q_{\overline{cse}}$.

We now make the following observations.

1- The SG represents the current status of the ordering of *all* the conflicting pairs in CP, whereas the solid arcs represent the ordering of those pairs for which the two actions have occurred, corresponding to the PG of section 2.2 (up to transitive closure of this one because it only draws arcs from the last user of an object to the current one; see Lemma 5.4).

2- The method of construction of the SG shows that a "dashed" or a "mixed" cycle will always result in a "solid" cycle, no matter how the execution is completed. (A solid cycle is only composed of solid arcs, a mixed cycle has at least one dashed arc and one solid arc, and a dashed cycle has only dashed arcs.)

3- The partial ordering of the conflicting pairs can remain consistent if and only if the current SG has no cycles, although the violation of serializability only occurs when there is a solid cycle. This is because the violation effectively happens only when all the actions involved in this cycle have occured. Hence, mixed and dashed cycles anticipate an unavoidable violation of serializability, even tough the execution can be serializable up to now (if there are no solid cycles).

These observations lead to the following results.

Lemma 3.3 : The state space Q_e has the following characterization:

(i) $q \in Q_{\overline{cse}}$ iff q is cycle free;

(ii) $q \in Q_{se-\overline{cse}}$ iff q has one or more cycles, each of these cycles containing at least one dashed arc, equivalently, iff q has at least one dashed or mixed cycle but no solid cycles;

(iii) $q \in Q_{e-se}$ iff q has one or more solid cycles.

Proof: (i) Complete serializable executions and their prefixes correspond to cycle free states.

(ii) Those executions which are serializable but which cannot be completed to serializable ones correspond to the states in $Q_{se-\overline{cse}}$ and so these states must have cycles, although none of them can be solid.

(iii) Finally, non-serializable executions correspond to states with solid cycles as observed above. □

Corollary 3.4 :

(i) The state cannot jump from $Q_{\overline{cse}}$ directly into Q_{e-se}.

(ii) A state $q \in Q_{se-\overline{cse}}$ can never return to $Q_{\overline{cse}}$ and will inevitably go to Q_{e-se} upon completion of the execution.

Proof: Follows from the above observations. □

Example 3.2 : In Figure 3.1, q_0 and q_1 are in $Q_{\overline{cse}}$, but the presence of dashed and mixed cycles in q_2 and q_3, respectively, indicates that these states are in $Q_{se-\overline{cse}}$, i.e. that although the corresponding (incomplete) executions are serializable, they cannot be completed to a serializable one. In fact, $q_4 \in Q_{e-se}$ shows that E^c_{1-3} of Example 3.1 is not serializable. □

The situation when the state jumps out of $Q_{\overline{cse}}$ will be called *deadlock*. This term means that in order to obtain a complete serializable execution, it is necessary to back-up (or undo) the current execution (partially, i.e. until the state returns to $Q_{\overline{cse}}$). This is in analogy with the usual meaning of deadlock in control by locking [2]. We are concerned

with deadlock detection only; we will not consider the problem of deadlock resolution. Each time it will be detected, we will assume some deadlock resolution procedure is invoked after which the execution is resumed.

3.3 - The concurrency control problem

We have in mind the situation where a controller acts on the system described by G by accepting or rejecting inputs. Its decisions are based on a division of the state space into a "legal" region and an "illegal" one. All transitions inside the legal region are accepted, whereas transitions from the legal part to the illegal one are always disabled (i.e. the input is rejected). Given a controller, executions which are accepted by it are termed *achievable* and their corresponding states (through Φ) are said to be *reachable*.

The control objective for our dynamical system is to obtain only complete serializable executions. To have more concurrency means to allow for more elements of Σ_{cse}, maximum concurrency meaning to allow for all of Σ_{cse}. We stress the fact that all complete serializable executions are assumed to be good per se (i.e. none are less desirable). Therefore, wanting to allow for all of them is a reasonable objective because more concurrency implies less waiting on the part of the transactions.

Our model is not only useful to characterize the status of an execution concerning serializability (by using Lemma 3.3), but it also provides for a measure of concurrency via the state space and its subsets as we now show.

Lemma 3.5 : $E \in \Sigma_{\overline{cse}}$ iff $Traj(E) \subset Q_{\overline{cse}}$.

Proof: (only if) If $E \in \Sigma_{\overline{cse}}$, all its prefixes are also in $\Sigma_{\overline{cse}}$. Hence, the SG's of these prefixes are in $Q_{\overline{cse}}$, and since they constitute $Traj(E)$, this trajectory remains inside $Q_{\overline{cse}}$. (if) Consider the states q_i in $Traj(E)$, $i = 1, \cdots n$. By definition of $Q_{\overline{cse}}$, the inverse image $\Phi^{-1}(q_i)$ is some set $S(i) \subset \Sigma_{\overline{cse}}$. But on the other hand, $Traj(E)$ uniquely determines E (Lemma 3.2). This means E is the only execution such that its i-th prefix is in $S(i)$ for all $i = 1, \cdots n$. In particular, $E \in S(n)$. Hence, $E \in \Sigma_{\overline{cse}}$. □

Corollary 3.6 : $E^c \in \Sigma_{cse}$ iff $Traj(E^c) \subset Q_{\overline{cse}}$.

Proof: Follows from Lemma 3.5 and the fact that the only complete executions in $\Sigma_{\overline{cse}}$ are those in Σ_{cse}, by definition of these two sets. □

Lemma 3.7 : Given a controller for G, an element of Σ_{cse} is not achievable iff one or more states in $Q_{\overline{cse}}$ are not reachable.

Proof: (if) Suppose the state q is not reachable by the controller. Clearly, this controller does not achieve any of the trajectories that go through q, and by definition of $Q_{\overline{cse}}$, there has to be at least one such trajectory (any $q \in Q_{\overline{cse}}$ is the image under Φ of some prefix of an element of Σ_{cse}).

(only if) By way of contradiction, suppose that all states in $Q_{\overline{cse}}$ are reachable but that some element E^c of Σ_{cse} is not achievable. Recall that the controller acts by disabling transitions out of a "legal" subset of the state space. Hence, all reachable states are necessarily "legal." But all states in $Traj(E^c)$ are reachable. Therefore it is not possible that each state in a trajectory be individually reachable (hence legal) but that the execution be not achievable (because the controller always accepts transitions inside the legal region). We have a contradiction. □

The above result justifies the fact that the portion of $Q_{\overline{cse}}$ that a controller can reach is an aprropriate measure of the concurrency it can achieve. Of course, the "ideal" legal region is $Q_{\overline{cse}}$. A controller that reaches $Q_{\overline{cse}}$ exactly solves optimally (in terms of serializability requirements and achievable concurrency) the concurrency control problem. We say "ideally" because, for this to be possible, the controller must know q_0 beforehand, i.e. it must have complete state information. This is because this objective requires that all state transitions out of $Q_{\overline{cse}}$ be detected. Lack of knowledge of some (dashed) arcs can result in cycles in the SG that are not detected by the controller.

In general, q_0 is unknown to the controller since it contains information about future actions. By looking at a current execution, the controller can only identify the elements of

CP for which the two actions have already occured, so that the best it can do is to determine if $q \in Q_{se}$ or if $q \in Q_{e-se}$ (by using Theorem 2.1). To determine membership inside Q_{se} requires more information. Therefore, the concurrency control problem is one of control with *partial state information*.

However, a necessary requirement is that the state remains within Q_{se} at all times, because an execution has to be serializable, even if incomplete. Hence, given the information available to the controller, our control objectives are:

(i) to bring the set of reachable states by the controlled system as close as possible to $Q_{\overline{cse}}$;

(ii) to detect deadlock occurence as soon as possible, and guarantee detection before the state jumps into Q_{e-se} (recall Corollary 3.4).

Remark 3.2 : As formulated, our model considers the dynamics of the problem for a fixed set of N transactions. In the case where transactions leave upon completion and are replaced by new ones, the state space model will be time-varying. By this we mean that Σ, Q_e and its subsets, and the state q will "jump" to new values each time there is a change among the N transactions. However, the basic properties of the model (nesting of states and executions subsets, characterization of state space, admissible states, etc.) will be unchanged so that the analysis and results in this paper still apply to this more general case. □

4. CONTROL BY LOCKING

Locking provides an effective means for concurrency control when a *locking protocol* is specified (see [1-8]). In this section, we introduce the necessary background about the method of control by locking and see how it fits in our model.

4.1 - Locking actions and constraints

There are two basic locking actions: "lock" and "unlock." A new one, called "declare," was introduced in [8]. These are the only three types we will consider in this paper.

Locking actions are added to a transaction to produce an *augmented transaction*, and any interleaving of augmented transactions is termed an *augmented execution*. We represent locking actions in an augmented transaction T_i by the symbols $d_i(b), l_i(b), u_i(b)$ to denote T_i declares, locks and unlocks object b, respectively.

An augmented transaction T has to satisfy the following locking constraints: (i) every object used by T is declared before it is first locked, (ii) every object is locked before it is used, and (iii) to every lock there is a corresponding unlock before the end of the transaction. Each upgrading voids the previous state of lock.

An augmented execution E is called a *locking execution* if it satisfies the following locking constraints: (i) every transaction in E is augmented correctly, (ii) locks are exclusive, i.e. there is never more than one lock on an object at any given time. Locking executions are the only augmented executions we are interested in.

Observe that a declare must precede a lock, but not necessarily immediately. Upon being promoted to a lock, it becomes void. So when we say "all transactions currently holding a declare on a," we mean all transactions that have declared but not yet locked object a. In contrast to locks, "declares" do not conflict with each other, neither do they conflict with "lock." This means that any number of transactions can simultaneously hold "declare" on the same object, even if it is locked by another transaction.

Remark 4.1 : In this paper, we consider both control with two types of locking actions (when only "lock" and "unlock" are used) and with three types of locking actions (when "declare" is also employed), the former being a special case of the latter where "declare" actions are deleted. □

4.2 - Locking executions

It is not hard to see that every element of Σ_e has a corresponding locking execution. The following augmentation procedure (based on one in [8]) gives a way of obtaining a locking execution from a given execution.

Standard augmentation procedure

Let E be an execution: $E = e_1 e_2 \cdots e_n$ where the e_i's are actions. We begin by augmenting e_1 in E with the addition of locking actions to produce E_1. Then, we augment e_2 in E_1 to produce E_2, and so forth. At the beginning of the k-th step, we have e_k in the form of $\tau_T(a)$ where T is a transaction and a is an object. In E_{k-1}, one of the following situations prevails.

(a) At the time of e_k, T holds the lock on a; in this case, we set $E_k = E_{k-1}$.

(b) At the time of e_k, a is unlocked; in this case, we replace $e_k = \tau_T(a)$ in E_{k-1} by $d_T(a)l_T(a)\tau_T(a)$ to produce E_k.

(c) At the time of e_k, some transaction $S \neq T$ holds the lock on a; in this case, we replace $e_k = \tau_T(a)$ in E_{k-1} by $u_S(a)d_T(a)l_T(a)\tau_T(a)$ to produce E_k.

After step n, we remove all redundant "declares" in E_n (retaining only the first declare on each object for each transaction) to get E_n'. Finally, if E is not a complete execution, we set $E^l = E_n'$, whereas if it is complete, we add all needed unlocks at the end of E_n' to get E^l. E^l is called the *standard locking execution* corresponding to E. □

Example 4.1 : Consider the three transactions:

$$T_1 = \tau_1(a)\tau_1(b) \quad T_4 = \tau_4(b) \quad T_5 = \tau_5(a)\tau_5(a).$$

Here is an example of the construction of the standard locking execution for the following complete (serializable) execution:

$$E^c_{1-4-5} = \tau_1(a)\tau_5(a)\tau_5(a)\tau_4(b)\tau_1(b).$$

At each step, we obtain the following sequences:

$E_1 = d_1(a)l_1(a)\tau_1(a)\tau_5(a)\tau_5(a)\tau_4(b)\tau_1(b)$
$E_2 = d_1(a)l_1(a)\tau_1(a)u_1(a)d_5(a)l_5(a)\tau_5(a)\tau_5(a)\tau_4(b)\tau_1(b)$
$E_3 = E_2$ *(T_5 has the lock on a)*
$E_4 = d_1(a)l_1(a)\tau_1(a)u_1(a)d_5(a)l_5(a)\tau_5(a)\tau_5(a)d_4(b)l_4(b)\tau_4(b)\tau_1(b)$
$E_5 = d_1(a)l_1(a)\tau_1(a)u_1(a)d_5(a)l_5(a)\tau_5(a)\tau_5(a)$
$\quad d_4(b)l_4(b)\tau_4(b)u_4(b)d_1(b)l_1(b)\tau_1(b)$
$E_5' = E_5$ *(because no redundant declares in E_5).*

Finally, adding the missing unlocks (this is only necessary because E^c_{1-4-5} is complete) we get the locking execution:

$$E^{c,l}_{1-4-5} = d_1(a)l_1(a)\tau_1(a)u_1(a)d_5(a)l_5(a)\tau_5(a)\tau_5(a)$$
$$d_4(b)l_4(b)\tau_4(b)u_4(b)l_1(b)\tau_1(b)u_5(a)u_1(b). \quad \square$$

4.3 - Locking protocols

The usefulness of locking actions comes when they are used as a means for concurrency control via a locking protocol, abbreviated LP. A LP for the augmentation of an execution is described by two sets of constraints:

category (1): constraints on the augmented execution itself, and

category (2): constraints on each transaction (as a whole) present in this execution.

Category (1) always contains the condition that the augmented execution be a locking execution, plus some other constraints on the state estimate corresponding to it. In category (2) are regrouped conditions which concern a transaction in its entirety: the constraints may not only apply to the actions from this transaction that are in the (incomplete) execution, but might also involve future actions by it (e.g., the two-phase and declare-before-unlock conditions discussed in sections 6 and 7). Some LP's will be analyzed in detail in sections 6 and 7. For now, we wish to introduce more terminology.

Let LPx denote any locking protocol. An *LPx−execution* is an augmented execution such that itself and the transactions present in it satify the requirements of protocol LPx. An execution is said to be *LPx−augmentable* if there exists an augmentation of that execution that is an LPx-execution. The "language" of protocol LPx is the set of executions that it can achieve:

$$\Sigma_{LPx} := \{E \in \Sigma_e : E \text{ is } LPx\text{−augmentable}\}$$

and the set of states reachable by this protocol is Q_{LPx}, the image under the mapping Φ of its language:

$$Q_{LPx} := \{q : \text{there exists } E \in \Sigma_{LPx} \text{ such that } q = \Phi(E)\}.$$

A state in Q_{LPx} is termed *LPx–reachable*.

Lemma 4.1 : An execution E is LPx-augmentable iff $Traj(E) \subset Q_{LPx}$.

Proof: As in Lemma 3.5. Just replace $\Sigma_{\overline{cse}}$ and $Q_{\overline{cse}}$ in that proof by Σ_{LPx} and Q_{LPx}, respectively. □

5. GRAPH REPRESENTATIONS OF CONCURRENT AUGMENTED TRANSACTIONS

In this section, we define and compare three different directed graphs for the representation of a set of concurrent augmented transactions that is being executed according to some locking protocol. These graphs are in effect state estimates, i.e. sub-graphs of the SG, and they are constructed from the incoming locking actions of a locking execution. (All augmented executions considered in this section are assumed to be locking executions.) The locking protocols we study in sections 6 and 7 use these graphs for control (via constraints of category (1)). (Some results in this section have appeared in [8]. They are repeated here for the sake of completeness.)

5.1 - Definitions of graphs

In the following graphs, to each transaction will correspond a node. Arcs are added according to the specifications given below. They are labeled by the objects that give rise to them.

Note: When an object is not locked, *most recent lock-owner* will mean the last transaction that held a lock on that object, if there is any.

Precedence graph (PG)

This graph was defined in section 2. In terms of locking actions, this graph can be redefined as follows:

when transaction T locks object a, draw arc $S \rightarrow T$ where S is the last lock-owner of a. (In such a case, we say "T reads a from S.")

Must-precede graph (MPG)

For this graph, arcs are added in the following way:

(a) when transaction T declares object a, draw arc $P \rightarrow T$ where P is the most recent lock-owner of a;

(b) when transaction T locks object a, draw arc $T \rightarrow F$ for each F that is currently holding a declare on a.

Concerning the PG, MPG and SG, we have the following terminology. If there is an arc from S to T, then S is called an *immediate predecessor* of T and T an *immediate follower* of S. If there is a path from S to T, then S is called a *predecessor* of T and T a *follower* of S.

Wait-for graph (WFG)

This graph is constructed as follows:

(a) when T requests a lock on a (hence T has already declared a) and a is currently locked by S, draw arc $S \rightarrow T$ ("T is waiting for S");

(b) update the graph at each new lock or unlock.

5.2 - Inclusion results

Our objective is to compare the above graphs and the SG. We will assume for the updating of the SG that an action input occurs simultaneously with the obtention of the corresponding "lock" for this action. (There is no loss of generality in doing so, even if a transaction acts more than once on an object.)

Lemma 5.1 : The MPG is a sub-graph of the SG.

Proof: According to the method of construction of the MPG, an arc $S \rightarrow T$ with label a is added only if one of the two following cases occurs.

(i) T declares a for which S is the most recent lock-owner. This means there exist integers k and m such that $((T, k), (S, m), a) \in CP$ and the action input (S, m, a) has occured before. Hence, there is already a directed dashed arc $S \rightarrow T$ in the SG.

(ii) S locks a and T holds declare on a. Again, there exist integers k and m such that

$((T, k), (S, m), a) \in CP$, but the action input (T, k, a) has not already occured. Hence, the arc $S \rightarrow T$ is added to the MPG at the same time as a corresponding dashed one becomes directed in the SG. □

Corollary 5.2 : A cycle in the MPG is anticipated by or occurs at the same time as a similar dashed or mixed cycle in the SG.

Proof: Follows from the proof of Lemma 5.1. Observe that the cycle cannot be solid in the SG when it occurs in the MPG. □

Lemma 5.3 [8] : The PG is a sub-graph of the MPG.

Proof: There is an arc from S to T in the PG only if there exists some object c such that T reads c from S. Necessarily, the locking execution must look like:

$$l_S(c) \cdots u_S(c) \cdots l_T(c)$$

with no other transaction locking c between S and T. Because of the locking constraints, $d_T(c)$ occurs before $l_T(c)$. If $d_T(c)$ occurs after $l_S(c)$, then the arc $S \rightarrow T$ is added to the MPG at $d_T(c)$; if $d_T(c)$ occurs before $l_S(c)$, then the arc $S \rightarrow T$ is added to the MPG at $l_S(c)$. This proves the result. □

Lemma 5.4 :

(i) The PG is a sub-graph of the solid arcs in the SG.

(ii) There is a cycle in the PG iff there is a solid cycle in the SG.

Proof: (i) Suppose an arc labeled a is added from S to T in the PG. This means there exist integers k and m such that $((T, k), (S, m), a) \in CP$ and now (T, k, a) is the second action in this pair to occur. Hence, the directed dashed arc from S to T corresponding to this pair becomes solid in the SG.

(ii) Observe that there might be more solid arcs in the SG than what appears in the PG (recall observation 1, section 3.2), but the extra ones can be obtained by transitivity from those in the PG. Therefore, the PG contains enough information for the detection of solid cycles in the SG. □

Lemma 5.5 [8] : The WFG is a sub-graph of the MPG.

Proof: If we have the arc $S \rightarrow T$ labeled a in the WFG, i.e. T is waiting for S to get the lock on a, then necessarily the same arc was added before to the MPG, either when:

i) T declared a which was already locked by S, or

ii) S locked a for which T was holding declare at the time. □

Corollary 5.6 [8] : Cycles in the PG and in the WFG are always anticipated by similar cycles in the MPG.

Proof: Follows from the proofs of Lemmas 5.3 and 5.5. □

Example 5.1 : Recall the transactions of Example 3.1. Consider the execution

$$E_3 = \tau_1(a)\tau_3(b)\tau_3(a)$$

and a locking execution of it (not the standard one)

$$E_{3-l} = d_1(a)d_1(b)l_1(a)\tau_1(a)d_3(b)l_3(b)\tau_3(b)u_1(a)d_3(a)l_3(a)\tau_3(a).$$

The SG of E_3 is q_3 in Figure 3.1. The PG and MPG of E_{3-l} are drawn in Figure 5.1. □

5.3 - Serializability results

We want to use the results of section 3.2 and 5.2 to determine the interpretation of cycles in the WFG, PG and MPG. Unless otherwise mentioned, the theorems in this sub-section are valid for *any* locking execution of some given execution $E \in \Sigma_e$. The state is just $\Phi(E)$, and the WFG, PG and MPG are constructed from a locking execution of E.

Theorem 5.7 : A cycle in the WFG implies that the state has previously jumped out of $Q_{\overline{csr}}$. However, a cycle free WFG does not imply that the state is in Q_{sr}.

Proof: The first statement follows from Corollaries 5.6 and 5.2, and Lemma 3.3. For the second statement, observe that the WFG is only concerned with waiting; for instance, any sequence of inputs constituting a standard locking execution (section 4.2) keeps the WFG cycle free. But every element of Σ_e has such an augmentation. □

Theorem 5.8 : The PG is cycle free iff the state is in Q_{sc}.

Proof: Follows from Lemmas 5.4 and 3.3. □

This result shows that the PG cannot detect transitions out of $Q_{\overline{cse}}$, but only those out of Q_{se}.

Theorem 5.9 :

(i) If the MPG is cycle free, then the state is in Q_{se}.

(ii) If the state is in $Q_{\overline{cse}}$, then the MPG is cycle free.

(iii) To any state q in Q_{se} corresponds a locking execution whose MPG is cycle free.

Proof: (i) Follows from Corollary 5.2 and Lemma 3.3, or observe that if the MPG has no cycles, then the PG has no cycles, and so the result is true by Theorem 5.8.

(ii) Follows from Lemma 3.3 and Theorem 5.1.

(iii) Take any element in $\Phi^{-1}(q)$ and augment it by using the standard augmentation procedure of section 4.2. The PG of this locking execution is cycle free because it is serializable. But since in this standard augmentation all "declares" immediately precede corresponding "locks," one can see that: (i) arcs in the MPG of a standard locking execution are only added at "declares," and (ii) if such an arc causes a cycle, the same cycle appears immediately after in the PG when the corresponding lock is requested. Hence, the MPG of this locking execution is also cycle free. □

In view of Lemma 3.3, the objective is to detect all cycles appearing in the SG, since it has to remain cycle free in order to get a complete serializable execution. But Theorems 5.7 to 5.9 show that:

- the WFG is of no help unless we impose some other conditions (see Lemma 6.3 (ii));
- the PG detects cycles only when they become solid in the SG; dashed and mixed cycles cannot be detected;
- the MPG is as good as the PG and furthermore it has the possibility to detect dashed and mixed cycles provided "declares" occur early enough in the locking execution. (For instance, compare the PG and MPG in Example 5.1.) This means that in contrast to the PG, states in $Q_{se-\overline{cse}}$ can potentially be detected by cycles in the MPG. This important

observation is the basis for the LP's discussed in section 7 and is one the justifications for using the declare locking action.

We conclude this section with a result about complete locking executions.

Theorem 5.10 : Let $q = \Phi(E^c)$ for some complete execution $E^c \in \Sigma_e$. Then $q \in Q_{cse}$ iff the PG is cycle free iff the MPG is cycle free iff q is cycle free.

Proof: Once the execution is complete, all arcs in the SG are solid, and so the three graphs differ only by arcs that can be obtained by transitivity. Hence, the complete execution E^c is serializable if and only if the graphs have no cycles. □

6. ANALYSIS OF LOCKING PROTOCOLS - PART I

In this section, we consider control with two types of locking actions.

6.1 - Basic locking protocol

We have found useful to present as a *basic locking protocol*, abbreviated LP0, a requirement that is common to all LP's in order to outline its effect precisely.

Protocol LP0

(1) : The augmented execution must be a locking execution.

(2) : A transaction cannot acquire a new lock on an object after it has unlocked it. □

The second requirement means that a transaction can only request one lock per object. Its effect on concurrency is as follows.

Theorem 6.1 :

(i) $Q_{\overline{cse}} \subset Q_{LP0} \subset Q_e$.[4]

(ii) $Q_{se} \not\subset Q_{LP0}$ and $Q_{LP0} \not\subset Q_{se}$.

Proof: We first make the following observation. In the SG, an arc has a label of the form $((i, k), (j, m), b)$. The effect of condition (2) of LP0 is to force all arcs with same object label (i.e. b part in the tuple) between two given nodes to have the same direction. This

[4] In sections 6 and 7, unless otherwise mentioned, all inclusions are strict in general.

direction is determined when the lock on this object is first obtained by one of these two transactions.

(i) In a cycle free state $q \in Q_{\overline{cse}}$, no two arcs between two nodes can have opposite directions. Hence, all these states are LP0-reachable. Of course, not all states in Q_e satisfy the requirement of condition (2).

(ii) Clearly, there are many states in Q_{se} that don't satisfy the above condition: e.g. consider a dashed cycle between two nodes. This shows the first non-inclusion.

On the other hand, that all arcs between two given nodes have same direction is no guarantee that solid cycles will not appear. This proves the second non-inclusion. □

Example 6.1 : The non-inclusions of (ii) above are also true for the sets of executions, i.e. $\Sigma_{se} \not\subset \Sigma_{LP0}$ and $\Sigma_{LP0} \not\subset \Sigma_{se}$ (see remark 3.1). Here are examples for the executions indicated in Figure 6.1. Recall

$$T_1 = r_1(a)r_1(b) \quad T_3 = r_3(b)r_3(a) \quad T_5 = r_5(a)r_5(a).$$

$E^c_{1-5} = r_5(a)r_1(a)r_5(a)r_1(b) \notin \Sigma_{se}$ (Theorem 2.1), and also $E^c_{1-5} \notin \Sigma_{LP0}$ because T_5 violates condition (2) of LP0.

$E_{(1-5)} = r_5(a)r_1(a)$ is serializable but again T_5 violates (2) of LP0.

$E^c_{1-3} = r_1(a)r_3(b)r_1(b)r_3(a)$ is LP0-augmentable (e.g. take $E^{c\prime}_{1-3}$ given by the standard augmentation procedure of section 4.2 and delete "declares") but is not serializable (by Theorem 2.1). □

Observe that protocol LP0 is not an acceptable concurrency control method since it does not guarantee that the state remains inside Q_{se}. Due to the second part of Theorem 6.1, we define $\Sigma_{se\,0} := \Sigma_{se} \bigcap \Sigma_{LP0}$ and $\Sigma_{se\,0-\overline{cse}} := \Sigma_{se\,0} - \Sigma_{\overline{cse}}$. Similarly for $Q_{se\,0}$ and $Q_{se\,0-\overline{cse}}$.

When condition (2) of LP0 is in force, we group all arcs between two nodes which have same object label into a single one since one arc carries all information we need for our purposes. This induces a concatenation of some states in Q_e and eliminates other states in it, yielding Q_{LP0}. Also, the arcs of the SG need only have an object label now.

However, this means we cannot completely reconstruct E from $Traj(E)$ when E has more than one action on an object because we only know when the first such action occurs (which is when the arc becomes directed), and an upper bound for the time when the last one occurs (which is when this object is used by another transaction, i.e. when the arc becomes solid). Nevertheless this undeterminacy is of no consequence for our analysis of concurrency. This is because if a concatenated state is reachable, all original ones before concatenation are reachable too. Hence, Lemma 4.1 is still true.

Remark 6.1 : Example 6.1 brings to the attention the fact that, in a practical situation, a transaction might not know if it can unlock an object because it does not know yet if it will need it later in the future. This in fact applies to all requirements of category (2) in the LP's we will consider. Clearly, no model can account for such situations. Hence, we stress that the proper interpretation to the concept of LPx-augmentability is that, given an execution, *there exists* an augmentation of it that is an LPx-execution. In some sense, this is "optimistic" for states in $Q_{\overline{cse}}$ but "pessimistic" for states out of $Q_{\overline{cse}}$. □

6.2 - Two-Phase Locking

We now wish to analyze the well-known *Two−Phase Locking Protocol* [1], abbreviated LP-2ϕ, in our framework.

Protocol LP-2ϕ

(1) : The augmented execution must be a locking execution.

(1') : The WFG must remain cycle free.

(2ϕ *condition*) : A transaction has to acquire all needed locks before it can unlock any object. □

(Observe that the 2ϕ condition implies condition (2) of LP0.)

Lemma 6.2 : Consider an execution and any locking execution of it.

(i) The 2ϕ condition guarantees that no solid arc is preceded by a dashed arc in the SG.

(ii) When the 2ϕ condition is in force, all dashed cycles in the SG eventually appear in the WFG.

Proof: (i) By way of contradiction, suppose a solid arc is preceded by a (directed or not) dashed arc in the SG, and call T the transaction node where these arcs are attached. The solid arc out of T means that T has unlocked some object. But the dashed arc attached to T means that T has not yet locked the object labeling this arc. This contradicts the 2ϕ condition.

(ii) Consider a dashed cycle in the SG. This means that only one of the two actions of each respective conflicting pair has occured yet. But each transaction involved in this cycle will eventually request a lock for the object labeling the arc going into it. Since all transactions observe the 2ϕ condition, all these requests will be placed before any lock is released by any of these transactions. Therefore, none of these requests can ever be granted and the same cycle eventually appears in the WFG. □

Theorem 6.3 :

(i) $Q_{LP-2\phi} \subset Q_{se\,0}$.

(ii) $Q_{\overline{cse}} \not\subset Q_{LP-2\phi}$.

Proof: (i) Lemma 6.2 (i) guarantees that no solid or dashed cycles will ever appear in the SG, and so by Lemma 3.3 and the fact LP-2ϕ is more restrictive than LP0, the state remains within $Q_{se\,0}$. The inclusion is strict because $Q_{se\,0}$ surely contains states where a solid arc follows a dashed one, whereas $Q_{LP-2\phi}$ does not.

(ii) Clearly, $Q_{\overline{cse}}$ contains many states where a solid arc follows a dashed one.

(Examples are given below.) □

The interpretation of the above results is that the 2ϕ condition is so strong that the controller need only keep the WFG as state estimate. The state remains all the time within Q_{se} (in fact $Q_{se\,0}$), and transitions out of $Q_{\overline{cse}}$ (i.e. deadlocks) are eventually detected by cycles in the WFG (from Lemma 6.2 (ii)). The price to pay for using such a simple controller is that only a fraction of $Q_{\overline{cse}}$ is now reachable, meaning that

concurrency is considerably less than what is admissible.

Example 6.2 : (i) Recall the transactions in Example 3.1. The execution $E_3 = \tau_1(a)\tau_3(b)\tau_3(a)$, whose SG is q_3 in Figure 3.1, is in $\Sigma_{se\,0}$ because it is serializable and the following is an LP0-execution if it:

$$E_{3-LP0} = l_1(a)r_1(a)u_1(a)l_3(b)r_3(b)l_3(a)r_3(a).$$

But $q_3 \notin Q_{LP-2\phi}$ (from Lemma 6.2 (i)), and so $E_3 \notin \Sigma_{LP-2\phi}$ by an application of Lemma 4.1.

(ii) Recall the complete (serializable) execution of Example 4.1.

$$E^c_{1-4-5} = \tau_1(a)\tau_5(a)\tau_5(a)\tau_4(b)\tau_1(b).$$

Figure 6.2 shows $Traj(E^c_{1-4-5})$. q_5 cycle free shows the execution is serializable. But $q_2 = q_3$ and q_4 are not LP-2ϕ-reachable, so E^c_{1-4-5} is not LP-2ϕ-augmentable. □

7. ANALYSIS OF LOCKING PROTOCOLS - PART II

7.1 - Protocols based on graphs

In the spirit of section 6.1, we now present and study two protocols, not for their practical interest, but because that will help in understanding the Declare-Before-Unlock protocol that we discuss in the next sub-section. Protocol LP-PG is for control with two types of locking actions, and protocol LP-MPG is for the case when "declare" is also employed.

Protocols LP-PG and LP-MPG

(1) and (2) : As in protocol LP0.

(1') : The PG (MPG respectively) must remain cycle free. □

Theorem 7.1 : $Q_{LP-PG} = Q_{LP-MPG} = Q_{se\,0}$.

Proof: (i) $Q_{LP-PG} = Q_{se\,0}$. From Theorem 5.8 and the fact that LP0 is contained in LP-PG, we have that $Q_{LP-PG} = Q_{LP0} \bigcap Q_{se} =: Q_{se\,0}$.

(ii) $Q_{LP-MPG} = Q_{se\,0}$. As in (i), but this time using Theorem 5.9. □

Condition (1') means that locking inputs "lock" (and "declare" for LP-MPG) are rejected by the controller if they cause a cycle to appear in the PG (MPG). In the case of LP-PG, it means the state is already in $Q_{se\,0-\overline{cse}}$ and so there is deadlock (recall Lemmas 5.4, 3.3 and Corollary 3.4).

In the case of LP-MPG, the situation is different and two cases have to be considered. First, suppose a request by T for "declare" on object a is the cause of the cycle detected in the MPG. The proof of Lemma 5.1 shows that a similar cycle is already present in the SG. Hence, the state jumped out of $Q_{\overline{cse}}$ before and we have deadlock. Intuitively, this "declare" is coming too late. The transition out of $Q_{\overline{cse}}$ occurred when the first conflicting action in this pair was accepted (because of insufficient information then).

Second, suppose that a request by T for "lock" on a is the cause of the cycle. This time, as mentioned in the proof of Lemma 5.1, a similar cycle appears simultaneously in the SG. Provided there are no other (dashed or mixed) cycles in the SG, this means the state would jump out of $Q_{\overline{cse}}$ if the lock were granted. So there is no deadlock and the strategy is not to grant the lock and ask T to wait until its predecessors are done with object a. Refer to [8] for a detailed treatment of this issue. (If there are other cycles in the SG, they will eventually appear in the MPG (at a "declare" request) and this will result in deadlock.)

These remarks show that although in the worst case LP-MPG does no better than LP-PG in terms of reachable states, it has the potential to do better in the sense that *cycles caused by lock requests do not correspond to deadlock but can be resolved by waiting*. This capability for improved performance is execution dependent and is a function of how early "declare" locking actions are placed. For instance, standard locking executions never cause such cycles because "declare" always immediately precedes "lock" and so only arcs of type (a) in the definition of the MPG are ever drawn. Hence, more stringent requirements than merely "declare before corresponding lock" are a necessity for reducing the set of reachable states. This is of crucial importance to understand the usefulness of the "declare" locking

action. Such ways to improve LP-MPG are discussed in the next sub-section and in appendix B.

7.2 - Declare-Before-Unlock protocol

The Declare-Before-Unlock protocol (denoted LP-DBU here) is proposed in [8] as a protocol that achieves all complete serializable executions and permits early detection of deadlocks. It is a stronger version of protocol LP-MPG where a condition of category (2) is added, namely the *DBU* condition.

Protocol LP-DBU

(1), (2) and (1') : As in protocol LP-MPG.

(*DBU condition*) : A transaction must declare all objects it needs before it can unlock any object. □

More details about LP-DBU can be found in [8]. Here, our objective is to identify Q_{LP-DBU}.

Lemma 7.2 : The *DBU* condition guarantees that all cycles in the SG that are not in the MPG (called undetected cycles) must contain at least two consecutive dashed arcs.

Proof: First observe that all solid arcs in the SG necessarily appear in the MPG (Lemmas 5.3 and 5.4), and so all undetected cycles must have at least one dashed arc. The undetected arcs are due to "declare" actions that have not been placed yet. By way of contradiction, suppose there is an undetected cycle in the SG with some dashed arcs, but no two consecutive. At least one of these arcs is not present in the MPG. Consider Figure 7.1 and suppose $S \rightarrow T$ is such an arc. T has unlocked object b (because the corresponding arc is solid) but has not declared object a yet (because of the arc $S \rightarrow T$ is not in the MPG). This means T has violated the *DBU* condition. □

Theorem 7.3 : $Q_{\overline{CSR}} \subset Q_{LP-DBU} \subset Q_{SR0}$.

Proof: By an application of Lemma 7.2 and previous results about LP0. The first inclusion follows by observing that all cycle free SG's are reachable by LP-DBU since they do not

violate any of its constraints. The second one is due to the fact that: (i) no states out of Q_{se0} are reachable since LP-DBU is more restrictive than LP-MPG, and (ii) states in Q_{se0} with mixed cycles that do not contain at least two consecutive dashed arcs are not reachable by LP-DBU. □

Corollary 7.3 : $Q_{LP-2\phi} \subset Q_{LP-DBU}$.

Proof: This inclusion follows from the results in Lemmas 6.2 and 7.2. □

Example 7.1 : Consider Figure 7.2. The following are examples of executions identified in this figure.

- E^c_{1-4-5} : see Example 6.2 (ii).

- For E_2, recall T_1 and T_3 of Example 3.1 and observe that

$$E_2 = \tau_1(a)\tau_3(b)$$

whose SG is q_2 in Figure 3.1, will result in a cycle in the WFG, i.e. deadlock.

- E_3 : see Examples 5.1, 6.2 (i) and Figures 3.1, 5.1.

- For E_4, consider the three transactions:

$$T_6 = \tau_6(c)\tau_6(b) \qquad T_7 = \tau_7(a)\tau_7(b)\tau_7(c) \qquad T_8 = \tau_8(a)$$

and the execution

$$E_4 = \tau_7(a)\tau_8(a)\tau_6(c)\tau_7(b).$$

$Traj(E_4)$ is given in Figure 7.3. It shows that E_4 is not LP-2ϕ-augmentable, but is LP-DBU-augmentable. □

As mentioned when protocol LP-MPG was presented, deadlock occurence is a function of how early declare actions are placed in a locking execution. If a transaction knows beforehand all objects it needs, the *Prior Declaration Protocol* of [8] guarantees deadlock never occur. In this protocol, the *DBU* condition is replaced by a *DBL* condition: "each transaction has to declare all objects before its first lock." In this case, the MPG corresponds exactly to the SG, and the set of reachable states is exactly $Q_{\overline{csc}}$. (As mentioned in section 3.3, optimal performance is possible when complete state information is

available.) The problem is that the *DBL* condition can rarely be met in practice.

Appropriate conditions are those that delay unlocking or locking until more or all declares are placed. An example is the *DBU* condition which is simple and easy to implement. A disadvantage of LP-DBU however is that more states in $Q_{se\,0-\overline{cse}}$ are reachable than with LP-2ϕ. Our investigations to specify a protocol that reaches all of $Q_{\overline{cse}}$ but about the same portion of $Q_{se\,0-\overline{cse}}$ as LP-2ϕ, i.e. a protocol that achieves maximum concurrency but about the same number of deadlock states as LP-2ϕ, have led to the *No-Declaring-Phase Protocol* presented in appendix B.

8. CONCLUSION

In this paper, we have presented a state space model for the study of concurrency control in database management systems. We have shown that this problem is in effect one of control with partial state information. This means that there does not exist a control method that will achieve maximum concurrency without avoiding roll-back of one or more transactions at some point in the execution.

We have applied our model to the analysis of some locking protocols. In our view, when the degree of concurrency achievable by two-phase locking is insufficient for some application, the Declare-Before-Unlock protocol of [8] is a good alternative because it permits to achieve maximum concurrency and has relatively simple conditions. At the core of this protocol is the locking action "declare." We have showed how this action performs the task of prediction and permits the construction of a better state estimate, the must-precede graph.

All results in this paper are generalizable to the case where one distinguishes between read and write actions and where different grades of locks are used (e.g. read and write locks), provided one uses as correctness criterion conflict-serializability. This is because our analysis is based on conflicting pairs of actions. In this case, these pairs will also depend on the action types (see section 2.2) and they will determine a compatibility

matrix [2] of the corresponding locks. However, the model and its properties remain the same: only the construction of the initial state q_0 is affected. (See [8] for the modifications to the MPG and to LP-DBU.)

The problem of concurrency control is intrinsically more complicated when the database is distributed at different sites [2, 3, 11]. Specific results about the complexity of this problem in a given framework for distributed databases are presented in [11]. In our model, we made no restrictions concerning the physical location of the objects. We assumed that the actions in all the transactions and executions were totally ordered, although our framework could be generalized to partially ordered transactions and executions (as considered in [11] for instance). However, we analyzed only the case of a unique central controller for the system of transactions. In appendix C, we discuss the issue of decentralized control when the database is distributed and propose a method for aggregating the state and state estimates at each site.

Finally, we mention that our model can be used to analyze other concurrency control techniques than locking, in particular timestamp-based techniques [2]. The idea is to determine what effects these techniques have on the state graph, in a way similar to what is done in sections 6 and 7.

APPENDIX A - Relation to Supervisory Control of Discrete Event Processes

A.1 - Controlled generator model

In the terminology of [10], our control problem is (ideally) to construct a controller or supervisor for the generator $G = (Q_e, \Sigma, \phi_e, q_0, Q_m)$, such that the language generated by the controlled generator is $\Sigma_{\overline{cse}}$ only. The supervisor can be viewed as another generator $S = (X, \Sigma, \xi, x_0, X_m)$ with state space X, state transition function ξ, initial state x_0, and same set of inputs Σ (X_m need not be specified for now). The controlled generator, denoted S/G, corresponds to the situation where S and G are coupled in the following sense: (i) the state transitions of S are forced by that of G, and (ii) the state transitions of G are constrained by a feedback control map depending on the state of S; this feedback map acts on G by enabling or disabling state transitions.

Roughly speaking, the theory in [10] says that if q_0 is known beforehand, then there exists a supervisor that will attain the ideal control objective. This is because the language $\Sigma_{\overline{cse}}$ is controllable (as defined there). But in our case the emphasis is on a different issue: partial state information. There are no controllability constraints on the languages we are interested in because we assume any incoming action can be accepted or rejected. However, the information available to our supervisor is incomplete. Typically, $x \in X$ is a partial version of the state of the system q. The design task is hence two-fold: (i) construct a good state estimate to be used by the supervisor, and (ii) define the feedback map $S \rightarrow G$ so that the serializability requirements are satisfied.

A.2 - Incorporation of locking to the generator

It is convenient to view locking as a partially decentralized control strategy for our sytem, each transaction being responsible for its own locking actions and for the constraints that concern it alone (including the constraints of category (2), section 4.3). In other words, we augment the set of inputs Σ to include the locking actions and consider the problem of scheduling concurrent augmented transactions. Specifically,

$$\Sigma^l := \Sigma \bigcup \{ \bigcup_{j=1}^{N} \bigcup_{\text{all objects } o \text{ used by } T_j} \{d_j(o), l_j(o), u_j(o)\} \}$$

is the new set of inputs. The new generator is $G^l = (Q_e, \Sigma^l, \phi_e^l, q_0, Q_m)$ where $\phi_e^l : \Sigma^l \times Q_e \to Q_e$ is only defined for "action" inputs (i.e. inputs in Σ):

$$\phi_e^l(\sigma, q) = \begin{cases} \phi_e(\sigma, q) & \textit{whenever } \sigma \in \Sigma \text{ and } \phi_e(\sigma, q) \textit{ is defined;} \\ \textit{undefined otherwise.} \end{cases}$$

The supervisor is now of the form $S^l = (X, \Sigma^l, \xi, x_0, X_m)$, where we can assume without loss of generality that the transition function ξ is only defined for "locking" inputs. This is because, as a consequence of the locking constraints of section 4.1, the information carried by the locking inputs of a locking execution contains that carried by the action inputs in it.

The action of the LP's we study in sections 6 and 7 can be interpreted as follows.

1- Conditions of category (1) involving the state x of the supervisor (typically, cycle free conditions on x) define the feedback map from S^l to G^l. For instance, all inputs causing a cycle in x are rejected and the transitions they would have caused in G^l are disabled.

2- Conditions of category (2) on the augmentation of a transaction correspond to restricting the set of strings of elements of Σ^l that can constitute inputs. In some cases (such as (2) of LP0 in section 6.1 and the 2ϕ condition of section 6.2) this also restricts the set of strings of Σ that can constitute inputs (i.e. the non-augmented inputs to which the controlled G^l responds). For instance, Lemma 6.2 gives a description of states that are no more reachable when 2ϕ is in force. All executions whose trajectories go through these states are then eliminated as possible input strings.

In other cases, conditions of category (2) (such as the *DBU* condition of section 7.2) do not directly restrict the possible non-augmented inputs but instead enable the supervisor to construct a better state estimate x, therefore making the feedback map more complete. (This is the role of the "declare" locking action.)

APPENDIX B - No-Declaring-Phase Protocol

When one wants to reduce the incidence of deadlock with respect to LP-DBU but still achieve maximum concurrency, one has to find conditions ensuring that less mixed and dashed cycles in the SG will go undetected in the MPG. Recalling Lemma 6.2, we see that the 2ϕ condition ensures no mixed cycles can be reached. However, since the effect of this condition is on "sequences of arcs" rather than on "cycles," it also considerably reduces the portion of $Q_{\overline{cse}}$ that is reachable. Our objective here is to specify conditions that do not impair concurrency but that guarantee all mixed cycles (and maybe some dashed ones too) in the SG will be detected in the MPG.

For this purpose, it is necessary to distinguish between solid and dashed arcs in the MPG as well. Therefore, let all arcs added to the MPG in its definition in section 5.1 be dashed, and add the following part to its construction:

(c) when transaction T locks object a, replace all dashed arcs into T with label a by solid arcs in the same direction.

This way, the arcs in the MPG have same type as they have in the SG.

Next, we need to introduce more terminology. S is called a *predecessor through* c of T, and T a *follower through* c of S, if in the path from S to T in the MPG, one of the arcs has c as label. S is called a *solid predecessor* of T, and T a *solid follower* of S, if the path from S to T in the MPG contains at least one solid arc. A transaction is said to be in the *declaring phase* if it has not declared all its objects yet. An object is said to be in state dsf if one or more of the transactions that hold declare on it have a solid follower in the MPG. We now state the No-Declaring-Phase protocol, abbreviated LP-NDP.

Protocol LP-NDP

(1) (2) and (1') : As in protocol LP-MPG.

(3) : No lock on object a is granted to transaction T if one of the two following conditions is satisfied:

(i) T has a predecessor through a in the MPG that is still in the declaring phase.

or

(ii) T or a predecessor of it in the MPG is in the declaring phase and a is in state dsf. □

Lemma B.1 : Condition (3) of LP-NDP guarantees that all mixed cycles and all dashed cycles preceding a solid arc in the SG are detected in the MPG.

Proof: (Outline) Condition (3-i) implies that when an arc becomes solid in the MPG, all the transactions in all the paths to the left of this arc in the MPG have declared all their objects. Hence, as far as these transactions are concerned, the MPG has complete information and it will detect all cycles going through these nodes in the SG.

Condition (3-ii) is to prevent the addition of new predecessors to the left of a solid arc when one of these predecessors is still in the declaring phase. This guarantees that when new predecessors are added to the left of a solid arc in the SG, they are also added in the MPG, and all the information about them is known because they have finished declaring. The two conditions together imply the result. □

Theorem B.2 : $Q_{\overline{cse}} \subset Q_{LP-NDP} \subset \{q \in Q_{se0} : q$ is cycle free or has dashed cycles only $\}$.

Proof: Clearly, all cycle free SG's are reachable by LP-NDP since they do not violate any of the conditions of this protocol. (The transactions need only have declared for condition (3) of LP-NDP to be satisfied, and "declares" are non-conflicting actions.) The second inclusion is a consequence of Lemma B.1 and Theorem 7.1, because now condition (3) guarantees that all dashed cycles preceding a solid arc and all mixed cycles are detected in the MPG. □

Corollary B.3 : $Q_{LP-2\phi} \subset Q_{LP-NDP}$

Proof: By observing that the 2ϕ condition implies that condition (3) of LP-NDP is always satisfied. □

Observe however that the deadlock states that are reachable by LP-NDP but not by LP-2ϕ are not a cause of concern since the extra arcs they contain (typically dashed arcs

followed by solid arcs preceding or not connected to a dashed cycle) are not involved in the cycles causing deadlock. Also observe that the *DBU* condition implies that the immediate predecessor through a of T never causes condition (3-i) to be satisfied when T requests the lock on a. (The same is true for all immediate solid predecessors of a transaction, since by *DBU* they have declared all their objects.) This is why in this case undetected mixed cycles always contain at least two consecutive dashed arcs (Lemma 7.2).

Protocol LP-NDP is of theoretical interest because of the above results. However, its practical usefulness is limited because it is difficult to implement.

APPENDIX C - Decentralized Concurrency Control for Distributed Databases

Consider the situation where the database is distributed at many sites and it is desired that the control be decentralized (in the sense of local controllers at each site instead of a central one). Suppose also that it is required that the properties of the "global system" concerning serializability and concurrency (as developed in section 3) be preserved, i.e. that the local controllers do as good as a central controller would do. Clearly, one way to achieve this objective is to maintain a copy of the complete SG (or its estimate used for control) at each site. Here, we present a way of aggregating locally the SG (or the MPG) that permits to achieve the same performance as a central controller without the need to have all the information about the global system at each site. (This corresponds to the idea of model aggregation in decentralized control theory, see e.g. [12].)

We want each site to have all necessary information about the status of serializability (in the global system) of the transactions acting at this site. For this purpose, we assume each transaction knows beforehand if it needs objects:

(i) from only one site, in which case it informs the given site and it is said to be a *local transaction* ;

or (ii) from more than one site, in which case it informs all the sites that it is a *common transaction* .

Each local graph (be it a SG or a MPG) has a node for each local transaction at this site (termed a local node), and a node for each common transaction in the global system (termed a common node). Consider the initial global SG, and do a transitive closure of the arcs between the common nodes (it is not necessary to put a label on the new arcs). Then each initial local SG is the restriction of this modified initial global SG to the common nodes and the given local nodes, and to the arcs attached to them.

We now treat the updating of the local SG's and MPG's simultaneously. Upon the arrival of an incoming action (locking action for the MPG) on an object at a given site, update the local graph following the same rules as before (i.e. as in sections 3.1 and 5.1). Moreover, if the newly added directed arcs create a path (solid path) between two common nodes in this local graph, inform all the other sites to put a directed arc (directed solid arc) between these nodes in their graphs. (This arc could for instance have for label the site number where the path is created.)

This means that each site has complete information about the current partial ordering of its local transactions and all the common transactions. Hence, we can show that: a (dashed, mixed or solid) cycle occurs in the global SG (MPG) iff a similar cycle occurs in some local SG (MPG). (The proof is straightforward but is omitted for the sake of brevity.)

Clearly, the level of aggregation that is possible will be function of the proportion of local transactions, but the above procedure gives the maximum aggregation that is possible under the constraints that there is no loss in performance and that the decisions are made locally.

FIGURES

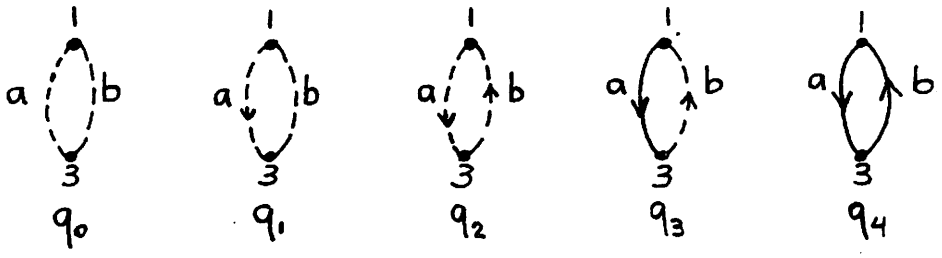

Figure 3.1 - $Traj(E^c_{1-3})$ of Example 3.1

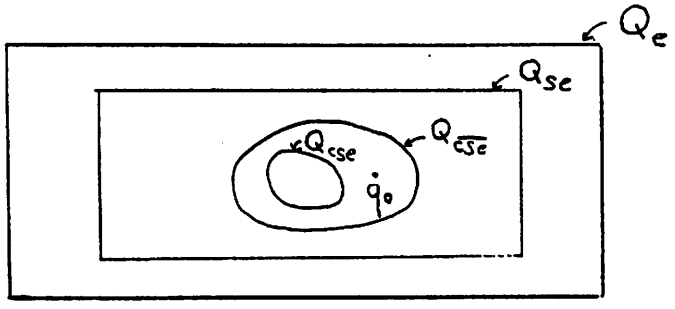

Figure 3.2 - State space subsets

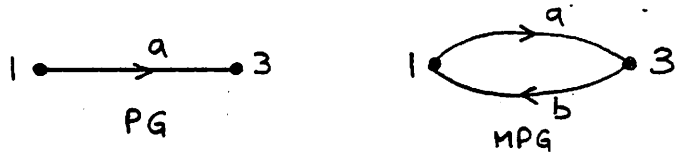

Figure 5.1 - PG and MPG of E_{3-1} of Example 5.1

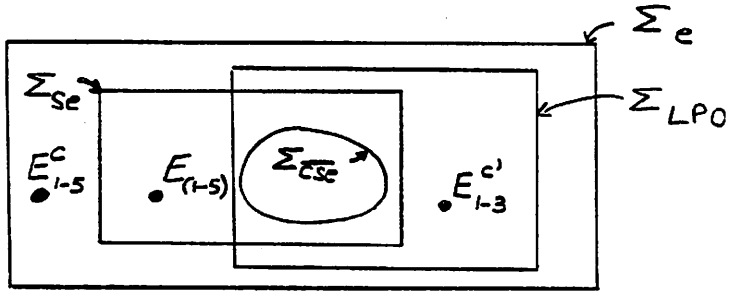

Figure 6.1 - Executions of Example 6.1

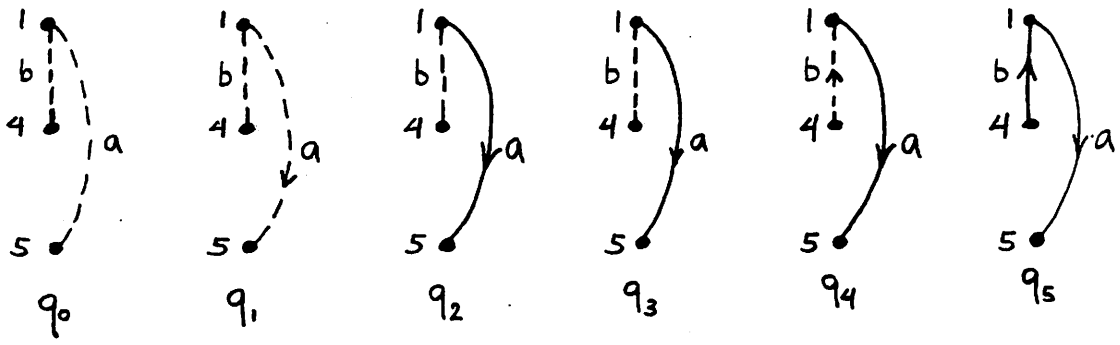

Figure 6.2 - $Traj(E^c_{1-4-5})$ of Examples 4.1 and 6.2 (ii)

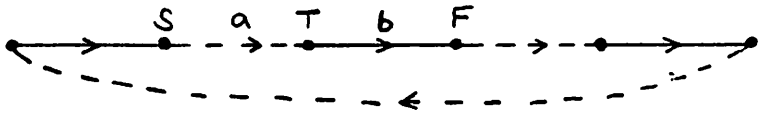

Figure 7.1 - Proof of Lemma 7.2

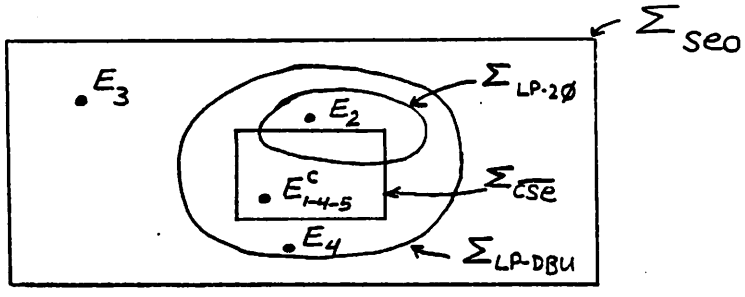

Figure 7.2 - Executions used as examples in the paper

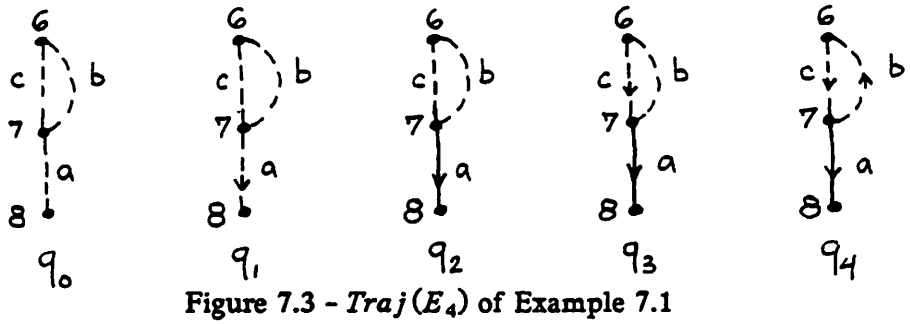

Figure 7.3 - $Traj(E_4)$ of Example 7.1

REFERENCES

[1] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, Vol. 19, No. 11, November 1976, pp. 624-633.

[2] C. J. Date, *An Introduction to Database Systems - Volume II*, Reading, MA: Addison-Wesley, 1983.

[3] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 185-221.

[4] J. Gray, "Notes on Database Operating Systems," in R. Bayer, R. M. Graham and G. Seegmuller (eds.), *Operating Systems: An Advanced Course*, Springer-Verlag, 1978.

[5] C. H. Papadimitriou, "Concurrency Control by Locking," *SIAM Journal on Computing*, Vol. 12, No. 2, May 1983, pp. 215-226.

[6] C. H. Papadimitriou, "Serializability of Concurrent Updates," *Journal of the ACM*, Vol. 26, No. 4, October 1979, pp. 631-653.

[7] M. Yannakakis, "Serializability by Locking," *Journal of the ACM*, Vol. 31, No. 2, April 1984, pp. 227-244.

[8] S. Lafortune and E. Wong, "A New Locking Protocol That Achieves All Serializable Executions," Memorandum No. UCB/ERL M 84/77, Electronics Research Laboratory, University of California, Berkeley, CA 94720, September 1984. Submitted for publication.

[9] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Reading, MA: Addison-Wesley, 1979.

[10] P. J. Ramadge and M. W. Wonham, "Supervisory Control of a Class of Discrete Event Processes," Systems Control Group Report #8311, Department of Electrical Engineering, University of Toronto, Canada, October 1983. An earlier version of this work appeared in *Feedback Control of Linear and Nonlinear Systems*, Lecture Notes in Con-

trol and Information Sciences No. 39, Springer-Verlag, Berlin, pp. 202-214.

[11] P. C. Kanellakis and C. H. Papadimitriou, "The Complexity of Distributed Concurrency Control," *SIAM Journal on Computing*, Vol. 14, No. 1, February 1985, pp. 52-74.

[12] N. R. Sandell, Jr., P. Varaiya, M. Athans, and M. G. Safonov, "Survey of Decentralized Control Methods for Large Scale Systems," *IEEE Transactions on Automatic Control*, Vol. AC-23, No. 2, April 1978, pp. 108-128.