Towards a unified theory of concurrency control and recovery

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Informationssysteme

Hans-Jörg Schek
Gerhard Weikum
Haiyan Ye

Towards a Unified Theory of Concurrency Control and Recovery

December 1992

190

ETH Zürich
Departement Informatik
Institut für Informationssysteme
Prof. Dr. Hans-Jörg Schek

Author's address:

Institut für Informationssysteme
ETH Zentrum, CH-8092 Zurich

e-mail: {schek,weikum,ye}@inf.ethz.ch

Towards a Unified Theory of Concurrency Control and Recovery

Hans-Jörg Schek, Gerhard Weikum, Haiyan Ye
Department of Computer Science
ETH Zurich
CH-8092 Zurich, Switzerland

December 1992

Abstract

The theory of transaction management is based on two different and independent criteria for the correct execution of transactions. The first criterion, serializability, ensures correct execution of parallel transactions under the assumption that no failures occur. The second criterion, strictness, ensures correct recovery in case of failures.

In this paper we develop a unified model that allows reasoning about the correctness of concurrency control and recovery within the same framework. We introduce several correctness criteria and show their relationships to the traditional notions.

1 Introduction

The theory of transaction management is based on two different and independent criteria for the correct execution of transactions [4]. The first criterion is serializability for the correct execution of parallel transactions under the (implicit) assumption that no failures occur. The second criterion complements the first one and deals with correct recovery in case of a failure. Transaction management must be implemented in a way that both correctness criteria are satisfied. As shown in [4, 9], there exist schedules which are serializable but cannot be correctly recovered from a failure. There are also schedules which satisfy the criterion for correct recovery but are not serializable. Rather than dealing with two incomparable criteria, we would prefer a single correctness criterion which ensures the correctness of parallel executions even in the presence of failures.

In this paper we develop a unified model that allows reasoning about the correctness of concurrency control and recovery within the same framework. We introduce *one* correctness criterion called **prefix-reducibility** and show how it is related to the traditional definitions. It is shown that the class of prefix-reducible schedules is less restrictive than the class of serializable and strict schedules, which is usually considered as the standard class of acceptable schedules. We also introduce a constructive correctness criterion called **expanded serializability** that allows efficient correctness testing based on serialization graph methods.

The unified model that is proposed in this paper builds on our previous work [3, 20], where we introduced the notion of "complete serializability". The basic idea of this notion is to represent all recovery-related actions explicitly in the schedule and to use serializability arguments to reason about such "completed schedules". In the current paper, we rename the notions of "completed schedules" and "complete serializability" by "expanded schedules" and "expanded serializability", to avoid confusion with the unrelated term "complete history" of [4]. As far as the correctness of recovery is concerned, our previous work stayed informal and we did not elaborate the relationships between our notion of "complete serializability" and the classical correctness criteria.

Recently, the notion of rigorousness has been proposed as a unified correctness criterion [5]. However, the class of rigorous schedules is even more restrictive than the class of serializable and strict schedules. The only other work along the lines of "complete serializability" are [14], [17], and [19]. However, these models are unnecessarily complex. We claim that prefix-reducibility and expanded serializability are simpler criteria that capture the intuition and are easy to deal with in a formal model. Another important advantage of our model is that it can be generalized to cover also non-conventional recovery paradigms like compensation of high-level actions. In this paper, however, we restrict ourselves to the classical setting of read and write actions.

The paper is organized as follows. Section 2 summarizes the standard terminology of the traditional transaction model. In Section 3 we present our unified model. In Section 4 we conclude with a brief outlook on further extensions of our model.

2 The Traditional Notions of Serializability and Correct Recovery

2.1 Standard Terminology

The database is a set of data objects. Actions on objects are reads and writes. Read access and write access to object x by transaction T_i are denoted by $r_i[x]$ and $w_i[x]$, respectively. The commit or abort of transaction T_i is abbreviated as c_i or a_i, respectively. Each transaction must have a termination action, which is either a commit or an abort action, following all other actions of the transaction. A partial order $<_i$ specifies the execution order of actions in a transaction.

Two actions $op_i[x]$ and $op_j[x]$ are in *conflict*, if they operate on the same object and at least one of them is a write. Otherwise they are *non-conflicting*. We say that *non-conflicting* actions are commutable.

A *schedule*, called a *history* in [4], is a pair (A, <) where A is the union of the actions of all submitted transactions and the partial order < specifies the execution order of these actions. For each transaction T_i the order $<_i$ must be preserved in the schedule (i.e., $<_i \subseteq <$). Also, all conflicting actions in the schedule must be <-ordered.

The committed projection C(S) of schedule S is obtained from S by deleting all actions that do not belong to committed transactions in S. A *complete schedule* is a schedule in which all transactions are terminated (i.e., committed or aborted).

2.2 Correct Concurrency Control

We take the basic definitions on (conflict-)serializability from [4].

Definition 2.1 (Serializability (SR))
A schedule S is serializable (SR), if its committed projection C(S) is equivalent to a serial schedule S', where equivalence means that all pairs of conflicting actions appear in the same order in both schedules. □

The well-known fundamental theorem of serializability theory is

Theorem 2.1 *A schedule S is SR iff the serialization graph SG(C(S)) is acyclic.* □

Note that it is important to distinguish between a schedule and its committed projection. The committed projection takes only committed transactions into consideration. Aborted transactions are disregarded. Therefore, additional correctness rules for recovery must be introduced.

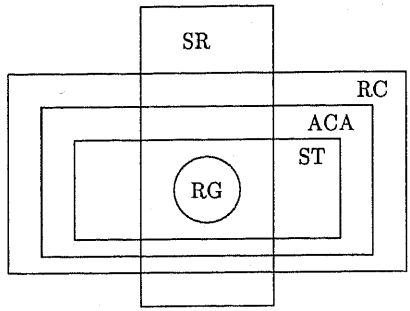

Figure 1: Relationships between RG, ST, ACA, RC and CPSR

2.3 Correct Recovery

We review the definitions of strictness (ST) and recoverability (RC) from [4]. We also recall the notion of rigorousness from [5], and point out the relationships between these properties.

A schedule S is called *recoverable* (RC), if each transaction commits after the commitment of all transactions (other than itself) from which it read.

A schedule S is *strict* (ST), if whenever $w_j[x] < o_i[x]$ $(i \neq j)$, then either $a_j < o_i[x]$ or $c_j < o_i[x]$ where $o_i[x]$ is $r_i[x]$ or $w_i[x]$. More intuitively, no object may be read or overwritten until all transactions that previously wrote into it terminate (i.e., abort or commit).

A schedule is *rigorous* (RG), if it guarantees strictness and in addition, no object can be overwritten until all transactions that previously read it terminate (i.e., abort or commit).

It has been shown [4, 5] that RG $\subset$ ST $\subset$ RC. The class SR, on the other hand, is incomparable to these classes, except that RG $\subset$ SR (see Figure 1). For example the schedule $S_1 =< r_j[x]\ w_i[x]\ w_i[y]\ c_i\ r_j[y]\ c_j >$ is in ST but not in SR and $S_2 =< w_i[x]\ r_j[x] w_i[y]\ c_i\ r_j[y]\ c_j >$ is in SR but not in ST.

3 The Unified Model

In this section, we present our model that aims to unify the concepts of serializability and correct recovery. The rationale of the model is based on the following two ideas:

1. All recovery-related actions are made explicit and represented in the same history of events that represents the execution order of regular actions.
2. Then, serializability arguments are used to reason about the correctness of action histories including the recovery actions, and, especially, about the interference of recovery-related and regular actions.

We restrict ourselves to modeling transaction aborts in concurrent transaction executions. Crash recovery is disregarded in this paper; our long-term goal is to extend the model so that crash recovery is included.

We assume that recovery is based on logging rather than deferred update. So, transaction aborts are implemented by undoing the write actions of the transaction, whereas transaction commits do not require any recovery actions. Note that virtually all commercial systems employ such log-based recovery method [7, 13], whereas deferred update methods have not been adopted in practice.

3.1 Expanded Schedules

In this section, we elaborate the principle of making recovery-related actions explicit. The idea is to replace transaction aborts by a sequence of *undo actions*. For a write action $w_i[x]$, the undo action is again a write to the same object, denoted as $w_i^{-1}[x]$, where the restored value of x is the old value of x that was overwritten by the original $w_i[x]$ action.

For read actions, no undo action is necessary. However, read actions of aborted transactions can be regarded as null actions in the sense that they are rendered meaningless by the abort. In particular, a transaction may be aborted because the system can no longer guarantee a serializable view for this transaction. In this case, the transaction may have read seemingly inconsistent data, and the read data is declared invalid, in retrospect, by the abort.

Each undo action undoes one of the transaction's original "forward" actions, and the order of undo actions is the reverse order of the original actions. Once an aborted transaction is expanded in this manner, we can as well consider the transaction as committed, since it leaves no effect on the database.

These expansion rules become slightly more complicated when we consider incomplete schedules, that is, schedules that contain "active" transactions without termination action. Such schedules occur as prefixes of complete schedules. Since a dynamic scheduler needs to make its decisions based on knowing only a prefix of a complete schedule, it is important to consider incomplete schedules.

We interpret an incomplete schedule as if a system crash occurred after the last action. So we assume that all active transactions of the schedule are implicitly aborted. Again, we need to make these implicit aborts explicit and represent the necessary undo actions in the schedule. The obvious idea to do this is to extend the schedule by an abort action for each active transaction, and to expand these aborts as in a complete schedule. The problem with this idea is that it is not clear how the additional aborts should be ordered in the schedule. In fact, it is possible that the aborts cannot be properly ordered at all. Consider, for example, the incomplete schedule

$w_1[x]\ w_2[x]\ w_2[y]\ w_1[y]$.

The schedule contains a conflict cycle, but since none of the two transactions is committed, this situation is perfectly acceptable so far. Now, if we add abort actions a_1 and a_2 for the two transactions, then no serial order of a_1 and a_2 can achieve correct recovery. The expansion of

$w_1[x]\ w_2[x]\ w_2[y]\ w_1[y]\ a_1\ a_2$.

would be

$w_1[x]\ w_2[x]\ w_2[y]\ w_1[y]\ \mathbf{w_1^{-1}[y]\ w_1^{-1}[x]\ c_1\ w_2^{-1}[y]\ w_2^{-1}[x]\ c_2}$.

which is intuitively incorrect, and the expansion of

$w_1[x]\ w_2[x]\ w_2[y]\ w_1[y]\ a_2\ a_1$.

would be

$w_1[x]\ w_2[x]\ w_2[y]\ w_1[y]\ \mathbf{w_2^{-1}[y]\ w_2^{-1}[x]\ c_2\ w_1^{-1}[y]\ w_1^{-1}[x]\ c_1}$.

which is incorrect, too. The situation can be resolved only if we allow the undo actions of multiple aborts to be interleaved. The correct ordering of the undo actions is again the reverse order of the original actions that are undone, but in this case the order is not just within one transaction. Rather the order of the undo actions for non-terminated transactions is determined globally. So we basically emulate a global undo procedure for all non-terminated transactions. In fact, this is exactly what we would intuitively expect a crash recovery method to do if a crash occurred right after the original schedule's last action. For the above example schedule, the intuitively correct expansion is

$w_1[x]\ w_2[x]\ w_2[y]\ w_1[y]\ \mathbf{w_1^{-1}[y]\ w_2^{-1}[y]\ w_2^{-1}[x]\ w_1^{-1}[x]}$.

These informal considerations are captured by the following definition.

Definition 3.1 (Expanded Schedule $\tilde{S}$)

Let $S = (A, <)$ be a (possibly incomplete) schedule. The **expansion** $\tilde{S}$ *of S, also denoted as the* **expanded schedule** $\tilde{S}$*, is a quadruple $(\tilde{A}, \tilde{<}, UNDO, NULL)$ where:*

1. *$\tilde{A}$ is a set of actions which is derived from A in the following way:*
   (a) *For all transactions T_i that are committed in S, $\widetilde{A_i}$ is the action set A_i of T_i in S.*
   (b) *For all transaction T_i that are aborted in S, $\widetilde{A_i}$ is obtained from A_i by adding an undo action $w_i^{-1}[x]$ for each regular write action $w_i[x]$ in A_i, and replacing the abort action of T_i by a commit action.*
   (c) *For all transactions T_i that are not terminated in S, $\widetilde{A_i}$ is obtained from A_i by adding an undo action $w_i^{-1}[x]$ for each regular write action $w_i[x]$ in A_i, and adding a commit action.*
   (d) *$\tilde{A}$ is the union of the action sets $\widetilde{A_i}$ for all transactions T_i in S.*
2. *$\tilde{<}$ is a partial order on $\tilde{A}$ such that:*
   (a) *For the original A actions, the order $\tilde{<}$ is identical to $<$.*
   (b) *All additional undo actions of a transaction follow the transaction's original actions and precede the transaction's commit action.*
   (c) *All additional undo actions of originally non-terminated transactions follow all original actions of the schedule and precede all commit actions of the originally non-terminated transactions.*

(d) *The ordering between the additional undo actions is the reverse order of the corresponding original actions.*

3. *UNDO contains all pairs* $(w_i[x], w_i^{-1}[x]) \in \tilde{A} \times \tilde{A}$, *where* $w_i^{-1}[x]$ *is the undo action of* $w_i[x]$.
4. *NULL is the set of read actions of transactions that are aborted in S or not terminated in S.* □

Example 3.1
$S = r_1[x]\ w_1[x]\ r_2[y]\ w_2[y]\ r_3[z]\ w_3[z]\ r_4[y]\ w_4[y]\ a_1\ c_3\ r_2[z]\ w_2[z].$
$\tilde{S} = r_1[x]\ w_1[x]\ r_2[y]\ w_2[y]\ r_3[z]\ w_3[z]\ r_4[y]\ w_4[y]\ \mathbf{w_1^{-1}[x]}\ \mathbf{c_1}\ c_3\ r_2[z]\ w_2[z]$
$\mathbf{w_2^{-1}[z]}\ \mathbf{w_4^{-1}[y]}\ \mathbf{w_2^{-1}[y]}\ \mathbf{c_2}\ \mathbf{c_4}.$
with UNDO $= \{< w_1[x], w_1^{-1}[x] >, < w_2[y], w_2^{-1}[y] >< w_2[z], w_2^{-1}[z] >, < w_4[y], w_4^{-1}[y] >\}.$
and NULL $= \{r_1[x],\ r_2[y],\ r_2[z],\ r_4[y]\}.$ □

3.2 Correctness Criteria

In this section, we elaborate the idea of using a unified correctness criterion for expanded schedules. Intuitively, correctness of an expanded schedule requires

1. that the write actions of aborted transactions are undone (i.e., failure atomicity),
2. that the subschedule formed by the committed transactions is serializable (i.e., isolation), and
3. that the undo actions of aborted transactions do not incorrectly interfere with the committed transactions (i.e., isolation of recovery).

The first condition is self-guaranteed since the expansion of a schedule adds the proper undo actions. The second and third condition boil down to requiring serializability for the expanded schedule, that is, for both committed and aborted transactions where aborted transactions include their undo actions. This consideration motivates our first correctness criterion.

Definition 3.2 (Expanded Serializability (XSR)) .
Let $S = (A, <)$ *be a (possibly incomplete) schedule. S is* **expanded-serializable (XSR)** *if the expanded schedule* $\tilde{S}$ *is serializable.* □

Example 3.2
$S_1 = r_1[x]\ w_1[x]\ r_2[x]\ a_1\ c_2$; $S_2 = r_1[x]\ w_1[x]\ a_1\ r_2[x]\ c_2.$
$\widetilde{S_1} = r_1[x]\ w_1[x]\ r_2[x]\ \mathbf{w_1^{-1}[x]}\ \mathbf{c_1}\ c_2$; $\widetilde{S_2} = r_1[x]\ w_1[x]\ \mathbf{w_1^{-1}[x]}\ \mathbf{c_1}\ r_2[x]\ c_2.$
S_1 is not XSR, S_2 is XSR □

Expanded serializability is a simple criterion that aims to unify the correctness criteria of concurrency control and recovery. Unfortunately, there are schedules for which expanded serializability is unnecessarily restrictive. Consider, for example, the schedule

$S_1 = r_1[x]\ w_1[x]\ r_2[x]\ w_2[x]\ a_2\ a_1.$

Its expanded schedule is

$r_1[x]\ w_1[x]\ r_2[x]\ w_2[x]\ \mathbf{w_2^{-1}[x]\ c_2\ w_1^{-1}[x]\ c_1}.$

This expanded schedule is not serializable. However, it is perfectly valid to view the schedule as correct from both the concurrency control and the recovery perspective. One may assume that the conflict cycle in the expanded schedule is irrelevant since both transactions are aborted anyway. In this case, this assumption is indeed justified. However, it is not valid to disregard all conflict cycles among aborted transactions. Consider the schedule

$S_2 = r_1[x]\ w_1[x]\ r_2[x]\ w_2[x]\ a_1\ a_2.$

The expanded schedule is

$r_1[x]\ w_1[x]\ r_2[x]\ w_2[x]\ \mathbf{w_1^{-1}[x]\ c_1\ w_2^{-1}[x]\ c_2}.$

In this case, the conflict cycle is relevant. The problem is that the undo actions of the two transactions interfere incorrectly. The undo action of T_2, $w_2^{-1}[x]$, would restore the value of x as of the time when the original $w_2[x]$ was executed. This would erroneously restore the after-value of T_1 for object x. Since the abort of T_1 precedes the abort of T_2, the overall effect would be that the update of T_1 remains in the database.

The essential difference between the two schedules S_1 and S_2 is that in S_1 the undo actions of one transaction, T_2, are adjacent (with respect to $\tilde{<}$) to the original actions of T_2, which is not the case in S_2. The adjacency of an original "forward" action and the corresponding undo action in S_1 means that the original action is effectively undone so that both the original and the undo action can be removed from the schedule before we further consider the serializability of the "remainder" schedule.

The adjacency of original action and undo action is not the weakest condition to guarantee correct behavior. Rather it is sufficient that there is no conflicting action between the original action and its corresponding undo action. These considerations finally lead to the following correctness criterion.

Definition 3.3 (Reducibility (RED)) .

A schedule $S = (A, <)$ is **reducible (RED)** *if its expanded schedule $\tilde{S} = (\tilde{A}, \tilde{<}, UNDO, NULL)$ can be transformed into a serial schedule by applying the following three transformation rules finitely many times.*

1. **Commutativity Rule:** *If $p, q \in \tilde{A}$ such that $p \tilde{<} q$ and (p, q) are not in conflict and there is no other action $o \in \tilde{A}$ with $p \tilde{<} o \tilde{<} q$, then the ordering $p \tilde{<} q$ can be replaced by $q \tilde{<} p$.*
2. **Undo Rule:** *If $p, q \in \tilde{A}$ such that $p \tilde{<} q$ and $(p, q) \in UNDO$ and there is no other action $o \in \tilde{A}$ with $p \tilde{<} o \tilde{<} q$, then p, q can be omitted from the schedule.*

3. **Null Action Rule:** *If $p \in \tilde{A}$ and $p \in NULL$, then p can be omitted from the schedule.* □

The Null Action Rule means that it is valid to disregard all read actions of aborted transactions. The intuition behind this rule is that the values that were "seen" by these reads are rendered meaningless by the abort of the transaction (see also Section 3.1 above).

Example 3.3
$S = r_1[x]\ w_1[x]\ r_2[x]\ w_2[x]\ a_2\ a_1.$
$\tilde{S} = r_1[x]\ w_1[x]\ r_2[x]\ w_2[x]\ \mathrm{w}_2^{-1}[\mathrm{x}]\ \mathrm{c}_2\ \mathrm{w}_1^{-1}[\mathrm{x}]\ \mathrm{c}_1.$
$\tilde{S}$ can be reduced to the serial schedule $S' = c_2\ c_1$ of "empty", that is, effect-free transactions by applying the Null Action Rule to the read actions of aborted transactions and applying the Undo Rule twice. □

To develop real schedulers, it is necessary to make decisions based on the knowledge of a prefix of a schedule [4, 16]. This requirement is taken into account by our final correctness criterion. Consider, for example, the schedule $w_1[x]\ w_2[x]\ c_2\ c_1$. This schedule is serializable, expanded-serializable, and also reducible. However, a scheduler that has processed the prefix $w_1[x]\ w_2[x]\ c_2$ should be careful about allowing transaction T_2 to commit, because the continuation of this prefix is only correct if T_1 is going to commit, too. If, on the other hand, T_1 is going to abort, then the schedule would no longer be expanded-serializable nor reducible. The expansion of the above prefix is $w_1[x]\ w_2[x]\ c_2\ w_1^{-1}[x]\ c_1$ and it can be easily seen that this expanded prefix cannot be transformed into a serial schedule by using the three rules of Definition 3.3. So, our final correctness criterion, which takes these considerations into account, is the following.

Definition 3.4 (Prefix-Reducibility (PRED)) .
A schedule $S = (A, <)$ is **prefix-reducible (PRED)** *if every prefix of S is reducible.* □

3.3 Results

In this section, we elaborate the relationships between the traditional correctness criteria for concurrency control and recovery and the newly introduced correctness criteria of Section 3.2. As already pointed out, our aim is to express correct recovery in terms of serializability applied to an expanded schedule that includes the undo actions. We denote by SR-ST the class of schedules that are serializable and strict, by SR-RC the class of schedules that are serializable and recoverable. These classes are compared to our new classes XSR, RED, and PRED defined in Section 3.2.

3.3.1 Serialization Graph Testing

In this part we investigate how far we are able to check serializability and correct recovery simply by testing whether the serialization graph (SG) of the expanded schedule $\widetilde{S}$ is acyclic. If successful, this approach would lead to a simple and constructive way of testing correctness. All practical techniques ensuring that cycles in the serialization graph are avoided would be applicable.

For the following we must carefully distinguish between a schedule S (that may contain committed, aborted, and active transactions), its committed projection $C(S)$ (that contains only committed transactions), and the expanded schedule $\widetilde{S}$ (that expands S by undo actions for aborted and for active transactions). Note further that the serialization graph solely depends on the notion of conflicting actions and is therefore defined for all kinds of schedules. Our first, fairly obvious result is the following:

Lemma 3.1 *1. $SG(C(S)) \subseteq SG(S) \subseteq SG(\tilde{S})$; 2. $XSR \subset SR$.* □

Proof: The schedule C(S) is a sub-schedule of S, and S is a sub-schedule of $\widetilde{S}$. By dropping actions in a schedule, we can only *lose* conflict pairs. This proves the inclusion relationship between the three serialization graphs. Consequently, XSR $\subset$ SR holds, too. To see that this inclusion is proper, consider the schedule $S = w_1[x]\ w_2[x]\ c_2\ a_1$, which is in SR, but not in XSR. □

It is known that a rigorous schedule S is also serializable. Even more, its serialization graph SG(S), not only SG(C(S)), must not be cyclic, as the following lemma shows.

Lemma 3.2 *A rigorous schedule S has an acyclic serialization graph SG(S).* □

Proof: Assume that SG(S) has a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_k \rightarrow T_1$. Because S is RG, any two conflicting actions p_i and q_j of two transactions T_i and T_j in S are separated by a termination action t_i which is either a c_i or an a_i. So we have $p_i < t_i < q_j$ in S. Therefore the cycle above leads to $t_1 < t_2 < \ldots t_k < t_1$ which is impossible. It shows that SG(S) of a rigorous schedule S is acyclic. □

Rigorousness is a rather strong criterion. Therefore we are interested in a weaker criterion which nevertheless ensures correctness for recovery and concurrency.

Theorem 3.1 *If a schedule S is strict and its serialization graph SG(S) is acyclic, then S is expanded-serializable.* □

Proof: If S does not contain any aborted transactions, then clearly S = $\widetilde{S}$ and SG(S) = SG($\widetilde{S}$) is acyclic, i.e., S is in XSR. In the more interesting case S contains one or more aborted transactions. Assume that S is not in XSR, i.e., SG($\widetilde{S}$) has a cycle. $\widetilde{S}$ includes all actions of S. Since SG(S) is acyclic, the cycle in SG($\widetilde{S}$) must be introduced by at least one new edge (T_i, T_k) due to an undo action p_i^{-1} of an aborted transaction T_i. Given any pair

of conflicting actions q_k and p_i, the cases $q_k < p_i < p_i^{-1}$ or $p_i < p_i^{-1} < q_k$ do not lead to a new edge because q_k conflicts already with p_i. The only case left is $p_i < q_k < p_i^{-1}$. However this case is impossible, because p_i is a write action and so it contradicts the premise that S is strict, which requires a termination action t_i of T_i before q_k. □

A corollary of Lemma 3.1 and 3.2 is the following

Lemma 3.3 *A rigorous schedule is expanded-serializable.* □

In view of these results we are interested in whether also the class SR-ST, not only RG could be subsumed by XSR. If so we could have replaced the combined criterion SR-ST by XSR and reached our objective. Unfortunately this is not the case. The following schedules, S_1 and S_2, show that SR-ST and XSR are incomparable, that is, neither of them is contained in the other.

$S_1 = r_1[x]\ w_2[x]\ r_2[y]\ w_1[y]\ a_2\ c_1$ is SR-ST but not XSR.

Clearly $C(S_1) = r_1[x]\ w_1[y]\ c_1$ is SR, and S_1 is strict because no uncommitted objects are read or overwritten. Nevertheless $\widetilde{S_1} = r_1[x]\ w_2[x]\ r_2[y]\ w_1[y]\ w_2^{-1}[x]\ c_2\ c_1$ is not SR. Note that already SG(S_1) must have a cycle (in accordance with Lemma 3.1). Therefore it is questionable whether we should consider such a schedule at all. Notice that already before $w_1[y]$ was executed it was clear that the schedule S_1 can only be turned into a correct one by either aborting T_1 or T_2.

$S_2 = w_1[x]\ w_2[x]\ w_1[y]\ w_2[y]\ a_2\ c_1$ is XSR but not SR-ST.

The schedule S_2 shows that we "lose" correct schedules, because

$\widetilde{S_2} = w_1[x]\ w_2[x]\ w_1[y]\ w_2[y]\ w_2^{-1}[y]\ w_2^{-1}[x]\ c_2\ c_1$

is SR and equivalent to a serial execution of T_1 followed by T_2. However, S_2 is not strict. Note that in this case SG(S_2) is acyclic.

The following schedule S_3 shows that RG is weaker than the class of strict schedules S with acyclic SG(S):

$S_3 = r_1[x]\ w_2[x]\ r_2[y]\ w_3[y]\ c_3\ a_2\ c_1$

is in ST but not in RG. The graph SG(S) is acyclic. More generally we have

Theorem 3.2 *If a schedule S is strict, then S is XSR if and only if SG(S) is acyclic.* □

Proof: The 'if'-part has been shown in Theorem 3.1. The 'only-if' part follows from Lemma 3.1 because SG(S) cannot have a cycle if SG($\widetilde{S}$) has no cycle. □

Let us summarize the results so far:

- RG ⊂ (SG(S) acyclic and ST) ⊂ XSR.
- SR-ST and XSR are incomparable.
- A strict schedule is XSR iff SG(S) is acyclic. □

While XSR is a constructive criterion, the disadvantage is that schedules such as S_1 are not captured. S_1 is correct in the sense of SR-ST but not according to XSR.

3.3.2 Reducibility is Correct Concurrency Control and Recovery

The problem that is not handled by SR-ST nor by XSR are schedules of the form

$S_4 = w_1[x]\ w_2[x]\ a_2\ c_1.$

This schedule is not in ST and is not in XSR. The expanded schedule is

$\widetilde{S_4} = w_1[x]\ w_2[x]\ w_2^{-1}[x]\ c_2\ c_1.$

The serialization graph of $\widetilde{S_4}$ has a cycle due to the undo action. For such cases we need a reduction of a schedule which drops aborted transactions whenever possible. This is what the criteria RED and PRED do.

It is obvious that reducibility of all prefixes of a schedule implies the reducibility of the schedule itself. However, reducibility of the schedule does not imply reducibility of all prefixes, as the following schedule shows.

$S_5 = w_1[x]\ w_2[x]\ c_2\ c_1.$

S_5 is reducible; however, the prefix that excludes c_1 is not reducible. Thus we have the following lemma:

Lemma 3.4 *PRED* $\subset$ *RED*

Let us now compare the criterion XSR with RED and PRED. The following lemma states the relationships between XSR and the other two criteria.

Lemma 3.5

1. *If a schedule is XSR, then it is also reducible.*
2. *The class XSR is a proper subset of the class of RED of reducible schedules.*
3. *XSR is not contained in PRED.* □

Proof: If S is in XSR, we know that SG($\tilde{S}$) is acyclic. Therefore, a serial schedule $\widetilde{S'}$ exists into which $\tilde{S}$ can be transformed. This shows that S is reducible and proves claim 1. The example schedule S_4 shows that RED is a proper supset of XSR (claim 2). In order to prove claim 3 consider S_5,which is in XSR but not in PRED. □

So, while XSR is included in RED, it is incomparable to PRED. However, we can show that the class SR-ST is a subset of PRED, which is the first main result of this section.

Theorem 3.3 *If a schedule is serializable and strict, then it is also prefix-reducible.*

Proof: Consider a prefix s of a schedule S which is in SR-ST. It follows from the definition of SR-ST that s is also SR and ST. Now consider the expansion $\tilde{s}$ of s. $\tilde{s}$ cannot be transformed to a serial schedule. Since s is serializable, the problem must come from undo actions in $\tilde{s}$. There must be a pair of UNDO-related actions $w_i[x]$ and $w_i^{-1}[x]$ which cannot be made adjacent w.r.t. $\tilde{<}$ in any transformation of $\tilde{s}$. The reason for this must be that there is an action p such that $w_i[x] \tilde{<} p \tilde{<} w_i^{-1}[x]$ and p is conflicting with $w_i[x]$ and $w_i^{-1}[x]$.

There are three cases for this scenario.

- Case 1: $w_i[x] \tilde{<} r_j[x] \tilde{<} w_i^{-1}[x]$ with $j \neq i$.
  T_j must be committed; otherwise, its read actions would have been omitted from $\tilde{s}$. This case is impossible because s is strict.
- Case 2: $w_i[x] \tilde{<} w_j[x] \tilde{<} w_i^{-1}[x]$ with $j \neq i$.
  This again is impossible because s is strict.
- Case 3: $w_i[x] \tilde{<} w_j^{-1}[x] \tilde{<} w_i^{-1}[x]$ with $j \neq i$.
  In this case, $\tilde{s}$ must also contain a corresponding forward action $w_j[x]$, preceding $w_j^{-1}[x]$. Because of the strictness of s, $w_i[x] \tilde{<} w_j[x]$ cannot hold (see Case 2 above). However, $w_j[x] \tilde{<} w_i[x]$ cannot hold either because of the same (symmetric) argument. Since the two conflicting actions must be $\tilde{<}$-ordered, this case is impossible. □

The converse of Theorem3.3 does not hold. If S is reducible to a serial schedule, then S is not necessarily serializable and strict. The schedule $S_6 = w_1[x]\ w_2[x]\ a_2\ a_1$ shows that SR-ST is a proper subclass of PRED. The expanded schedule is $\widetilde{S_6} = w_1[x]\ w_2[x]\ w_2^{-1}[x]\ c_2\ w_1^{-1}[x]\ c_1$, which is reducible, yet S_6 is not strict.

The following theorem is the second main result of this section.

Theorem 3.4 *PRED* $\subset$ *SR-RC*

Proof: Assume that S is not serializable. Then, we must have a conflict cycle of the form $T_1, T_2, \ldots, T_n, T_1$ where all transactions are committed. It follows that $\tilde{S}$ cannot be reducible. Assume that S is not recoverable. This can occur because of one of the following three cases.

- Case 1: $w_i[x] < r_j[x] < c_j < c_i$.
  The prefix of S that excludes c_i is not reducible to a serial schedule, because the expansion of the prefix contains the ordering $w_i[x] \tilde{<} r_j[x] \tilde{<} w_i^{-1}[x]$.
- Case 2: $w_i[x] < r_j[x] < c_j < a_i$.
  The expansion of S is not reducible to a serial schedule, because it contains the ordering $w_i[x] \tilde{<} r_j[x] \tilde{<} w_i^{-1}[x]$.
- Case 3: $w_i[x] < r_j[x] < a_i < c_j$.
  Again, the expansion of S is not reducible to a serial schedule, because it contains the ordering $w_i[x] \tilde{<} r_j[x] \tilde{<} w_i^{-1}[x]$. □

If S is serializable and recoverable, then S is not necessarily reducible, as shown by the example

$S = w_1[x]\ w_2[x]\ a_1\ a_2$.

The expanded schedule is

$\tilde{S} = w_1[x]\ w_2[x]\ w_1^{-1}[x]\ w_2^{-1}[x]$

which is not reducible.

We can further observe that RED and SR-RC are incomparable.

The following schedule is reducible but not recoverable

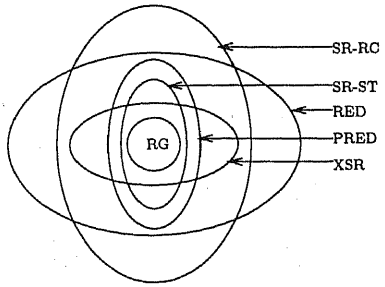

Figure 2: Relationships between various classes

$S_7 = w1[x]\ r2[x]\ c2\ c1$

S_7 is reducible to a serial schedule with T_1 preceding T_2 but S_7 is not recoverable because the uncommitted x is read. On the other hand, the schedule

$S_8 = w1[x]\ w2[x]\ a1\ a2$

is recoverable and serializable but not reducible, because the expanded schedule $\widetilde{S_8} = w_1[x]\ w_2[x]\ w_1^{-1}[x]\ w_2^{-1}[x]$ cannot be transformed to any serial schedule.

Let us again summarize our results:

- XSR $\subset$ RED.
- SR-ST $\subset$ PRED.
- PRED $\subset$ SR-RC.
- PRED $\subset$ RED.
- PRED and XSR are incomparable.
- RED and SR-RC are incomparable. □

These results are illustrated in Figure 2. The main insight from these results is the following. The newly proposed correctness criterion of prefix-reducibility corresponds to a subset of the class of serializable and recoverable schedules. In addition to the intuitive arguments given in Section 3.2, this is an indication that PRED is a reasonable correctness criterion. In fact, as discussed in Section 3.2, we believe that PRED is the weakest criterion that accepts all schedules that are intuitively correct from both a concurrency control and a recovery perspective. In particular, we have shown that the criterion PRED is less restrictive than the class of serializable and strict schedules.

4 Conclusions

We have developed a model of concurrent transaction executions including transaction aborts that aims to unify the correctness considerations for concurrency control and recovery. In particular, we have proposed the notion of prefix-reducibility as a criterion that captures all intuitively correct executions. We have shown that this relatively simple criterion is less restrictive than the classical criterion of requiring both serializability and strictness. We have also started to investigate a second, more restrictive criterion, called expanded serializability, that aims towards practical schedulers based on graph cycle testing.

One obvious next step in this research is to further explore the class of prefix-reducible schedules, trying to identify a maximum subclass that can be recognized by a dynamic scheduler (with acceptable efficiency). Another goal beyond the current work is to consider also the correctness of crash recovery procedures. We already used some arguments about global recovery in defining the expansion of an incomplete schedule, and we believe that our model is suitable to be extended to include crash recovery as well.

In the current paper, we restricted ourselves to schedules of uninterpreted read and write actions. However, the model has actually been motivated, to a large extent, by the goal of providing rigid correctness criteria for non-conventional recovery paradigms like compensation of high-level actions [6, 14, 3, 17, 11, 20, 13, 10, 7]. Our preliminary investigations into this direction indicate that the presented model can be carried over to transactions with semantically rich, high-level ADT operations (in the sense of [1, 18]) in a straightforward way. In particular, our criteria of prefix-reducibility and expanded serializability are still applicable within such an extended model, whereas the classical notions of strictness and recoverability cannot be easily extended to schedules beyond reads and writes. In fact, these limitations of the classical criteria for correct recovery have been realized in a number of papers [14, 19, 17], and concepts similar to our notion of expanded schedules have been considered. However, none of the previous work has been able to come up with a formal model that is simple and intuitive enough to serve as a possible "standard" model.

In addition to the extension towards semantically rich operations, we are working on an even more ambitious extension to cover also multilevel transactions and open nested transactions [3, 2, 8, 12, 15, 20]. We would like to allow a high-level action to spawn further subactions that represent the action's implementation at a lower level of abstraction. This extension is necessary to reflect the fact that high-level actions are not necessarily atomic (i.e., indivisible w.r.t. concurrency and failures). We then have to deal with forests of transaction trees rather than flat schedules, where the nodes of a forest correspond to semantically rich operations at different semantic levels (e.g., in an ADT implementation hierarchy). Once we introduce the notion of compensating actions to this model, we can extend our definition of expanded schedules to such forests. The concept of (prefix-)reducibility, however, needs an additional rule which allows us to prune "isolated" subtrees. This rule is fairly well understood in the previous work on open nested transactions [2, 20, 15]. The long-term goal of our work is to combine the powerful model of open nested transactions with our unified model that we have presented in this paper.

Acknowledgement

The ideas of this paper have been shaped in discussions with a number of colleagues. Notably, many discussions with Catriel Beeri have contributed significantly to develop the proposed unified model.

References

[1] B.R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM TODS*, 17(1), March 1992.

[2] C. Beeri, P.A. Bernstein, and N. Goodman. A model for concurrecy in nested transactions systems. *Journal of the ACM*, 36(1), 1989.

[3] C. Beeri, H.-J. Schek, and G. Weikum. Multi-level transaction management, theoretical art or practical need? In *International Conference on Extending Database Technology (EDBT), Sprenger*, 1988.

[4] P.A. Bernstein, V.Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, 1987.

[5] Y. Breitbart, D.Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17(9):954–960, September 1991.

[6] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM TODS*, 8(2), June 1983.

[7] J. Gray and A. Reuter. *Transaction processing: concepts and techniquies.* Morgan Kaufmann Publishers, 1993.

[8] T. Hadzilacos and V. Hadzilacos. Transaction synchronization in object bases. In *ACM PODS Conf.*, 1988.

[9] V. Hadzilacos. A theory of reliability in database systems. *Journal of the ACM*, 35(1), 1988.

[10] P. Karychniak, M. Rusinkiewicz, A. Sheth, and G. Thomas. Bounding the effects of compensation under pelaxed multi-level serializability. Technical memorandum: Tm-tsv-021509/1, Bellcore, June 1992.

[11] E. Levy, H.F. Korth, and A. Silberschatz. A theory of relaxed atomicity. In *ACM PODS Conf.*, 1991.

[12] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions.* Morgan Kaufmann, 1993. to appear.

[13] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1), March 1992.

[14] J. Moss, N. Griffeth, and M Graham. Abstraction in recovery management. In *Proc. ACM SIGMOD Conference*, 1986.

[15] P. Muth, T.C. Rakow, G. Weikum, P. Brössler, and C. Hasse. Semantic concurrecy control in object-oriented database systems. In *Proceedings of IEEE Nineth International Conference on Data Engineering*, April 1993. to appear.

[16] C. Papadimitriou. *The theory of database concurrency control.* Computer Science Press, 1986.

[17] J. Veijalainen. *Transaction concepts in autonomous database environments.* PhD thesis, GMD-Bericht Nr.183. Oldenbourg Verlag, 1990.

[18] W.E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12), 1988.

[19] W.E. Weihl. The impact of recovery on concurrency control. *ACM PODS Conf.*, 1989.

[20] G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

Gelbe Berichte des Departements Informatik

171	K. Simon, P. Trunz	On Transitive Orientation
172	A. Rosenthal, Ch. Rich, M.H. Scholl	Reducing Duplicate Work in Relational Join(s): A Unified Approach
173	P. Schäuble, B. Wüthrich	On the Expressive Power of Query Languages
174	M. Brandis, R. Crelier M. Franz, J. Templ	The Oberon System Family
175	P. Scheuermann, G. Weikum, P. Zabback	Automatic Tuning of Data Placement and Load Balancing in Disk Arrays
176	H. Hinterberger J.C. French (eds.)	Proceedings of the Sixth International Working Conference on Scientific and Statistical Database Management (price SFr. 40.-)
177	J. Burse	ProQuel: Using Prolog to Implement a Deductive Database System
178	P. Arbenz, M. Oettli	Block Implementations of the Symmetric QR and Jacobi Algorithms
179	B. Hösli	3-wertige Logiken und stabile Logik
180	K. Zuse	Computerarchitektur aus damaliger und heutiger Sicht
181	B. Sanders	A Predicate Transformer Approach to Knowledge and Knowledge-based Protocols
182	O. Lorenz	Retrieval von Grafikdokumenten
183	W.B. Teeuw, Ch. Rich, M.H. Scholl, H.M. Blanken	An Evaluation of Physical Disk I/Os for Complex Object Processing
184	L. Knecht, G.H. Gonnet	Alignment of Nucleotide with Peptide Sequences
185	T. Roos, P. Widmayer	Computing the Minimum of the k-Level of an Arrangement with Applications
186	E. Margulis	Using *nP*-based Analysis in Information Retrieval
187	D. Gruntz	Limit Computation in Computer Algebra
188	S. Mentzer	Analyse von Methoden und Werkzeugen zur Entwicklung grosser Datenbank-Anwendungs-Systeme
189	S.J. Leon	Maximizing Bilinear Forms Subject to Linear Constraints