LOCKING POLICIES: SAFETY AND FREEDOM FROM DEADLOCK

Mihalis Yannakakis
Bell Laboratories
Murray Hill, NJ 07974

and

C. H. Papadimitriou
Massachusetts Institute of Technology
Cambridge, MA 02139

and

H. T. Kung**
Carnegie-Mellon University
Pittsburg, PA 15213

1 INTRODUCTION

A database consists of *entities* which relate to each other in certain ways, i.e., they satisfy certain *consistency constraints*. Many times, when a user updates the database, he may have to update temporarily these constraints in order to eventually transform the database into a new, consistent state. For this reason, atomic actions by the same user are grouped together into units of consistency called *transactions*. In practice, a transaction may be either an interactive session, or the execution of a user update program. When, however, many transactions access and update the same database concurrently, there must be some kind of coordination to ensure that the resulting sequence of interleaved atomic actions (or *schedule*) is *correct*. This means that all transactions have a consistent view of the data, and furthermore the database is left at the end in some consistent state.

This required coordination is achieved via the *concurrency control mechanism* of the database. Considerable research effort has been devoted recently to the theoretical aspects of the design of such a system [EGLT1, SLR, SK, KS, Pa, PBR, KP]. The theory of database concurrency control bears a superficial similarity to the operating systems-inspired concurrency theory [KM, CD]. The difference is that in operating systems we have cooperating, monitoring, and monitored, processes, and the goal is to prevent bad cooperation or management (e.g. indeterminacy, deadlocks). In databases, we have a population of users that are unaware of each other's presence; the goal is to protect them from the dangers of this ignorance (e.g., creation of bad data because of an unfortunate sequence of accesses and updates originated from two users). Deadlocks are important only in conjunction with correctness, as possible defects of the concurrency control mechanism.

A concurrency control mechanism is evaluated basically in terms of the *parallelism* that it supports — roughly, the class of all schedules that are possible responses of the system to incoming user requests. The goal is therefore to design a concurrency control mechanism that has as rich such class as possible, while at the same time allowing only *correct* schedules.

The right notion of correctness in this context is not immediately obvious. Virtually all researchers in the area [e.g. EGLT1, SK, Pa, PBR] have adopted the notion of *serializability*. A sequence of atomic steps is serializable if it is equivalent (in a schema-theoretic sense) to a *serial* schedule, one in which the users execute their programs sequentially, one at a time. In fact, in [KP] it was shown that this is indeed the right notion of correctness when only syntactic information on the transactions is available — as is usually the case. If some semantic information is also available, then more relaxed definitions are possible. In this paper we focus on serializability.

With the notable exception of the SDD1 system [BGRP], all solutions to the concurrency control problem proposed thus far are based on *locking*, i.e., binary semaphores controlling the access to data. Each transaction locks entities according to some *locking policy* in such a way that when the transaction runs concurrently with any possible set of transactions that follow the same locking policy, any schedule that may result is guaranteed to be correct, (that is, serializable). Such a locking policy is called *safe*. The paradigm of safe locking policies is the *two-phase locking* (2PL) policy proposed in [EGLT1]. In

* Work partially supported by NSF Grants MCS 77-01193, MCS 77-05314.
** Work partially supported by NSF Grant MCS 75-222-55 and Office of Naval Research Contracts NOOO14-76-C-0370, NR 044-422.

2PL a transaction must lock any entity that it needs before its access, and may unlock it at any time after its access. However, after some entity is unlocked, the transaction cannot lock any more entities. Thus a transaction has two phases: the *locking phase*, during which the transaction may request, but does not release, locks, and the *unlocking phase*. It is shown in [EGLT1] that 2PL is a safe locking policy, and furthermore that it is necessary for safety, in the sense that if transaction T_1 is not two-phase locked, then there is another transaction, T_2, such that the pair $\{T_1, T_2\}$ is unsafe.

However, in [SK] another safe policy, the *tree policy* (TP), was proposed. Here the entities are arranged in a rooted tree, and transactions access whole subtrees of entities. A transaction T may access unconditionally the root of its subtree by first locking it. Subsequently, T may lock an entity only if its father in the tree is presently locked. Notice that TP is a *family* of policies (one for each underlying tree) rather than a single locking policy. It may result in transactions that are not two-phase locked, and still it is provably safe.

In this paper we embark on a theoretical examination of locking policies and safety in general. In Section 2 we describe our model and formally define our terminology. Section 3 is devoted to the following question: Is locking a good concurrency control primitive? We present an answer (very simple analytically), which we interpret as negative. In particular, we show that the set of schedules, that are possible responses of any concurrency control mechanism based on locking must satisfy a very rigid "obliviousness" condition, which appears to forbid the use of any sophisticated methodology of increasing parallelism.

In Sections 4 and 5 we characterize safe locking policies. We first give a characterization of safety for the case of two transactions. In doing so, we employ a geometric methodology reminiscent of that used by Dijkstra for studying deadlocks [CES].

Here we use it in a very different way to study *incorrect completions*, ignoring deadlocks. Besides its independent interest and elegance, the two-transactions solution is the building block for solving the general case (Section 5). It turns out that a locking policy defined on d>2 transactions is safe iff all of its restrictions to two transactions are safe, plus a combinatorial condition. This combinatorial condition is shown to be NP-hard [Ka, GJ], but it is simple enough to have interesting corollaries. For example, the safety of 2PL and TP follows very easily. Furthermore a generalization of TP called *digraph policy* (DP, in which the entities are arranged on a directed acyclic graph instead of a tree) is also shown safe.

It can be trivially shown that if no structure is to be imposed on the entities — i.e., a policy must remain safe under any arbitrary renaming of the entities — then 2PL is essentially necessary for safety. Now, locking policies like TP and DP get around this by imposing a structure (a tree and a DAG respectively) on the entities. In practice, such structures may reflect either a physical (e.g. trees or DAG's of pointers) or a logical (e.g. flow of consistency constraints) organization of the entities. The idea in TP and DP is to take advantage of this structure so as to gain parallelism. Are all safe locking policies, then, expressible as policies operating on appropriately general, "nice" (in some intuitive sense) structures? The answer for the general case is most likely "no", (unless NP = co-NP) since safety was shown to be NP-hard. Therefore in Section 6 we restrict our attention to a natural subclass of locking policies called *L-policies*. *L*-policies are those that can be stated in terms of conditions that determine whether a given entity can be locked or not, based on the portion of the transaction up to this point. We show that safe *L*-policies can be modeled by a certain policy HP that operates on appropriately defined *hypergraphs*.

We also examine the issue of *deadlocks* (Section 7). It was known that TP is deadlock-free [SK]. Here we extend this to show that DP is deadlock-free, and that, in fact, it is the most general safe *and* deadlock-free policy for a pair of transactions. However, we show that deciding whether a set of transactions is not deadlock-free is NP-complete, even when the transactions are restricted to be two-phase locked. We show that for safe *L*-policies freedom from deadlock depends only on the order in which entities are locked (and not on where they are unlocked, i.e. how safety is enforced), and describe some ways for achieving freedom from deadlock.

2 DEFINITIONS

A *transaction system* $\tau = \{T_1, \ldots, T_d\}$ is a set of *transactions*. A transaction $T_i = (T_{i1}, \ldots, T_{im_i})$ is a sequence of *actions*. Each action T_{ij} has associated with it an *entity*, $x_{ij} \in E$, where E is a set of entities. The x_{ij}'s need not be distinct.

Each action T_{ij} is thought of as the indivisible execution of the following

$$T_{ij} : t_{ij} := x_{ij}$$
$$x_{ij} := f_{ij}(t_{i1}, \ldots, t_{ij})$$

The first instruction stores the current value of x_{ij} to a local variable t_{ij}, not in E, and the second changes x_{ij} in the most general possible way based on all available local (to the transaction) information. The t_{ij}'s are all distinct. We let $R(T_i)$ be the set $\{x_{ij} : j = 1, \ldots, m_i\}$. A *schedule* s of τ is a permutation of all steps of τ such that $j < k \leq m_i$ implies $s(T_{ij}) < s(T_{ik})$. The set of all schedules in S. s is called *serial* if, for all i and $j < m_i$ $s(T_{ij}) + 1 = s(T_{i,j+1})$. Two schedules are *equivalent* if they are equivalent as parallel program schemata with uninterpreted f_{ij}'s. s is *serializable* (notation: $s \in SR$) if it is equivalent to some serial schedule.

Deciding serializability of a schedule s is known to be NP-complete if we distinguish between reading and writing steps [Pa, PBR], but can be easily done in our model as follows [EGLT1]: Construct a digraph $D(s)$ by corresponding a node v_i to each transaction T_i, and drawing an arc (T_i, T_j) whenever, in the schedule s, T_i updates an entity before T_j does. Then s is serializable iff $D(s)$ is acyclic. A *locked transaction system* $L(\tau)$ is a special augmented version of the (ordinary) transaction system τ. The operator L performing this augmentation is

called *locking*. The entities of $L(\tau)$ are $E \cup LV$, where LV is a set of special entities called *locking variables* — intuitively, the locking bits of the entities. L transforms each transaction T_i in τ to $L(T_i)$ by inserting pairs of "lock X ... unlock X" steps, where $X \in LV$. The step "lock X" has the fixed interpretation "X := if X = 0 *then* 1 *else error*"; similarly for "unlock X". The set of all schedules of $L(\tau)$ is denoted by $L(S)$.

The set $L(S)$ of schedules is entrusted to a special scheduler M, called the *lock manager;* the output set of schedules from M is $M(L(S))$: formally, $M(L(S))$ is the set of all schedules in $L(S)$ that leave invariant the predicate

$$\bigwedge_{X \in LV} X=0 .$$

Now, if R is a set of schedules from $L(S)$, let $L^{-1}(R)$ be the same set with the "lock" — "unlock" steps removed. Then the class of schedules $L^{-1}(M(L(S)))$ abbreviated $O(L)$, is the *output set* of the locked transaction system $L(T)$ and is a measure of the parallelism supported by $L(T)$. $L(T)$ is called *safe* if $O(L) \subseteq SR$.

3 A CHARACTERIZATION OF LOCKING

How general output sets $O(L)$ can be produced by increasingly sophisticated lockings L? To study this, let us generalize locking to *d-locking*. In d-locking the locking variables may assume d values $0,1,...,d-1$, and the locked state is $d-1$. In other words, any proper subset of the d transactions may share a variable. Trivially, ordinary locks can be simulated by d-locks in that the same output set may be achieved (ignoring deadlocks for the time being).

For a prefix p of a schedule s, let steps(p) be the set of transaction steps involved in p. We call a class C of schedules *order oblivious* if whenever $s_1 s_2 \in C$, $s_3 s_4 \in C$, and steps(s_1) = steps(s_3) then also $s_1 s_4 \in C$. This condition states that once a history has executed several steps, it has "forgotten" the exact order, as far as membership in C is concerned.

Theorem 1. $C=O(L^d)$ for some d-locking L^d iff C is order-oblivious.

Corollary 1. If $C=O(L)$ for some (ordinary) locking L, then C is order-oblivious.

In fact, an exact characterization is immediate.

Corollary 2. $C=O(L)$ for some locking L iff all projections of C to pairs of transactions are order-oblivious.

4 THE GEOMETRY OF LOCKING

For the subsequent Sections we shall assume that all locked transaction systems are *well-formed,* in that

1) There is a natural isomorphism between E and LV via the mapping $x \leftrightarrow X$, $y \leftrightarrow Y$, etc. Then "lock X" will be abbreviated as Lx, and "unlock X" as Ux.
2) All variables are locked at most once in each transaction.
3) If T_{ij} is not a Lx or Ux step, then it is included in a Lx_{ij}-Ux_{ij} pair of steps.

We will assume also that

4) Any Lx-Ux pair of steps contains a step T_{ij} with $x_{ij}=x$.

These assumptions are made invariably throughout the literature on locking. By "transaction system" we shall henceforth understand "well-formed locked transaction system" with 4) satisfied.

Consider a transaction system $\tau = \{T_1, T_2\}$

A point p in the coordinated plane (Figure 1) represents a possible state of progress made towards executing T_1 and T_2. The lock-unlock instructions of the system τ have the effect of creating a forbidden region (possibly disconnected) which is the union of rectangular *blocks* (Figure 1). The region D is one of *deadlock,* whereas U is unreachable, yet not in any block. A schedule S corresponds to an *increasing curve* from O to F that avoids all blocks (still Figure 1). (For the sake of informality we will disregard the point that all such "curves" must be staircase). The two serial histories are the curves OT_1F and OT_2F. At this point we need a lemma:

Lemma Two schedules are equivalent iff they can be transformed to one another by a sequence of "switchings" of adjacent steps not involving the same entity.

A "switching" is shown in Figure 2. If such a switching is illegal in that $x_{1i} = x_{2j}$, then it cannot be performed because of a forbidden block. It turns out that schedule equivalence is the same with curve *homotopy;* two curves are homotopic if they can be transformed to one another by continuous transformations within the rectangle OT_1FT_2 avoiding all blocks. Hence we have (see Figure 3):

Theorem 2. τ is unsafe iff there exists an increasing block-avoiding curve from O to F that separates two blocks.

Let R be any region (possibly disconnected). Call two points (x_1,y_1) and (x_2,y_2) *incomparable* if $(x_1-x_2)(y_1-y_2)<0$ (Figure 4, points p and q). Then R is closed if, for any two connected incomparable points (x_1,y_1) and (x_2,y_2) in R, the points (x_1,y_2) and (x_2,y_1) are also in R. The *closure* of R is the (well-defined) smallest closed region containing R (Figure 5). We have

Theorem 3. τ is safe iff the closure of the forbidden region is connected.

Corollary 3. τ can be checked for safety in $O(n^3)$ time.

In fact, this can be done in $O(n \log n \log\log n)$ time [Li]. Notice how intuitive 2PL becomes now. 2PL says that all blocks must contain the point P whose projections P_1 and P_2 are the phase-shift points of the transactions T_1 and T_2 (see Figure 6). Thus the blocks are connected, and 2PL correct.

5 THE d>2 CASE

Consider now a set of $d>2$ locked transactions $\tau = \{T_1,...,T_d\}$, and define the graph $G(\tau) = (\tau, E)$,where $[T_i, T_j] \in E$ iff the transactions T_i and T_j have an entity in common. If the restriction of τ to any pair of transac-

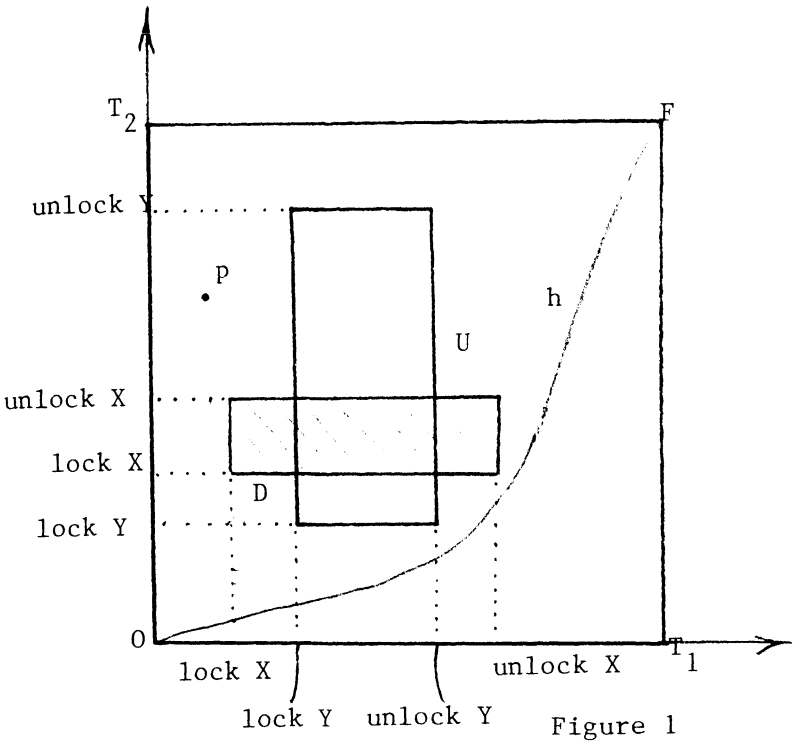


Figure 1

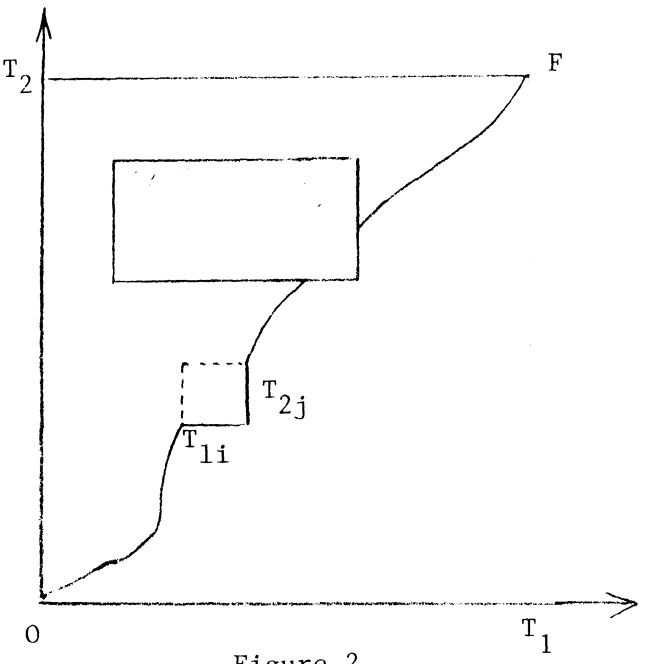


Figure 2

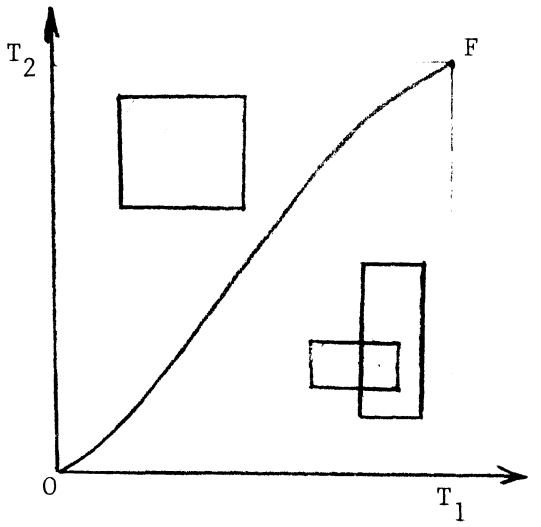


Figure 3

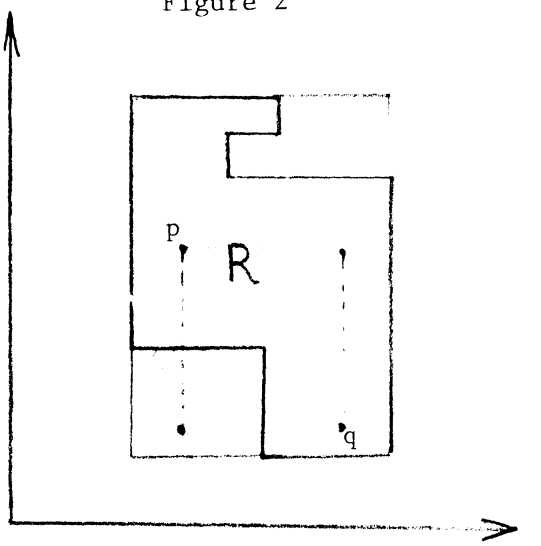
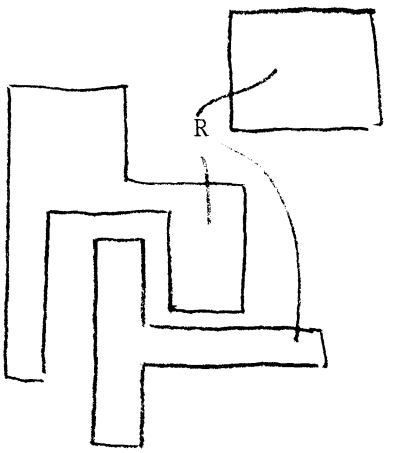


Figure 4



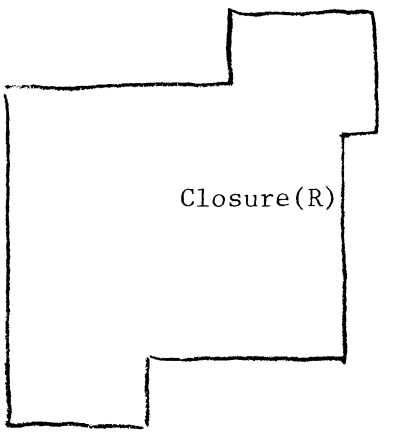


Figure 5

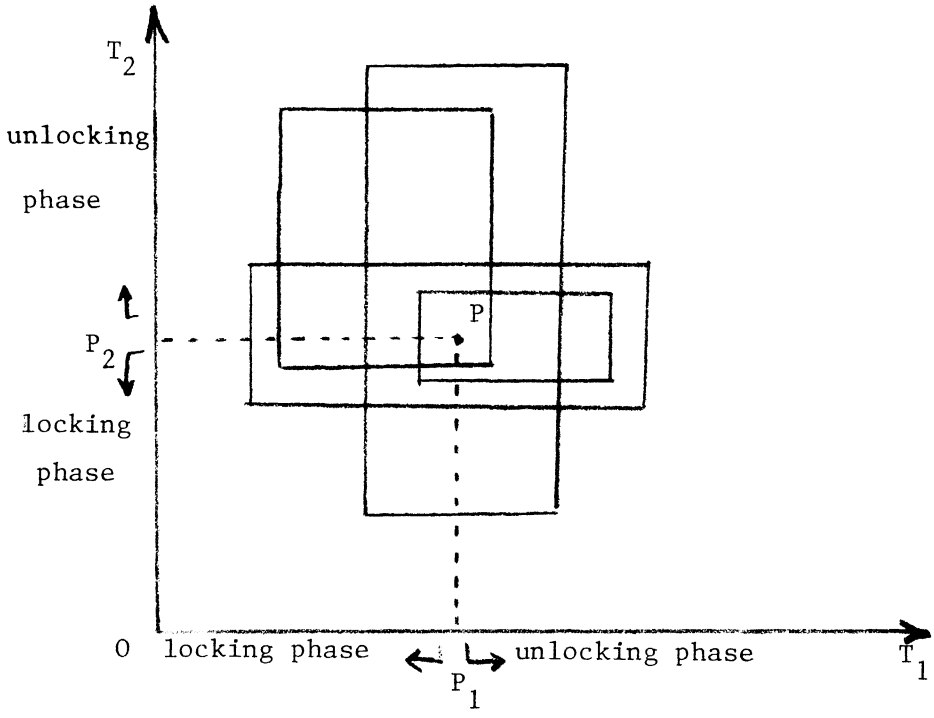
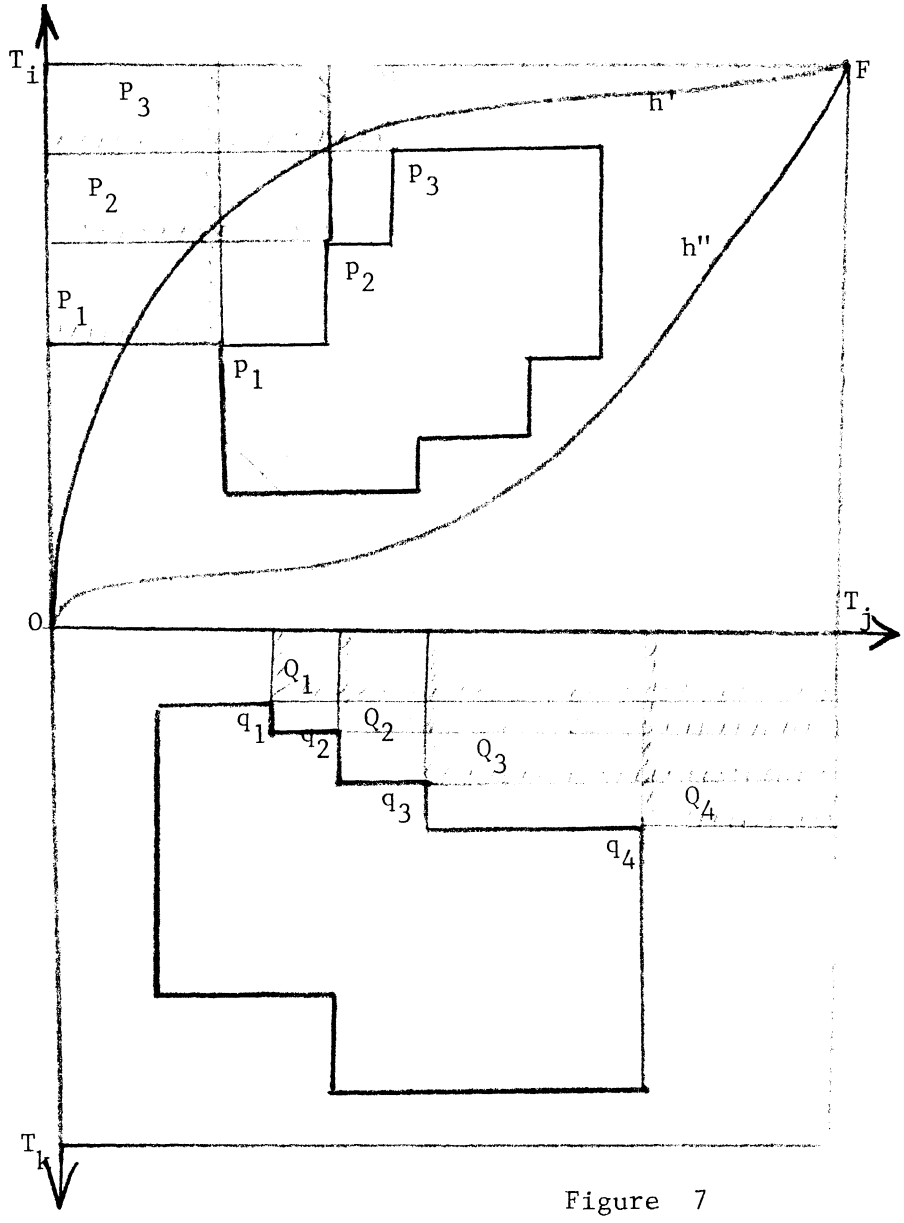


Figure 6



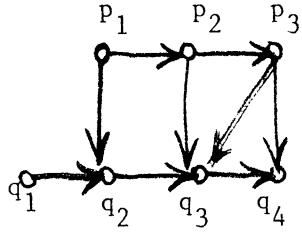


Figure 7

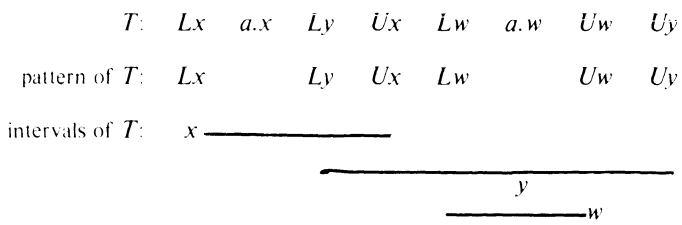


Figure 8

tions is incorrect (i.e., it violates Theorem 3), then the overall system is incorrect. So, let us assume that for all $[T_i,T_j] \in E$ the closure of the forbidden region in the T_i-T_j plane is either equivalent to OT_iF (s' in Figure 7) or to OT_jF (s'' in Figure 7). In the former case we write $T_i <_s T_j$; in the latter $T_j <_s T_i$.

Lemma s is serializable iff the $<_s$ relation is acyclic.

Therefore for τ to be safe, for each directed cycle that corresponds to an undirected cycle in $G(\tau)$ there must be a reason why no s exists that has this same cycle in the graph of $<_s$. Intuitively, the reason is that there is a contradiction in the order in which the curve of s intersects the prisms with bases marked $P_1,P_2,\ldots,Q_3,Q_4$ in Figure 6. This is captured as follows: with each pair $([T_i,T_j], [T_j,T_k])$ of edges in E we associate a digraph B_{ijk}. The vertices of this digraph are the vertices of the P_m and Q_n regions (see Figure 7). There is an arc from u to v iff (a) either u is a p_m or v is a q_n (or both), and (b) the T_j-coordinate of u is smaller than the T_j-coordinate of v. The construction is illustrated in Figure 7. Finally, if C is a directed cycle corresponding to a simple undirected cycle in $G(\tau)$, we let B_C be the union of all B_{ijk} digraphs for all consecutive triples (T_i,T_j,T_k) of C. The result is the following:

Theorem 4. τ is safe iff

(a) the restrictions of τ to all pairs of transactions are safe, and

(b) for all directed cycles C corresponding to undirected minimal cycles in $G(\tau)$ the digraph B_C has a cycle.

One can now derive extremely easy proofs of correctness of different locking policies, based on Theorem 4:

Corollary 4. Any transaction system obeying 2PL is safe.

Proof: That any transaction system obeying 2PL satisfies Theorem 3 is immediate. Condition (b) of Theorem 4 follows from the fact that in 2PL the graphs B_{ijk} are complete bipartite. □

Corollary 5. Any transaction system obeying *TP* is safe.

Proof: Since the common variables of any two transactions form a rooted tree in *TP*, condition (a) of Theorem 4 is trivial. Condition (b) also follows easily from the tree structure. □

Consider now the following special case in which any two transactions have at most one variable in common -- or, equivalently, the closure of the forbidden region on all planes is either empty or rectangular.

Corollary 6. Under the above assumptions τ is safe iff the restrictions of τ to every biconnected component of $G(\tau)$ obeys 2PL.

Another consequence of Theorem 4 is an algorithm for checking a transaction system for safety.

Corollary 7. Checking a transaction system τ for safety can be done in time polynomial in the number of minimal cycles of $G(\tau)$.

In general, of course, $G(\tau)$ will have an exponential number of minimal cycles, and thus Corollary 7 does not imply a genuine polynomial-time algorithm. In fact, such an algorithm is quite unlikely in view of the next result:

Theorem 5. Testing a transaction system for nonsafety is NP-complete.

Thus Corollary 6 suggests a polynomial-time algorithm for a special case of the NP-complete safety problem: the case in which any two transactions have at most one entity in common. It turns out that safety is NP-complete even if we restrict any two transactions to have at most *two* entities in common.

6 LOCKING POLICIES AND L-POLICIES

2PL is a *locking policy*. Intuitively it is a set of rules which govern but do not completely specify the transformation of any transaction system to a locked one. Further, these rules have the desirable property that they focus on each individual transaction, and do not regulate the *interaction* of any two or more transactions. A *locking policy* P is a mapping from the set of transactions on E to the power set of (well-formed) locked transactions on E, which satisfies the property: if $\bar{T} \in P(T)$ then T and $\bar{T}$ contain exactly the same actions in the same order. The locking policy P is safe if for any finite set $\tau = \{T_1,\ldots,T_m\}$ of transactions with $P(T_i) \neq \emptyset$, for all i, any set $\bar{\tau} = \{\bar{T}_1, \ldots, \bar{T}_m\}$ of locked transactions with $\bar{T}_i \in P(T_i)$ is safe. The *locking pattern* $p(T)$ of a locked transaction T is the subsequence of T formed by deleting its actions. It can be viewed as a set of intervals each one associated with an entity $x \in E$ which is locked in the intermediate steps (See Figure 8).

An interval I associated with entity x is an *action interval* if there is an action on x in I. It is easy to see that if τ is a safe locked transaction system and we rearrange in each transaction some actions within their action intervals in any way, then the resulting system τ' will also be safe. In other words, safety of a locked transaction system depends only on the locking patterns and the action intervals of its transactions.

Thus, we can view a locking policy P as a collection of locking patterns together with their action intervals. In

this paper we will consider a locking policy P as a collection of locking patterns allowed by P *without* a specification of which are the action intervals. In other words, we assume that if $\bar{T} \in P(T)$, for some transaction T, then there is another transaction T' which acts on all the entities locked by $\bar{T}$, and $\bar{T}' \in P(T')$ with $p(\bar{T}) = p(\bar{T}')$. The reason for this assumption is the following: One could define policies that use for locking, any set LV of special variables - not related to the set E of entities It is easy to see that any such policy P can be embedded to a locking policy P' which locks only entities (by expanding the set E of entities and the policy P in an appropriate way). We don't know if there are any non-artificial such policies. It is very easy however to construct many artificial ones, and our assumption serves to rule out such policies. (Note however that our complexity and sufficiency results carry over to the general setting.)

For simplicity we are going to assume that each pattern has at most one interval associated with each entity. It should be easy for the reader to modify the statements of the theorems, whenever necessary, in order to handle patterns with multiple intervals. We shall use the term *transaction* to refer both to a locked transaction T with an action on an entity x in the interval associated with x, and to the pattern of T. We will use the term *policy* P to refer (1) to a mapping from unlocked to locked transactions, as in the definition we gave, and (2) to a collection of transactions - i.e. locked transactions or their patterns - which form the image set of this mapping. If Δ is a class of structures on E (e.g. relations, graphs, etc.) a *structured policy* ΔP *operating* on Δ is a family of locking policies, one for each structure $D \in \Delta$.

Theorem 5 of the previous section, besides suggesting that testing safety is in general probably intractable, tells us also something about the limitations of "nice" locking policies.

Intuitively, a "nice" structured policy ΔP operating on a set of structures Δ should possess several properties, such as: (1) the set of structures Δ should be efficiently recognizable (e.g. trees, graphs, DAGs, etc.), (2) the policy should be operating in polynomial time; i.e. for each $D \in \Delta$, and for every unlocked transaction T, the mapping $DP(T)$ of the corresponding policy DP should be computable by a (nondeterministic) algorithm running in polynomial time in $|D|$ and $|E|$. Let us say that a structure D of Δ *covers* a set τ of transactions if DP (regarded as a set of transactions) includes τ. Theorem 5 then implies that unless $NP = co\text{-}NP$, no "nice" structured policy can cover all safe transaction systems in a succinct way, i.e. there will always be transaction systems τ that need a very large structure D of Δ (not bounded by any polynomial in $|\tau|$) to be covered. This means in particular that "nice" policies operating on simple structures, such as trees, graphs, etc., cannot possibly cover all safe transaction systems.

In this Section we will focus on a natural class of policies, one that includes all locking policies proposed thus far, and show that all policies in the class can be covered by a "nice" structured policy.

We say that a locking policy P is an *L-policy* if P can be described by a set of conditions that state whether a given entity can be locked at a certain moment in a transaction, depending on the portion of the transaction up to this moment. In other words, with each entity x there is associated a set $W(x)$ of prefixes of transactions; a transaction T is in P iff for each entity x referenced by T, the prefix of T to the left of Lx belongs to $W(x)$. For example, the two-phase locking policy has for every $x \in E$, $W(x) = \{\bar{T} | \bar{T}$ does not contain any unlock steps$\}$.

A *truncation* of a transaction T at the j-th step is a transaction T' that agrees with T in the first j steps, and then unlocks (in any order) the entities locked by T through the $j+1$-th step. The *closure under truncation* $Ct(\tau)$ of a transaction system τ is $Ct(\tau) = \tau \cup [\bigcup_{T \in \tau} Tr(T)]$, where $Tr(T)$ is the set of truncations of a transaction T. A system τ is *closed under truncation* if $Ct(\tau) = \tau$. Thus a policy P is an L-policy if (when viewed as a transaction system) it is closed under truncation.

We will show that if a system τ that is closed under truncation is not safe, then it has a particular nonserializable schedule; one in which all transactions of τ but one are executed serially. A *hypergraph* $H = (N,F)$ has a set of nodes N and a set of hyperedges F. Each hyperedge is a subset of N. With every transaction system τ we can associate a hypergraph $H(\tau)$, which has one node for each entity and a hyperedge $R(T)$ for each transaction T of τ. Let us denote by $L_T(i)$ the set of entities locked by T through step i.

Theorem 6 . A transaction system τ, that is closed under truncation, is safe if and only if for every $T \in \tau$, and x, y in $R(T)$ such that Ux occurs in T before Ly, the set $L_T(Ly)$ (or equivalently $L_T(Ux) - \{x\}$) separates x from y in $H(\tau)$.

Corollary 8. Given a transaction system τ, we can test in polynomial time if its closure under truncation is safe.

We will now define a policy and show as an example how the criterion of Theorem 6 can be used to show its safety.

DAG policy(DP): The entities are arranged on (correspond to the nodes of) a single source directed acyclic graph (DAG) D.

The rules of the policy are as follows:

(1) First lock is arbitrary,

(2) Subsequently, an entity x can be locked only if (a) all its fathers (immediate predecessors) have been mentioned in the transaction up to this point, and (b) at least one father is currently locked.

(3) Each entity is locked at most once.*

Formally, a transaction T with at most one interval associated with each entity, is in DP iff $x \in R(T) \Rightarrow [R_T(Lx) = \emptyset]$ or $[F(x) \subseteq R_T(Lx)$ and $F(x) \cap L_T(Lx) \neq \emptyset]$, where $F(x)$ is the set of fathers of x in D.

Thus, if the underlying DAG D is a rooted tree, DP becomes the tree policy.

* We mention this here explicitly because it is an essential part of the rules, in order to guarantee safety of DP.

If T is any transaction following DP, and z the first entity locked by T, then an easy induction can show that,
(i) z is an ancestor of $R(T)$; in fact z *dominates* all elements of $R(T)$, i.e. any path from the source of D to an element x of $R(T)$ has to pass through z (See [AHU] p.210), and
(ii) for each x in $R(T)$, all nodes that are ancestors of x and descendants of z are in $R_T(Lx)$.

Let us prove now that DP is a safe policy. Suppose it is not. Then there is a transaction T and entities x and y of it such that Ux occurs before Ly in T, and $L_T(Ly)$ does not separate x from y in H(DP). Let $R(T_1),\ldots, R(T_k)$ be an x-y path that avoids $L_T(Ly)$ with $x \in T_1$ and $y \in T_k$. Since y is not the first entity locked by T, $F(y) \cap L_T(Ly) \neq \varnothing$. Since $L_T(Ly) \cap R_{T_k}(Uy) = \varnothing$, we have $F(y) \not\subseteq R_{T_k}(Ly)$, and therefore y must be the first entity locked by T_k. Let x_{k-1} be an element of $R(T_{k-1}) \cap R(T_k)$ such that $F(x_{k-1}) \cap R(T_{k-1}) \cap R(T_k) = \varnothing$. Since x_{k-1} is not the first element locked by T_k we have $F(x_{k-1}) \subseteq R(T_k)$, $F(x_{k-1}) \cap R(T_{k-1}) = \varnothing$, and therefore x_{k-1} must be the first element locked by T_{k-1}. Thus by property (i) of DP, y is an ancestor of $R(T_k) \cup R(T_{k-1})$. Proceeding similarly we can deduce that y is an ancestor of $\bigcup_{j=1}^{k} R(T_j)$, and consequently of x. But then by property (ii) of DP, $y \in R_T(Lx)$ contradicting the rule that each entity is locked at most once.

A *directed hypergraph* $DH = (N,F)$ is a hypergraph, each hyperedge A of which has a node specified as its *head*. The rest of the nodes of A form its *tail*. The underlying hypergraph of DH is simply $H = (N,F)$ without head-tail specification. When we'll talk about "paths," "cycles," "separators," etc. in a directed hypergraph we are referring to the underlying hypergraphs.

Hypergraph policy (HP): The entities are arranged in a directed hypergraph H. The rules are:
(1) First lock arbitrary,
Subsequently, an entity x can be locked iff
(2) There is a hyperedge A of H with head x, whose tail has been mentioned in the transaction up to this point, and
(3) For each y previously unlocked, the set of entities that are currently locked separate x from y.

Corollary 9. An L-policy is safe if and only if it is covered by the hypergraph policy for some directed hypergraph H.

Proof. Both directions follow trivially from Theorem 6. However let us construct from an L-policy P a directed hypergraph without redundant hyperedges. For each transaction T, let $l(T)$ be the entity locked last by T. H has a hyperedge $R(T)$ with head $l(T)$ if there is no transaction T' of P such that $R(T') \subseteq R(T)$ and $l(T) \in R(T')$. Clearly HP on this hypergraph H covers P.□

Examples

1. *Two-phase policy.* The hypergraph H is a complete symmetric digraph (a clique). In order that a hypergraph H be nontrivial , i.e. have $HP(T) \neq \varnothing$ for at least one T which acts upon more than one entities, H must have at least one arc (hyperedge of cardinality 2). Then the clique is the only hypergraph that has all possible automorphisms (i.e. 2PL is the only nontrivial safe L-policy that treats all entities uniformly).
2. *Tree policy.* The hypergraph constructed above is the underlying tree.
3. *DAG policy.* The dominator of a node x is the (unique) lowest ancestor of x that dominates x. The hypergraph of *DP* contains for each node x a hyperedge A_x with head x. The tail of A_x is the set of all ancestors of x that are descendants of its dominator. Although a DAG can be more easily visualized than a hypergraph, the hypergraph makes explicit restrictions that are "hidden" in the rules of the policy: if we want to act in a transaction T upon entity x then we have to lock beforehand all of A_x, unless the rest of entities acted upon by T are all dominated by x. Also the hypergraph shows that rule 2(b) of DP is stricter than necessary: it could be replaced by 2(b') at least one ancestor of x but not of its dominator is currently locked. □

Since a general hypergraph can have an exponential number of hyperedges, the question that arises is whether there is a more succinct representation of L-policies. We will show that there are "too many" distinct (incomparable) L-policies, and therefore this is not the case. We will call two safe policies P_1,P_2 *distinct* if there is no safe policy P_3 that includes both of them. A safe L-policy is *maximal in* L if there is no other safe L-policy that includes it. For example, an L-policy whose hypergraph contains an arc (x,y) but not the symmetric arc (y,x) is not maximal in L. On the other hand HP operating on a symmetric digraph (an undirected graph) can be shown to be maximal in L. Maximality of a policy P in L does not imply that P is maximally safe; in fact it is easy to see that 2PL is the only maximally safe L-policy.

Lemma. If $P_1 \neq P_2$ are two safe L-policies, maximal in L, then P_1 and P_2 are distinct.

Corollary 10. There is a doubly exponential number (in $|E|$) of (mutually) distinct safe L-policies.

The results of this section can help also answer questions such as the one examined in [KS]: If ΔP is a particular L-policy operating on a set of structures Δ, find the set of structures $\Delta' \subseteq \Delta$ for which ΔP is safe. Since ΔP is an L-policy, we can construct as in the proof of Corollary 9 a set H of hypergraphs, one for each structure of Δ. The problem then reduces to finding the set of structures Δ' for which rule (3) of HP is enforced by ΔP. For example, in [KS] Δ is the set of directed graphs, and ΔP is the policy with rules
(1) First lock arbitrary,
(2) Subsequently entity x can be locked if there is a father y of x that is currently locked.

It was shown there that ΔP is safe if and only if the underlying graph of the digraph is a tree. Let us see how this result can be derived from Corollary 9.
The hypergraph H that corresponds to a digraph D is clearly D itself. For a digraph D, rule (2) of ΔP implies rule (3) of HP if and only if for every $x \in E$, for every

possible transaction T of ΔP with $x \notin R(T)$, and for every $y \in R(T)$ such that $y \rightarrow x$, y separates $R(T)$ from x (again "separates" refers to the underlying graph G of D), since by the rules of ΔP we can unlock all of $R(T)$but y, and then lock x (and this is the worst that can happen). Since any pair of adjacent (in G) nodes can be $R(T)$ for some $T \in \Delta P$, for every arc $y \rightarrow x$, node y must separate x from all nodes adjacent in G to y; i.e. every edge of G is a bridge and G is a tree. Conversely, if y separates x from all the nodes adjacent to y, then it separates x also from all the nodes of $R(T)$ with $x \notin R(T)$, $y \in R(T)$, since the subgraph of G induced by $R(T)$ is connected.

7 FREEDOM FROM DEADLOCK

A *partial schedule* s of a transaction system $\tau = \{T_1,...,T_m\}$ is a legal schedule of any prefices of the transactions of τ. The *state* $J(s)$ of a partial schedule s is the vector $<j_1,...,j_m>$ that describes the next step to be executed for each transaction of τ. The state J is a *deadlock state* if for all i the j_i-th step of every unfinished transaction T_i is Lx_i for some entity x_i locked at J. A transaction system τ - and the associated policy P - is *deadlock-free* if the state $J(s)$ of any partial schedule s of τ is not a deadlock state. In other words any partial schedule of τ can be extended to (is a prefix of) a (complete) schedule of τ.

From a partial schedule s of τ we can construct, as with a complete schedule, a directed graph $D(s)$ by corresponding a node v_i to each transaction T_i, and having an arc (v_i, v_j) labelled x, if T_i locks x in s before T_j does (even if the Lx step of T_j has not been executed yet in s). Then τ is safe and deadlock-free iff for every partial schedule s of τ, the digraph $D(s)$ is acyclic.

Let us consider a deadlock state J. There is a set of transactions $\{T_1,...,T_k\}$ such that the next step of T_i is Lx_i where x_i is currently locked by T_{i+1} ($x_i \in L_{T_{i+1}}(Lx_{i-1})$) - See Figure 9.

T_1: . . . Lx_k . . . Lx_1 . . . Ux_k .
T_2: . . . Lx_1 . . . Lx_2 . . . Ux_1 .
.
.
.
T_k: . . . Lx_{k-1} . . . Lx_k . . . Ux_{k-1} .

Figure 9

Thus in the partial schedule S, transaction T_i accesses x_{i-1} before T_{i-1}; if S could possibly finish in any way then the resulting schedule would not be correct. In other words, deadlocks prevent some wrong schedules from finishing. Let us show that the DAG policy is deadlock-free using this fact.

Theorem 7. The DAG policy is deadlock-free.

Proof. Suppose S is a partial schedule, of $T_1,...,T_k$ deadlocked at state J where the next step of each T_i is Lx_i as in Figure 9. We can assume without loss of generality that Lx_i is the last locking step of T_i. Suppose that for some i, the transaction T_i' obtained from T_i by moving the Ux_{i-1} step right before the Lx_i step is also a transaction of the DAG policy. Then the partial schedule S could be extended to a nonserializable schedule of the system $[\bigcup_{j \neq i} \{T_j\}] \cup \{T_i'\}$, contradicting the safety of the DAG policy. Therefore, for each i, x_{i-1} must be a father of x_i, which implies that the DAG has a cycle $\{x_1,...,x_k\}$. □

If we modified the DAG policy by changing the locking rule (2) into: (2') an entity x can be locked if at least one father of x is currently locked, and all fathers of x are mentioned until the Ux step, then the modified policy (which is not an L-policy any more) is safe. However, it is easy to see that it is not any more deadlock-free: postponing the locking of some father of x allows a (partial) schedule to start wrongly, and then be stopped later on by a deadlock.

Testing for freedom from deadlock

We will now characterize safety and freedom from deadlock for a pair of transactions. If $F \subseteq E$ is a set of entities, the *restriction of T on F*, is the sequence of steps of T that involve entities from F.

Theorem 8. $\tau = \{T_1, T_2\}$ is a safe and deadlock-free pair of transactions if and only if the restrictions T_1', T_2', of T_1 and T_2 on their common entities $(R(T_1) \cap R(T_2))$ follow the DAG policy for some DAG D on $R(T_1) \cap R(T_2)$. [Note that entities referenced only by one of the transactions do not affect safety or freedom from deadlock.]

It is not hard to see that the tree policy does not suffice to cover all safe and deadlock-free pairs of transactions.

Unfortunately freedom from deadlock cannot be tested efficiently (and characterized) in general even for L-policies :

Theorem 9. It is NP-complete to decide whether a set of two phase transactions is not deadlock-free.

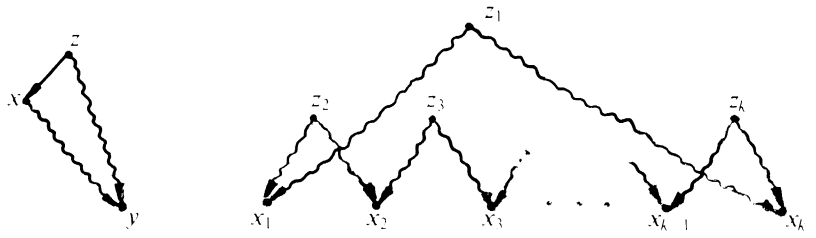
In Section 5 we showed that safety can be tested in time polynomial in the number of minimal cycles. The direct analogue of this result for freedom from deadlock does not hold however. The reason for this difference is the fact that if a transaction system is not safe then this is due to some chordless cycle, whereas deadlock may be possible due to some cycle with chords, even though all chordless cycles are deadlock-free. Thus the correct analogue of Theorem 4 and Corollary 7 is:

Theorem 10. Freedom from deadlock of a safe transaction system can be decided in time polynomial in the number of cycles.

Deadlock-free L-policies

Theorem 11. If τ is a safe transaction system that is closed under truncation, then τ is deadlock-free if and only if there do not exist transactions $T_1,...,T_k$, and entities $x_1,...,x_k$, where $x_i \in R(T_i) \cap R(T_{i-1}) - [\bigcup_{j \neq i, i+1} R(T_j)]$ and $R_{T_i}(Lx_i) \cap [\bigcup_{j \neq i} R(T_j)] = \emptyset$.

From Theorem 11 it follows that whether a safe L-policy is deadlock-free or not depends only on the order



(a) (b)

Figure 11

in which entities get locked, and not on how the unlock steps are placed within the transactions. More formally we have:

Corollary 11. Suppose that $\tau = \{T_1,...,T_m\}$ and $\tau' = \{T_1',...,T_m'\}$ are two safe transaction systems that are closed under truncation, and such that for every i, there is a j, where T_i' locks the same entities as T_i in the same order. Then, if τ' is deadlock-free, then so is τ.

Note that Corollary 11 is not true for general policies (systems that are not closed under truncation). A consequence of Corollary 11 is that if H is the underlying directed hypergraph of a safe L-policy which uses the full freedom of rule (2) of HP, then whether P is deadlock-free or not depends only on H and not on how rule (3) of HP is enforced (how safety is ensured in P). Thus the following two problems are suggested by this fact : (1) characterize those directed hypergraphs H for which HP is deadlock-free, and (2) find the "correct" restriction of rule (2) of HP for freedom from deadlock - "correct" in the sense that it describes all deadlock-free L-policies (in the same way that rule (3) of HP is the "correct" rule for safety). Theorem 9 (the proof of it rather) implies that there is probably no solution to these problems that can be efficiently tested. In the remainder of this Section we will give a partial answer to these problems, and show how Corollary 11 can be used to prove the freedom from deadlock of L-policies.

With every transaction system τ we can associate a directed graph $D(\tau)$ as follows: the nodes of $D(\tau)$ are the entities, and there is an arc (x,y) if there is a transaction T of τ that starts by locking x and references y. Suppose that τ is safe and deadlock-free. Then, it is easy to see that

(a) *$D(\tau)$ is acyclic.*

(b) *If x is an ancestor of y, then in all transactions that contain both x and y, x gets locked before y.*

Let $F_1(x)$ be the set of fathers y of x, for which there is a transaction T that starts with y, contains x, and there is no ancestor z of x in $R_T(Lx)-y$. Denote by $D'(\tau)$ the subgraph of $D(\tau)$, where the arc (y,x) is in $D'(\tau)$ if there is a transaction T starting with y, containing x, and such that $F_1(x) \cap [R_T(Lx)-y] = \varnothing$. ($D'(\tau)$ has the same transitive closure as $D(\tau)$, but is not necessarily its transitive reduction.) Clearly every transaction T of τ has the property:

(c) *If $x \in R(T)$ is not the first entity locked by T, then at least some father y of x in $D'(\tau)$ is referenced by T before Lx.*

Consequently every transaction T references a connected subgraph of $D'(\tau)$ which can be reached from the first entity locked by T. For example, if τ is the tree policy, then $D'(\tau)$ is the tree itself; in general if τ is the hypergraph policy on a (directed) graph G, then $D'(\tau)$ is the graph G. If τ is the DAG policy, then $D'(\tau)$ is the dominator tree of the DAG. In general, if P is an L-policy with $D'(P)$ a tree (not necessarily rooted), then P is deadlock-free (by Corollary 11 and the freedom from deadlock of the tree policy). This is not the case however for general policies. For example, if $\tau = \{T_1,T_2\}$,where
T_1 : LA LC UA LB UB UC
T_2 : LA LB UA LC UB UC
then $D'(\tau)$ is the tree of Figure 10, τ is safe (but its closure under truncation $Ct(\tau)$ is not), and τ is not deadlock-free.

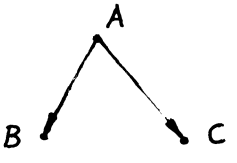


Figure 10

This is not a coincidence: if any such τ is deadlock-free, then it can be extended to a safe (and of course still deadlock-free) L-policy.

Theorem 12. Suppose that τ is a safe and deadlock-free transaction system with $D'(\tau)$ a tree. Then $Ct(\tau)$ is also safe and deadlock-free.

Suppose now that τ is an L-policy . Deadlocks may arise because of undirected cycles in $D'(\tau)$ (cycles in the underlying undirected graph of $D'(\tau)$)
In a general $D'(\tau)$ we can distinguish between two kinds of cycles - See Figure 11(a),(b). A cycle as in Figure 11(a) may give rise to a pair of transactions $\{T_1,T_2\}$, where T_1 starts from x and follows the path to y, and T_2 starts from z, follows the path to y, and then goes on to lock x; thus, T_2 does not satisfy condition (b). (Note that if τ is the hypergraph policy with the graph $D'(\tau)$ as

the underlying hypergraph, then both transactions are allowed.) In this case we have:

Theorem 13. Let P be a safe L-policy whose digraph $D'(P)$ is acyclic and contains no (undirected) cycles as in Figure 11(b). Then P is deadlock-free if and only if it satisfies condition (b).

Note that if D' is a tree, then (b) is satisfied automatically. Also, note that if a cycle as in Figure 11(b) exists in D', and P is the hypergraph policy operating on D', with condition (b) checked in addition, such a cycle gives rise to a deadlock. Deadlock from such a cycle can be avoided either if we lock nodes according to some specified (acyclic) order or prevent the x_i's from being the first common entities of the corresponding transactions T_i, by forcing the transactions to start locking higher in the DAG D'. For example the following rule guarantees freedom from deadlock (assuming that (b) is enforced): if $x \in R_T(Ly)$ and x is not an ancestor of y, then $R_T(Ly)$ and the descendants of x separate x and y in the underlying graph of D'. Thus, for example, the DAG policy enforces this rule by requiring all fathers of x to be locked before x (which results in $D'(DP)$ being a tree rather than the original DAG). Note however that the previous rule (or any other simple rule) is not necessary, because of our NP-completeness result of the previous Section.

8 DISCUSSION

In this paper we examined locking as a concurrency control mechanism in database systems. In Section 3 we characterized the class of schedules that can be produced if we use locking. Corollary 1 is the price in parallelism that we have to pay for the conceptual simplicity of locking-based schedulers. It is a dear price. All sophisticated serializability techniques introduced in [Pa] involve some notion of "remembering" which transaction read data first from which, and therefore they cannot be implemented by order-oblivious primitives such as locking. In contrast, *all* subsets C of the set of all schedules S can in principle be the output sets of some scheduler. In fact, it is shown in [Pa] that there is a polynomial-time scheduler A such that $A(S)=C$ iff the set of prefixes of C is in P. It remains to be seen how well locking can be complemented as a concurrency control technique by other, order-conscious primitives such as queues. The *SDD*-1 system [BGRP] is an instance of this.

In Sections 4 and 5 we characterized safety of locked transaction systems , and showed that testing for safety is an NP-hard task.

In Sections 6 and 7 we analyzed locking as a mechanism for preserving consistency in a database that has a given structure. We showed that safe locking policies that fall into a "natural" class - the class of those policies that can be stated in terms of conditions that describe when each entity can be locked - can be described by a certain policy operating on hypergraphs. This policy must visit entities along the paths of a hypergraph H. In order to avoid unnecessary extra locking, the hypergraph must be chosen to resemble the structure of the consistency constraints; i.e. the hyperedges should correspond to the consistency constraints (Recall that a transaction can be viewed as the unit of consistency - the set of actions needed to rectify temporary inconsistencies.) Rule (3) of the hypergraph policy then (see Section 6) describes when an entity can be unlocked before the end of the transaction, in order to guarantee safety. The higher the connectivity of H is, the less early unlocking is allowed (but possibly less extra locking might be needed). Thus, if H is a graph, the two extreme cases are H = complete (2PL), and H = tree (tree policy). How faithfully H should represent the consistency constraints depends on the information available about the particular application; for example, if entities x and y are connected by some constraint C which is rarely violated, i.e. most updates on x and y do not affect C, then it might be advantageous not to represent C in H in order to achieve a higher degree of concurrency, at the expense of doing (rarely) some extra locking.

In our model we viewed a locking policy as an algorithm that takes a transaction - a sequence of actions - and sets locks. It might be the case however that the whole transaction is not known at the beginning, but is found out dynamically, i.e. the result of an action determines subsequent actions. The conditions we gave still hold if we look at the locked transactions that are produced. In the case of the hypergraph policy, the transaction must start from an entity that can reach all entities that might be needed, and can unlock some entity x only if the set of locked entities at this point separate x from any other entity y that might be needed later on (in order to ensure rule (3)).

Another choice involved in the design of such a policy, is how rule (3) is enforced; that is one might choose not to use the full freedom of it in order to get a more efficient policy (at the expense of a loss in concurrency). For example, in the case of the DAG policy, we could just require that when x is locked, then an ancestor of x but not of its dominator be locked, instead of requiring a father of x to be locked. This policy is also safe and deadlock-free (by Theorem 6 and Corollary 11). In general, Corollary 11 implies that the way rule (3) of HP is enforced does not affect the freedom from deadlock of the policy.

In this paper we did not distinguish between read- and write-actions. However probably our results generalize to this case (where "serializability" is as in [EGLT1]) in the same way that 2PL is generalized in [GLPT] and [LW].

Note.

This is the merging of results obtained independently by the first author (MY) on the one hand, and the second and third authors (CHP and HTK) on the other. Section 3 contains results by CHP and HTK, whereas Sections 6 and 7 contain results due to MY. Sections 4 and 5 contain essentially common, yet independently obtained results; the exposition of these Sections follows CHP and HTK.

REFERENCES

[AHU] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, 1974.

[BGRP] P. A. Bernstein, M. Goodman, J. B. Rothnie, and C. H. Papadimitriou, "Analysis of Serializability of SSD-1: A System of Distributed Databases (The Fully Redundant-Case)," IEEE Trans. Soft. Eng. SE-4(3), 154-168, (1978).

[CD] E. G. Coffman Jr., and P. J. Denning, Operating Systems Theory, Prentice-Hall, 1973.

[CES] E. G. Coffman Jr., M. J. Elphick, and A. Shoshani, "Systems Deadlock," Computing Surveys 3(2), 67-68, (1971).

[EGLT1] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", CACM 19(11), 624-633, (1976).

[EGLT2] _____, "On the Notions of Consistency and Predicate Locks", IBM Research Report RJ 1487, (1974).

[GJ] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, 1978.

[GLPT] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base", IBM Research Report RJ 1654, (1975).

[Ka] R. M. Karp, "Reducibility among Combinatorial Problems," in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher (eds.), Plenum, 85-103, 1972.

[KM] R. M. Karp, and R. E. Miller, "Properties of a model for parallel computations: determinacy, termination and queuing," SIAM J. Appl. Math. 14(6), 1390-1410, (1966).

[KS] Z. Kedem, and A. Silberschatz, "A characterization of Database Graphs admitting a simple Locking Protocol," Technical report 49, University of Texas at Dallas, (1979).

[KP] H. T. Kung and C. H. Papadimitriou, "An Optimality Theory of Concurrency Control for Databases", ACM-SIGMOD Conference (1979).

[LW] Y. E. Lien and P. J. Weinberger, "Consistency, Concurrency, and Crash Recovery", ACM-SIGMOD Conference (1978).

[Li] W. Lipski Jr., private communication.

[Pa] C. H. Papadimitriou, "Serializability of Concurrent Updates", Technical report, Harvard University, (1978).

[PBR] C. H. Papadimitriou, P. A. Bernstein, and J. B. Rothnie, "Computational Problems related to Database Concurrency Control", Conf. on Theoretical Computer Science, University of Waterloo, 275-282, (1977).

[SK] A. Silberschatz and Z. Kedem, "Consistency in Hierarchical Database Systems", to appear in JACM, (1978).

[SLR] R. E. Stearns, P. M. Lewis, and D. J. Rosenkrantz, "Concurrency Control for Database Systems", Proc. of the 17th Annual Symp. on Foundations of Computer Science, 19-32, (1976).