ERRORS IN 'PROCESS SYNCHRONIZATION IN DATABASE SYSTEMS'*

by

Philip A. Bernstein
Marco A. Casanova
Nathan Goodman

TR-09-79

*This work was supported by the National Science Foundation, Grant Numbers MCS-77-05314 and MCS-79-07762, by the Advanced Research Projects Agency of the Department of Defense under Contract Number N00039-78-G-0020 and by the Conselho Nacional de Pesquisas, CNPq-Brasil, Grant Number 1112.1248/76.

Abstract

This paper disproves several results pertaining to database concurrency control that are claimed in [8]. The results we disprove are

- theorems 3.1, 3.2, 3.6 -- which claim a polynomial time algorithm for testing whether transaction schedules are serializable, and
- theorems 4.2 and 4.7 -- which claim a necessary and sufficient mechanism for preserving the "weak consistency" of databases.

In addition, we demonstrate that the notion of "weak consistency" introduced in [8] admits database states that are strictly inconsistent.

1. Introduction

In a recent paper [8], G. Schlageter introduced a formal theory of database concurrency control that is similar in appearance to the serializability theory developed in [1,2,3,5,6,7,9] and elsewhere. One of the results claimed in [8] implies that serializability of transaction schedules can be tested in polynomial time, contradicting NP-completeness results in [5,6] (unless P=NP). In addition, Schlageter [8] introduces a notion of consistency that is strictly weaker than serializability, and mechanisms that claim to achieve this weak form of consistency.

Unfortunately Schlageter's theory contains serious errors. We present counterexamples to the alleged fast serializability test, thus preserving the results of [5,6] (and leaving open the question P=NP!). We demonstrate that the weak notion of consistency admits database states that are strictly inconsistent. And we prove that the mechanism introduced to attain weak consistency is neither necessary nor sufficient for that purpose.

2. Review of Serializability Theory

Many problems in Schlageter's theory derive from imprecision in his model of computation. To verify or refute Schlageter's results, the first step is to formalize database computations with sufficient rigor. We do so by reviewing the database concurrency control theory of [1,5,6] using Schlageter's terminology.*

2.1 Terminology

The system consists of n processes, $\{P_1,\ldots,P_n\}$, and a set of data items V. Each process can perform read and/or write actions on each data item in V. We denote the processing of a read by process P_i on data item $x \in V$ by $r_i[x]$.** Similarly, a write by P_i on x is denoted $w_i[x]$. Two actions conflict if both operate on the same data item and at least one of them is a write. The set of all read and write actions for process P_i is denoted A_i, and $A_s = \bigcup_{i=1}^{n} A_i$ denotes the set of all actions for the system.

Each write operation produces a new data item value that is a function of data item values previously read by the process performing the write.

---

*This first step is inherently subjective; it is possible that our formalization misinterprets Schlageter's model. We must leave it to the reader to judge for himself the accuracy of our formalization.

**If a process, P_i, reads x twice, then the denotation of both reads is the same, i.e., $r_i[x]$. This ambiguity can be avoided by another subscript. However, since we will not encounter the ambiguity in this paper, we will keep the notation simple and ignore the problem.

The exact nature of this function is left open to arbitrary interpretations.

An execution of the system is modelled by a schedule, which, intuitively, models the exact order in which read and write actions are performed by the database management system (abbr. DBMS). Formally, a schedule is a total order $(A_s, <<)$ where $<<$ denotes the "executes-before" relation. It is convenient to write schedules as sequences of reads and writes, leaving $<<$ implicit, e.g., $S = r_1[x]\ w_2[x]\ w_1[y]$ denotes the schedule $(\{r_1[x], w_2[x], w_1[y]\}, \{(r_1[x], w_2[x]), (w_2[x], w_1[y])\})$.

To avoid dealing with the state of the database before and after the execution of a schedule, we will augment all schedules with processes P_{in} and P_{out} that produce the initial state and read the final state of the database respectively. Formally, we define an augmented schedule $S' = (A_S', <<')$ of S as follows

1. $A_S' = A_S \cup A_{in} \cup A_{out}$, where
   $A_{in} = \{w_{in}[x] \mid x \in V\}$ and
   $A_{out} = \{r_{out}[x] \mid x \in V\}$

2. $<<' = << \cup <<_{in} \cup <<_{out}$, where*
   $<<_{in} = \{(a_{in}, a_j) \mid a_{in} \in A_{in}, a_j \in A_j$ and $in \neq j\}$
   $<<_{out} = \{(a_j, a_{out}) \mid a_{out} \in A_{out}, a_j \in A_j$ and $out \neq j\}$.

Augmented schedules are a technical convenience that will simplify some of our discussion. In what follows, we assume all schedules are augmented.

---

*To be rigorous, the actions of P_{in} and P_{out} should be totally ordered within each process, although the order itself is obviously arbitrary.

2.2 Serializable Schedules

A set of processes is correctly synchronized if it is "equivalent" to a system in which the processes execute serially, i.e., one in which each process runs to completion before the next one begins [8]. In this section we formalize this notion in terms of schedules and present necessary and sufficient conditions for a schedule to be correctly synchronized. The material in this section is drawn from [1,5,6].

Two schedules are equivalent iff for every initial state of V both schedules produce the same final state of V for all interpretations of $\{P_1,\ldots,P_n\}$. Outputs to the external environment are modelled by "write-once/read-never" data items, i.e., each output is modelled by a write action $w_i[x]$ where x appears in no other action. This notion of equivalence is identical to that of [8].

To characterize the concept of equivalent schedules, we define a reads-from relation on schedule $S = (A_S, <<)$ as follows: $r_i[x]$ reads x from $w_j[x]$ in S if $w_j[x] << r_i[x]$ and there is no $w_k[x]$ with $w_j[x] << w_k[x] << r_i[x]$. Intuitively, $r_i[x]$ reads x from $w_j[x]$ in S if the value of x read by $r_i[x]$ was actually produced by $w_j[x]$.

Not every process has an effect on the final database state produced by a schedule. Those processes that do have an effect are called live and are defined as follows:

1. P_{out} is live;
2. If P_i is live and $r_i[x]$ reads x from some $w_j[x]$ in S, then P_j is live; and
3. A process is live iff it so follows by (1) and (2).

A process that is not live is called dead.

From a practical standpoint, we do not expect to see dead processes. At the very least, a process should record the fact that it ran by writing on an output device. From a theoretical standpoint, however, deadness is significant. A precise concurrency control theory must either take deadness into account or must incorporate assumptions that eliminate it.

We are now ready to characterize equivalence of schedules.

Lemma E [6] Two (augmented) schedules $S_1 = (A_S, <<_1)$ and $S_2 = (A_S, <<_2)$ are equivalent if and only if they have the same set of live processes and for each $r_i[x]$ and $w_j[x]$ in A_S such that P_i and P_j are live, if $r_i[x]$ reads x from $w_j[x]$ in S_1 then $r_i[x]$ reads x from $w_j[x]$ in S_2.

Proof This is a standard program schema theoretic result that can be proved using Herbrand interpretations for each process, e.g., see [4]. □

The schedule $(A_S, <<)$ is serial if for all $a_i \in A_i$ and $a_j \in A_j (i \neq j)$, if $a_i << a_j$ then there is no $a_i' \in A_i$ and $a_j' \in A_j$ such that $a_j' << a_i'$. A schedule is serializable (abbr. SR) if it is equivalent to a serial schedule. SR schedules are precisely those that are correctly synchronized in the sense of [8].

Unfortunately, the question "is schedule S serializable?" is NP-complete [5,6]. This result is based on a graph model of schedules we call serialization graphs. A serialization graph, $SG(S) = \langle N,E \rangle$, for schedule S is a node-labelled directed graph where:

$$N = \{P_1, \ldots, P_n\}$$

$$E = E_{\text{reads-from}} \cup E_{\text{interferes}}$$

$$E_{\text{reads-from}} = \{(P_i, P_j) \mid P_j \text{ is live and } r_j[x] \text{ reads } x \text{ from } w_i[x] \text{ for some } x \in V\} \cup \{(P_{in}, P_i) \mid i \neq in\}$$

$$E_{\text{interferes}} = \{\text{either } (P_k, P_i) \text{ or } (P_j, P_k) \mid j \neq out \text{ and for some } x \in V \; r_j[x] \text{ reads } x \text{ from } w_i[x] \text{ and } P_k \text{ writes into } x\} \cup \{(P_k, P_i) \mid \text{for some } x \in V \; r_{out}[x] \text{ reads } x \text{ from } w_i[x] \text{ and } P_k \text{ writes into } x\}$$

The edges in SG(S) reflect the notion of "happened before". $E_{\text{read-from}}$ models the reads-from relationship between processes and $E_{\text{interferes}}$ ensures that write operations do not interfere with any of these read-from relationships. Note that there are many choices of edges for $E_{\text{interferes}}$, so there are many possible serialization graphs for a given schedule.

Serialization graphs exactly characterize serializable schedules.

Theorem SR1 [5,6] A schedule S is SR iff at least one of its serialization graphs is acyclic. □

Theorem SR2 [5,6] The question "does S have an acyclic serialization graph?" is NP-complete. [5,6] □

Theorem SR2 implies that any necessary and sufficient condition for serializability must itself be NP-complete. This fact leads directly to the most serious error in [8]. By restricting the system model, however, the NP-completeness of theorem SR2 can be circumvented.

Theorem SR3 [9] If every $w_i[x]$ is preceded by $r_i[x]$, (i.e. if every process is required to read all variables that it writes), then the question "does S have an acyclic serialization graph?" can be answered in polynomial time. □

Notice, however, that many database systems do not constrain user processes in the manner required by theorem SR3 (e.g. the SDD-1 distributed database system [2] does not make this requirement). Thus the special case suggested by this theorem is not always a practical means of circumventing the previous NP-completeness result.

3. Serializability Results of [8]

The concurrency control theory of [8] is based on a notion of "precedes" (denoted $<$) that is stronger than $<<$, and the notion of "reduced dependency graphs". Let $a,b \in A_s$; Schlageter [8] defines $a < b$ iff $a << b$ and a and b conflict. In addition the reduced dependency graph, $G_s(W,U)$, of S is defined by:

$$W = \{P_i \mid P_i \text{ is a process}\} \text{ and}$$

$$U = \{(P_i,P_j) \mid \exists\, a \in A_i,\ b \in A_j : a < b \text{ in } S\}.$$

We remark that G_s is essentially the same as the "augmented ancestor graph" of [9], and is an instance of a serialization graph.

An attempt is made in section 3 of [8] to characterize SR logs using $<$ and G_s. Two theorems are presented.

Theorem 3.1 If schedule S is correctly synchronized then for all pairs of processes P_i, P_j:

$$(\exists a \in A_i,\ b \in A_j)\ (a < b) \Rightarrow \neg(\exists c \in A_i,\ d \in A_j)\ (d < c).^{*} \qquad (+)\ \square$$

Theorem 3.2 Schedule S is serializable if and only if G_S does not contain a cycle. $\square$

Both of these theorems are false, as the following counter-example proves.

---

*This theorem as it appears in print contains an obvious typographical error; the $\neg$ is omitted in the implicand. The version we state is apparently what was intended.

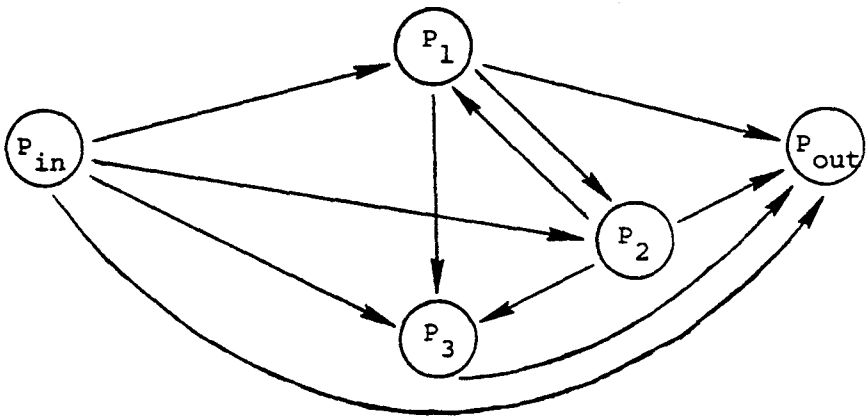
Example 1

$$S_1 = w_{in}[x]\ w_{in}[y]\ w_{in}[z]\ r_1[x]\ w_1[z]\ r_2[y]\ w_2[x]\ w_1[x]\ r_3[z]$$
$$w_3[x]\ r_{out}[x]\ r_{out}[y]\ r_{out}[z]$$

$$S_1' = w_{in}[x]\ w_{in}[y]\ w_{in}[z]\ r_1[x]\ w_1[z]\ w_1[x]\ r_2[y]\ w_2[x]$$
$$r_3[z]\ w_3[x]\ r_{out}[x]\ r_{out}[y]\ r_{out}[z]$$

Condition (+) requires that whenever two processes have conflicting actions, all pairs of conflicting actions must appear in the same order. S_1 does not satisfy condition (+) because $r_1[x] < w_2[x]$, yet $w_2[x] < w_1[x]$. Nonetheless S_1 is equivalent to S_1' which is a serial schedule. Thus S_1 is SR even though condition (+) is false, contradicting theorem 3.1.

Theorem 3.2 fails on the same example. S_1 produces the following reduced dependency graph



which contains the cycle $P_1 \rightarrow P_2 \rightarrow P_1$. But S_1 is SR. Contradiction! □

Notice that theorem 3.2 posits a computationally efficient means for testing serializability: given S, G_S can clearly be constructed in polynomial time; and given G_S, cycles can be detected in linear time. Consequently this theorem contradicts theorem SR2 (or else proves P = NP!).

A key property of schedule S_1 is that the value of x produced by $w_1[x]$ is overwritten by $w_3[x]$ before it is ever read. This situation, in which $w_1[x]$ is "dead", is what allows S_1 to arise. Note, though, that P_1 is not dead, because $w_1[z]$ is read by P_3, which is live. A more complex counterexample to theorem 3.1 with a similar deadness property appears in figure 4c of [6].

One way to avoid dead actions is to require that all processes read each data item they write before writing it. With this proviso, theorems 3.1 and 3.2 become true; indeed theorem 3.2 is then essentially identical to theorem 1 in [9]; of course, serializability is not NP-complete in this environment, as stated by theorem SR3. The treatment in [8] is not entirely explicit on this important point. When discussing the system model [8, pg. 250], the author takes care to distinguish between the data items that P_i reads (denoted I_i in [8]), those that it reads and does not write (denoted IR_i in [8]), and those that it writes (denoted O_i in [8]). The author states, "$IR_i \subseteq I_i$ ... [and] in general $I_i \cap O_i \neq \emptyset$, but $IR_i \cap O_i = \emptyset$" [8, pg. 250]. We infer from this discussion that $I_i \supseteq O_i$ is <u>not</u> intended, and therefore dead actions are possible in the model of [8].

Theorem 3.6, which is a consequence of theorem 3.2, also fails (in one direction) for the same reason as theorem 3.2. It describes a concurrency control that can produce any schedule with an acyclic reduced dependency graph. The additional claim that this includes <u>all</u> serializable schedules and therefore displays maximal parallelism is wrong, since there are serializable schedules for which G_S has a cycle (e.g. S_1).

4. Weak Consistency

Intuitively, process P_i obtains a weakly consistent view of the database if P_i forms a serializable system with each process that either reads data items that P_i writes, or writes data items that P_i reads or writes. This concept is formalized as follows: P_i obtains a weakly consistent view if for each P_j condition (+) of Theorem 3.1 holds.

The treatment of weak consistency in [8] is puzzling. Theorem 3.3 of [8] proves that weak consistency is strictly weaker than serializability, hence (we presume) unacceptable. However, Section 4 of [8] is concerned in large part with mechanisms for attaining weak consistency!

In this section, we demonstrate that "weak" consistency is a misnomer at best--a "weakly" consistent database can, in general, be inconsistent. Thus there is little motivation for attempting to attain this condition. Moreover the mechanisms proposed in [8] are neither necessary nor sufficient for this purpose.

4.1 Weak Consistency Means Inconsistency

Consider the following example.

Example 2

$$S_2 = w_{in}[x]\ w_{in}[y]\ r_1[x]\ r_2[x]\ r_2[y]\ w_2[x]\ w_2[y]\ r_1[y]$$
$$w_1[x]\ w_1[y]\ r_3[x]\ r_3[y]\ r_{out}[x]\ r_{out}[y]$$

P_3 satisfies the formal definition of weakly consistent view, but we can find an interpretation for P_1, P_2 and P_3 and a consistency criterion where P_3 obtains inconsistent data. Consider this scenario:

consistency criterion: $x = y$

Processes:	$\underline{P_1}$	$\underline{P_2}$	$\underline{P_3}$
	$r_1[x]\ x_1 := x;$	$r_2[x]\ x_2 := x;$	$r_3[x]\ x_3 := x;$
	$r_1[y]\ y_1 := y;$	$r_2[y]\ y_2 := y;$	$r_3[y]\ y_3 := y;$
	$w_1[x]\ x := 2x_1;$	$w_2[x]\ x := x_2+1;$	
	$w_1[y]\ y := 2y_1;$	$w_2[y]\ y := y_2+1;$	

Intuitively, P_3 obtains data that results from a race condition between P_1 and P_2, which in this case is inconsistent. □

4.2 Physical Locking by Value

Section 4.1 of [8] considers a database system in which each process P_i accesses only those data items that satisfy a predicate $C_i[v]$ associated with that process. To simplify the treatment we shall assume that C_i is a quantifier-free first order formula in which v is the only variable. Instead of the read action, $r_i[x]$, we have two test-and-read actions: $t_i^+[x]$ means P_i read x and $C_i(x)$ was found to be true; $t_i^-[x]$ means P_i read x and $C_i(x)$ was found to be false. Each process must first access a data item x by $t_i^+[x]$ before it can perform any writes on x.*

The set of data items that satisfy C_i in a particular state Z of V is denoted V_i^Z. Usually Z is understood to be the state seen by P_i in a particular schedule, and is therefore dropped. It is assumed that V

---

*For notational convenience we exempt P_{in} and P_{out} from this requirement. Alternatively, we could let C_{in} = true, and precede each $w_{in}[x]$ action by $t_{in}^+[x]$. Similarly we could define C_{out} = true, and replace each $r_{out}[x]$ by $t_{out}^+[x]$.

must be exhaustively searched by P_i to determine V_i.*

The definition of conflict is extended to test-and-read actions, treating $t_i^+[x]$ and $t_i^-[x]$ exactly as $r_i[x]$ in this respect. And a variation of reduced dependency graphs, called weighted dependency graph, is introduced:

> The reduced dependency graph $G_S(W,U)$ is extended to the weighted dependency graph $G'_S(W,U,c)$, where $c : U \to \{+,-\}$ is defined by
>
> $$c(P_i,P_j) = \begin{cases} -, \text{ if } (\forall a \in A_i \ \ \forall b \in A_j)\ (a < b \text{ implies } typ(a) = t^-) \\ +, \text{ otherwise.} \end{cases}$$

Intuitively, the arc (P_i,P_j) is labelled '-' if only t_i^- actions in P_i precede conflicting actions in P_j.

Weakly consistent schedules and G'_S are claimed to be related by the following theorem:

Theorem 4.2 If G'_S does not contain a cycle of length 2 with at least one arc of weight +, then V_i is weakly consistent for all processes P_i. □

This theorem is false, as the following counterexample proves.

Example 3

$$S_3 = w_{in}[x]\ w_{in}[y]\ t_1^-[x]\ t_2^-[y]\ t_2^+[x]\ w_2[x]\ t_1^+[y]\ w_1[y]\ r_{out}[x]\ r_{out}[y]$$

---

* In [8], it is assumed that only a subset $\tilde{V}_i$ of V needs be searched to determine V_i. Our treatment is a special case, hence the counter-examples that follow refute the general case as well.

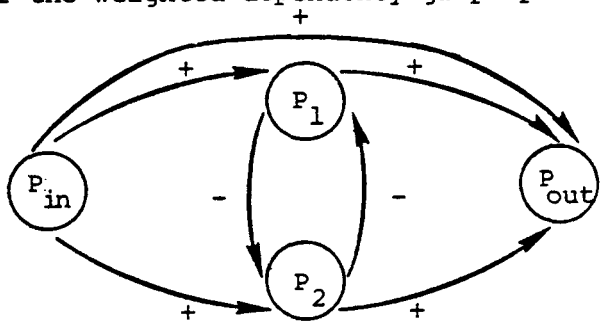
Observe that $t_1^-[x]$ conflicts with $w_2[x]$ and precedes it, so $t_1^-[x] < w_2[x]$; and $t_2^-[y]$ conflicts with $w_1[y]$ and precedes it, so $t_2^-[y] < w_1[x]$. Substituting these actions into the formula for condition (+) we find

$$(t_i^-[x] \in A_1 \wedge w_2[x] \in A_2 \wedge t_1^-[x] < w_2[x])$$

$$\Rightarrow \neg (w_1[y] \in A_1 \wedge t_2^-[y] \in A_2 \wedge t_2^-[y] < w_1[y])$$

which is false. Thus P_1 and P_2 each fail the definition of weak consistency, and neither V_1 nor V_2 is a weakly consistent view.

But consider the weighted dependency graph produced by S3.



The only cycle is $P_1 \rightarrow P_2 \rightarrow P_1$, and all arcs have weight '-'. Hence by theorem 4.2 every V_i is weakly consistent. Contradiction! □

The proof of theorem 4.2 states that if P_i and P_j are connected by a 2-edge cycle with arc weights -, then $V_i \cap V_j = \emptyset$. The conclusion is that P_i and P_j "do not interfere mutually" [8, p263]. It is unclear what this phrase means. The most plausible interpretation (to us) is : if P_i and P_j each read consistent data, then under the conditions of the theorem any database produced by their execution is also consistent. Unfortunately this interpretation is false, as example 4 illustrates.

Example 4

We can find an interpretation for processes P_1 and P_2 such that in schedule S_3, the final database state is inconsistent.

consistency criterion: $x \nrightarrow y$ $(x,y \in \{true, false\})$

values written by P_{in}: $x = false$, $y = true$

process, predicate:

P_1, $C_1(v) = v$	P_2, $C_2(v) = \neg v$
if $C_1(x)$	if $C_2(y)$
then no action;	then no action;
else if $C_1(y)$	else if $C_2(x)$
then $y := false$;	then $x := false$;

Notice that P_1, P_2 each leave x,y unchanged unless $x = false$, $y = true$;

P_1 maps $(x = false, y = true)$ into $(x = false, y = false)$,

P_2 maps $(x = false, y = true)$ into $(x = true, y = true)$.

The interleaved execution of P_1 and P_2 in S_3 maps $(x = false, y = true)$ into $(x = true, y = false)$ violating the consistency criterion. □

Example 4 exploits the fact that negative tests and positive tests have identical logical power: a negative answer to predicate C is identical to a positive answer to $\neg C$.

4.3 Synchronization by Adaptation

Section 4.2 of [8] discusses a synchronization strategy called adaptation. The system concept is based on physical locking by value, but in addition assumes that each process locks all objects it requires in a distinct lock phase before doing any processing. While the paper does not define "lock phase" precisely, apparently the following definition is intended: process P_i enters its lock phase upon issuing its first t_i^+ or t_i^- action, and

leaves its lock phase upon issuing its first w_i; the objects locked are V_i. P_i adapts to P_j is defined to mean: P_j terminates before P_i and after the termination of P_j, P_i issues $t_i^+[x]$ or $t_i^-[x]$ actions for all $x \in V_j$.

It is claimed by theorem 4.7 that synchronization by adaptation is a mechanism for attaining weakly consistent schedules.

Theorem 4.7 A schedule S is weakly consistent if and only if for all processes P_i, P_j if P_i is in its lock phase when P_j terminates, then P_i adapts to P_j. □

This theorem is false in both directions, as examples 5 and 6 demonstrate.

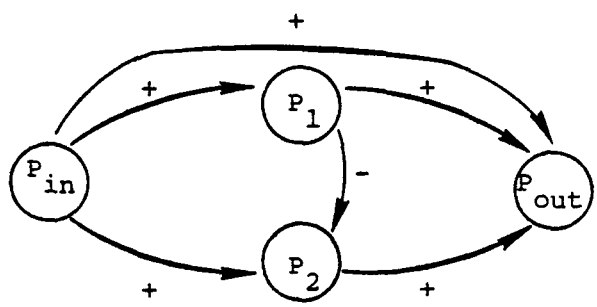
Example 5

This is a counter-example to the forward implication of the theorem:

S is weakly consistent implies $(\forall P_i, P_j)$ (if P_i is in its lock phase when P_j terminates, then P_i adapts to P_j)

$$S_5 = w_{in}[x]\ w_{in}[y]\ t_1^-[x]\ t_1^+[y]\ t_2^+[x]\ w_2[x]\ \underset{*}{\uparrow}\ w_1[y]\ r_{out}[x]\ r_{out}[y]$$

G'_{S_5} is



Since G'_{S_5} is acyclic, S_5 is certainly weakly consistent. (In fact, it is serializable.) Moreover, at time *, P_2 has terminated and P_1 is in its lock phase. Nonetheless P_1 does not re-test $V_2 = \{x\}$. □

Example 6

This is a counter-example to the reverse implication of the theorem; $(\forall P_i, P_j)(P_i$ in lock phase when P_j terminates implies P_i adapts to $P_j)$ implies S is weakly consistent.

$$S_6 = w_{in}[x]\ w_{in}[y]\ w_{in}[z_1]\ w_{in}[z_2]$$
$$t_1^-[x]\ t_2^-[y]\ t_1^+[y]\ t_2^+[x]\ t_1^+[z_1]\ t_2^+[z_2]\ \overset{*}{\uparrow}\ w_1[y]\ w_2[x]\ w_2[z_2]\ \overset{**}{\uparrow}$$
$$w_1[z_1]\ r_{out}[x]\ r_{out}[y]\ r_{out}[z_1]\ r_{out}[z_2].$$

P_2 is the first process to terminate and does so at time **; P_1 ends its lock phase at time * which is before ** Hence the requirement that P_1 adapt to P_2 if P_1 is in its lock phase when P_2 finishes is trivially satisfied. Yet S_6 is not weakly consistent because $t_1^-[x] < w_2[x] \wedge t_2^-[y] < w_1[y]$. □

Remark This counter-example depends upon the failure of theorem 4.2; G'_{S_6} contains only one cycle of length 2, and both its arcs have weight '-'.

5. Conclusion

Database consurrency control has been studied formally by several authors in recent years and a general theoretical framework is starting to evolve. Naturally, as new work appears it must be compared to this general theory and absorbed, modified, or rejected. When [8] appeared in print, we attempted this comparison and discovered the errors noted in this paper.

We believe the fundamental cause for these errors to be the weakness of the schedule concept in [8]. Schedules are defined there solely in terms of the partial order $<$, rather than the total order $<<$. Notice that $<$ combines the concepts of "precedes in time" and "conflicts", although these concepts are qualitatively different: "conflicts" is a property of actions independent of a particular execution, while "precedes" depends solely upon execution order. By combining these two concepts, $<$ confuses the description of processes with their instances; and by building schedules on top of $<$, the theory is left with a confused and unwieldy model of computation.

The errors resulting from this combination of 'schedule' and 'conflict' reaffirm our belief that the distinction between these concepts and the formalisms developed around them are major methodological contributions of serializability theory.

REFERENCES

1. Bernstein, P. A., D. W. Shipman, and W. S. Wong, "Formal Aspects of Serializability in Database Concurrency Control", IEEE Trans. on Software Eng., Vol. SE-5, No. 3 (May '79), pp. 203-216.

2. Bernstein, P.A., J. B. Rothnie, N. Goodman, and C. H. Papadimitriou, "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", IEEE Trans. on Software Eng., Vol. SE-4, No. 3 (May '78), pp. 154-168.

3. Eswaran, K. P., J. N. Gray, R. A. Lorie, I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Databse System", Comm. ACM Vol. 19, No. 11, November 1976.

4. Manna, Z., Mathematical Theory of Computation, McGraw-Hill, N.Y., 1974.

5. Papadimitriou, C. H., "Serializability of Concurrent Updates", Technical Report TR-14-78, Center for Research in Computing Technology, Harvard University, to appear in Jour. of the ACM.

6. Papadimitriou, C. H., P. A. Bernstein, and J. B. Rothnie, "Some Computational Problems Related to Database Concurrency Control", Proc. Conference on Theoretical Computer Science, University of Waterloo, 1977, pp. 275-282.

7. Rosenkrantz, D. J., R. E. Stearns, and P. M. Lewis II, "System Level Concurrency Control for Distributed Database Systems", ACM Trans. on Database Sys., Vol. 3, No. 3 (June '78), pp. 178-198.

8. Schlageter, G., "Process Synchronization in Database Systems", ACM Trans. on Database Sys., Vol. 3, No. 3 (Sept.'78), pp. 248-271.

9. Stearns, R. E., D. J. Rosenkrantz and P. M. Lewis II, "Concurrency Control for Database Systems", IEEE Proc. 1976 Symp. on Foundations of Computer Science, pp. 19-32.