SIAM J. COMPUT.
Vol. 14, No. 1, February 1985

004

THE COMPLEXITY OF DISTRIBUTED CONCURRENCY CONTROL*

PARIS C. KANELLAKIS† AND CHRISTOS H. PAPADIMITRIOU‡

Abstract. We present a formal framework for distributed databases, and we study the complexity of the concurrency control problem in this framework. Our transactions are partially ordered sets of actions, as opposed to the straight-line programs of the centralized case. The concurrency control algorithm, or scheduler, is itself a distributed program. Three notions of performance of the scheduler are studied and interrelated: (1) its parallelism, (2) the computational complexity of the problems it needs to solve and (3) the cost of communication between the various parts of the scheduler. We show that the number of messages necessary and sufficient to support a given level of parallelism is equal to the minimax value of a combinatorial game. We show that this game is PSPACE-complete. It follows that, unless NP = PSPACE, a scheduler cannot simultaneously minimize communication and be computationally efficient. This result, we argue, captures the quantum jump in complexity of the transition from centralized to distributed concurrency control problems.

Key words. distributed database, concurrency control, games, complexity, PSPACE-complete

1. Introduction. There is now considerable literature, both theoretical and applied, concerning the *database concurrency control problem*—that is, maintaining the integrity of a database in the face of concurrent updates. Most of the theoretical work so far has been concerned with the *centralized* problem, in which the database resides at one site, and the update requests are submitted to a single process, called the *scheduler*, which implements the concurrency control policy of the database [4], [8], [11], [15], [17], [18]. There is also some interesting applied work on *distributed* databases [1], [2], [13], [16]. It is often said that the concurrency control problem is much trickier and harder in the distributed case than in the centralized case. This is evidenced by the existing solutions, which are extremely complex and sometimes incorrect.

In this paper we present a model of distributed databases, which captures the intricacies of distributed computation that are most pertinent to the database domain. Some novelties of our model are:

(a) *Transactions* are *partial orders* of atomic steps, thus generalizing the straight-line programs of the centralized case [8]. The partial order corresponds to both time-precedence and information flow, and it captures the notion of *distributed time* [10].

(b) The *scheduler*, the concurrency control agent of the system, is itself a *distributed program*, consisting of communicating sequential processes [6], one for each site.

(c) *Redundancy* (the requirement that two entities stored at different sites be copies of the same "virtual entity") is not treated at the syntactic level, but is considered as part of the integrity constraints of the database. Redundancy was at the root of the complexities of most previous attempts to formalize distributed databases.

* Received by the editors October 28, 1983. This research was supported by the National Science Foundation under grants ECS-79-19880 and MCS-79-08965.

† Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. Present address, Department of Computer Science, Brown University, Providence, Rhode Island 02912.

‡ Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, and National Technical University of Athens, Athens, Greece.

As a consequence, there are *three measures of performance* in a distributed database (centralized theory deals with the first two):

(1) *parallelism*, measured by the set of allowable interleavings of atomic steps,

(2) *complexity* of the computational problems that the scheduler must solve,

(3) *communication*, measured as number of messages exchanged by scheduler processes.

There are some interesting tradeoffs here. For example, let us fix (1) (think of it as the parallelism specifications of the system). By expending many messages, we can reduce the problem of distributed concurrency control to the centralized one (by broadcasting each request) and thus solve it in polynomial time for most reasonable parallelism specifications [11]. It turns out that, based on a priori information about transactions, we can minimize the number of messages sent in exponential time (and polynomial space; this is the upper bound of our main result). Finally we cannot have a scheduler simultaneously using the minimum number of messages and running in polynomial time at each site, unless NP = PSPACE (this follows from the lower bound).

Specifically our main result states that: *for a certain parallelism specification, which in fact can be fixed to be the popular serializability principle, minimizing communication costs is a computational problem complete for* PSPACE [3], [5], [14]. Thus, our result appears to be concrete mathematical evidence suggesting that distributed concurrency control is indeed an inherently more complex problem than centralized concurrency control (under quite general conditions, centralized schedulers can be implemented in polynomial time and always in nondeterministic polynomial time [11], [15], [17], [18]).

Our result also adds to the literature on *distributed computation*, independently of its database context. It states, loosely speaking, that one cannot tell efficiently whether distributed processes can cooperate successfully for performing an (otherwise easy) *on-line* computational task, at fixed communication cost. It can therefore be considered as complementing the result of Ladner for lockout properties of "antagonistic" processes [9]. On the other hand, A. Yao has asked [19] whether minimizing communication costs for some distributed combinatorial computation is computationally intractable; NP-complete for the *off-line* problem. We answer this question for *on-line* computation. Yao's original conjecture was recently answered in the affirmative [12].

We provide both upper and lower bounds. For the upper bound, we need a characterization (Theorem 3) of the incomplete executions of transactions that can be completed within a fixed number of messages. This upper bound holds for most parallelism specifications that can be achieved efficiently in a centralized manner. For the lower bound we relate distributed scheduling to a game played on graphs (the conflict graph of the transactions). Intuitively one player (Player I) is an adversary who submits update requests so as to force the scheduler to use as many messages as possible, whereas the other player (Player II) is the distributed scheduler. Player I wants to prolong the game as much as possible, whereas Player II tries to bring it to an end as soon as possible; other than that there is no winner or loser. The rules are related in a simple way to the cycles of the graph. The minimax length of the game corresponds to the optimal communication cost. We prove that this game is complete for PSPACE, and then show that our constructs can faithfully reflect a special kind of distributed concurrency control situation. This new kind of game may be of independent interest.

Section 2 describes the model used, § 3 the upper bound and the game on graphs, and finally § 4 has the PSPACE-completeness reduction (Theorem 4) and its implications.

2. A model of distributed database concurrency control.

2.1. Distributed database. A distributed database is a collection of sites. Each site has its own processor and data. The sites are interconnected by a network and are controlled by a distributed database management system (DDBMS). In Fig. 1 we show the architecture of a two-site system; distributed programs on this system consist of communicating sequential processes [6], one for each site, (horizontal arrows join parts of the same distributed program). Formally, a distributed database is defined as follows:

DEFINITION 1. A *distributed database* (DD) is a triple $\langle G, D, \textit{stored-at}\rangle$ where:

(a) $G=(U, L)$ is an undirected graph, where every node corresponds to a *site* and every link to a two-way communication link between sites.

(b) D is a set of *entities*, denoted $\{x, y, z, \cdots\}$.

(c) *stored-at*: $D \rightarrow U$ is a function determining the site, where each entity is stored.

The entities are the *physical* data items. Multiple copies of the same *logical* data item are considered as different physical data items stored at different sites. The fact that they are copies and must remain identical for reasons of consistency is part of the *integrity constraints* [1], and is not treated separately. We assume that the DD is fixed and given.

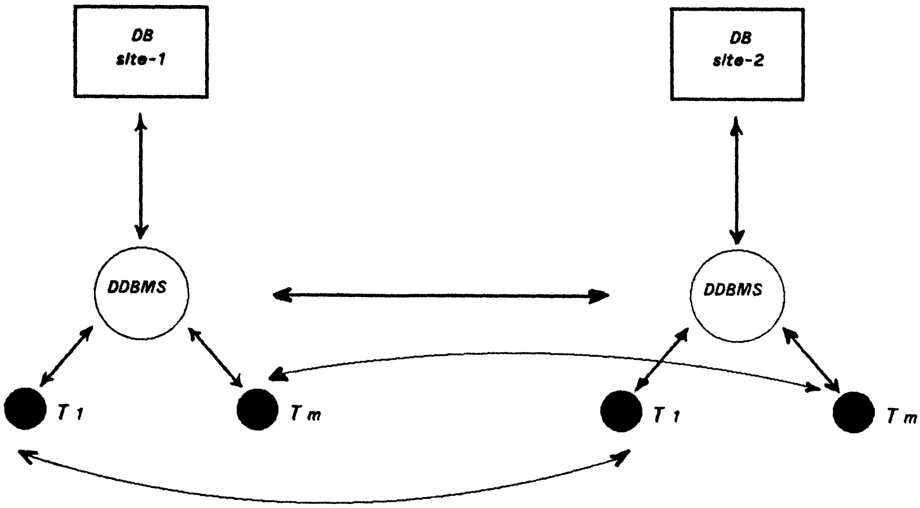

FIG. 1. *Architecture of a two-site system.*

2.2. Transactions and schedules. The users interact with the database using transactions. In our model a transaction is a distributed program, not identified with a particular site.

DEFINITION 2. A *transaction* T, in a DD, is a directed acyclic graph (dag) $T=(N, A)$ such that:

(a) Every node p is associated with one of the sites of the system, $\textit{site}(p)\in U$ and with an entity x_p for which $\textit{stored-at}(x_p)=\textit{site}(p)$.

(b) Nodes associated with the same site are totally ordered in A, (we denote the partial order imposed by T on its nodes as $\geqq_T$). A *transaction system* $\boldsymbol{T}$ is a set of transactions $\{T_i, 1\leqq i\leqq m\}$.

An example is shown in Fig. 2. Nodes are also called *actions*, since they are intended to represent update actions on the corresponding entity. An action p represents an indivisible read and write operation on x_p [8] (we do not distinguish between read and write operations as in [11]). Action p, as in [8], depends only on actions preceding it in its transaction. Each transaction T represents a distributed program, consisting of communicating sequential processes [6], one per site. Let the t_i's be variables local to this program, and the f_i's be uninterpreted function symbols, then the semantics of action p of transaction T is the indivisible execution of the two instructions: $t_p := x_p$; $x_p := f_p(t_p, t_{p_1}, t_{p_2}, \cdots, t_{p_k})$, where $p_1, p_2, \cdots, p_k$ are all the actions preceding p in $\geqq_T$.

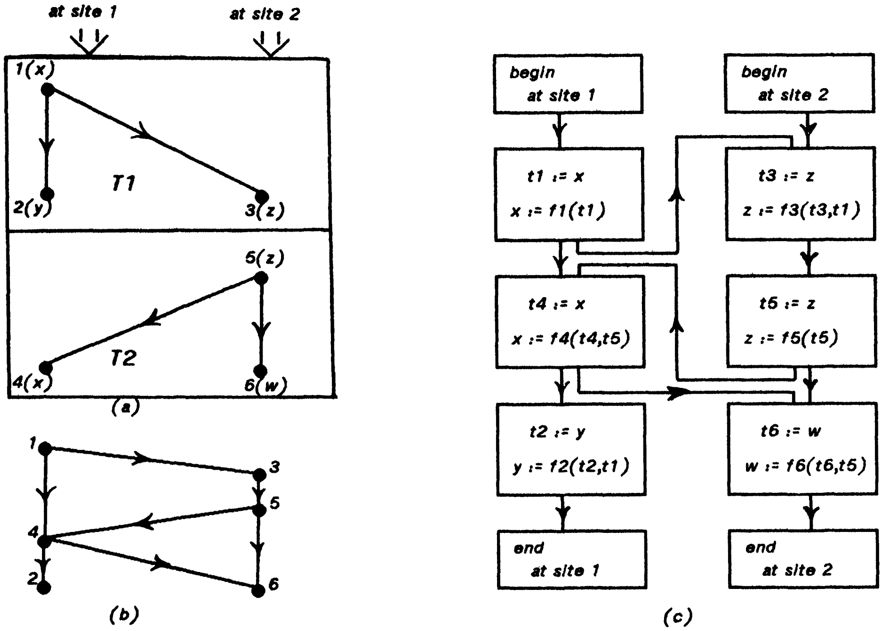

FIG. 2. (a) *Transaction system* $T=\{T_1, T_2\}$ *(e.g., action* 1 *updates* x*)*. (b) *Schedule* $s=\langle T, \pi\rangle$. (c) *The semantics of the actions in schedule* s.

Precedence between actions in a transaction T denotes both temporal precedence and a transfer of information (i.e., in Fig. 2a action 3 needs data from action 1 and is executed after action 1). Arcs in a transaction T between actions at different sites are called *cross-arcs defined in* T. A cross-arc defined in T indicates information transfer between processes of T at different sites.

A schedule is a description of a set of transactions and the process of their execution on the system. In a distributed system it is in general impossible to tell which one of two events occurred first (because communication is not always instantaneous). Because of this uncertainty, we describe the execution order of the actions by a partial order. If two events are incomparable in this partial order, any one could have preceded the other. There are two restrictions on the partial orders. First, what happens at every site is totally ordered; this is consistent with the centralized problem and guarantees that the result of the execution is uniquely determined, as in the case of individual transactions. Second, precedences specified by the transactions are always respected.

Formally:

DEFINITION 3. A *schedule* is a pair $\langle \boldsymbol{T}, \pi\rangle$, where $\boldsymbol{T} = \{T_i, 1 \leqq i \leqq m\}$ is a transaction system and π is a directed acyclic graph (dag) on the nodes of the transactions T_i such that:

(a) Nodes p with the same *site*(p) are totally ordered.

(b) For any transaction T_i and actions $p, q \in T_i$ with $p \geqq_{T_i} q$ we have that $p \geqq_\pi q$ (where $\geqq_\pi$ denotes the partial order of π).

A *prefix* of a schedule $s = \langle \boldsymbol{T}, \pi\rangle$ is a pair $\langle \boldsymbol{T}, \alpha\rangle$, where α is the subgraph of π induced by a subset of its nodes and such that if action $q \in \alpha$ all $p \geqq_\pi q$ belong to α.

Let S denote the set of all schedules. Recall that a partial order can be considered as a set of total orders (those compatible with it). Let S^+ denote the set of all schedules $\langle \boldsymbol{T}, \pi\rangle$, where π is a total order. Therefore a schedule s represents a particular subset of this basic set S^+. Arcs in a schedule, between actions at different sites are called *cross-arcs.* The schedules with only transaction defined cross-arcs are maximal when considered as sets of total orders. Yet schedules can have other cross-arcs also (e.g., arc (4, 6) in Fig. 2b), whose presence restricts the represented total orders of actions. The goal of *concurrency control* is to recognize *on-line* large sets of correct total orders.

As in the centralized case, synchronization is necessary only between actions of a transaction system, which operate on the same entities (i.e., conflict). These conflicts are represented by the conflict graph $G(\boldsymbol{T})$. We denote undirected edges by ij and arcs by (ij).

DEFINITION 4. For the transaction system $\boldsymbol{T} = \{T_i, 1 \leqq i \leqq m\}$, the *conflict graph* $G(\boldsymbol{T})$ is an undirected multigraph (V, E), with a partial order $\geqq_i$ on the edges incident upon each node i, such that:

(a) $V = \{i | 1 \leqq i \leqq m\}$, where node i corresponds to transaction T_i.

(b) E is a multiset of edges.

$$E = \{\text{copies of edge } ij | \text{for every copy of } ij \text{ there is a distinct pair of actions } p, q \text{ with } p \in T_i, q \in T_j, i \neq j \text{ and } x_p = x_q\}.$$

(c) For two edges incident at node i we have $ij \geqq_i ik$ iff the action in T_i corresponding to ij precedes the action in T_i corresponding to ik.

Note that an edge in E denotes a conflict between two transactions. Every edge ij in E corresponds to a pair of actions $\{p, q\}$ which update the same entity. Based on where this entity is stored we can partition E into as many multisets as there are sites: **red** and **green** edges for two sites. An example is presented in Figs. 3a and 3b.

An *ordered mixed multigraph* $G = (V, E, A, \{\geqq_i\})$ is a mixed multigraph, with E a multiset of edges, A a multiset of arcs and $\{\geqq_i\}$ partial orders at each node i of the edges *and* arcs incident at the node. An *ordered undirected multigraph* has $A = \varnothing$ (e.g., conflict graphs are such combinatorial objects). An *ordered directed multigraph* has $E = \varnothing$.

Since a conflict (an edge in $G(\boldsymbol{T})$) corresponds to two actions at the same site and a schedule $s = \langle \boldsymbol{T}, \pi\rangle$ has a total order of the actions at each site, we say that a schedule *resolves all conflicts.* That is, if edge ij corresponds to the pair of actions $\{p, q\}$, $p \in T_i$, $q \in T_j$, $i \neq j$, we direct ij as (ij) iff $p \geqq_\pi q$. Thus the schedule s determines a unique ordered directed multigraph $G^\pi(\boldsymbol{T})$.

DEFINITION 5. A *prefix* $\langle \boldsymbol{T}, \alpha\rangle$ assigns a direction (ij) to an edge ij of the conflict graph $G(\boldsymbol{T})$ iff *all* schedules, which have $\langle \boldsymbol{T}, \alpha\rangle$ as prefix, assign ij the direction (ij). Therefore a prefix $\langle \boldsymbol{T}, \alpha\rangle$ determines an *assignment of directions* to some edges of $G(\boldsymbol{T})$. Conversely an assignment of directions to edges of the conflict graph is *realizable* by a prefix, if there is a prefix of a schedule assigning these directions and no others.

Thus a prefix $\langle \boldsymbol{T}, \alpha\rangle$ determines a unique ordered mixed multigraph $G^\alpha(\boldsymbol{T})$, which is $G(\boldsymbol{T})$ with some of its edges directed. In Fig. 3c we have a nonrealizable assignment of directions. Moreover we have the following complete characterization of realizable assignments of directions, which are, after all, the assignments of interest.

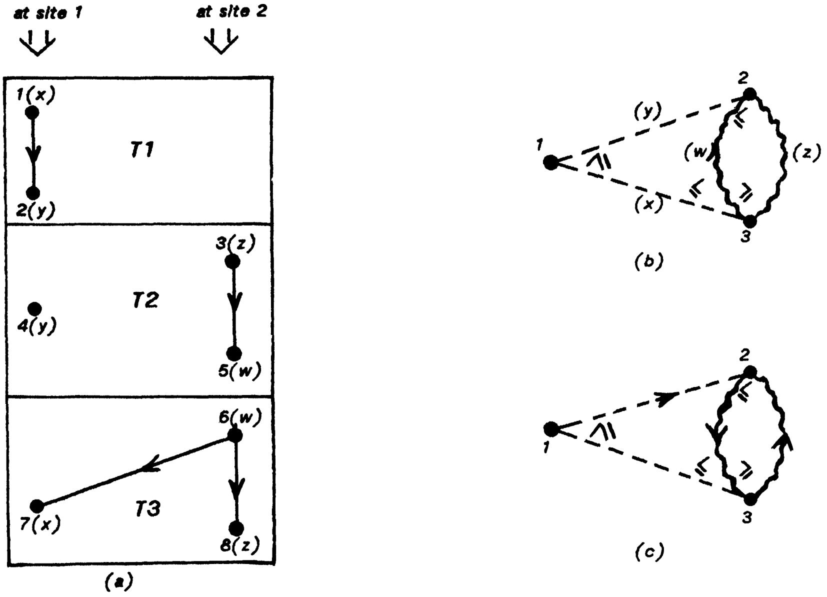

FIG. 3. (a) *Transactions.* (b) *Conflict graph.* (c) *A nonrealizable assignment; red* ●- - -● = *conflicts at site* 1, *green* ●〰● = *conflicts at site* 2.

LEMMA 1. *Given a conflict graph $G(\boldsymbol{T}) = (V, E, \emptyset, \{\geqq_i\})$, an assignment of directions to a multiset X of its edges, producing the ordered mixed multigraph $(V, E\backslash X, A, \{\geqq_i\})$ is realizable iff:*

(a) *$ij \in X$, is directed as $(ij) \in A$, and $ik \geqq_i ij \Rightarrow ik \in X$.*

(b) *A has no directed cycles $(i_1 i_2 i_3 \cdots i_n i_1)$ such that*

$$i_1 i_2 \geqq_{i_2} i_2 i_3, \quad i_2 i_3 \geqq_{i_3} i_3 i_4, \quad \cdots, \quad i_n i_1 \geqq_{i_1} i_1 i_2.$$

Proof. "only if". Given a prefix $\langle \boldsymbol{T}, \alpha\rangle$ of a schedule let us first assign the direction (ij) to any edge ij in $G(\boldsymbol{T})$, corresponding to a pair of conflicting actions $\{p, q\}$, with $p \in T_i$, $q \in T_j$, under the following conditions:

$$p \in \alpha \quad \text{and} \quad \text{if } q \in \alpha \text{ then } p \geqq_\alpha q.$$

It is easily seen that both conditions (a) and (b) hold for the directions constructed above. Obviously all schedules, which have $\langle \boldsymbol{T}, \alpha\rangle$ as prefix, resolve these conflicts in the same way. Moreover if an edge has not been given a direction then both its actions p^*, q^* are not in α. We can complete $\langle \boldsymbol{T}, \alpha\rangle$ using two different schedules, one having p^* before q^* and the other q^* before p^*. One schedule results from completely executing the transaction of p^* first and the other is symmetric. This proves that the directions we have constructed are exactly those assigned by $\langle \boldsymbol{T}, \alpha\rangle$.

Sufficiency. Given an assignment A we construct the following digraph (V^*, A^*) from A and $\boldsymbol{T}$

$$V^* = \{p \mid \exists (ij) \in A, \text{ where the conflicting action of } ij \text{ in } T_i \text{ is } p \text{ or one of } p\text{'s predecessors in } T_i\},$$

$$A^* = \{(pq) \mid \text{if } (pq) \text{ is part of some } \geqq_{T_i} \text{ or if } (pq) \text{ corresponds to an } (ij) \in A\}.$$

Since (b) is true (V^*, A^*) is acyclic, and since (a) is true transaction precedences are respected. Thus (V^*, A^*) has the same nodes as some prefix and respects all its conflict resolving orderings. □

2.3. Serializability. Only a subset of the possible schedules are considered correct for the operation of the database. The object of concurrency control is to develop algorithms, which monitor the execution of transactions, and disallow incorrect schedules. Actually, our results can be stated in a manner independent of the notion of correctness used in the system. We can show, however, that our negative results hold even when this correctness criterion is a practically important one, that of serializability, which we introduce next.

Serializability can be defined *semantically* [8], [11]. Since we are interested in simplifying our model, in order to bring out the complications inherent to distributed databases, we shall adopt instead a simple *syntactic* definition of serializability. This definition will not require our formally dealing with the semantics of actions and was, interestingly, the first to be proposed [4]. It turns out to be equivalent to the semantic one, if we think of the nodes of the transactions as indivisible read and write operations (see [8]), as opposed to operations that entail either reading or writing an entity [11]. The example of Fig. 2c illustrates the semantics of updates, in terms of program schemes [8], [11]. In fact, the following syntactic definitions suffice for the results presented in this paper.

DEFINITION 6. Two schedules $\langle \boldsymbol{T}, \pi\rangle, \langle \boldsymbol{T}, \rho\rangle$ are *equivalent* if they determine the same ordered directed multigraph, (i.e., $G^\pi(\boldsymbol{T}) = G^\rho(\boldsymbol{T})$).

DEFINITION 7. A schedule $\langle \boldsymbol{T}, \pi\rangle$ is *serial* iff

(a) The execution of actions at every site introduces a total order of transactions at that site (i.e., there are no T_i, T_j, $i \neq j$ with actions $p, q \in T_i$, $r \in T_j$ at the same site with $p \geqq_\pi r$ and $r \geqq_\pi q$).

(b) If T_i precedes T_j at one site it does so at all sites, where both transactions have actions.

A schedule is *serializable* iff it is equivalent to a serial schedule.

We denote the set of serializable schedules by SR ($SR \subseteq S$). What is remarkable, is that deciding whether a schedule is serializable in a centralized or distributed model are practically identical tasks [11]. We state this as follows:

THEOREM 1. *A schedule $\langle \boldsymbol{T}, \pi\rangle$ is serializable iff it resolves conflicts without creating directed cycles in $G(\boldsymbol{T})$ (i.e., $G^\pi(\boldsymbol{T})$ is acyclic). Similarly, a prefix $\langle \boldsymbol{T}, \alpha\rangle$ has a serializable completion iff the already resolved conflicts do not create a directed cycle in $G(\boldsymbol{T})$ (i.e., $G^\alpha(\boldsymbol{T})$ has no directed cycles).*

Proof. Easily follows from the analysis of [11]. □

2.4. Schedulers. Up until now the distributed problem appears to be a straightforward generalization of the centralized case. What is considerably more complex in the distributed case is the subject of schedulers and their design to meet performance specifications. For an exposition of the relatively simple theory for the centralized case see [11].

Our schedulers will be distributed programs characterized by the parallelism they provide and by their efficiency. We will measure parallelism using the subset C of schedules, which the scheduler allows to be executed as requested. The efficiency of the scheduler will be measured by the worst-case number of steps it executes and the worst-case number of messages it sends. We will be interested in the following kinds of C's:

DEFINITION 8. Consider a set of schedules $C \subseteq S$, such that for each $s \in C$ the only cross-arcs are defined by the transactions. Such a C we shall call a *concurrency control principle.*

Each schedule s corresponds to a set of *total orders* $\{\sigma | \sigma$ is a total order compatible with $s\}$. This set is also denoted by s. If C is a set of schedules, we let $C^+ = \bigcup_{s \in C} s$. Recall that S is the set of all schedules and S^+ the set of all total orders. For a particular transaction system $\boldsymbol{T}$, with n actions, $\sigma \in S^+$ is a string of length n over N, where N is the set of $\boldsymbol{T}$'s actions. The jth symbol of σ is denoted σ_j.

The cardinality of $C^+ \subseteq S^+$ will be the measure of parallelism. The larger C^+ is, the higher the level of parallelism supported by this concurrency control principle. For example, if SR are the serializable schedules then $SR^+ = \bigcup_{s \in SR} s$. Note that, SR^+ is also the set of total orders of a concurrency control principle, the serializable schedules with only transaction defined cross-arcs; this easily follows from Theorem 1 and Lemma 1. We will hence use the notation SR for this concurrency control principle, without any loss of generality. Similarly serial execution provides another example of a concurrency control principle, which obviously supports less parallelism. Thus concurrency control principles are very natural classes of schedules measuring parallelism, although not all subsets of S can be expressed as such.

A *scheduler* **A** is a *distributed program.* We do not explicitly specify the model of computation; we use a model equivalent to [6], although we employ a simple concurrent language notation as needed (e.g., a *send-message* instruction). Our distributed programs consist of a set of communicating sequential processes [6], one for each site. Their instructions may denote:

(a) local computation;

(b) receiving an execution request for an action q;

(c) granting an execution request of an action q;

(d) sending a message to another site;

(e) receiving a message from another site.

We shall now formalize the input–output behavior of the scheduler. Intuitively, a scheduler receives a schedule as its input and outputs another schedule. There is a difficulty though in defining this mapping precisely, because it is essentially a nondeterministic mapping. Although the scheduler has perfectly deterministic algorithms as its processes, the interaction of these algorithms is conducted via *messages*, whose delivery time is unpredictable. M. Fischer uses the term *indeterminism* [20] for this kind of unpredictable behavior (nondeterminism would not be an appropriate term, since we wish to produce correct computations in all cases). To model indeterminism of a scheduler, we must somehow introduce some notion of *time.*

(1) *The input of a scheduler is a string in S^+.* Thus we assume that the arrivals of the requests for executions of the nodes of the schedule-input are totally ordered in time. This is only a simplifying tool (a formalism of the familiar notion of a *timestamp* [10]), and is not used by the scheduler, whose processes still perceive the world in terms of partial orders. We therefore have introduced a *global clock*, whose ticks are the arrivals of the action requests.

(2) What is the output of a scheduler? It is a schedule, of course. However, it

must also add some more information. Namely, it must tell us whether an action was granted before or after the arrival of another request. *The output of the scheduler is an n-tuple of strings* $(\tau_1, \tau_2, \cdots, \tau_n) \in (N^*)^n$. Here τ_j denotes the sequence of granted requests between the jth and $(j+1)$st (after the jth if $j = n$) arrivals of requests. N^* is the set of all strings constructed from the set of actions N and includes the empty string. The concatenation of the n strings, conc $(\tau_1, \tau_2, \cdots, \tau_n)$, should be in S^+.

(3) We shall now formalize the indeterministic part of the scheduler, namely the communication delays. A *delay vector* $\mathbf{d}$ is a sequence of nonnegative real numbers. Intuitively, the jth component is the delay of the jth message sent by the scheduler. With a given delay vector the operation of a scheduler $\mathbf{A}$ on some input σ is completely specified (exactly as the operation of a nondeterministic algorithm becomes specified if we supply a sequence of choices for the nondeterministic steps). To find the resulting output, we do the following. For each site, we keep a calendar of events (i.e., arrivals of actions or messages, operations of the scheduler), with the precise times at which they occur. An event may trigger a finite sequence of operations of the scheduler, which we execute. If an operation involves sending a message to another site, we add the next component of $\mathbf{d}$ to the time of the present event and we insert the arrival of this message in the calendar of the other site at the time of the sum. We thus assume that, *all local operations of the scheduler take* 0 *time*. We break ties on the times of events in a systematic fashion (e.g., arrivals of actions first, then messages from site 1, etc). We can now produce the output of the scheduler for this input σ and this delay vector $\mathbf{d}$ in the obvious way from the calendars of events. This output $(\tau_1, \tau_2, \cdots, \tau_n)$, we denote by $\mathbf{A}_\mathbf{d}(\sigma)$. Not all delay vectors can lead to meaningful executions, however. What can go wrong is that a long delay can postpone the granting of an action p until after the successor q of p in its transaction has been received. Delay vectors for which no such anomaly occurs for an input σ are called *feasible* for σ. The *zero sequence* $\mathbf{d} = \mathbf{0}$ is always feasible.

Therefore the operation of a scheduler is formulated by the function $\mathbf{A}_\mathbf{d}: S^+ \rightarrow (N^*)^n$.

Consider a concurrency control principle C. We say that scheduler $\mathbf{A}$ implements C if, intuitively, all outputs of $\mathbf{A}$ are in C and, furthermore, if $\mathbf{A}$ is fed with a schedule in C and all delays are 0, then $\mathbf{A}$ grants all requests immediately upon receipt. It is argued in [11] that these are traits, in the centralized case, of all schedulers that are *on-line* and *optimistic* (i.e., the scheduler does not intervene to unnecessarily delay an action if the input schedule is so far correct). The same arguments are applicable to justify Definition 9.

DEFINITION 9. We say that $\mathbf{A}$ is an *implementation* of concurrency control principle C iff

(a) conc $(\mathbf{A}_\mathbf{d}(\sigma)) \in C^+$ for all $\sigma \in S^+$ and feasible delay vectors $\mathbf{d}$, and

(b) $\mathbf{A}_\mathbf{0}(\sigma) = (\sigma_1, \cdots, \sigma_n)$ for all $\sigma \in C^+$.

There is a fundamental asymmetry in Definition 9. If the input is in C^+, then condition (b) is in effect, and the scheduler must leave it intact, unless forced to do otherwise because of the delays. If, however, the input is not in C^+, then the output can be any schedule in C^+. In practice, we would expect of a scheduler to change a schedule not in C^+ as little as possible in order to transform it into one in C^+. Unfortunately, there does not seem to be a clean way to express this mathematically in the distributed or centralized case. We have adopted the above convention in the interest of keeping our model and subsequent proofs as simple as possible.

DEFINITION 10. The *computational complexity* of $\mathbf{A}$ is the sum of the step-counts of all local computations by $\mathbf{A}$ over all processes of $\mathbf{A}$, maximized over all σ and feasible **d**. The *communication complexity* of **A** is the number of all *send-message* instructions executed by all processes of **A**, maximized over all σ and feasible **d**.

Note that apart from the messages generated by the scheduler processes of the system, there is also user defined communication, implied by transaction defined cross-arcs (e.g., some action at site 2 needs data from site 1). *This communication is assumed free, since it is unavoidable.* Such messages, between the processes of a transaction, can be used to pass information between scheduler processes at no cost.

A scheduler **A** is polynomial-time bounded (or *computationally efficient*) if its computational complexity is bounded by a polynomial in n (where $n=|N|$ and N is the set of actions of $\boldsymbol{T}$). Similarly, with [11] we can prove:

THEOREM 2. *C has a computationally efficient implementation iff the set of prefixes of C is in* P.

Proof. By broadcasting each arrival of a request we can reduce the distributed to the centralized problem, and use [11, Thm. 10]. Note that this solution is wasteful in terms of messages. □

Finally in order to characterize communication complexity we define the following classes of prefixes $M_c(b)$:

DEFINITION 11. For concurrency control principle C, its set of prefixes $PR(C)$ and integer $b \geqq 0$,

$$\begin{aligned} M_c(b) = &\{\text{prefixes not in } PR(C)\} \\ &\cup \{\langle \boldsymbol{T}, \alpha\rangle \mid \langle \boldsymbol{T}, \alpha\rangle \in PR(C) \text{ and there is an implementation } \mathbf{A} \text{ of } C, \\ &\qquad \text{which, given that } \langle \boldsymbol{T}, \alpha\rangle \text{ has been granted,} \\ &\qquad\qquad \text{proceeds using at most } b \textit{ send-message's}\}. \end{aligned}$$

Let $b^*(\boldsymbol{T})$ be the least b for which $\langle \boldsymbol{T}, \varnothing\rangle \in M_c(b)$. A scheduler which achieves $b^*(\boldsymbol{T})$, for every $\boldsymbol{T}$, is called *communication-optimal* with respect to C.

Note that for $b<0$ we can define $M_c(b)=\varnothing$ and then for all b we have $M_c(b) \subseteq M_c(b+1)$. If $\langle \boldsymbol{T}, \alpha\rangle$ is a prefix of $\langle \boldsymbol{T}, \beta\rangle$ and $\langle \boldsymbol{T}, \alpha\rangle \in M_c(b)$, then also $\langle \boldsymbol{T}, \beta\rangle \in M_c(b)$. By our convention if $\langle \boldsymbol{T}, \alpha\rangle$ is not a prefix of a schedule in C then $\langle \boldsymbol{T}, \alpha\rangle \in M_c(0)$. Intuitively, if all sites know of an incorrect input they can output a predetermined correct completion without communication.

In essence, what Definition 11 says is that: the scheduler might use a priori information, available to all scheduler processes, in order to enhance the communication performance (worst-case, number of messages used at run time) of the concurrency control mechanism. For example, a scheduler that implements serializability (for all transaction systems $\boldsymbol{T}$), might also examine the available syntax of transaction system $\boldsymbol{T}$, in order to develop a more economical communication strategy between its processes. This is analogous to the conflict graph analysis used to improve parallelism in SDD-1 [1], [2]. A communication-optimal scheduler is the limit in message performance attainable, subject to a parallelism requirement C. In the following section we will show, in a constructive fashion, that such schedulers exist for concurrency control principles.

3. Communication-optimal schedulers and games.

The performance measure of a concurrency control algorithm is a set of schedules C. We require C to be a concurrency control principle (see Definition 8). Let $PR(C)$ be the set of prefixes of schedules in C. We assume that we have an efficient (polynomial time in n) test of membership of a prefix in $PR(C)$. For example, if $C=SR$ Theorem 1 provides us with such a test. If no such test is possible, concurrency control is quite hopeless, even in the centralized case [11]. We also assume that we have a two-site system. This is no loss for the negative results of the next section. As for the positive results, they can be restated without much difficulty, although less succinctly, for the general case.

Let us briefly review the notation used. A prefix is denoted as a pair $\langle \boldsymbol{T}, \alpha \rangle$, or simply α when there is no ambiguity. In order to make our notation simple we will omit $\boldsymbol{T}$, the obvious transaction system, whenever possible. We use $M_c(b)$, for the set of all prefixes $\langle \boldsymbol{T}, \alpha \rangle$ of C such that there is an implementation of C, which, when started with $\langle \boldsymbol{T}, \alpha \rangle$, sends b or fewer messages. Now let α be a prefix of β, then $(\beta/\alpha)_i$ denotes the prefix of β, that contains α and all actions of β at site i. We call this the *projection* of β at site i given α (see Fig. 4 for an example of this important notion).

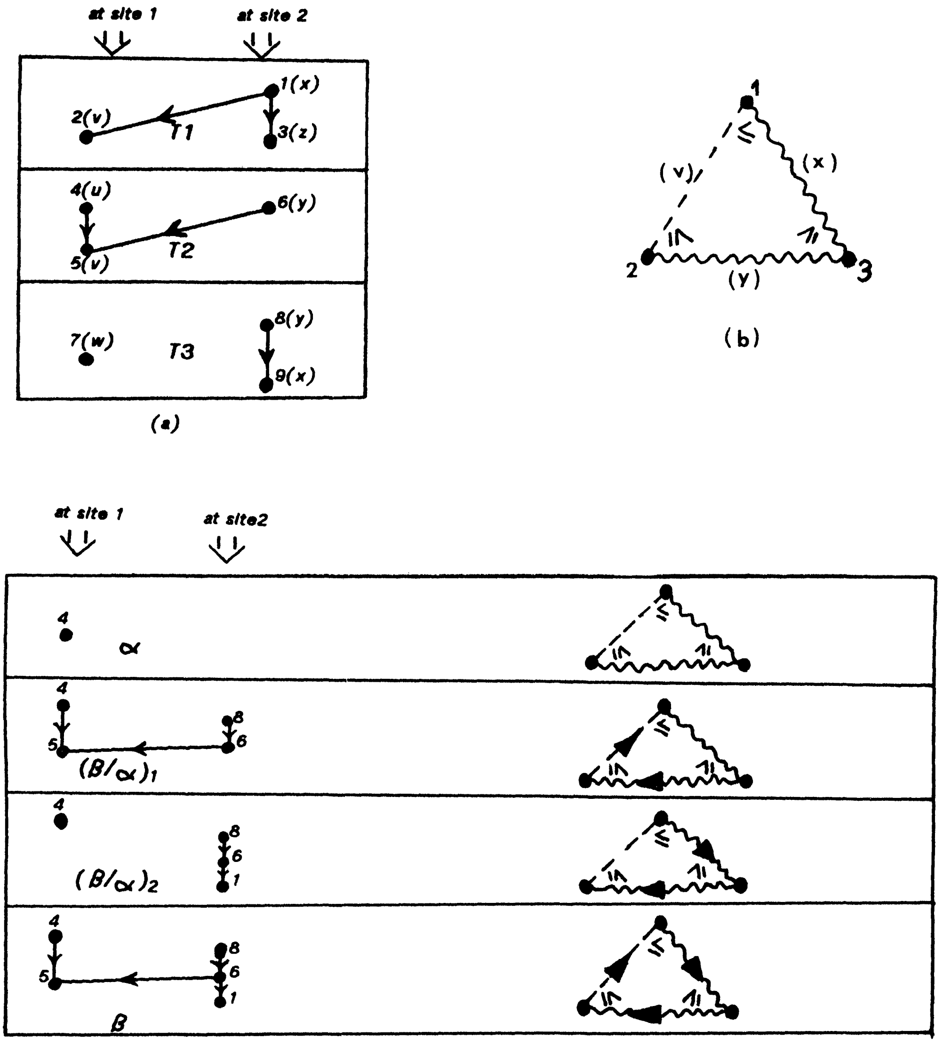

FIG. 4. (a) *Transactions* (u, v, w *at site* 1, x, y, z *at site* 2). (b) *Conflict graph* (*red* ●– – –● = *conflicts at site* 1, *green* ●∿∿∿● = *conflicts at site* 2). (c) *Illustrating a bad* β. *Left*: *prefixes. Right*: *assignments of directions.*

DEFINITION 12. Let $\langle \boldsymbol{T}, \alpha\rangle \in PR(C)$, and let α be a prefix of β. We call β *bad*, with respect to α, if

(a) $(\beta/\alpha)_1, (\beta/\alpha)_2 \in PR(C)$; and

(b) $\beta \not\in PR(C)$.

It is only bad prefixes that force the scheduler to communicate, in rounds of two messages. This, as well as a description of the possible strategies for guarding against bad prefixes, is captured by the following theorem.

THEOREM 3. *Let C be a concurrency control principle, $\langle \boldsymbol{T}, \alpha\rangle \in PR(C)$ and $b \geqq 0$. Let i denote the site number, $i \in \{1, 2\}$. Then statements* (I) *and* (II) *are equivalent*:

(I) $\alpha \in M_c(b)$.

(II) *For all bad β, with respect to, α*: (1) $(\beta/\alpha)_i \in M_c(b)$ *for* $i = 1, 2$ *and* (2) *at least one of the* $(\beta/\alpha)_i \in M_c(b-2)$.

The intuitive interpretation of the theorem is the following: Suppose that there is a possible scenario (see Fig. 4 for an example) in which both sites see projections $(\beta/\alpha)_i$, that are perfectly legal locally (i.e., both are in $PR(C)$) and still, they are not legal when put together (i.e., β is not in $PR(C)$). This clearly calls for communication. The theorem says that, in the worst case, two messages are both necessary and sufficient to overcome this problem.

Proof. "only if". To show that (I) implies (II), suppose that a scheduler **A** can start from α and implement C with only b messages. Let β be bad with respect to α. It is easy to see that (II.1) is satisfied. We must now show that (II.2) is also true.

What should site i do if it is presented with requests for the actions in $(\beta/\alpha)_i$? Clearly, it should have a way of granting them, perhaps after certain communication, since **A** is supposed to implement C (i.e., see Definition 9 for the on-line property). If site i grants the requests without waiting for any messages, then site $j = 3 - i$ must guard against this eventuality, when presented with $(\beta/\alpha)_j$, by asking site i's state. This takes two messages, which synchronize the two processes, and thus **A** must implement C starting from $(\beta/\alpha)_j$ within $b-2$ messages; thus property (II.2) holds. This leaves us with the case in which site i waits for a message before granting $(\beta/\alpha)_i$. It cannot wait for a message triggered by any event at site j other than an arrival of a message from i; this follows from the fact that **A** must implement C. We are therefore reduced to the previously examined case.

"if". To show that condition (II) is sufficient, we shall construct an explicit algorithm that implements C, starting from α and using no more than b messages, assuming (II) holds. The algorithm is recursive, and is shown in Fig. 5.

The algorithm, **localscheduler**, is the process run by each site. Its arguments are the prefix $\langle \boldsymbol{T}, \alpha\rangle$ of granted actions at the instant it takes over and the number b of messages that it can use. For example if no actions have been granted, both sites start by running **localscheduler**($\langle \boldsymbol{T}, \emptyset\rangle$, b).

The variable *localstate* represents the actions that the site knows are granted (through its own granting actions and other messages), whereas *commonstate* is the information this site knows the other site already has. The values of these variables are prefixes in $PR(C)$. They are both initialized to $\langle \boldsymbol{T}, \alpha\rangle$ and updated appropriately whenever:

(a) An action is granted at this site, through the function **grant**(p).

(b) A message is exchanged by scheduler processes, through the functions **askstate** and **reportstate**.

(c) A message is exchanged by transaction processes, because of a transaction defined cross-arc.

In the last case the *localstate* at one site, may be passed to the other at no communication cost. The detailed code for performing these updates or the functions **grant**, **askstate** and **reportstate** is straightforward and is not shown in Fig. 5. The low level details of all these functions can be found in [7].

When a request p arrives, the scheduler first decides whether it is necessary to communicate. This is the first test in Fig. 5. Communication is forced just in the case that a prefix β exists, such that:

(a) it violates the concurrency control principle C (i.e., $\beta \not\subseteq PR(C)$);

(b) its projection at the other site given *commonstate* is in $PR(C)$;

(c) its local projection is *localstate*$\|p$ (where $\|$ denotes concatenation), and moreover it is amenable to scheduling with $b-2$ messages. In other words, condition (II.2) of the theorem is satisfied with i equal to the present site.

```
procedure localscheduler($\langle \boldsymbol{T}, \alpha\rangle$, $b$)
localstate := commonstate := $\langle \boldsymbol{T}, \alpha\rangle$;
on request-arrival $p$ do
if there is a prefix $\beta \not\subseteq PR(C)$, whose projection at the other site
   given commonstate is in $PR(C)$, and whose local projection is
   localstate$\|p \in M_c(b-2)$
    then        begin
                localstate := askstate( );
                if localstate$\|p \in PR(C)$
                   then grant($p$); localscheduler(localstate,$b-2$)
                   else tablelookup( )
                end
    else if localstate$\|p \in PR(C)$
            then grant($p$)
            else tablelookup( )
end localscheduler
```

FIG. 5. *The process* **localscheduler**.

By convention, any prefix not in $PR(C)$ needs 0 messages and therefore the prefix *localstate*$\|p \not\subseteq PR(C)$ would pass the test only if $b>0$. Except for this case a β, such that the above conditions are true, is one satisfying (II) of the theorem.

If the above conditions are met, the scheduler decides to communicate. The function **askstate** learns the state of the other site at the cost of two messages. Presumably the return message is sent by a function **reportstate** at the other site, which also does the appropriate updating. If p is found to be safe, it is granted, and **localscheduler** is called recursively with the new arguments (note that *localstate* would be appropriately updated by **grant**). Since the test succeeded, we know that it can carry out its task within $b-2$ messages. If now *localstate*$\|p \not\subseteq PR(C)$, then the arriving stream of requests is not in C, and therefore we have no contract to fill (recall the paragraph right after Definition 9); both sites continue scheduling by some **tablelookup**, agreed upon in advance between the **sites**.

If the first test fails, then we must proceed with locally available information. If p looks safe, we grant it. We know we are not risking anything since, by (II), the other site will pass the test, and will communicate before it grants its part of any bad β. If p is not safe, we again resort to **tablelookup**, but now since $b=0$ and (II) is true both sites can proceed independently with no risk.

The formal proof that the algorithm, as specified above, correctly schedules a given prefix within the given number of messages is now straightforward, by induction on the number of actions in any suffix of $\langle \mathbf{T}, \alpha\rangle$. It should also be noted that we use the fact that C is a concurrency control principle when we test if a *localstate*$\|$*p* is in $PR(C)$. Since schedules in C have only transaction defined cross-arcs this test can be done locally. □

COROLLARY 3.1. *If C, a concurrency control principle, has a computationally efficient implementation, then it has a communication-optimal implementation, which uses space polynomial in n (n = number of actions of $\mathbf{T}$).*

Proof. The hardest computation performed by **localscheduler** in the proof of Theorem 3 is testing whether a prefix is in $M_c(b)$. This, however, can be expressed as a predicate with polynomial matrix and b alternations of quantifiers. It is therefore in PSPACE [3]. □

Distributed scheduling is related to a game on prefixes called PREFIX. The rules of this game are displayed in Fig. 6. In this game Player I corresponds to a malicious adversary who wishes to force communication. His move is a continuation β of the current position α, which satisfies the conditions of Theorem 3. Player II corresponds to the two cooperating scheduler processes. Each one of his choices i^* indicates, which of the two processes has the responsibility of guarding against the continuation β (by questioning the other process before proceeding). Player I wants to prolong the game as much as possible, whereas Player II tries to bring it to an end as soon as possible (other than that there is no winner or loser). Players I and II take turns moving.

COROLLARY 3.2. *The minimum number of messages used by a communication-optimal implementation of C equals the length of* PREFIX($\langle \mathbf{T}, \varnothing\rangle$) *if both players play optimally, (we call this the minimax length).*

Proof. It follows from Theorem 3 and the theory of alternation [3]. Note that although in general we define PREFIX from an arbitrary initial position $\langle \mathbf{T}, \alpha\rangle$, we are in fact interested in $\alpha = \varnothing$, ($\mathbf{T}$ represents the static a priori information on transactions, that is used to optimize communication). As a result the question: "$\langle \mathbf{T}, \alpha\rangle \notin M_c$(b)?" is equivalent to "can Player I make PREFIX($\langle \mathbf{T}, \alpha\rangle$) last more than b moves?" □

PREFIX($\langle \mathbf{T}, \alpha\rangle$)
Position before player I's move: A prefix $\langle \mathbf{T}, \alpha\rangle$
Player I: Select a prefix β, which has α as a prefix **such that:**
(1) $(\beta/\alpha)_1, (\beta/\alpha)_2 \in PR(C)$
(2) $\beta \notin PR(C)$
Player II: Select $i^* \in \{1, 2\}$ **and set** $\alpha := (\beta/\alpha)_{i^*}$

FIG. 6. *The game* PREFIX.

4. The complexity of PREFIX.

In this section we prove the following theorem:

THEOREM 4. *Let $C = SR$. Given $\mathbf{T}$ and $b \geqq 0$, determining whether the minimax length of the game* PREFIX($\langle \mathbf{T}, \varnothing\rangle$) *equals b is* PSPACE-*complete.*

This theorem, as is pointed out explicitly in a series of corollaries, is a fundamental negative complexity result for distributed concurrency control.

It turns out that PREFIX, with $C = SR$, is closely related to a game played on the conflict graph of $\mathbf{T}$. Recall that the conflict graph is an ordered undirected multigraph with edges colored red or green. The game, called CONFLICT, is displayed in Fig. 7.

CONFLICT(G)

Position before player I's move: An ordered mixed multigraph $G = (V, E, A, \{\geqq_i\})$, with E partitioned into red and green, and A closed.

Player I: Select a set X of edges and assign directions to them. Let $X_r(X_g)$ be a subset of X containing all its red (green) edges and let $A_r(A_g)$ be the corresponding arcs. The sets A_r, A_g must be **such that:**

(1) $A \cup A_r$, $A \cup A_g$ are acyclic and closed;

(2) $A \cup A_r \cup A_g$ has a cycle and is closed;

Player II: Select $c \in \{r, g\}$ **and set** $E := E \backslash X_c$; $A := A \cup A_c$.

FIG. 7. *The game* CONFLICT.

A round (i.e., of moves by the two players) starts with a position, which is an ordered mixed multigraph $G = (V, E, A, \{\geqq_i\})$. Player I gives directions to certain undirected edges X, with subsets X_r, X_g, such that already existing arcs (i.e., A) and each new directed subset A_r or A_g do not create a cycle, whereas all arcs together do. Player II picks a color (i.e., red or green) and fixes the directions proposed by I (i.e., creates a new A). In the absence of the partial orders $\geqq_i$, the moves of Player I are very simple: He picks a *two-color cycle* that contains some red edges (X_r), some green edges (X_g) and possibly some arcs, and the arcs are all directed with the same sense around the cycle. These rules are complicated a little by the existence of the partial orders on edges and arcs. Again Player I chooses a set of undirected edges X and assigns directions to them, but now the sets of arcs $A \cup A_r$, $A \cup A_g$ and $A \cup A_r \cup A_g$ must be *closed* (e.g., each one of X_r, X_g contains all edges of one color in X and might contain some edges of the other color) where formally:

"arc (ij) is in a *closed* set of arcs and $ik \geqq_i (ij) \Rightarrow ik$ is in this set as arc (ik) or (ki)"

Again, as in PREFIX, Player I's goal is to prolong and Player II's is to shorten the game. The intuition behind CONFLICT and its relation to concurrency control is the following:

Concurrency control means to direct somehow all edges of the conflict graph, without forming directed cycles. (The color, red or green, of an edge is the site that is responsible for directing it.) To carry out this task in a distributed fashion, we may have to communicate, in order to prevent two-color cycles. Single-color cycles are benign, since they can be detected locally and prevented without communication. Player I's move is an orchestrated stream of requests for conflict resolutions, that forces such a communication. Player II, the distributed scheduler, chooses the site (color) that will send a message, trying to block long sequences of legal moves for I (i.e., trying to save messages). The connection between the concurrency control problem and PREFIX was established in Corollary 3.2. The connection between PREFIX and CONFLICT discussed above, can be formalized in the following, straightforward lemma:

LEMMA 2. *The minimax length of the game* PREFIX($\langle \boldsymbol{T}, \emptyset \rangle$), *with* $C = SR$, *equals the minimax length of the game* CONFLICT($G(\boldsymbol{T})$), (*i.e.,* $G(\boldsymbol{T})$ *is the conflict graph of* $\boldsymbol{T}$).

Proof. The correspondence between PREFIX($\langle \boldsymbol{T}, \alpha \rangle$), and CONFLICT($G^\alpha(\boldsymbol{T})$) is easily seen to be as follows:

α corresponds to A (i.e., the conflicts of $G(\boldsymbol{T})$ resolved by α);

β corresponds to $A \cup A_r \cup A_g$ (i.e., a nonserializable input);

$(\beta/\alpha)_1$ corresponds to $A \cup A_r$ (i.e., a serializable projection at site 1 given α);

$(\beta/\alpha)_2$ corresponds to $A \cup A_g$ (i.e., a serializable projection at site 2 given α).

$A, A \cup A_r \cup A_g, A \cup A_r, A \cup A_g$ have to be closed, because the moves in CONFLICT must be realizable by prefixes (see Lemma 1, § 2.2). □

It is easy to see that CONFLICT is in PSPACE (that is, computing the minimax length is in PSPACE). To show Theorem 4, we shall first prove that CONFLICT is PSPACE-complete. We start by proving a weaker result, whose proof is indicative of the method used [3], [5], [14].

LEMMA 3. *Computing the minimax length of* CONFLICT *is* Π_2^p*-hard, (even when the initial mixed graph has no orders on the edges).*

Proof. Let F be an AE-quantified Boolean formula

$$F = \forall x_2 \forall x_4 \cdots \forall x_n \exists x_1 \exists x_3 \cdots \exists x_{n-1} F^*(x_1, \cdots, x_n),$$

where F^* is a $3CNF$ formula with n variables (n is even) and m clauses. We shall construct a mixed graph G such that the minimax number of rounds of CONFLICT (rounds of moves by the two players), started on G, is equal to $(n/2)+1$ iff F is true. G is constructed as follows:

For each existentially quantified variable x_i, $i = 1, 3, \cdots, n-1$, we add to G a copy of the $\exists$-graph shown in Fig. 8c. For each universally quantified variable x_i, $i = 2, 4, \cdots, n$, we add to G a copy of the $\forall$-graph in Fig. 8a. Finally, for each clause C_k, we add to G the C-graph in Fig. 9. All these subgraphs are connected as indicated from vertex names (i.e., "in tandem"), with $\exists$-graphs alternating with $\forall$-graphs, followed by the C-graphs (that is, $S_{n+1} = C_1$). The "cycle" is closed by a green edge S_1C_{m+1} (see Fig. 10 for an example).

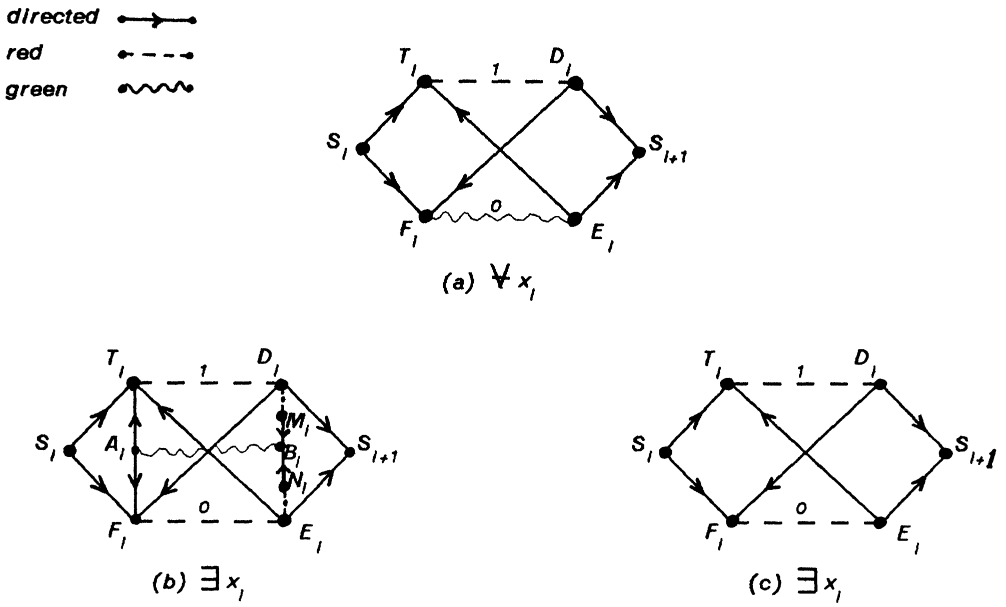

FIG. 8

So far we have only taken into account the numbers n and m. To encode the structure of F^* into G, we must look at the C-graphs of Fig. 9 in some detail. The C-graph consists of 7 paths, numbered from 001 to 111. These are the 7 truth assignments to the literals u, v, w of the clause, that satisfy the clause. Thus each of the 21 red edges of a C-graph, say e, is associated with a literal $l(e)$ and a truth value

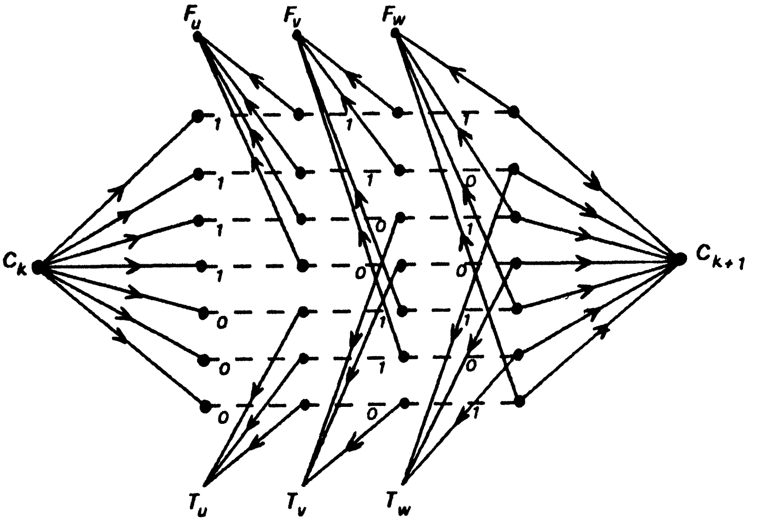

FIG. 9

$t(e)$. We now connect e's right endpoint with appropriate $\exists$- and $\forall$-graphs. We draw an arc from the right endpoint of the edge e

to F_j if ($l(e) = x_j$ and $t(e) = 1$), or ($l(e) = \neg x_j$ and $t(e) = 0$); and
to T_j if ($l(e) = x_j$ and $t(e) = 0$), or ($l(e) = \neg x_j$ and $t(e) = 1$).

These arcs are called *backarcs.*

This completes the construction of G (i.e., all orders $\geqq_i$ are empty). In Fig. 10 we have an example of the construction if we ignore the nodes A_i, B_i, M_i, N_i $i = 1, 3$ and A_{13}, A_{12}, A_{34}.

We now claim that, from the mixed graph G the minimax number of rounds (rounds of two moves each) is $(n/2)+1$ iff F is true. Clearly, since there are $(n/2)+1$ green edges, this number is at most $(n/2)+1$. We shall show that Player I can force $(n/2)+1$ rounds iff F is true.

Since the orders are empty, Player I's moves consist of choosing two-color directed cycles. These contain just one green edge (if I is to play $n/2+1$ times), and, if we disregard this green edge, there is no directed cycle in the graph with the proposed directions of red edges. It is easy to see that each green edge can be used only in one move, even if Player II does not explicitly direct it after this move (i.e., if his choice is red, in the new A, he has created a directed path between the endpoints of the green edge, and thus implicitly fixed its direction). Without loss of generality, the first $n/2$ moves will involve the green edges F_iE_i of the $\forall$-graphs. The two-color cycle $(F_iE_iT_iD_iF_i)$ is such a possibility. The choices of Player II can be thought of as fixing the direction of F_iE_i to: $(F_iE_i)-(x_i = 0)$ or $(E_iF_i)-(x_i = 1)$.

The claim is that Player I has an $[n/2+1]$st move, no matter what Player II plays, iff F is true. Player I has a $[n/2+1]$st move iff at the end there is a two-color cycle, which contains the only green edge left, $(C_{m+1}S_1)$, some red edges, some directed edges and no directed cycle without the green edge. Picking red edges is no problem—one has to do this to "pass through" the C-graphs and the $\exists$-graphs. In the $\forall$-graphs, the path must follow either $(S_iT_iD_iS_{i+1})$ or $(S_iF_iE_iS_{i+1})$. It follows the latter iff (F_iE_i) was picked by Player II in the corresponding move—otherwise a cycle $(F_iE_iT_iD_iF_i)$ would be created. In the $\exists$-graphs, this choice can be thought of as an assignment of the truth value to x_i by Player I (i.e., 1 if $(S_iT_iD_iS_{i+1})$ was picked, 0 if $(S_iF_iE_iS_{i+1})$ was

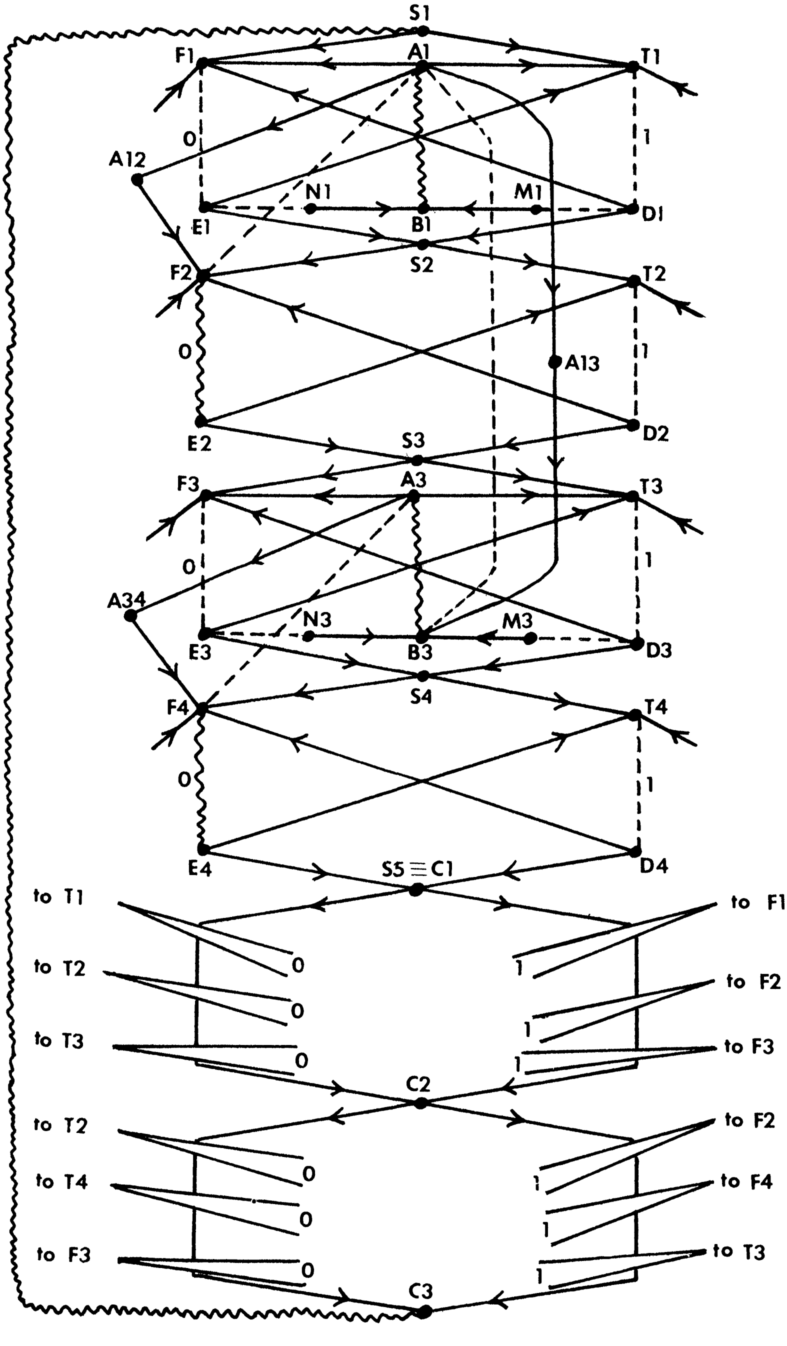

FIG. 10. $\exists x_1 \forall x_2 \exists x_3 \forall x_4\ (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_4 \vee \bar{x}_3)$.

picked). Finally, in each of the C-graphs, Player I must pick one of the 7 paths, which would not create cycles because of the backarcs. Therefore this path corresponds to a truth assignment, which agrees with the one chosen at the ∀- and ∃-graphs. It follows that such a path (indeed, such an $[n/2+1]$st move by Player I) exists iff F^* is satisfiable no matter what the values of $x_2, x_4, \cdots, x_n$ are, or, equivalently iff F is true. □

LEMMA 4. *Computing the minimax length of* CONFLICT *is* PSPACE-*complete,* (*even when the initial ordered graph is undirected*).

Proof. There are two directions in which we must extend the previous proof. First, we must encode in G the n alternations of quantifiers. We do this by designing a more elaborate $\exists$-graph (containing a green edge too) and using the partial orders $\{\geqq_i\}$. Second, we must get rid of the directed arcs of G. We do this last, by replacing each directed arc by a triangle, and using the partial orders.

Starting from the *QBF* instance $F = \exists x_1 \forall x_2 \exists x_3 \cdots \exists x_{n-1} \forall x_n F^*(x_1, \cdots, x_n)$ we construct an ordered mixed graph G by putting together the $\exists$-graphs of Fig. 8b (not 8c), the $\forall$-graphs of Fig. 8a and the C-graphs of Fig. 9, as in Lemma 3. We also have the following edges connecting neighboring $\forall$- and $\exists$-graphs:

$$\begin{array}{lll} \text{arcs} & (A_iA_{i,i+2}), (A_{i,i+2}B_{i+2}), & i = 1, 3, \cdots, n-3, \\ & (A_iA_{i,i+1}), (A_{i,i+1}F_{i+1}), & i = 1, 3, \cdots, n-1, \\ \text{red edges} & A_iB_{i+2}, & i = 1, 3, \cdots, n-3, \\ & A_iF_{i+1}, & i = 1, 3, \cdots, n-1. \end{array}$$

(These connections will guarantee that the order of moves by Player I will respect the order of quantification.) A full example is shown in Fig. 10.

Notice that, so far, we have not specified the orders $\{\geqq_i\}$. The orders for the arcs can be empty and for the undirected edges arbitrary total orders exist at all nodes except for the A_j, B_j, F_j nodes. There they are designed in such a way that Player I must play the green edges in their quantificational order (if the closure properties are to hold):

$$\begin{array}{ll} \text{at } A_i, & A_iB_i \geqq A_iF_{i+1} \geqq A_iB_{i+2},\ i = 1, 3, \cdots, n-1 \text{ (the last for } i \neq n-1), \\ \text{at } F_{i+1}, & F_{i+1}A_i \geqq F_{i+1}E_{i+1},\ i = 1, 3, \cdots, n-1, \\ \text{at } B_{i+2}, & B_{i+2}A_i \geqq B_{i+2}A_{i+2},\ i = 1, 3, \cdots, n-3. \end{array}$$

We can indicate these total orders by assigning the integers 1, 2, 3 to the undirected edges at each node and using the ordering of these numbers (see Fig. 11a).

We claim that the minimax number of rounds equals $n+1$ (again, the number of green edges in G) iff F is true. This would prove the lemma, modulo the presence of directed edges. The proof parallels that of Lemma 3, but is slightly harder.

It is easy to see that if Player I wishes to play $n+1$ rounds each one of his moves has to contain exactly one green edge, whose direction has not been fixed by previous moves. Therefore, as in Lemma 3, a game in which Player I can force $n+1$ rounds is essentially a permutation of the $n+1$ green edges. We will thus name his moves after their green edge. We will demonstrate that A_iB_i-moves $i = 1, 3, \cdots, n-1$ will correspond to Player I assigning values for the $\exists$-variables of F and F_iE_i-moves $i = 2, 4, \cdots, n$ to Player II assigning values to the $\forall$-variables of F. Moreover in a game where both players play optimally these choices alternate. The matter will consequently be reduced to the existence of an $[n+1]$st round, which will be equivalent to the validity of F.

Necessity. Assume the *QBF* instance F is false. We will describe a strategy for Player II, that will make the $[n+1]$st round impossible.

If Player I wishes to play $n+1$ rounds his game will be constrained in a variety of ways:

(a) Every $A_{i-2}B_{i-2}$-move must precede the A_iB_i- and $F_{i-1}E_{i-1}$-moves $i = 3, 5, \cdots, n+1$. Since the arcs of G have to be respected, we can only have $(B_iA_i) \in A_g$ and $(F_{i-1}E_{i-1}) \in A_g$ for legal assignments in these moves. This is because $A_g \cup A_r \cup A$

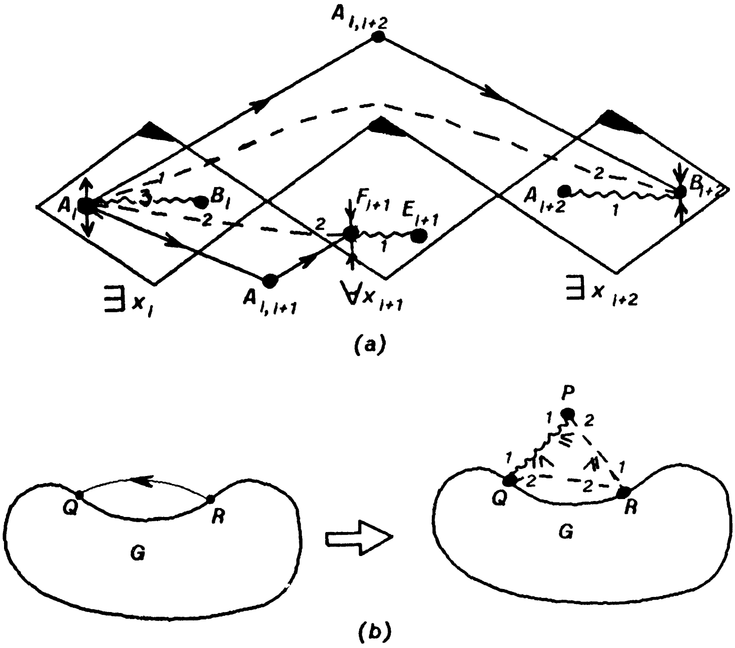

FIG. 11. (a) *Forcing alternation.* (b) *Forcing directions*: *red* ●– – –●; *green* ●~~~~●; *directed* ●→—●.

must contain a cycle whose orientation is determined by the existing arcs. Now we can justify the construction in Fig. 11a. If $(B_iA_i)\in A_g$ from the $\geqq_{B_i}$ order and the closure property of moves we have that $(A_{i-2}B_i)\in A_g\cup A$ (e.g., the direction of $A_{i-2}B_i$ is fixed because of the existing directed path $(A_{i-2}A_{i-2,i}B_i)$ in G). From the $\geqq_{A_{i-2}}$ order we have now that $A_{i-2}B_{i-2}$ must already be assigned a direction. Thus the $A_{i-2}B_{i-2}$-move must have already taken place. A similar argument holds for $F_{i-1}E_{i-1}$. It is easy to see also that the $C_{m+1}S_1$-move has to follow the F_nE_n-move.

(b) The F_iE_i-move corresponds to Player II assigning a value to x_i, $i=2,4,\cdots,n$; as in Lemma 3.

(c) The A_iB_i-move corresponds to Player I assigning a value to x_i, $i=1,3,\cdots,n-1$. The only possible choices of cycles are $(B_iA_iT_iD_iM_iB_i)$ corresponding to $x_i=1$ and $(B_iA_iF_iE_iN_iB_i)$ corresponding to $x_i=0$. For $x_i=1$ ($x_i=0$ is symmetric) the choice is forced by the existing arcs and because:

$(B_iA_iB_{i+2}A_{i+2}\cdots)$ would use up $B_{i+2}A_{i+2}$;

$(B_iA_iT_iD_iF_iE_i\cdots)$ would introduce a cycle in $A_r\cup A$;

$(B_iA_iT_iD_iS_{i+1}\cdots)$ would fix the direction of $F_{i+1}E_{i+1}$.

The strategy of Player II in response to these moves will be always to play red, fixing the directions of T_iD_i, and F_iE_i and making vertex A_i inaccessible from S_i.

Obviously the best Player I can do is assign a value to x_1 (by the A_1B_1-move), force Player II to show his hand by assigning a value to x_2 (by the F_2E_2-move), assign a value to x_3 etc. As a result the choices for the $C_{m+1}S_1$-move are constrained as in Lemma 3. Consequently the existence of a legal $[n+1]$st round depends on whether the assignment of values to the x_i's has made $F^*(x_1,\cdots,x_n)$ true. Since the QBF instance F is not valid Player II can always pick values for x_i, $i=2,4,\cdots,n$ that make $F^*(x_1,\cdots,x_n)$ false and the $[n+1]$st round impossible.

Sufficiency. Assume F is true. Player I's game follows the same structure as above. Only now, because of the validity of F, he can choose an assignment for x_i, $i = 1, 3, \cdots, n-1$ which will make $F^*(x_1, \cdots, x_n)$ true and the $[n+1]$st round possible.

Finally, we must eliminate the arcs of the graph in the construction above. We accomplish this by replacing each arc (RQ) by an undirected *triangle* RQP, where P is a new node, PQ is green and RQ and RP are red (Fig. 11b). At the nodes R, P, Q the three edges are ordered as indicated in Fig. 11b. The triangles themselves ($K_{n,m}$ in number) can be ordered. We can add $kK_{n,m}$ to the numbers 1, 2 at the edges of the kth rectangle, that indicate the orderings. Thus all $\{\geqq_i\}$ become total orders. We have therefore constructed an ordered undirected graph G^* from an arbitrary QBF instance F. We claim that the minimax number of rounds equals the number of green edges in G^* iff F is true.

Let us look at legal PQ-moves, that is moves whose green unfixed edge belongs to a triangle. If this move $(A_r \cup A_g \cup A)$ produces a cycle $(RQPR)$, we can infer the following: The arc (RQ) must belong to $A_r \cup A$ and $A_g \cup A$. This is because $A_r \cup A$ must contain a directed path $(P \cdots Q)$ and $QR \geqq_Q QP$. (Recall that QP is the only green edge without a previously fixed direction.) Thus no matter what the response of Player II is to such a PQ-move the arc (RQ) becomes part of A. On the other hand a PQ-move producing a cycle $(QRPQ)$ is never legal. This is because $A_g \cup A$ must contain $\{(PQ), (QR), (RP)\}$ a cycle. The existence of a path $(Q \cdots P)$ in $A_r \cup A$ and the fact that $RQ \geqq_R PR \geqq_P QP$ force this situation. Thus PQ-moves fix the direction of QR to (RQ). Finally if Player I were ever to use a QR in the direction (QR), in some other e-move (e a green unfixed edge), then a response of red by Player II would consume two green edges (i.e., e and PQ).

Now in order for Player I to play as many times as there are green edges in G^*, he must move using the green edges in the triangles and forcing the desired directions. This completes the proof of Lemma 4. □

Proof of Theorem 4. The theorem now follows by observing that the ordered graph $G = (V, E, \emptyset, \{\geqq_i\})$ in Lemma 4 is indeed the conflict graph of a transaction system $\boldsymbol{T}$. For each vertex i in V there is a transaction T_i in $\boldsymbol{T}$. For each edge $e = ij$ in E, there is an entity x_e updated by both T_i and T_j. If e is red, x_e is stored at site 1, if green at site 2. For the (total) orders $\geqq_i$, we simply order the actions of transaction T_i accordingly. □

As more-or-less immediate consequences of Theorem 4 and its proof we can obtain complexity characterizations for several special cases. Let us slightly abuse our notation, and use PREFIX($\langle \boldsymbol{T}, \alpha\rangle, b$) to denote the decision problem:

Is the minimax length of game PREFIX($\langle \boldsymbol{T}, \alpha\rangle$) larger than b?

We have the following cases depending on the structure of $\langle \boldsymbol{T}, \alpha\rangle$ and b.

COROLLARY 4.1. (a) PREFIX($\langle \boldsymbol{T}, \emptyset\rangle, b$) *is* PSPACE-*complete.*

(b) PREFIX($\langle \boldsymbol{T}, \alpha\rangle, b$) *is* PSPACE-*complete and* PREFIX($\langle \boldsymbol{T}, \alpha\rangle, 0$) *is* NP-*complete, even if* $\boldsymbol{T}$ *contains no cross-arcs.*

(c) PREFIX($\langle \boldsymbol{T}, \emptyset\rangle, 0$), *if* $\boldsymbol{T}$ *contains no cross-arcs, is in* P.

Furthermore, (a) *and* (b) *hold even when there are no more than six actions per transaction.*

Proof. Note that (a) follows directly from Theorem 4 and (b) can be easily shown by extending the proofs of Lemmas 3 and 4. By minor modifications [7] to the subgraphs of Figs. 8 and 9 we can make the nodes (after substituting triangles for directed edges) have at most degree 6. For case (c) all we have to test for is if $G(\boldsymbol{T})$ contains a two-color cycle. □

We finally obtain the following result on the complexity of distributed concurrency control.

COROLLARY 4.2. *Unless* NP = PSPACE, *there is no scheduler for SR, which is both computationally efficient and communication-optimal; even if we restrict* $\boldsymbol{T}$ *to sets of transactions which are total orders and have six actions each.*

Proof. If such a general scheduler existed, we would have a nondeterministic polynomial-time algorithm for solving the PSPACE-complete problem PREFIX $(\langle \boldsymbol{T}, \varnothing\rangle, b)$, as follows:

On input $\langle \boldsymbol{T}, \varnothing\rangle, b$:

1. *Guess* a schedule in *SR*, check it in polynomial time.
2. Simulate (in a centralized manner) the operation of the scheduler on this schedule. Whenever a *send-message* instruction occurs, *guess* a delay *d*, and increase a message count. (The delay *d* can be chosen to be a number bounded by a polynomial in size of the input).
3. In the end, if more than *b* messages were used, then report "yes", else report "no". □

5. Conclusions. Our main result shows that concurrency control, an on-line problem clearly in NP (P for serializability) in the centralized case, is PSPACE-complete in the distributed case. This result is quite strong, in that it holds for transaction systems of rather ordinary appearance (e.g., transactions which are total orders with at most six actions each). Also, the negative implications of our result (Corollary 4.2) are quite robust. For example, even if the scheduler is equipped with a powerful oracle belonging anywhere in the polynomial hierarchy, it still cannot minimize communication efficiently, unless the polynomial hierarchy collapses.

In the process of proving this negative result, we have related distributed concurrency control to certain combinatorial games played on graphs. It could be that this connection is of some practical value. There is a more-or-less immediate heuristic for approximating an optimal strategy in the game CONFLICT. This heuristic is based on the following purely combinatorial problem:

Given an undirected graph with its edges colored red and green, find a "small" set of edges that have to be deleted in order for the resulting graph to have no two-color cycle.

Acknowledgments. We would like to thank Prof. R. G. Gallager for many helpful discussions and Prof. P. Elias, Dr. A. Bruss, Vassos Hadzilacos and the anonymous referees for their comments on the presentation of these results.

REFERENCES

[1] P. A. BERNSTEIN AND N. GOODMAN, *Concurrency control in distributed database systems*, ACM Computing Surveys, 13 (1981), pp. 185–223.

[2] P. A. BERNSTEIN, D. W. SHIPMAN AND J. B. ROTHNIE, *Concurrency control in a system of distributed databases* (SDD-1), ACM Trans. Database Systems, 5 (1980), pp. 18–51.

[3] A. K. CHANDRA AND L. J. STOCKMEYER, *Alternation*, Proc. 17th Foundations of Computer Science Conference, 1976, pp. 98–108.

[4] K. P. ESWARAN, J. N. GRAY, R. A. LORIE AND I. L. TRAIGER, *The notions of consistency and predicate locks in a database system*, Comm. ACM, 19 (1976), pp. 624–634.

[5] S. EVEN AND R. E. TARJAN, *A combinatorial problem which is complete in polynomial space*, J. Assoc. Comput. Mach., 23 (1976), pp. 710–719.

[6] C. A. R. HOARE, *Communicating sequential processes*, Comm. ACM, 21 (1978), pp. 666–677.

[7] P. C. KANELLAKIS, *The complexity of concurrency control for distributed databases*, MIT/LCS/TR-269, Aug. 1981.

[8] H. T. KUNG AND C. H. PAPADIMITRIOU, *An optimality theory of database concurrency control,* Proc. 1979 SIGMOD, pp. 116–126.

[9] R. E. LADNER, *The complexity of problems in systems of communicating sequential processes,* Proc. 11th ACM Symposium on Theory of Computing, 1979, pp. 214–223.

[10] L. LAMPORT, *Time, clocks and the ordering of events in a distributed system,* Comm. ACM, 21 (1978), pp. 558–565.

[11] C. H. PAPADIMITRIOU, *Serializability of database updates,* J. Assoc. Comput. Mach., 26 (1979), pp. 631–653.

[12] C. H. PAPADIMITRIOU AND J. TSITSIKLIS, *On the complexity of designing distributed protocols,* manuscript, Aug. 1982.

[13] D. J. ROSENKRANTZ, R. E. STEARNS AND P. M. LEWIS, *System level concurrency control for distributed database systems* ACM Trans. Database Systems, 3 (1978), pp. 178–198.

[14] T. J. SCHAEFER, *Complexity of some perfect information games,* J. Comput. System Sci., 16 (1978), pp. 185–225.

[15] R. S. STEARNS, P. M. LEWIS AND D. J. ROSENKRANTZ, *Concurrency control for database systems,* Proc. 16th Foundations of Computer Science Conference, 1976, pp. 19–32.

[16] R. H. THOMAS, *A majority consensus approach to concurrency control for multiple copy databases,* ACM Trans. Database Systems, 4 (1979), pp. 180–209.

[17] J. D. ULLMAN, *Principles of Database Systems,* Computer Science Press, Potomac, MD, 1982 second edition.

[18] M. YANNAKAKIS, C. H. PAPADIMITRIOU AND H. T. KUNG, *Locking policies: safety and freedom from deadlock,* Proc. 20th Foundations of Computer Science Conference, 1979, pp. 283–287.

[19] A. C. YAO, *Some complexity questions related to distributive computing,* Proc. 11th ACM Symposium on Theory of Computing, 1979, pp. 209–213.

[20] *Eden Project Proposal: Research in Integrated Distributed Computing,* Dept. of Computer Science, Univ. Washington, Pullman, TR 80-10-1, October 1980.