Unified transaction model for semantically rich operations

Unified Transaction Model for Semantically Rich Operations *

Radek Vingralek Haiyan Ye Yuri Breitbart H.-J. Schek

Abstract

We present here an unified transaction model for database systems with semantically rich operations. Based on the work in [SWY93], we develop constructive correctness criteria that encompass both serializability and failure atomicity in an uniform manner. As it turns out, the exact characterization of the class of prefix reducible schedules that was introduced for the simple read/write model in [AVA+94] is infeasible. Thus, we propose here two sufficiently rich subclasses of prefix reducible schedules, argue that serializability and atomicity can be unified by considering schedules from these classes, and design concurrency control protocols that assure also failure atomicity for schedules from these classes. We also show that the previously proposed correctness criteria [MGG86a, MGG86b] and [RKS92, RKS93] are subsumed by our model.

1 Introduction

Transaction management in database systems with semantically rich operations [MGG86a, MGG86b, Wei88, HH88, Wei89, HH91, BR92, RKS92, RKS93, LMWF94] is becoming increasingly important. In this paper we develop one such model that is based on commutativity. Such an approach can be captured by a conflict detection method defined on pairs of operation invocations. Importing such a method into a transaction manager facilitates an extensible approach to providing semantically serializable and semantically atomic transactions. Our main objective is to develop a model that would allow us to reason about transaction atomicity and consistent execution in an uniform manner. Our approach is based on the assumption that with each operation a backward (or undo) operation must be given to undo the observable effects of the operation, if necessary.

Recently, the work by [SWY93, AVA+94] has introduced an unified model of transaction management based on the read/write model. In particular, it introduced the class $PRED$ of prefix reducible schedules that captures the correctness criteria of serializability and failure atomicity in an uniform manner. In this paper, we generalize the previous work of [SWY93] and [AVA+94] in that we provide constructive characterizations for prefix reducible schedules with *semantically rich operations*.

When we started this work, we assumed that generalizing the results of [AVA+94] for read/write operations should be straightforward. Unfortunately, it proved to be not as simple. It turns out that we must distinguish the case where the backward operations have the same conflict behaviour as the forward operations

*Department of Computer Science, ETH Zurich, CH-8092, Switzerland. This material is based in part upon work supported by NSF grants IRI-9221947, IRI-9012902 and IRI-9117904 and by grant from Hewlett-Packard Corporation. This work has been performed while Y. Breitbart was on one year sabbatical and R. Vingralek was visiting the database research group at ETH, Zurich.

from the case where this does not hold. In the latter case, it turns out, the characterization of prefix reducible schedules given in [AVA+94] is not sufficient to characterize such schedules in a semantically rich model of operations.

To obtain practically feasible protocols, we introduce the class of *safe* schedules, which is a proper subclass of prefix reducible schedules. We discuss the properties of these schedules and argue that safe schedules are practically feasible and allow an uniform treatment of serializability and atomicity in transaction models with semantically rich operations.

Our definition of commutativity closely relates to the definitions given in [Wei88, Wei89]. However, unlike [Wei88, Wei89], our definition of commutativity considers also the effects of the undo related operations in addition to the effects of the forward operations. Moss, Griffith and Graham in [MGG86a, MGG86b] introduced the notion of *revokable* schedules to handle the transaction atomicity. We show here that the class of revokable and serializable schedules is a proper subclass of the schedules introduced here. Rastogi, Korth and Silberschatz [RKS92, RKS93] develop a theory of *strict* schedules. Our model is more general in that the class of strict schedules from [RKS92, RKS93] is properly contained in the class of schedules introduced here.

The rest of the paper is organized as follows. In the next section we introduce our transaction management model. Section 3 contains our main theoretical results. Section 3.1 contains a characterization of reducible schedules that are introduced in the previous section. Section 3.2 contains an algorithm to recognize prefix reducible schedules. Section 3.3 introduces a class of $safe$ schedules. Section 3.4 defines restrictions on the commutativity relation such that the class of prefix reducible schedules can be constructively characterized similarly to [AVA+94]. In Section 4 we define a class of protocols for safe schedules and prove their correctness. Section 5 concludes the paper.

2 Model Description

In this section we define our transaction model. The main purpose of this model is to unify concepts of semantic serializability and semantic failure atomicity of concurrently executed transactions. Our model is an extension of a similar model that was defined in [SWY93].

2.1 Operations

A database DB consists of a set D of data objects d and a set of O of operations o (called in the sequel **forward** operations). In order to manipulate D or retrieve values from D the only means is "applying" or "invoking" an operation o from O. An operation invocation [Wei88, Wei89] therefore is an operation o from O that has one or several data objects d from D as input that returns a value to the caller who has invoked the operation. In [Wei88, Wei89], an operation execution is subdivided into two events: an operation invocation event followed by an operation response event delivering the return value. Generally, if two operations are $<_S$ ordered in schedule S, then their invocation and response events can interleave. We, however, assume that if $o_1 <_S o_2$, then the o_1's response event precedes the o_2's invocation event. Thus, in what follows when we talk about an operation, we always understand an operation invocation.

We assume that operations from O and only these operations are available to database users and/or transactions to access and manipulate database objects. Consequently, the database states can be inspected only through the return values of operations from O. We assume that in addition to operations from O there are two special termination operations: **abort** (denoted by a) and **commit** (denoted by c).

The details of return values are not important for our model, except that we assume that the return value of an operation is a function of the changes that the operation performed on the database. For example, in the read/write model, return value of $read(x)$ is the value of x and the return value of $write(x)$ is an indication whether a write was successfully completed. On the other hand, for the embedded SQL operations *insert*, *select*, *delete* and *update* the return values of these operations include the $SQLCA$ area.

For each operation o from O we introduce an **undo** or **backward** operation o^{-1} which backs out all "recognizable" effects of the corresponding forward operation (i.e. those changes in the database caused by o that can be detected by other operations through their return values). Let O^{-1} be the set of all undo operations Thus, after executing forward operation o immediately followed by its corresponding backward operation o^{-1}, no "effect" of o is left in the database as far as it can be detected through the return values of any other operation executed after o and o^{-1}. We say that a sequence of operations α is **well-formed** if each undo operation o^{-1} in α is preceded by its corresponding forward operation o.

Definition 1 *We call a sequence of operations σ* **effect-free** *if, for all possible sequences of operations α and β such that $< \alpha\ \sigma\ \beta >$ and $< \alpha\ \beta >$ are well-formed, the sequence of the return values of β in the $< \alpha\ \sigma\ \beta >$ is the same as in $< \alpha\ \beta >$.*

We say that o^{-1} is undo operation for o if and only if sequence $o\ o^{-1}$ is effect-free. In some cases the undo operation does not need to do anything (for example $read^{-1}$ operation in read/write model). Thus, we introduce a **null** operation and denote it by λ. The return value of the null operation is an empty sequence.

We can reasonably assume that whoever designs the forward operation o also provides the undo operation o^{-1}, since it is him/her who knows the semantics of o and thus also knows how to undo it. For example, in the read/write model, for a *write* operation of the DBMS, the DBMS maintains a log and uses the log information to generate a $write^{-1}$ operation. Alternatively, if a high-level interface to a SQL database is provided by using an embedded SQL to design an operation o, it is the programmer who designs the undo operation o^{-1} (see Examples 1 and 3 below). We assume that the undo operation is dependent on its forward operation in the sense that the forward operation's return value is passed to the corresponding undo operation as one of the input parameters.

As we mentioned earlier, the main purpose of undo operations is to undo the recognizable effects of corresponding forward operations. From this viewpoint any undo operation must successfully complete. In addition, if undo operations are used only for undoing forward operations and cannot be used as forward operations themselves, then it is reasonable to assume that a return code of any undo operation is the same (we assume throughout the paper that a return code of any undo operation is 1 with the only exception of the null operation returning an empty sequence).

2.2 Commutativity

Our notion of conflicting operations is based on the notion of the *commutativity* of operations, which we discuss next. Consider the sequence $S : < \alpha\ p\ q\ \beta >$. If permuting p and q in the sequence does not change their return values and also the return values of β, then we call p and q commutative. There are two possible cases that lead to two alternative definitions of commutativity:

- Permuting p and q in S does not change their return values regardless of which α precedes them in S.
- Permuting p and q in S does not change their return values only for some α.

Thus, we define two notions of commutativity as follows:

Definition 2 *We say that two operations p and q from $O \cup O^{-1}$* **state-independently commute** *if and only if, for all possible operation sequences α and β such that $< \alpha\ p\ q\ \beta >$ and $< \alpha\ q\ p\ \beta >$ are well-formed, the sequence of the return values of β and the return values of p and q in $< \alpha\ p\ q\ \beta >$ are the same as in $< \alpha\ q\ p\ \beta >$.*

Definition 3 *We say that two operations p and q from $O \cup O^{-1}$* **state-dependently commute** *with respect to sequence α_0[1] if and only if, for any sequence of operations β such that $< \alpha_0\ p\ q\ \beta >$ and $< \alpha_0\ q\ p\ \beta >$ are well-formed, the sequence of the return values of β and the return values of p and q in $< \alpha_0\ p\ q\ \beta >$ are the same as in $< \alpha_0\ q\ p\ \beta >$.*

We say that two operations **conflict** if and only if they do not commute. Example 1 below illustrates the above concepts:

Example 1: Let the database consist of a single set S with the following operations defined on elements of it:

SInsert(x)	Inserts element x into S, if it is not already there and returns x. Otherwise it returns constant 0.
$SInsert^{-1}(x)$	X is a return value of the corresponding forward operation. If x is not 0, then it deletes element x from S, otherwise does nothing. It always returns constant 1.
SDelete(x)	Deletes element x from S, if it is there and returns x. Otherwise it returns constant 0.
$SDelete^{-1}(x)$	X is a return value of the corresponding forward operation. If x is not 0, then it inserts element x from S, otherwise does nothing. It always returns constant 1.
Test(x)	Returns YES if element x is in S, otherwise it returns NO.
$Test^{-1}(x)$	It is λ operation that returns an empty sequence.

The commutativity relation for these operations is shown in Figure 1[2]. We illustrate some of the cases shown in Figure 1:

- $SInsert(x)$ does not commute with itself. Indeed, it is easy to verify that return values in the sequences $< SDelete(x)\ SInsert_1(x)\ SInsert_2(x) >$ and $< SDelete(x)\ SInsert_2(x)\ SInsert_1(x) >$ are not the same.

- $SInsert^{-1}(x)$ commutes with itself. We know that $SInsert^{-1}(x)$ always returns 1. Consequently, it can be only return values of some sequence β that can distinguish $< \alpha\ SInsert_1^{-1}(x)\ SInsert_2^{-1}(x)\ \beta >$ and $< \alpha\ SInsert_2^{-1}(x)\ SInsert_1^{-1}(x)\ \beta >$. To guarantee well-formedness, every α must contain both $SInsert_1(x)$ and $SInsert_2(x)$. If any of the two *SInsert*'s returns zero, then one of the two backward operations does not modify the set and thus it is easy to see that no β can distinguish the two sequences. If, on the other hand, both *SInsert*'s return x (this can happen if there is a *SDelete* in between them), then both undoes delete x and thus no β can recognize the difference between $< \alpha\ SInsert_1^{-1}(x)\ SInsert_2^{-1}(x) >$ and $< \alpha\ SInsert_2^{-1}(x)\ SInsert_1^{-1}(x) >$ since x is removed from the set in both cases.

□

Sequences α and β in our definition of commutativity can contain both *forward* and *backward* operations. The traditional commutativity definition [Wei88, Wei89] allowed only *forward* operations to occur

[1] Note that the sequence α_0 corresponds to the existence of some *state* [MGG86a, MGG86b, RKS92, RKS93] in which the two operations commute.

[2] As usual, we assume that operations invoked on different elements always commute.

	SInsert	SDelete	Test	SInsert^{-1}	SDelete^{-1}	Test^{-1}
SInsert	-	-	-	-	-	+
SDelete	-	-	-	-	-	+
Test	-	-	+	-	-	+
SInsert^{-1}	-	-	-	+	-	+
SDelete^{-1}	-	-	-	-	+	+
Test^{-1}	+	+	+	+	+	+

Figure 1: **Commutativity relation**

in sequences α and β. In that, if two operations from O commute in a traditional way, they would not necessarily commute according to our definition.

Example 2: Consider operation *SInsert(x)* from Example 1. Assume that the operation always returns a constant value 1 (we assume that the undo operations use some form of a log to determine their actions). Let us denote such a modified insert operation as *SI(x)*. Under this assumption it is easy to show that two *SI(x)* operations commute according to the traditional definition of commutativity. However, they do not commute according to our definition since operation *Test(x)* returns YES in the sequence $< SI_2(x)\ SI_1(x)\ SI_1^{-1}(x)\ Test(x)\ >$ and NO in the sequence $<\ SI_1(x)\ SI_2(x)\ SI_1^{-1}(x)\ Test(x)\ >$. □

Any scheduler can execute commutative operations concurrently. In order to make the scheduler, a conflict detection method CON must be provided. CON will return true if the two operation invocations conflict and false otherwise. If a concurrency control is based on a state-dependent commutativity, it can, in general, allow more concurrency. However, for the mechanism CON to decide whether two operation invocations are conflicting CON must know the whole prior history. In some cases, it is possible to design such a concurrency control mechanism. For example, in [AD94] operations that state-dependently commute are allowed to run concurrently, provided that they are executed in certain *contexts*.

If the conflict detection method works only on the operation invocations that are independent of the state, it may still require a sophisticated implementation. For example, if CON is applied to two SQL operation invocations, it must determine whether the read and the write sets of their where-predicates are disjoint. Note that for practical purposes we do not require CON to detect all conflicting pairs correctly. What we need is that if operations conflict, then CON will detect it. But sometimes CON may decide that operations conflict even if they do not. For the rest of the paper we assume that such a conflict detection method on operation invocations is provided. That is, the method decides whether or not two operations are in conflict at the time of their invocation. In the most simple cases such a conflict detection can be described by a commutativity relation. Thus, in what follows, by commutativity we mean the state independent one. In the sequel, if two operations o_1 and o_2 are in conflict it is denoted by $(o_1, o_2) \in CON$ and if they do not conflict $(o_1, o_2) \notin CON$.

2.3 Transactions

Database users access the database through **transactions**. A transaction, T_i, is a partial order, $<_i$, of operations (o_i) from O with either *commit* (c_i) or *abort* (a_i) (but not both) as a maximal element of $<_i$. A **schedule** S over a set of transactions $\mathcal{T}$ is a partial order $<_S$ of all operations of all transactions in $\mathcal{T}$ such that for any transaction T_i in $\mathcal{T}$, $<_i$ is a subset of $<_S$. If $o_i <_S o_j$ in S, then we say that operation o_i is executed before operation o_j in S. In schedule S we also allow operation $a(T_{i_1}, \cdots, T_{i_k})$, where $T_{i_1}, \cdots, T_{i_k}$ are from the transaction set $\mathcal{T}$. This operation, called **group abort**, indicates that an abort should

be executed for each transaction from $T_{i_1}, \ldots, T_{i_k}$. However, the execution of these aborts is conducted concurrently.

Transaction T_i is said to be **committed** (**aborted**) in S if S contains c_i (a_i or $a(\ldots, T_i, \ldots)$) operation(s). Transaction T_i is **active** in S if it is neither committed nor aborted in S. The **committed projection** $C(S)$ of schedule S is obtained from S by deleting all operations that do not belong to the committed transactions in S. A **complete schedule** is a schedule in which all transactions are terminated (i.e., committed or aborted).

Let S be a schedule over a set of transactions $\mathcal{T}$. We require that any two conflicting operations from different transactions in the schedule are $<_S$ ordered, and as usually, $<_S$ must observe the intra-transaction order $<_i$. Two schedules are **conflict-equivalent** if they are defined on the same set of transactions, have the same operations and the same set of pairs of conflicting operations of committed transactions. Schedule S is **conflict serializable (SR)** if its committed projection is conflict-equivalent to a serial history.

2.4 Expanded schedules

In order to handle aborted transactions explicitly in a schedule we replace each transaction abort statement with a sequence of transaction undo operations to eliminate the partial effects of an aborted transaction and call the resulting schedule an expanded one. Thus, if a scheduler produces a serializable expanded schedule of transaction operations, where adjacent $o \cdot o^{-1}$ are eliminated from the consideration (since they do not make any effect on a schedule) then issues of serializability and atomicity are treated by such a scheduler in an uniform way. These ideas proposed in [SWY93, AVA+94] lead to the introduction of an **expanded schedule**. Following [AVA+94], for each schedule $< S, <_S >$, we define an expanded schedule $< \tilde{S}, <_{\tilde{S}} >$ as follows.

Definition 4 ([AVA+94]:) *Let $S = (A, <_S)$ be a schedule, where A is the set of operations in S and $<_S$ is a partial order over those operations. Its* **expansion**, *or* **expanded schedule**, *$\tilde{S}$, is a tuple $(\tilde{A}, <_{\tilde{S}})$ where:*

1. *$\tilde{A}$ is a set of operations which is derived from A in the following way:*
   (a) *For each transaction $T_i \in S$, if $o_i \in T_i$ and o_i is not an abort operation, then $o_i \in \tilde{S}$.*
   (b) *Active transactions are treated as aborted transactions, by adding a group abort $a(T_{i_1} \ldots T_{i_k})$ at the end of S, where $T_{i_1} \ldots T_{i_k}$ are all active transactions in S.*
   (c) *For each aborted transaction $T_j \in S$ and for every operation $o_j \in T_j$, there exists a backward operation $o_j^{-1} \in \tilde{S}$. An abort operation $a_j \in S$ is changed to $c_j \in \tilde{S}$. Operation $a(T_{i_1} \ldots T_{i_k})$ is replaced with a sequence of $c_{i_1}, \ldots, c_{i_k}$.*
2. *The partial order, $<_{\tilde{S}}$, is determined as follows:*
   (a) *For every two operations, o_i and o_j, if $o_i <_S o_j$ in S then $o_i <_{\tilde{S}} o_j$ in $\tilde{S}$.*
   (b) *If transactions T_i and T_j abort in S and their aborts are not $<_S$-ordered, then every two conflicting undo operations of transactions T_i and T_j are in $\tilde{S}$ in a reverse order of the two corresponding forward operations in S. If the forward operations are not $<_S$-ordered, then the two undo operations are in an arbitrary order.*
   (c) *All undo operations of every transaction T_i that does not commit in S follow the transaction's original operations and must precede c_i in $\tilde{S}$.*
   (d) *Whenever $o_n <_S a(T_{i_1}, \ldots, T_{i_k}) <_S o_m$ and some undo operation o_j^{-1} ($j \in \{i_1, \ldots, i_k\}$) conflicts with o_m (o_n), then $o_j^{-1} <_{\tilde{S}} o_m$ ($o_n <_{\tilde{S}} o_j^{-1}$).*

(e) *Whenever* $a(\ldots, T_i, \ldots) <_S a(\ldots, T_j, \ldots)$ *for some* $i \neq j$, *then for all conflicting undo operations of* T_i *and* T_j, o_i^{-1} *and* o_j^{-1}, $o_i^{-1} <_{\tilde{S}} o_j^{-1}$.

We say that schedule S is **reducible** (RED) [SWY93] if there exists at least one expanded schedule $\tilde{S}$ such that it can be transformed into a serializable schedule by applying the following two rules:

1. **Commutativity rule:** If o_1 and o_2 are two operations in $\tilde{S}$ such that $o_1 <_{\tilde{S}} o_2$ and $(o_1, o_2) \notin CON$ and there is no $p \in \tilde{S}$ such that $o_1 <_{\tilde{S}} p <_{\tilde{S}} o_2$, then the order $o_1 <_{\tilde{S}} o_2$ can be replaced with $o_2 <_{\tilde{S}} o_1$.

2. **Undo rule:** If o and o^{-1} are two operations in $\tilde{S}$ such that there is no $p \in \tilde{S}$ such that $o <_{\tilde{S}} p <_{\tilde{S}} o^{-1}$ then both o and o^{-1} can be removed from $\tilde{S}$.

To illustrate, consider schedule $S_1 : SDelete_1(x)\ SInsert_2(x)\ Test_3(x)\ c_2\ a_3$. Its expansion is $\tilde{S}_1 : SDelete_1(x)\ SInsert_2(x)\ Test_3(x)\ c_2\ Test_3^{-1}(x)\ c_3\ SDelete_1^{-1}(x)\ c_1$ and it is not reducible. On the other hand, schedule $S_2 : SDelete_1(x)\ SInsert_2(x)\ Test_3(x)\ c_2\ c_1\ a_3$ with expansion $\tilde{S}_2 : SDelete_1(x)\ SInsert_2(x)\ Test_3(x)\ c_2\ c_1\ Test_3^{-1}(x)\ c_3$ is reducible.

Unfortunately, the class of reducible schedules is not prefix-closed and hence cannot be used for online scheduling of transactions [SWY93]. We resolve it by requiring that any reducible schedule be prefix reducible, i.e not only S should be reducible but also any prefix of S must be reducible. For example, schedule S_2 given above is reducible but not prefix reducible, while schedule $S_3 : SDelete_1(x)\ SInsert_2(x)\ Test_3(x)\ c_1\ c_2\ a_3$ is prefix-reducible. Similarly to [SWY93] we consider a class of prefix reducible schedules as a class of schedules that allow to unify the notions of serializability and failure atomicity.

2.5 Restrictions on Commutativity Relations

The major question we deal with in this paper is a consequence of the following: non-symmetric behaviour of commutativity relations between forward operation and its related backward operation.

Definition 5 *We call a commutativity relation* **normal** *if for every two operations* p *and* q *the following condition holds: If* $(p, q) \in CON$ *and* $p^{-1} \neq \lambda$, *then* $(p^{-1}, q) \in CON$ *and* $(p^{-1}, q^{-1}) \in CON$, *if* $q^{-1} \neq \lambda$.

The practical consequence of a normal commutativity relation is the following: Assume that p and q conflict and p was executed before q. To abort the transaction that issued p we must assure that q could not have been executed, even if for serializability reasons q could have been scheduled after p. Scheduling q before p is undone, would lead to a non reducible schedule, which in our view is incorrect. Consider the following example of a normal commutativity relation.

Example 3: Let the database consist of a set of positive integers with the following operations defined on them:

Incr(x)	Increments x if $x > 0$ and returns 1. Otherwise does nothing and returns 0.
$Incr^{-1}(x,\ y)$	If y is the return value of the corresponding forward operation and it is not 0, then decrements x, otherwise does nothing. Always returns 1.
Reset(x)	Resets x to 1. Returns the old value of x.
$Reset^{-1}(x,\ y)$	Sets x to value y where y is the return value of the corresponding forward operation. Always returns 1.
cTest(x)	Returns the current value of x.
$cTest^{-1}(x)$	Is a λ operation and returns an empty sequence.

The commutativity relation for this set of operations is given in Figure 2 and below we illustrate two cases:

	Incr	Reset	cTest	Incr^{-1}	Reset^{-1}	cTest^{-1}
Incr	+	-	-	-	-	+
Reset	-	-	-	-	-	+
cTest	-	-	+	-	-	+
Incr^{-1}	-	-	-	+	-	+
Reset^{-1}	-	-	-	-	-	+
cTest^{-1}	+	+	+	+	+	+

Figure 2: **Commutativity relation**

- Operation *Incr(x)* commutes with itself. Consider two arbitrary sequences α and β. If $x \leq 0$ after α then both *Incr(x)* operations do nothing whatever their mutual ordering is. Consequently, their return values are the same in sequences $< \alpha\ Incr_1(x)\ Incr_2(x) >$ and $< \alpha\ Incr_2(x)\ Incr_1(x) >$. Also the value of x remains the same at the end of both sequences and thus no β can distinguish the two sequences. The case when $x > 0$ after α is similar.

- Operation *Incr(x)* does not commute with $Incr^{-1}(x,y)$. Assuming that $x > 0$, we can verify that operation *cTest(x)* returns 0 in sequence $< Incr_1(x)\ Reset(x)\ Incr_1^{-1}(x,y)\ Incr_2(x)\ cTest(x) >$ and returns 1 in sequence $< Incr_1(x)\ Reset(x)\ Incr_2(x)\ Incr_1^{-1}(x,y)\ cTest(x) >$.

□

A special case of normal commutativity is perfect commutativity. In contrast to normality, perfectness requires that if some combination of backward and forward operations conflict, then *all* combinations of backward and forward operations must conflict. Formally,

Definition 6 *We say that commutativity relation is* **perfect** *if for every two operations p and q either p^α commutes with q^β for all possible combinations of $\alpha, \beta \in \{-1, 1\}$ or p^γ does not commute with q^δ for all possible combinations of $\gamma, \delta \in \{-1, 1\}$ with the exception of λ as a backward operation commuting with everything.*

Perfectness is given in the read/write model because the undo of write is another write. Moreover, every model with a perfect commutativity relation is isomorphic to the read/write model. However, in the more general model this property is not given a priori. For instance, the commutativity relation in Examples 1 is not normal and not perfect. The operation $SInsert(x)$ does not commute with itself, but its backward operation $SInsert^{-1}(x)$ does commute with itself. We have seen that the commutativity relation in Example 3 is normal but it is not perfect, since operation $Incr(x)$ commutes with itself, but operations $Incr(x)$ and $Incr^{-1}(x,y)$ do not commute.

3 Unified Transaction Theory

In this section we present our main theoretical results. Our goal is to provide a constructive characterization of prefix reducible schedules in models with semantically rich operations that would easily lead to the construction of schedulers. In this section we identify the conditions under which the generalization of the characterization from [AVA+94] is exact. In the general case, we are still able to provide a constructive graph based characterization of prefix reducible schedules, however its complexity is too high (although polynomial) for the design of schedulers. We therefore define subclasses of prefix reducible schedules possessing a simple characterization amenable to protocol construction.

3.1 Reducible Schedules and Their Characterization

The definition of reducible schedules given in the previous section is not constructive. In this section we provide a constructive procedure to decide whether a given schedule is reducible. Consider a pair of operations (o_i, o_i^{-1}) in expanded schedule $\tilde{S}$. If there are no other operations between o_i and o_i^{-1} in $\tilde{S}$, then this pair can be removed using the undo rule. Assume now that there are some operations between o_i and o_i^{-1} in $\tilde{S}$. Let $o_1, \ldots o_n$ be operations between o_i and o_i^{-1} such that each o_l conflicts with o_{l+1}, o_i conflicts with o_1 and o_n conflicts with o_i^{-1}. Then, to eliminate the pair (o_i, o_i^{-1}) we need to break this chain of operations by eliminating at least one operation from the sequence by using the undo and commutativity rules. However, if each operation in the sequence belongs to a committed transaction, then none of o_l can be eliminated since no operation of a committed transaction can be eliminated from schedule $\tilde{S}$. In such case, S would not be reducible. Thus in order for S to be reducible, we need to know for each pair (o_i, o_i^{-1}) in $\tilde{S}$ whether it can be removed from the schedule.

Let S be a schedule and $\tilde{S}$ its expansion. To characterize a reducibility of schedule S we construct a **reducibility graph** $RG(\tilde{S})$ as follows: The nodes of the graph are *all* operations in $\tilde{S}$. If o_i from T_i $<_{\tilde{S}}$-precedes o_j from T_j $(i \neq j)$ and o_i conflicts with o_j, then $RG(\tilde{S})$ contains edge $< o_i, o_j >$.

Lemma 1 *Two operations o_i and o_i^{-1} can be moved together by use of the commutativity rule in $\tilde{S}$ if and only if there is no path between o_i and o_i^{-1} in $RG(\tilde{S})$.* □

Proof: Clearly, whenever there exists a path of pairwise conflicting operations from o_i to o_i^{-1} then o_i and o_i^{-1} cannot be moved together by use of commutativity rule only. On the other hand, assume that there is no such path. Consider operations on all paths coming out from node o_i. Out of these operations, those that are the $<_{\tilde{S}}$-maximal preceding o_i^{-1} can be moved beyond o_i^{-1} by use of the commutativity rule. This process can be applied until there are no operations conflicting with o_i between o_i and o_i^{-1}. Then o_i can be easily moved towards o_i^{-1} using the commutativity rule. □

Based on this lemma we can decide whether a given expanded schedule $\tilde{S}$ is reducible using the procedure defined below:

1. For $\tilde{S}$ construct $RG(\tilde{S})$.
2. Find a pair of nodes o_i and o_i^{-1} in $RG(\tilde{S})$ such that there is no path between them.
3. If such a pair does not exist and $\tilde{S}$ contains some backward operations, declare the schedule S non reducible and exit. If such a pair does not exist and $\tilde{S}$ does not contain any backward operations, exit.
4. If such a pair does exist, remove it from $RG(\tilde{S})$ along with all edges incidental to these nodes and also remove that pair from $\tilde{S}$.
5. Go to step 2.

If, as a result of the procedure, we obtain a serializable schedule of only forward operations, then S is reducible. Otherwise, S is not reducible. To illustrate, consider the following examples:

Example 4: Consider schedule S_3 : $SInsert_1(x)$ $SDelete_2(x)$ $SInsert_3(x)$ a_1 a_2 a_3. Its expansion is $\tilde{S_3} = SInsert_1(x)$ $SDelete_2(x)$ $SInsert_3(x)$ $SInsert_1^{-1}(x)$ $SDelete_2^{-1}(x)$ $SInsert_3^{-1}(x)$ c_1 c_2 c_3. Operation $SInsert_1(x)$ conflicts with $SDelete_2(x)$, $SDelete_2(x)$ conflicts with $SInsert_3(x)$, $SInsert_3(x)$ conflicts with $SInsert_1^{-1}(x)$, $SInsert_1^{-1}(x)$ conflicts with $SDelete_2^{-1}(x)$ and $SDelete_2^{-1}(x)$ conflicts with $SInsert_3^{-1}(x)$. The reducibility graph for $\tilde{S_3}$ consists of a single path: ($SInsert_1(x)$, $SDelete_2(x)$, $SInsert_3(x)$, $SInsert_1^{-1}$,

$SDelete_2^{-1}$, $SInsert_3^{-1}$). Thus, there is a path between any two forward operation and its corresponding backward operation. S_3 is not reducible. □

Example 5: Consider schedule S_4 : $SDelete_1(x)$ $SDelete_2(x)$ $SDelete_3(x)$ a_1 a_2 a_3. Its expansion is $\tilde{S_4}$ = $SDelete_1(x)$ $SDelete_2(x)$ $SDelete_3(x)$ $SDelete_1^{-1}(x)$ $SDelete_2^{-1}(x)$ $SDelete_3^{-1}(x)$ c_1 c_2 c_3. Operation $SDelete_1(x)$ conflicts with $SDelete_2(x)$, $SDelete_2(x)$ conflicts with $SDelete_3(x)$, $SDelete_3(x)$ conflicts with $SDelete_1^{-1}(x)$. The reducibility graph for $\tilde{S_4}$ contains path ($SDelete_1(x)$, $SDelete_2(x)$, $SDelete_3(x)$,$SDelete_1^{-1}$) and two isolated nodes $SDelete_2^{-1}(x)$ and $SDelete_3^{-1}(x)$. Consequently, after removing $SDelete_3(x)$ and $SDelete_3^{-1}(x)$ and $SDelete_2(x)$ and $SDelete_2^{-1}(x)$ from the graph, we obtain a graph that does not contain any path between $SDelete_1(x)$ and $SDelete_1^{-1}(x)$. Therefore, this pair of nodes also can be removed from the graph. Thus, S_4 is reducible. □

The construction of the reducibility graph requires $O(n^2)$ operations, where n is the number of operations in $\tilde{S}$. Testing whether there is at least one path from o_i to o_i^{-1} can be done in $O(n^2)$ steps. The test needs to be done for at most n pairs. Finally, the procedure steps 2, 3, and 4 have to be repeated in the worst case n times since not more than n pairs can be eliminated. Therefore the overall complexity of the procedure is $O(n^4)$ where n is the number of operations! This is relatively costly and therefore the procedure is not very practical. In Section 3.3 we consider much less complicated procedures that would allow us to generate relatively rich subclasses of reducible schedules.

We conclude this section by comparing the class of reducible schedules with the class of revokable schedules introduced by Moss, Griffith and Graham [MGG86a, MGG86b]. Their definition in our model can be restated as follows.

Definition 7 ([MGG86a, MGG86b])
Schedule S is **revokable (RV)** *iff for every two transactions T_i, T_j in S and every two operations $o_i \in T_i, o_j \in T_j$ such that $o_i <_S o_j$, a_i does not precede o_j in S and o_i^{-1} is in conflict with o_j then if T_i aborts in S then T_j also aborts in S and either $a_j <_S a_i$ or $a(\ldots, T_i, \ldots, T_j, \ldots) \in S$.*
Schedule $SInsert_1(x)$ $SInsert_2(x)$ a_2 a_1 is both revokable and reducible. Not every reducible schedule is revokable. Schedule $SInsert_1(x)$ $SInsert_2(x)$ a_1 a_2 is not revokable, however, it is reducible. Furthermore, not every revokable schedule is reducible. Schedule $SInsert_1(x)$ $SInsert_2(x)$ $SInsert_2(y)$ $SInsert_1(y)$ c_1 c_2 is revokable but not reducible, since it is not serializable. It appears, however, that the only revokable non-serializable schedules are not reducible.

Theorem 1 *Every revokable and serializable schedule is also reducible.* □

Proof: We first prove an auxiliary lemma:

Lemma 2 *Let S be a revokable schedule. Then all operations of transactions non-committed in S can be completely eliminated from some $\tilde{S}$ by finitely many applications of the* **commutativity** *and* **undo** *rules.*

Proof: (of Lemma 2) Consider an arbitrary pair of forward-backward operations o_i and o_i^{-1} in $\tilde{S}$. We will show that both o_i and o_i^{-1} can be eliminated from $\tilde{S}$. We can assume that o_i^{-1} is the $<_{\tilde{S}}$-minimal backward operation in $\tilde{S}$ (since if that is not the case we can repeat an elimination of $<_{\tilde{S}}$-minimal backward operations until o_i^{-1} itself becomes $<_{\tilde{S}}$-minimal).

To show that the pair o_i, o_i^{-1} can be eliminated from $\tilde{S}$ where o_i^{-1} is $<_{\tilde{S}}$-minimal, we proceed by induction on the number of operations between o_i and o_i^{-1}, k. The case $k = 0$ is trivial. Let us assume that the claim is true for all $l < k$ and we need to establish it for k. For that consider the o_i^{-1}'s predecessor, o_j. Clearly, o_j cannot be a backward operation since we assumed that o_i^{-1} is the $<_{\tilde{S}}$-minimal. Thus, o_j is a forward operation. If o_j commutes with o_i^{-1} then the two operations can be swapped and the induction hypothesis is used. If, on the other hand, o_j conflicts with o_i^{-1}, then from the revokability of S we obtain that T_j also aborts in S and $a_j <_S a_i$. Consequently, from the definition of expanded schedules it follows that $o_j^{-1} <_{\tilde{S}} o_i^{-1}$ which contradicts the $<_{\tilde{S}}$-minimality of o_i^{-1}. Thus the lemma is proven.

We continue now with the proof of Theorem 1. By Lemma 2 all operations of aborted transactions in S can be eliminated from $\tilde{S}$ by the commutativity and undo rules. Thus $\tilde{S}$ would contan after such elimination only operations of transactions committed in S. Since we assumed that S is serializable, $\tilde{S}$ is also serializable and consequently, S is reducible. □

3.2 Prefix Reducible Schedules

In [AVA+94] we characterized the class of prefix reducible schedules in the read/write model. However, it appears that a straightforward generalization of that result for the transaction model presented here does not work, as we demonstrate below. We first redefine the class of schedules **serializable with ordered termination** (SOT) defined in [AVA+94] for a semantically rich set of operations O.

Definition 8 *A schedule S is* **serializable with ordered termination (SOT)** *if it is serializable, and for every 2 transactions T_i, T_j in S and every 2 operations $o_i \in T_i, o_j \in T_j$ such that $o_i <_S o_j$, a_i does not precede o_j in S, o_i is in conflict with o_j and o_i^{-1} is in conflict with o_j, the following conditions hold:*

1. *if T_j commits in S then T_i commits in S and $c_i <_S c_j$.*
2. *if o_i^{-1} and o_j^{-1} are in conflict, and T_i aborts in S then T_j also aborts in S and either $a_j <_S a_i$ or $a(\ldots, T_i, \ldots, T_j, \ldots) \in S$.*

The first condition implies that commit operations of both transactions should be performed in the order of their conflicting operations. Without this condition, the schedule $S : SInsert_1(x)\ SInsert_2(x)\ c_2\ c_1$ is not prefix reducible. Indeed, consider $SInsert_1(x)\ SInsert_2(x)\ c_2$, which is a prefix of S. Its expansion $SInsert_1(x)\ SInsert_2(x)\ c_2\ SInsert_1^{-1}(x)\ c_1$ cannot be reduced since neither operations $SInsert_2(x)$ and $SInsert_1^{-1}(x)$ nor $SInsert_1(x)$ and $SInsert_2(x)$ can be swapped.

The second condition implies that abort operations of conflicting transactions should be performed in the order opposite to the execution of their conflicting operations. Without this condition the schedule $SInsert_1(x)\ SDelete_2(x)\ a_1\ a_2$ is not reducible, and, thus, is not prefix-reducible.

Thus, both conditions are required for a schedule to be prefix reducible. In the read/write model, however, these conditions were also sufficient. [AVA+94]. We first show that the above conditions are indeed necessary to ensure prefix reducibility for an arbitrary set of operations O. Namely, we prove that each prefix reducible schedule is also an SOT schedule.

Theorem 2 *Every prefix reducible schedule is also serializable with ordered termination.* □

Proof: Assume to the contrary, that there exists $S \in PRED - SOT$. Let us consider the following cases:

1. Consider $S \in PRED$, but the first condition in the definition of SOT is violated. Let o_i, o_j be a pair of operations satisfying the assumptions of the SOT definition. We assume that T_j commits, but either $c_i \notin S$ or $c_j <_S c_i$. In the first case from the definition of expanded schedules we derive that $o_i <_{\tilde{S}} o_j <_{\tilde{S}} o_i^{-1}$ for all $\tilde{S}$. Since $(o_i, o_j) \in CON$, $(o_i^{-1}, o_j) \in CON$ and o_j belongs to a committed transaction, $\tilde{S}$ cannot be reducible which h contradicts to the initial assumption that $S \in PRED$. In the second case we consider a prefix of S containing c_i but not c_j. Applying the arguments similar to the first case, we again derive a contradiction with $S \in PRED$.
2. Consider $S \in PRED$ but the second condition in the definition of SOT is violated. Let o_i, o_j be two operations satisfying the assumptions of the SOT definition. Assume that $(o_i^{-1}, o_j^{-1}) \in CON$ and T_i aborts in S, but either T_j does not abort in S or $a_i <_S a_j$. Consider first the case where $a_i <_S a_j$. From the definition of the expanded schedules it follows that $o_i <_{\tilde{S}} o_j <_{\tilde{S}} o_i^{-1} <_{\tilde{S}} o_j^{-1}$ holds in any

	Incr	Reset	cTest	$Incr^{-1}$	$Reset^{-1}$	$cTest^{-1}$	Decr	$Decr^{-1}$
Incr	+	-	-	-	-	+	-	-
Reset	-	-	-	-	-	+	-	-
cTest	-	-	+	-	-	+	-	-
$Incr^{-1}$	-	-	-	+	-	+	+	+
$Reset^{-1}$	-	-	-	-	-	+	-	-
$cTest^{-1}$	+	+	+	+	+	+	+	+
Decr	-	-	-	+	-	+	+	+
$Decr^{-1}$	-	-	-	+	-	+	+	+

Figure 3: **Commutativity relation**

$\tilde{S}$. Since $(o_i, o_j) \in CON$, $(o_i^{-1}, o_j) \in CON$ and $(o_i^{-1}, o_j^{-1}) \in CON$, $\tilde{S}$ cannot be reducible which contradicts the initial assumption that $S \in PRED$. Let us assume now that T_j does not abort in S. If it commits and under our assumptions T_i aborts, we violate the first condition of the SOT definition which we have already considered. Thus, T_j is active in S. Hence, in every $\tilde{S}$ it is treated as implicitly aborted at the end of the schedule. Thus the arguments from the case where $a_i <_S a_j$ apply also here.

The containment stated in Theorem 2 is proper as the following example demonstrates

Example 6: Consider the set of operations from Example 3 extended with the following operations:

Decr(x) Decrements x and returns 1.
$Decr^{-1}(x)$ Increments x and returns 1.

The commutativity relation for this extended set of operations is given in Figure 3 and below we illustrate two cases:

- Operation *Incr(x)* does not commute with *Decr(x)*. Assuming that $x = 1$ initially, we can verify that operation $Incr_1(x)$ returns 1 in sequence $< Incr_1(x)\ Decr_2(x) >$ and returns 0 in sequence $< Decr_2(x)\ Incr_1(x) >$.
- Operation *Decr(x)* commutes with $Incr^{-1}(x, y)$. Consider two arbitrary sequences α, β. Because of well-formedness, α must contain $Incr_2(x)$ in the sequences $< \alpha\ Decr_1(x)\ Incr_2^{-1}(x,y)\ \beta >$ and $< \alpha\ Incr_2^{-1}(x,y)\ Decr_1(x)\ \beta >$. In both sequences $Decr_1$ decrements x and returns 1. Assuming the value of x is k after α. If the return value of $Incr_2(x)$ is 1, then $Incr_2^{-1}(x, y)$ always decrements x and returns 1. Also the value of x is $k-2$ at the end of both sequences and thus no β can distinguish the two sequences. The case when the return value of $Incr_2(x)$ is 0 can be established in a similar way.

Consider now schedule $S_3 : Incr_1(x)\ Decr_2(x)\ Incr_3(x)\ a_1\ c_2\ c_3$. Its expansion is $\tilde{S_3} = Incr_1(x)\ Decr_2(x)\ Incr_3(x)\ Incr_1^{-1}(x,y)\ c_2\ c_3$. Operation $Incr_1(x)$ conflicts with $Decr_2(x)$, $Decr_2(x)$ conflicts with $Incr_3(x)$ and $Incr_3(x)$ conflicts with $Incr_1^{-1}(x,y)$. The reducibility graph for $\tilde{S_3}$ consists of a single path: ($Incr_1(x)$, $Decr_2(x)$, $Incr_3(x)$, $Incr_1^{-1}(x)$) and consequently, S_3 is not reducible. However, schedule S_3 satisfies the SOT conditions, since $Incr_1^{-1}(x)$ does not conflict with $Decr_2(x)$ and $Incr_1(x)$ does not conflict with $Incr_3(x)$. □

Thus, the requirement for a schedule to be SOT is not sufficient to ensure prefix reducibility. We know of only one way to check whether a schedule is $PRED$. Namely, we check for each prefix of S using the method described in section 3.1. However this process is expensive and highly impractical! To eliminate such complexity, one of two ways can be followed: either we should restrict the class of prefix reducible schedules or we should impose some restrictions on a commutativity relation. In the next section we investigate the first approach and in Section 3.4 we investigate the second approach.

3.3 Safe Schedules

In order to show that a given schedule is prefix reducible, it is necessary to eliminate all forward-backward operation pairs belonging to aborted transactions by the use of the commutativity and undo rules. In doing so, it is possible to combine the movements of both forward operations towards the backward operations and backward operations towards forward operations. Such a degree of freedom together with the fact that both forward operation and its backward operation can commute with different sets of operations contribute to the difficulties of a constructive characterization of $PRED$.

To restrict the $PRED$ class to a class that can be effectively handled by a scheduler and/or a recovery manager, consider in more detail what happens when an undo operation is scheduled. The purpose of the undo operation as we stated earlier is to undo all visible effects of the corresponding forward operation. Consider the situation that after executing $o_1\ o_2 \ldots o_k$ operations o_1^{-1} must be executed. To undo the effects of o_1 and also to guarantee the consistency of the resulting schedule, the scheduler scheduling o_1^{-1} must assure that no operation in the sequence $o_2 \ldots o_k$ would be affected by scheduling o_1^{-1}. This can be achieved in one of two ways:

- Operations $o_2 \ldots o_k$ do not conflict with o_1 and thus their visible effects are not affected by return value of o_1. Then o_1 can be safely moved to o_1^{-1} and both operations subsequently eliminated by the undo rule.
- Operation o_1^{-1} commutes with every operation in the sequence $o_2 \ldots o_k$. Then o_1^{-1} can be safely moved to o_1 and both operations can be subsequently eliminated by the undo rule.

These two cases can be formalized in the following definition:

Definition 9 *Schedule S is* **forward safe (FSF)** *(***backward safe (BSF)***) if and only if for every two transactions T_i, T_j in S and every two operations $o_i \in T_i, o_j \in T_j$ such that $o_i <_S o_j$, T_i does not abort before o_j in S and o_i (o_i^{-1}) is in conflict with o_j the following conditions hold:*

1. *If T_j commits in S, then T_i commits in S and $c_i <_S c_j$.*
2. *If T_i aborts in S and $o_j^{-1} \neq \lambda$ then T_j also aborts in S and either $a_j <_S a_i$ or $a(\ldots, T_i, \ldots, T_j, \ldots) \in S$.*

There exist forward safe schedules that are not backward safe and vice versa as the example below demonstrates:

Example 7: Consider schedule $S_1 : Incr_1(x)\ Decr_2(x)\ c_2\ a_1$. Since $Decr(x)$ conflicts with $Incr(x)$, but $Incr^{-1}(x, y)$ commutes with $Decr(x)$, the schedule is backward safe, but not forward safe. On the other hand, schedule $S_2 : Incr_1(z)\ Incr_2(z)\ c_2\ a_1$ is forward safe, but not backward safe, since $Incr(z)$ commutes

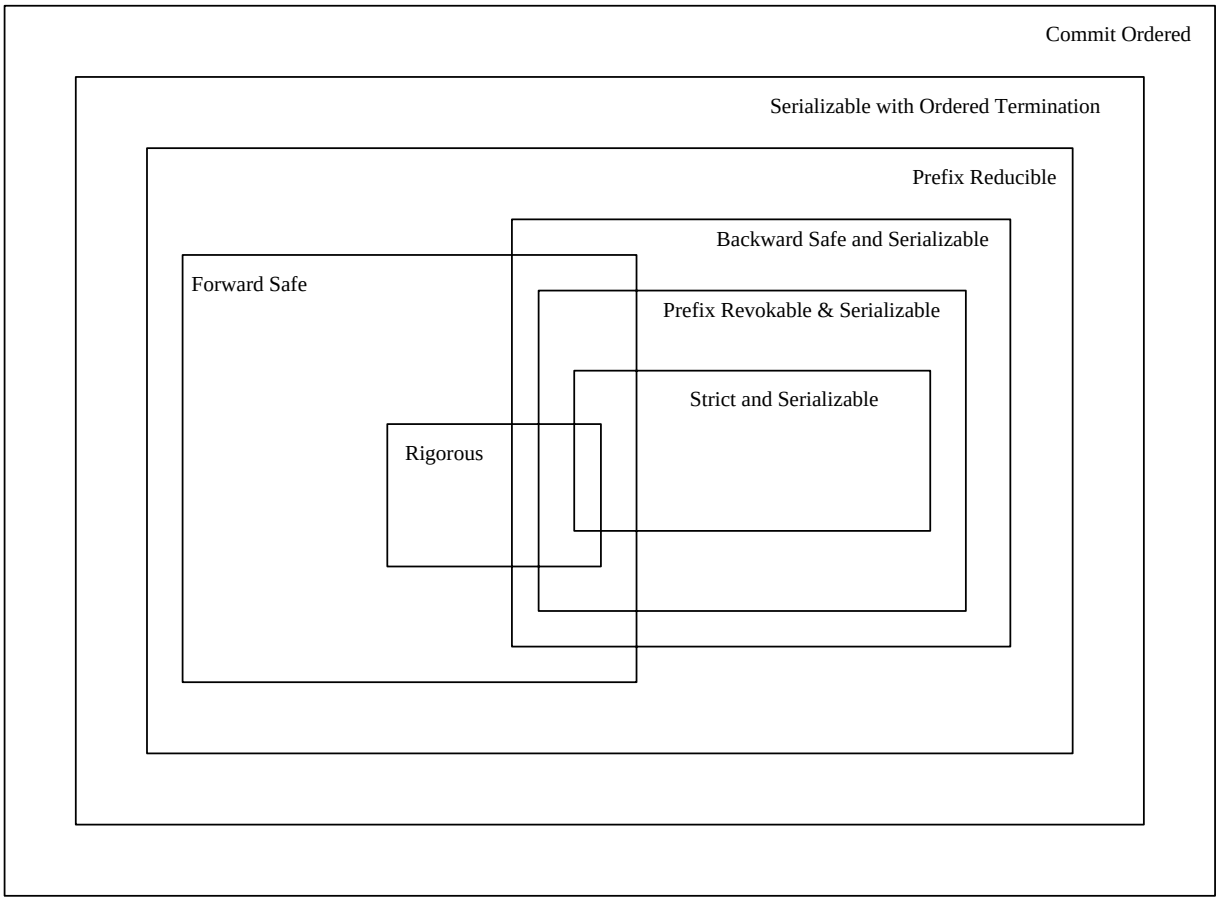

Figure 4: **Relationship between classes**

with itself, but is in conflict with $Incr^{-1}(z, y)$. Schedule $S_3 : Incr_1(x)\ Decr_2(x)\ c_2\ a_1\ Incr_3(y)\ Incr_4(y)\ c_4\ a_3$ is refix reducible, but neither forward nor backward safe □

Recall that to guarantee forward safeness we must consider conflicting pairs of forward operations. Alternatively, we must consider conflicting pairs of forward and backward operations to guarantee backward safeness. Consider, for example, a case in which transaction T_1 issues several select statements and transaction T_2 subsequently performs some update statements. If transaction T_1 aborts, then T_2 still can commit with guaranteeing of backward safeness. However, to guarantee forward safeness, T_2 would have to be aborted. On the other hand, suppose, for example, that forward operations of transaction T_1 are followed by forward operations of transaction T_2 which commute with all T_1's forward operations, but some of the backward operations of T_1 conflict with some of the T_2's forward operations. If transaction T_1 aborts, then transaction T_2 still can commit with guaranteeing of forward safeness. However, to guarantee backward safeness, T_2 would have to be aborted.

Since ordering of commit operations in backward safe schedules reflects conflicts between forward and backward operations rather than between two forward operations, a backward safe schedule is not necessarily serializable. For example, schedule $S : Incr_1(x)\ Decr_2(x)\ Incr_2(y)\ Decr_1(y)\ c_1\ c_2$ is backward safe, but not serializable. Alternatively, every forward safe schedule is serializable since it is a subclass of **commit ordered (CO)** [BS92] schedules which is, in turn, a subclass of serializable schedules [BS92, Raz92]. Every **rigorous (RG)** schedule [BGRS91] is also forward safe. On the other hand, schedule $S : Incr_1(x)\ Decr_2(x)\ c_1\ c_2$ is forward safe, but not rigorous.

To compare the class of backward and forward safe schedules with **revokable schedules**[MGG86a, MGG86b], we need to make a revokable property prefix closed. We therefore introduce the class of **prefix revokable** schedules as follows:

Definition 10 *Schedule S is* **prefix revokable (PRV)** *iff for every two transactions T_i, T_j in S and every two operations $o_i \in T_i, o_j \in T_j$ such that $o_i <_S o_j$, a_i does not precede o_j in S and o_i^{-1} is in conflict with o_j the following is true:*

1. *if T_j commits in S then T_i commits in S and $c_i <_S c_j$.*
2. *if T_i aborts in S then T_j also aborts in S and either $a_j <_S a_i$ or $a(\ldots, T_i, \ldots, T_j, \ldots) \in S$.*

It turns out that the class of backward safe schedules is broader than the class of prefix revokable schedules. That is, there are schedules that are not prefix revokable and yet are backward safe. For example, consider schedule $S : SInsert_1(x)\ Test_2(x)\ a_1\ a_2$. This schedule is backward safe, but it is not prefix revokable. On the other hand, there are prefix revokable schedules that are not forward safe and forward safe schedules that are not prefix revokable.

The class of *strict* schedules defined in [RKS92, RKS93] as schedules such that if for every two operations in S o_i and o_j such that $o_i <_S o_j$ and o_i^{-1} is in conflict with o_j, then transaction T_i terminates before o_j in S is a subclass of prefix revokable (and consequently also backward safe) schedules. On the other hand, there are prefix revokable schedules that are not strict. Consider for example schedule $S : SInsert_1(x)$ $SDelete_2(x)\ c_1\ c_2$. It is easy to see that S is not strict, but it is prefix revokable.

As opposed to the read/write model, there is no relationship between strict serializable and rigorous schedules in the model with semantically rich operations, i.e. there exist strict schedules that are not rigorous and vice versa. Consider schedule $S_1 : Incr_1(x)\ Decr_2(x)\ c_1\ c_2$. The schedule is strict and serializable, but not rigorous. Similarly, schedule $S_2 : Incr_1(y)\ Incr_2(y)\ c_1\ c_2$ is rigorous, but not strict.

The classes of forward safe and serializable backward safe schedules are proper subclasses of prefix reducible schedules, as the following theorems show:

Theorem 3 *Every forward safe schedule is prefix reducible.*

Proof: First, an auxiliary lemma is proven:

Lemma 3 *Let S be a forward safe schedule. Then all operations of transactions non-committed in S can be completely eliminated from some $\tilde{S}$ by finitely many applications of the* **commutativity** *and* **undo** *rules.*

Proof: (of Lemma 3)
The structure of the proof is identical to that of the proof of Lemma 2. We therefore show only the inductive step. Let us consider the pair of operations o_i and o_i^{-1} where o_i^{-1} is $<_{\tilde{S}}$-minimal and there are k operations in between o_i and o_i^{-1}. We show that this pair can be eliminated from some $\tilde{S}$.

Let o_j be an immediate successor of o_i in $\tilde{S}$. If o_j commutes with o_i, then we swap these operations and use the induction hypothesis. If, on the other hand, o_j does not commute with o_i then from the forward safeness of S it follows that T_j cannot be committed in S and $o_j^{-1} = \lambda$ (for if $o_j^{-1} \neq \lambda$ then $o_j^{-1} <_{\tilde{S}} o_i^{-1}$ in any $\tilde{S}$ which contradicts the assumption of o_i^{-1} being $<_{\tilde{S}}$-minimal). Consequently, since $o_j^{-1} = \lambda$ commutes with all operations, o_j and o_j^{-1} can be eliminated and the induction hypothesis can be used again. This completes the proof of the Lemma.

We continue now with the proof of Theorem 3. Let S_1 be an arbitrary prefix of S. We show that S_1 is reducible. Since a forward safeness is a prefix-closed property, all operations of transactions non-committed in S_1 can be eliminated from some $\tilde{S}_1$ by Lemma 3. Since S_1 is forward safe, it is also commit ordered

and thus it is also serializable[BGRS91, Raz92]. The only remaining operations in $\tilde{S}_1$ are operations of transactions committed in S_1. Thus, $\tilde{S}_1$ is serializable and consequently S_1 is reducible. Hence, S is prefix reducible. □

Theorem 4 *Every backward safe and serializable schedule is prefix reducible.*

Proof: First, an auxiliary is proven:

Lemma 4 *Let S be a backward safe schedule. Then all operations of transactions non-committed in S can be completely eliminated from some $\tilde{S}$ by finitely many applications of the* **commutativity** *and* **undo** *rules.*

Proof: (of Lemma 4)
The structure of the proof is identical to that of the proof of Lemma 2. We therefore show only the inductive step. Let us consider the pair of operations o_i and o_i^{-1} where o_i^{-1} is $<_{\tilde{S}}$-minimal and there are k operations in between o_i and o_i^{-1}. We show that this pair can be eliminated from some $\tilde{S}$.

Let o_j be an immediate predecessor of o_i^{-1} in $\tilde{S}$. If o_j commutes with o_i^{-1}, then we swap both operations and use the induction hypothesis. If, on the other hand, o_j does not commute with o_i^{-1} then from the backward safeness of S it follows that T_j cannot be committed in S and $o_j^{-1} = \lambda$ (for if $o_j^{-1} \neq \lambda$ then $o_j^{-1} <_{\tilde{S}} o_i^{-1}$ in any $\tilde{S}$ which contradicts the assumption of o_i^{-1} being $<_{\tilde{S}}$-minimal). Consequently, since $o_j^{-1} = \lambda$ commutes with all operations, o_j and o_j^{-1} can be eliminated and the induction hypothesis is used again. The rest of the proof is the same is in the proof of Theorem 3. □

The containments proved in the above two theorems are proper as Example 7 demonstrates. The relationship among schedule classes discussed so far is shown in Figure 4.

3.4 Normal and Perfect Commutativity Relations

In this subsection we investigate restrictions on the commutativity relation that allow us to characterize prefix reducible schedules. We show that the following theorem holds:

Theorem 5 *Let a commutativity relation be normal. Then a schedule is prefix reducible if and only if it is serializable with ordered termination.*

Proof: In Theorem 2 we have proven that each prefix reducible schedule is also an SOT one. Thus it remains to show only that each SOT schedule is also a prefix reducible. First, an auxiliary lemma is proven:

Lemma 5 *Let S be a serializable with ordered termination schedule and let a commutativity relation be normal. Then all operations of transactions non-committed in S can be completely eliminated from some $\tilde{S}$ by finitely many applications of the* **commutativity** *and* **undo** *rules.*

Proof: The structure of the proof is identical to that of the proof of Lemma 2. We therefore show only the inductive step. Let us consider the pair of operations o_i and o_i^{-1} where o_i^{-1} is a $<_{\tilde{S}}$-minimal and there are k operations in between o_i and o_i^{-1}. We show that this pair can be eliminated from some $\tilde{S}$.

Let o_j be an immediate successor of o_i in $\tilde{S}$. If o_j commutes with o_1, then we swap these operations and use the induction hypothesis. Consider now the case that o_i does not commute with o_j. If $o_i^{-1} = \lambda$, then o_i and o_i^{-1} can be trivially eliminated from $\tilde{S}$. If, on the other hand, $o_i^{-1} \neq \lambda$, then from the normality of a commutativity we obtain that $(o_i^{-1}, o_j) \in CON$ and $(o_i^{-1}, o_j^{-1}) \in CON$. Since S is SOT, T_j cannot commit in S and consequently, for all expanded schedules $\tilde{S}$ $o_j <_{\tilde{S}} o_j^{-1} <_{\tilde{S}} o_i^{-1}$ which contradicts to the $<_{\tilde{S}}$-minimality of o_i^{-1}. The lemma is proven.

The rest of the proof follows precisely the arguments given in the proof of Theorem 3. Thus we have shown $PRED = SOT$. □

Lemma 6 *If a commutativity relation is normal, then the class of rigorous schedules is a subset of the class of strict and serializable schedules.*

Proof: It follows directly from the definitions of rigorous [BGRS91] and strict [RKS93] schedules and a normality of the commutativity relation. Indeed, whenever a commutativity relation is normal then from $(o_i, o_j) \in CON$ it followsalso that $(o_i^{-1}, o_j) \in CON$ and $(o_i^{-1}, o_j^{-1}) \in CON$. □

As we have seen, the commutativity relation of Example 1 is not normal. Nevertheless, it is not difficult to prove that the classes of SOT and $PRED$ schedules coincide for the set of operations defined there. Thus, Theorem 5 does not provide a necessary condition on a commutativity relation to guarantee that the classes of SOT and $PRED$ schedules coincide.

The relationship between classes introduced so far for normal commutativity relation can be derived from Figure 4 by assuming that $PRED = SOT$ and RG is a subset of $ST\&SER$. An obvious consequence of normal commutativity relations is that all the protocols defined in [AVA+94] can be applied to generate prefix reducible schedules in models with semantically rich operations possessing normal commutativity relation. Note, under the condition of normal commutativity relation, the class of serializable with ordered termination schedules is still a proper superset of the class of backward safe and serializable schedules ($BSF\&SR$) as the following example demonstrates.

Example 8: Consider the schedule $S = Incr_1(y)\ Incr_2(y)\ Incr_1^{-1}(y, z_1)\ Incr_2^{-1}(y, z_2)$. It is easy to see that the commutativity relation in Example 3 is normal. S is SOT, since $Incr_1(y)$ commutes with $Incr_2(y)$. On the other hand, the schedule is not backward safe, since $Incr_2(y)$ conflicts with $Incr_1^{-1}(y, z_1)$ but $Incr_1^{-1}(y, z_1)$ precedes $Incr_2^{-1}(y, z_2)$ in the schedule. □

A perfect commutativity relation is a special case of a normal commutativity relation. The main appeal of models with a perfect commutativity relation lies in their "isomorphism" to the read/write model. With the perfectness, the classes of SOT and $BSF\&SR$ schedules coincide, as the following theorem states.

Theorem 6 *Let a commutativity relation be perfect. Then the classes of serializable with ordered termination and backward safe serializable schedules coincide.*

Proof: The equivalence claimed in the theorem follows directly from the definitions 8 and 9 and perfectness of the commutativity relation. Indeed, whenever a commutativity relation is perfect then from $(o_i^{-1}, o_j) \in CON$ follows also $(o_i, o_j) \in CON$ and $(o_i^{-1}, o_j^{-1}) \in CON$. □

Another consequence of perfectness of the commutativity relation is that the class of forward safe schedules becomes a proper subset of the class of backward safe serializable schedules (which is, in turn, equal to SOT and $PRED$). Schedule $S_1 : r_1(x)\ w_2(x)\ c_2\ a_1$ is an example of the schedule that is backward safe and serializable, but not forward safe. Similarly to the read/write model, the class of rigorous schedules becomes a proper subset of strict serializable schedules. Schedule S_1 gives an example of the strict serializable schedule that is not rigorous. The class of strict serializable schedules remains a proper subset of prefix revokable schedules which, in turn, remains a proper subset of backward safe serializable schedules. The class of strict serializable schedules remains also incomparable with the class of forward safe schedules. Schedule S_1 is strict serializable, but not forward safe, schedule $S_2 : w_1(x)\ r_2(x)\ c_1\ c_2$ is forward safe, but not strict serializable. The relationship among all the classes under the assumption of perfect commutativity relation is depicted in Figure 5.

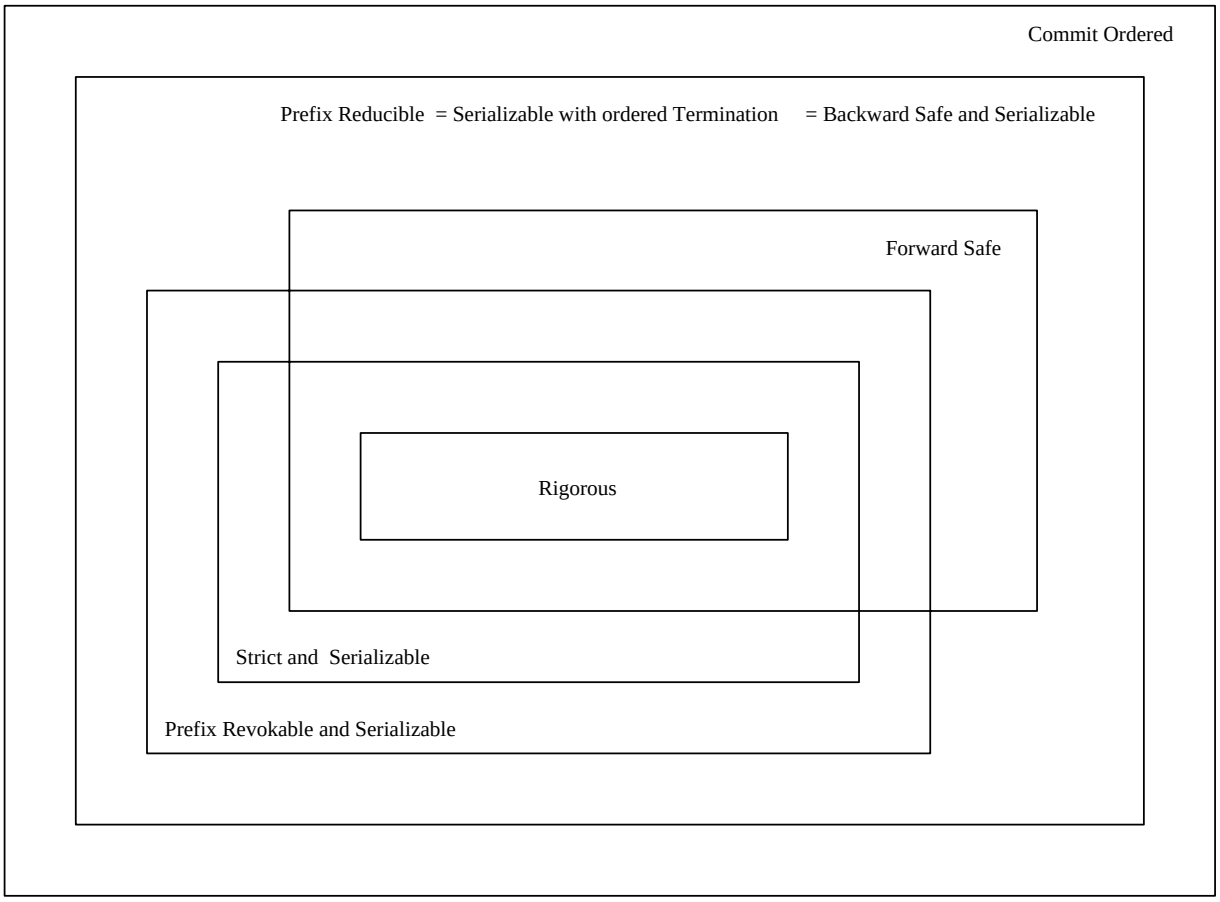

Figure 5: **Relationship between classes in perfect model**

As we have shown, we cannot assume in general that the state-independent commutativity relation is perfect. However, we can show that the following obvious property holds for the state-dependent commutativity in our model.

Lemma 7 *If two operations p and q state-dependently commute with respect to any sequence α, then p and q^{-1} or p^{-1} and q also state-dependently commute with respect to the sequences αq or αp, respectively. Furthermore, p^{-1} and q^{-1} state-dependently commute with respect to the sequence αpq.*

Proof: We show that if p state-dependently commutes with q with respect to the sequence α_1, the following three cases also hold:

- operation p^{-1} state-dependently commutes with q with respect to $\alpha_1 p$:

  The claim trivially holds when $p^{-1} = \lambda$. We consider the case $p^{-1} \neq \lambda$. Since p and q state-dependently commute with respect to the sequence α_1, then we know the return values in sequence $< \alpha_1\ p\ q\ \beta >$ are the same as in sequence $< \alpha_1\ q\ p\ \beta >$ for all possible sequences β.

  Consider $\alpha_2 = \alpha_1 p$. Since $< pp^{-1} >$ is effect-free, the return values of q and β are the same in $< \alpha_1\ p\ p^{-1}\ q\ \beta >$ and $< \alpha_1\ q\ \beta >$. Since p and q state-dependently commute, we obtain that the return values in $< \alpha_1\ p\ q\ p^{-1}\ \beta >$ are the same as in $< \alpha_1\ q\ p\ p^{-1}\ \beta >$. Since $< pp^{-1} >$ is effect-free, we obtain that the return values of q and β are the same in $\alpha_1\ q\ p\ p^{-1}\ \beta$ and $\alpha_1\ q\ \beta$. Since in addition p^{-1} always returns a constant value 1, we derive that the return values of q, p^{-1} and β are the same in sequences $< \alpha_1\ p\ q\ p^{-1}\ \beta >$ and $< \alpha_1\ p\ p^{-1}\ q\ \beta >$ and our claim holds.

- p^{-1} state-dependently commutes with q with respect to $\alpha_1 q$:

  Similarly to the above case.

- p^{-1} state-dependently commutes with q^{-1} with respect to the sequence $\alpha_1 pq$:

  Again, the claim holds trivially when either $p^{-1} = \lambda$ or $q^{-1} = \lambda$.

  Let us assume to the contrary that p^{-1} does not state-dependently commute with q^{-1}. Then the return values in sequence $\omega_1 = < \alpha\ p^{-1}\ q^{-1}\ \beta >$ are different from the return values in sequence $\omega_2 = < \alpha\ q^{-1}\ p^{-1}\ \beta >$ for every α. According to our assumption, operations q^{-1} and p^{-1} always return a constant value 1, therefore it could be only the sequence β which gives back different return values in ω_1 and ω_2.

  Let we select $\alpha = \alpha_1 pq$. Since p state-dependently commutes with q with respect to α_1, the return values of β in the sequence ω_1 are the same as in the sequence $< \alpha_1\ q\ p\ p^{-1}\ q^{-1}\ \beta >$. Since $< pp^{-1} >$ and $< qq^{-1} >$ are effect-free, the return values of β are the same in $< \alpha_1\ q\ p\ p^{-1}\ q^{-1}\ \beta >$ as in $< \alpha_1\ \beta >$. By similar arguments, we also obtain that the return values of β in ω_2 are the same as in the sequence $< \alpha_1\ \beta >$. A contradiction!

□

Note that the sequences of operations αq, αp and αpq naturally arise because well-formedness requires the previous execution of p or q if we talk about the commutativity of p^{-1} or q^{-1}.

4 Protocols

In this section we present protocols generating forward safe and backward safe serializable schedules. Recall that from the scheduler design point of view,a backward safeness does not guarantee serializability. Every backward safe protocol has to keep track and test for acyclic dependency not only for $< o_i, o_j >$ conflicts (for serializability), but also for $< o_i^{-1}, o_j >$ conflicts (for backward safeness). The forward safe protocol, on the other hand, needs to keep track of only $< o_i, o_j >$ conflicts. Thus, every protocol guaranteeing serializability can be easily extended with additional rules for ordering of transaction termination operations to generate forward safe schedules.

4.1 Forward Safe Protocols

In this section we first describe *forward safe pessimistic graph testing protocol* in detail and then show how other protocols generating forward safe schedules can be constructed. The forward safe pessimistic graph testing protocol uses serialization graph to generate schedules. The ordering of commit operations has to obey the order given by the graph, while the ordering of abort operations has to obey a reverse order than the one given by the graph (this can be always guaranteed by performing a group abort). The protocol is shown in Figure 6.

To illustrate the forward safe pessimistic graph testing protocol consider that the scheduler receives the following sequence of operations: $S : SInsert_1(x)\ SInsert_2(x)\ SInsert_3(y)\ SInsert_2(y)\ c_2\ c_3\ a_1$. After receiving the prefix $S : SInsert_1(x)\ SInsert_2(x)\ SInsert_3(y)\ SInsert_2(y)$ the serialization graph contains two edges: $< T_1, T_2 >$ and $< T_3, T_2 >$. When operation c_2 is received, transaction T_2 is put on the commit queue, since both T_1 and T_3 are still active. At the time c_3 is received, the transaction T_3 commits and node T_3 together with edge $< T_3, T_2 >$ is removed from the serialization graph. Nevertheless, T_2 still

1. Operation o_j (different from either commit or abort) is submitted. Insert appropriate edges and/or nodes to the serialization graph. If the graph contains a cycle, then o_j is rejected and a_j is submitted instead. Otherwise, submit o_j for execution.

2. c_j is submitted. If T_j has some predecessors in the serialization graph, put T_j on the commit queue. Otherwise, submit c_j for execution. After c_j gets executed, remove T_j from the serialization graph and test the commit queue whether any transactions can be committed.

3. a_j is submitted. If transaction T_j has been already aborted, do nothing. Otherwise, find the set of all transaction nodes reachable from T_j in the serialization graph, $\mathcal{T}$. Submit $a(\mathcal{T})$ for execution. After $a(\mathcal{T})$ is executed, all transactions from $\mathcal{T}$ are removed from the serialization graph and from the commit queue.

Figure 6: **Forward safe pessimistic graph testing protocol**

cannot commit, since its predecessor T_1 is not terminated yet. Finally, a_1 is received and both T_1 and T_2 are aborted by submitting $a(T_1, T_2)$.

To show that the protocol indeed generates forward safe schedules, it suffices to show that for any two operations o_i and o_j satisfying the assumptions of Definition 9, the following holds: if T_j commits then it does so after T_i and if T_i aborts, then it does so either after T_j or in parallel with it in a single group abort. Whenever two operations o_i and o_j satisfy the assumptions from Definition 9 and T_i is still active, the serialization graph contains an edge $< T_i, T_j >$. If T_i is already committed, then the edge $< T_i, T_j >$ is already removed. Consequently, T_j cannot commit until T_i does so due to the point 2 of the protocol (if T_i is still active, T_j is held on the commit queue until T_i terminates). Similarly, whenever T_i aborts, T_j is either already aborted (in which case it has been removed from the serialization graph) or it aborts together with T_j within a single group abort due to the point 3 of the protocol. The removal of nodes corresponding to committed transactions in the point 2 of the protocol does not lead to non-serializable schedules, since only the sinks of the graph are removed[Pap86]. The removed nodes from the graph in the point 3 of the protocol correspond to the aborted transactions and are irrelevant for serializability maintenance. Therefore, the forward safe pessimistic graph testing protocol indeed generates forward safe schedules.

Since the protocol delays transactions, it is necessary to show that it does not lead to deadlocks. Since each transaction waits only for its predecessors in a serialization graph to commit and the serialization graph is guaranteed to be acyclic at all times, the deadlock is impossible. Each transaction T_i remains in the commit queue until either all its predecessors commit, in which case T_i commits, or at least one of them aborts, in which case T_i is aborted too.

Several different protocols based on different paradigms can be constructed. Firstly, it is not difficult to see that the optimistic version of the forward safe serialization graph testing protocol can be easily obtained by performing the acyclicity test of the serialization graph lazily in the point 2 rather than in the point 1 of the protocol. Similarly, a non-blocking version of the protocol can be obtained by the following modification of the point 2 of the protocol: whenever there exists any predecessor of T_j in the serialization graph, rather than putting T_j on the commit queue, reject c_j and submit a_j instead.

It is also possible to extend any existing protocol (like the two phase locking, the timestamp ordering, etc.) with the rules 2 and 3 of the protocol to generate forward safe schedules. A combination of blocking caused by waiting for a lock and blocking caused by waiting to execute commit in the point 2 of the protocol cannot lead to deadlocks. Indeed, if transaction T_j waits for transaction T_i to release a lock on some data item,

the serialization graph contains edge $< T_i, T_j >$. Similarly, whenever transaction T_j waits for transaction T_i to commit, the serialization graph contains edge $< T_i, T_j >$. Therefore, a cycle in the wait-for-graph is also a cycle in the serialization graph.

As the reader has probably noticed, the point 3 of our protocol may lead to cascading aborts, i.e. an abort of one transaction may necessitate the abort of some other transactions in order to guarantee a forward safeness of the schedule. As it turns out, the class of rigorous schedules is the maximal subclass of forward safe schedules that avoid cascading aborts.

Theorem 7 *The class of rigorous schedules is the $\subseteq$-maximal subclass of the class of forward safe schedules that avoids cascading aborts.* □

Proof: Clearly, any scheduler generating forward safe schedules has to abort all transactions that are reachable from the aborting transaction in the serialization graph (otherwise forward safeness would be violated). Therefore, whenever the serialization graph contains only isolated nodes at all times (i.e. the schedule is rigorous), there are no cascading aborts. At the same time, a violation of rigourousness leads to cascading aborts, since each transaction can abort at any time. Therefore, the class of rigorous schedules is the $\subseteq$-maximal subclass of forward safe schedules avoiding cascading aborts. □

The cascading aborts are the price to be paid for the increased concurrency of forward safe schedules with respect to rigorous schedules. It is however reasonable to ask whether it is not possible to limit the number of transactions that are aborted as a consequence of aborting a single transaction T_i.

One possible way to limit the number of cascading aborts is to bound the length of every path in the serialization graph by some constant n. Therefore, not more than n transactions can be aborted at any time as a consequence of abort of any transaction T_i. The only modification required in the protocol is to modify the point 1: whenever a path longer than n should appear in the serialization graph as a consequence of scheduling operation o_j in the point 1 of the protocol, the transaction T_j is either delayed (and some deadlock detection is initiated) or aborted. Setting $n = 0$ reduces the class of schedules recognize by the modified protocol to the class of rigorous schedules. Whenever n grows, the degree of concurrency grows as well. However, also the number of transactions that may need to be aborted as a result of abort of some transaction is increased. Another way of limiting the number of cascading aborts is to decrease the conflict rate of forward operations by using state-dependent commutativity which is more liberal than state-independent commutativity.

4.2 Backward Safe Protocols

Since a backward safeness by itself does not guarantee serializability, the backward safe protocols therefore must guarantee not only a backward safeness, but also serializability. Serializability can be guaranteed by maintaining an acyclic serialization graph. In addition to that, the protocol must maintain a *termination graph* which is used to order the commit and abort operations. We define the termination graph as follows: the nodes of the graph are all non-aborted transactions in S. Whenever there are two operations in S, $o_i <_S o_j$ such that T_i does not abort before o_j and o_i^{-1} is in conflict with o_j, we add an oriented edge from T_i to T_j. Clearly, whenever the graph contains edge $< T_i, T_j >$ then T_j can commit only after T_i does so and T_i can either abort after T_j or in parallel with T_j in a single group abort. This implies that the committed projection of the termination graph has to be acyclic at all times (if it would contain a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ then by backward safeness we derive that $c_1 <_S c_2 <_S \ldots <_S c_n <_S c_1$. A contradiction!).

The acyclicity of both serialization and termination graphs can be maintained by any possible combination of pessimistic or optimistic, blocking or non-blocking and graph testing, two phase locking or timestamp ordering protocols. For illustration purposes, we show in Figure 7 a backward safe protocol using pessimistic

1. Operation o_j (different from commit and abort) is submitted. Perform two phase locking test. If it fails o_j is rejected and a_j is submitted instead. If T_j is delayed by waiting for a lock, trigger the appropriate deadlock detection method. Otherwise, add node T_j to the termination graph, if it is not already there. If there exists an operation o_i of non-aborted transaction T_i such that o_i^{-1} is in conflict with o_j, then add an edge $< T_i, T_j >$ into the termination graph.

2. c_j is submitted. Test whether all edges incident to T_j in the termination graph obey the timestamp order, i.e. the source of each edge has a greater timestamp than its sink. If not, reject c_j and submit a_j instead. Otherwise, test if there are any edges coming from T_j. Is yes, reject c_j and submit a_j instead, otherwise submit c_j for execution.

3. a_j is submitted. If transaction T_j has been already aborted, do nothing. Otherwise, find the set of all transaction nodes reachable from T_j in the serialization graph, $\mathcal{T}$. Submit $a(\mathcal{T})$ for execution. After $a(\mathcal{T})$ is executed, all transactions from $\mathcal{T}$ are removed from the serialization graph and from the commit queue.

Figure 7: **Backward Safe Hybrid Protocol**

blocking two phase locking to guarantee serializability and optimistic non-blocking timestamp ordering to guarantee backward safeness.

To illustrate the protocol from Figure 7 assume that the scheduler receives the following sequence of operations: $S : Incr_1(x)\ Incr_2(x)\ Incr_3(y)\ Incr_2(y)\ c_3\ c_2\ a_1$. Also assume that timestamps of the transactions are in the following order: $ts(T_3) < ts(T_2) < ts(T_1)$. If we assume that no transaction releases its locks until it submits its commit or abort operation, then the two phase locking serializability test admits the prefix $Incr_1(x)\ Incr_2(x)\ Incr_3(y)\ Incr_2(y)$ since there are no conflicts among the forward operations. At this point the termination graph contains two edges: $< T_1, T_2 >$ and $< T_3, T_2 >$. After operation c_3 is received, it gets submitted for execution since $ts(T_3) < ts(T_2)$ and there are no edges coming from node T_2 in the termination graph. The edge $< T_3, T_2 >$ together with the node T_3 are removed from the termination graph after c_3 is executed. When c_2 is received, it gets rejected and a_2 is submitted instead, since $ts(T_2) < ts(T_1)$, but edge $< T_1, T_2 >$ requires the opposite order of timestamps. After the abort of T_2 is received, $a(T_1, T_2)$ gets submitted for execution since T_1 is reachable from T_2. Finally, after receipt of the last operation of the schedule- a_1, nothing is done since T_1 has already been aborted. Note that the scenario would be the same even if $ts(T_1) < ts(T_2)$, since at the moment c_2 is received, there is an edge coming out from node T_2.

Similarly to the forward safe pessimistic graph testing protocol from Figure 6, the protocol might lead to cascading aborts. As it turns out, the strict schedules are the maximal subclass of the backward safe schedules avoiding cascading aborts:

Theorem 8 *The class of strict schedules is the $\subseteq$-maximal subclass of the class of backward safe schedules that avoids cascading aborts.* □

Proof: Similar to Theorem 7. □

All methods for limiting the number of cascading aborts mentioned in the context of forward safe protocols can be also applied here. The way we introduced backward operations in Section 2 implies that an backward operation depends only on the corresponding forward operation and does not depend at all on any other operations that were executed between the forward and its backward operation. This was done to simplify the way a recovery system performs undo by remembering only "old" values that the corresponding forward operation has changed. However, if we are willing to pay an extra price in complexity of the backward

operations, then some (not all!) conflicts (and consequently also cascading aborts) disappear. To illustrate, consider the following example:

Example 9: Consider a different implementation of a backward operation for operation $SInsert(x)$ from Example 1. We assume that such a backward operation depends not only on its corresponding forward operation, but also on all operations that have been invoked on the object Set after the forward operation. Namely, we assume that the backward operation is passed the return values of all non-aborted *SInsert()* and *SDelete()* operations that have been invoked on Set after the forward operation. The backward operation does not perform any update of the database provided there is at least *SInsert()* or *SDelete()* executed between the forward and the backward operation which overwrote effects of the forward operation. Whenever there is no such operation, the backward operation not only undoes effects of its corresponding forward operation, but also of all other forward operations that has been previously "deferred".

Such an implementation of a backward operation commutes with both $SInsert(x)$ and $SDelete(x)$. However, it does not commute with $Test(x)$. Thus the only cascading aborts are of those transactions that invoked $Test(x)$ after the forward operation has been issued. □

5 Conclusion

In this paper we discussed prefix reducibility within the framework of the general model with semantically rich operations. We have demonstrated that the class of schedules serializable with ordered termination, does not characterize all prefix reducible schedules in this general case. We, however, identified the conditions on the model under which such a characterization is exact. We have shown that with normal commutativity relations the class of serializabe with ordered termination schedules (SOT) and the class of prefix reducible schedules coincide and for perfect commutativity relations the general model becomes isomorphic to the read/write model. In the general case, we have argued that the only practically feasible classes of schedules that allow an uniform treatment atomicity and serializability are the classes of forward safe schedules and serializable backward safe schedules.

We believe that there are at least two cases when an unified treatment of transaction atomicity and serializability in models with semantically rich operations is important:

In distributed database environments a transaction is often considered as a partial order of different local sub-transactions. Each such sub-transaction can be in turn considered as an operation. Thus, to prove correctness of execution of such transactions in a failure prone distributed database environment a formal model of such transactions must be developed.

Recently the model of multilevel transactions[BBG89, Wei91, LMWF94] become widely accepted in modelling operations on abstract data types which can be correctly executed without requiring serializability of its read/write operations. It has been shown in [Wei91] that the correctness of the entire multilevel schedule can be under certain restrictions reduced to guaranteeing correctness for each level with respect to only operations on the level below. Since these operations in general are not only read/write accesses on pages, this again postulates a need for investigation of single level models with arbitrary set of operations.

Acknowledgement

The authors would like to thank Gerhard Weikum for numerous inspiring discussions, and many valuable comments and observations.

References

[AD94] P. Anastassopoulos and Jean Dollimore. A Unified Approach to Distributed Concurrency Control. In Won Kim and Fredrick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications, ACM Press*. Addison-Wesley Publishing Company, 1994.

[AAE94] G. Alonso, D. Agrawal and A. El Abbadi. Reducing Recovery Constrains on Locking Based Protocols. *Proc. ACM Principles of Database Systems*, 1994.

[AVA+94] G. Alonso, R. Vingralek, D. Agrawal, Y. Breitbart, A. Abbadi, H. Schek, and G. Weikum. A Unified Approach to Concurrency Control and Transaction Recovery. *Information Systems*, 19(1), 1994.

[BBG89] C. Beeri, P. Bernstein, and N. Goodman. A Model for Concurrecy in Nested Transactions Systems. *Journal of the ACM*, 36(2), 1989.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BS92] Y. Breitbart, and A. Silberschatz. Strong Recoverability in Multidatabase Environments. *Proceedings of RIDE-TQP IEEE*, 1992.

[BR92] B.R. Badrinath and K. Ramamritham. Semantics-based Concurrency Control: Beyond Commutativity. *ACM Transactions on Database Systems*, 17(1), March 1992.

[BGRS91] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz and A. Silberschatz. On Rigorous Transaction Scheduling. *IEEE Transactions on Software Engineering*, 17(9), 1991.

[HH88] T. Hadzilacos and V. Hadzilacos. Transaction Synchronization in Object Bases. *Proc. ACM Principles of Database Systems*, 1988.

[HH91] T. Hadzilacos and V. Hadzilacos. Transaction Synchronization in Object Bases. *Journal of Computer and System Sciences*, 1991.

[HW88] M. P. Herlihy and W. E. Weihl. Hybrid Concurrency Control for Abstract Data Types. *Proc. ACM Principles of Database Systems*, 1988.

[LMWF94] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgen Kaufmann, San Mateo, CA, 1994.

[MGG86a] J. Moss, N. Griffeth, and M. Graham. Abstraction in Concurrency Control and Recovery Management (revised). Tech. rept. coins Technical Report 86-20., University of Massachusetts at Amherst, March 1986.

[MGG86b] J. Moss, N. Griffeth, and M Graham. Abstraction in Recovery Management. *ACM SIGMOD Conference*, 15(1), 1986.

[O'N86] P.E. O'Neil. The Escrow Transactional Method. *ACM Transactions on Database Systems*, 11(4), 1986.

[Pap86] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[RKS92] R. Rastogi, H. F. Korth, and A. Silberschatz. Strict Histories in Object-Based Database Systems. Technical Report, Matsushita Information Technology Laboratory, 1992.

[RKS93] R. Rastogi, H. F. Korth, and A. Silberschatz. Strict Histories in Object-Based Database Systems. In *Symposium on Principles of Database Systems*, 1993.

[Raz92] Y. Raz. The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous, Multiresource Manager Enviroment Using Atomic Commitment. *Proc. 18th International Conference on Very Large Data Bases*, 1992.

[SWY93] H.-J. Schek, G. Weikum, and H. Ye. Towards a Unified Theory of Concurrency Control and Recovery. *Proc. ACM Principles of Database Systems*, 1993.

[Wei88] W.E. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, 37(12), 1988.

[Wei89] W.E. Weihl. The Impact of Recovery on Concurrency Control. *Proc. ACM Principles of Database Systems*, 1989.

[Wei91] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1), 1991.