Unifying concurrency control and recovery of transactions with semantically rich operations

R. Vingralek, H. Hasse-Ye, Y. Breitbart, H.-J. Schek

Department of Computer Science, University of Kentucky, Lexington, KY 40506, USA
Institute for Information Systems, ETH Zentrum, CH-8092 Zurich, Switzerland

Abstract

The classical theory of transaction management contains two different aspects, namely concurrency control and recovery, which ensure serializability and atomicity of transaction executions, respectively. Although concurrency control and recovery are not independent of each other, the criteria for these two aspects were developed orthogonally and as a result, in most cases these criteria are incompatible with each other.

Recently a unified theory of concurrency control and recovery for databases with read and write operations has been introduced in [19, 1] that allows reasoning about serializability and atomicity in a failure prone environment with read/write operations was introduced. Several protocols were developed to generate such schedules by a database concurrency control mechanism.

We present here a unified transaction model for databases with an arbitrary set of semantically rich operations. We investigate constructive characterization of the class of prefix reducible schedules with semantically rich operations. It turns out that unlike databases with only read/write operations, the exact characterization of prefix reducible schedules in databases with arbitrary operations is rather infeasible. Thus, we propose here several sufficiently rich subclasses of prefix reducible schedules, and design concurrency control protocols that guarantee both serializability and atomicity for schedules from these classes.

1. Introduction

Transaction management in database systems with an arbitrary set of operations [12, 13, 21, 8, 22, 9, 3, 15, 16, 11] is becoming increasingly important. In this paper we develop a transaction management model for transactions over an arbitrary but finite
set of operations. Our model is based on a semantic, high-level commutativity relation defined for any pair of operations. If two operations from different transactions commute, the transaction manager is free to execute these operations in any order. If two operations from different transactions do not commute (we call such operations conflicting), the transaction manager ensures "semantic" serializability, i.e. serializability with respect to the (semantic) commutativity relation.

Our approach is further based on the assumption that with each operation invocation an undo (or backward) operation must be given. The sole purpose of the undo operation is to erase from the database all observable effects of the corresponding operation invocation. In order to unify concurrency control and recovery, commutativity is also defined for the undo operations. Thus we are able to develop an unified theory based on (semantic) serializability with respect to the commutativity relation that encompasses the regular (forward) operations and the undo operations. Our model is able to reason in a uniform manner about transaction concurrency and recovery using both standard and nonstandard transaction concurrency and recoverability models (such as workflows for transactions and compensation for recoverability).

For a practical implementation we assume that a transaction manager is supplied with a conflict detection method that for any two invoked operations decides whether these two operations commute or not. Having such a method facilitates an extensible approach in unifying of serializability and atomicity notions and enables design of transaction managers that ensure both serializability and atomicity with a single algorithm.

Recently, in [19, 1] a similar unified model of transaction management with read/write operations was discussed. In particular, the authors introduced the class of prefix reducible schedules, \textit{PRED}, and argued that any transaction manager should not generate other than \textit{PRED} schedules to guarantee both serializability and atomicity of user transactions. In these papers, however, the authors mostly concentrated on the traditional notion of transactions. Issues of recovery have been dealt by issuing undo operations for write operations within limits of a transaction log kept by the transaction manager. By restricting the model to the classical read/write transaction model, the authors were able to provide a uniform correctness criteria for schedules that contained explicit recovery actions for aborted transactions.

In this paper, we generalize the previous work of [1] by expanding the notion of transaction. Unlike the previous work, we consider here transactions over an arbitrary but finite set of operations. Such an approach expands the traditional transaction notion by including transactions defined on data objects of different abstract data types (\textit{ADT}). We develop a unified approach to deal with a concurrency and failure atomicity by explicitly including transaction recovery actions (which in this case could include compensating operations as well!) into a transaction schedule. Following [1] we provide constructive characterizations for classes of schedules whose serializability guarantees both consistency and atomicity.

When we started this work, we assumed that generalizing the results of [1] for an arbitrary set of operations should be straightforward. Unfortunately, it proved to be not as simple. It turns out that we must distinguish the case where the undo or
compensating operations have the same conflict behaviour as their forward operations from the case where this does not hold. The practical consequence of that is that the protocol proposed in [1] that guarantees a serializability of schedules that include recovery actions could not be exploited in our new model.

To obtain practically feasible protocols, we explore two approaches: First, we restrict the class of so called prefix reducible schedules from [1] to a subclass of of safe schedules. We discuss the properties of these schedules and argue that safe schedules are practically feasible and allow a uniform treatment of serializability and atomicity in transaction models with an arbitrary set of operations. Second, we impose restrictions on commutativity relations that enable us to constructively characterize all prefix reducible schedules.

Our definition of commutativity closely relates to the definitions given in [21,22]. However, unlike [21,22], 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 [12,13] 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 reducible schedules introduced here. Rastogi et al. [15,16] develop a theory of strict schedules. In [20] we introduced a notion of safe schedules that is more general than the notion of strict schedules introduced in [15,16].

This paper extends preliminary results from [20] by considering several additional schedule classes, providing complete proofs of results announced in [20], and, in addition, by designing protocols that generate schedules from our classes and only schedules from our classes. The rest of the paper is organized as follows. In the next section we introduce our transaction 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 [1]. In Section 4 we define a protocols generating safe schedules and prove their correctness. Section 5 concludes the paper.

2. Model description

In this section we describe our transaction model. The main purpose of this model is to unify concepts of serializability and failure atomicity of concurrently executed transactions defined over an arbitrary but finite set of operations. Similar to [19], our model is based on three basic principles:

- Operations are defined on arbitrary abstract data types and for each do (or forward) operation, we define an undo (or backward) operation that undoes the effects of a corresponding do operation.
All recovery related operations (i.e. backward operations) must be explicitly present in transactions and consequently in the schedule which represents the execution of regular forward operations.

Serializability with respect to the commutativity relation of forward and backward operations is used to reason about the correctness of schedules including recovery operations, and especially, about the interference of recovery related backward and regular forward operations in a schedule.

To formalize these ideas, we first discuss in Section 2.1 a notion of forward operations defined on data objects of any abstract data type (ADT) and backward operations whose sole purpose is to recover from the effects of a corresponding forward operation. We model all possible database states by sequences of operations and their return values to avoid an explicit definition of database states. Based on this we define the notion of effect-free sequences of operations and use the notion of effect-free sequences for an introduction of backward operations.

We introduce then in Section 2.2 a richer notion of commutativity, which is not only valid for simple read and write operations but also valid for any arbitrary ADT operations. This semantic richer notion of commutativity is the backbone of extension of the traditional read/write model to our model with general operations. The classical definition of conflict preserving serializability (CSR) is based on the notion of commutativity, which was limited to the simple read and write operations. With our new richer notion of commutativity, the classical criterion CSR is immediately applicable for semantically rich operations. In Section 2.3 we reconsider the definitions of transactions, schedules and the criterion of CSR.

In Section 2.4 we complete the model description with an introduction of expanded schedules where all recovery actions are explicitly defined by adding an undo operation for every forward operation of an aborted transaction in the same schedule. Consequently, an expanded schedule consists of both forward and backward operations and we argue that if a scheduler guarantees serializability of an expanded schedule, then it guarantees both serializability and recoverability for a given set of users transactions. Furthermore we consider an elimination of forward–backward pairs of operations to model an intuitive notion that an execution of a forward operation immediately followed by an execution a corresponding backward operation leaves no effect neither on a database nor on the data viewed by other transactions. If such elimination allows us to get to a serializable schedule that consists of only forward operations, then such a schedule should be "correct" from both serializability and recoverability viewpoints. Such reducibility is captured with criteria of reducibility and prefix reducibility.

### 2.1. Operations, database states

A database $DB$ consists of a set $D$ of data objects $d$ of any abstract data type and a set of $O$ of operations $o$ (called in the sequel forward operations). An operation invocation $[21, 22]$ is an operation $o$ from $O$ that has one or several data objects $d$ from $D$ as input. In $[21, 22]$, an operation invocation event is followed by an operation execution event.
delivering the return value. Generally, a transaction manager may interleave operations invocations and executions events. We, however, for simplicity assume that operations invocation and execution events for any two operations may not interleave. Thus, in what follows when we talk about an operation, we always understand an operation invocation. We assume that in addition to operations from \( O \) there are two special termination operations: \texttt{abort} (denoted by \( a \)) and \texttt{commit} (denoted by \( c \)).

For each operation \( o \) from \( O \), apart from two special operations \texttt{commit} and \texttt{abort}, we introduce an \texttt{undo} or a \texttt{backward} operation \( o^{-1} \) and let \( O^{-1} \) be the set of all undo operations defined for operations in \( O \). We require that every operation \( o \in O \cup O^{-1} \) after it is executed, returns to the caller some value (called \textit{return value}). The details of operation 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 \texttt{read(x)} is the value of \( x \) and the return value of \texttt{write(x)} is the value which was overwritten. The return values of the embedded SQL operations \texttt{insert}, \texttt{select}, \texttt{delete} and \texttt{update} include the entire \texttt{SQLCA} area.

To introduce a notion of \textit{database state} we first discuss a notion of an operation sequence. Operation sequences over \( O \cup O^{-1} \) are denoted by \( \sigma, \alpha \) and \( \beta \). Operation sequence \( \alpha = p_1 p_2 \ldots p_n \) means that this sequence contains \( n \) operations \( p_1, p_2, \ldots, p_n \) and these operations are executed in the order they appear in the sequence. If several operation invocations of the same operation appear in the sequence we use indices to distinguish different invocations. For example, \( \beta = \text{write}_a \text{write}_b \) contains two write operation invocations. The operation sequence \( \alpha \sigma \beta \) means that operation sequence \( \alpha \) executes before \( \sigma \) and \( \sigma \) before \( \beta \).

A backward operation can be invoked only after the related forward operation. We say that a sequence of operations \( \alpha \) over the set \( O \cup O^{-1} \) is \textit{well-formed} if every backward operation \( o^{-1} \) in \( \alpha \) is preceded by its corresponding forward operation \( o \).

By definition, an empty sequence is an initial database state. Starting from the initial database state, called \( s_0 \), any database state \( s \) is defined as a sequence of return values for some well-formed sequence of operations \( \alpha \) over the set \( O \cup O^{-1} \). We denote this by \( s = s_0 \alpha \). Let \( s_1 = s_0x_1 \) and \( s_2 = s_0x_2 \) be two database states. We say that \( s_1 \) and \( s_2 \) are equivalent if and only if for any well-formed sequence of operations \( \beta \), return values of \( \beta \) applied to the database state \( s_1 \) are the same as return values of \( \beta \) applied to the database state \( s_2 \). Consequently, operation sequences are the only means to generate a database state or to detect an equivalence between any two database states.

The intuitive meaning of a backward operation \( o^{-1} \) is that all "recognizable" changes in the database that \( o \) did are backed out by executing the corresponding operation \( o^{-1} \). Therefore, the changes in the database caused by \( o \) that can be detected by other operations through their return values are undone by executing \( o^{-1} \). Below we formalize the above requirement on backward operations by introducing first the definition of \textit{effect-free} sequences.
Definition 1. We call a sequence of operations $\sigma$ effect-free if, for all possible sequences of operations $\alpha$ and $\beta$ such that sequences of operations $\alpha \sigma \beta$ and $\alpha \beta$ are well-formed, the return values of operations in $\beta$ from the sequence $\alpha \sigma \beta$ are the same as the return values of operations in $\beta$ from sequence $\alpha \beta$.

With the notion of effect-free sequences we can formalize the requirement of backward operations: For every operation invocation $o \in O$ and its corresponding backward operation invocation $o^{-1} \in O^{-1}$ we require that the sequence $o o^{-1}$ be an effect-free sequence.

The requirement, that, for every operation $o$ from $O$ and its inverse operation $o^{-1}$ from $O^{-1}$, the sequence $o o^{-1}$ is effect-free, has impact on the application designers: whoever designs the forward operation $o$ should also provide the undo operation $o^{-1}$, since it is him/her who knows the semantics of $o$ and thus also knows how to undo it.

The above requirement implies that backward operations are dependent on its forward operation. The forward operation's return value is passed to the corresponding backward operation as one of its input parameters when the undo operation is invoked. For example, in the read/write model, for a write operation, the value which is overwritten by the write operation is the input parameter of the corresponding backward operation write$^{-1}$.

A special case of an effect-free sequence is a sequence that contains only one operation, for example a read operation in the read/write model. Such an operation $o$ does not make any changes to a database. So its corresponding undo operation $o^{-1}$ does not need to do anything. The backward operation of an effect-free forward operation is called null operation and is denoted by $\lambda$.

As we mentioned earlier, the main purpose of backward operations is to undo the recognizable effects of corresponding forward operations. From this viewpoint any backward operation must successfully complete. In addition, if backward 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 value of any backward operation is the same. We assume throughout the paper that a return value of any backward operation $o^{-1} \neq \lambda$ is constant 0. If $o^{-1} = \lambda$, its return value is a reserved constant null.

2.2. Commutativity

Consider the well-formed sequence of operations $\alpha p q \beta$, where each operation is from $O \cup O^{-1}$. If permuting operations $p$ and $q$ does not change their return values and also the return values of operations from $\beta$, then we say that $p$ and $q$ commute. There are two possible cases that lead to two alternative definitions of commutativity:

- Permuting $p$ and $q$ does not change their return values regardless of which operation sequence $\alpha$ precedes them.
- Permuting $p$ and $q$ does not change their return values only for some $\alpha$.

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 operational sequences \( \alpha \) and \( \beta \) over \( O \cup O^{-1} \), such that \( \alpha p q \beta \) and \( \alpha q p \beta \) are well-formed, the return values of operations from \( \beta \) 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 an operational sequence \( \alpha_0 \) over \( O \cup O^{-1} \) if and only if, for any sequence of operations \( \beta \) over \( O \cup O^{-1} \), such that \( \alpha_0 p q \beta \) and \( \alpha_0 q p \beta \) are well-formed, the return values of operations from \( \beta \) and the return values of \( p \) and \( q \) in \( \alpha_0 p q \beta \) are the same as in \( \alpha_0 q p \beta \).

If we say that \( p \) and \( q \) state-dependently commute, it may very well mean that there are database states when \( p \) and \( q \) can be permuted as well as that there are database states where \( p \) and \( q \) cannot be permuted. On the other hand, saying that \( p \) and \( q \) state-dependently do not commute means that regardless of the database state, \( p \) and \( q \) cannot be permuted. Note that the sequence \( \alpha_0 \) corresponds to the existence of some database state \([12, 13, 15, 16]\) in which the two operations commute.

The following examples illustrate the above concepts. We use a commutativity matrix to represent the commutativity of each pair of operations over a given set of operations. In all examples we assume that operations invoked on different objects always commute. The matrix shows whether two operations on the same object commute (which is denoted by \(+\)) or not (which is denoted by \(-\)). Note in the general case, e.g. when SQL operations are used, the conflict test must consider all input parameters of the operation invocations \([7, 18]\).

Example 1. Reconsider the read and write operations. The description of the operations can be restated as in the following:

- \( \text{read}(x) \) — return value is the current value of object \( x \).
- \( \text{write}(x, v) \) — changes the value of object \( x \) with \( v \) and returns the value \( v_0 \) which was overwritten.
- \( \text{read}^{-1}(x) \) — a \( \lambda \) operation returning always null.
- \( \text{write}^{-1}(x, v_0) \) — writes the original value \( v_0 \) of the object \( x \) back. It returns always constant 0.

The state-independent commutativity relation is shown in Fig. 1. For illustration we consider the following case. The indices, \( a \) and \( b \), in the operation sequence are used to distinguish operation invocations of the same type.

- \( \text{write}(x, v) \) does not commute with itself. Consider the following two sequences \( \sigma_1 = \text{w}_a(x, v_1) \; \text{w}_b(x, v_2) \) and \( \sigma_2 = \text{w}_b(x, v_2) \; \text{w}_a(x, v_1) \). Assume at the beginning the value of the object \( x \) is \( v_0 \) and the values \( v_0, v_1 \) and \( v_2 \) are all different. The return value of \( \text{w}_a(x, v_1) \) in the sequence \( \sigma_1 \) is \( v_0 \) and the return value of \( \text{w}_b(x, v_1) \) in the sequence \( \sigma_2 \) is \( v_2 \). It is obvious, that \( \text{write}(x, v) \) does not commute with itself.
The commutativity relation represented in Fig. 1 has a special property: if two operations \( p \in O \) and \( q \in O \) do not commute with each other, then all other combinations of these two operations, i.e. \( p \) with \( q^{-1} \), \( p^{-1} \) with \( q \) and \( p^{-1} \) with \( q^{-1} \) do not commute either (provided that none of these operations is null operation. A null operation commutes with any operation, by definition). If two operations \( p \in O \) and \( q \in O \) do commute with each other, then all other combinations of these two operations, i.e. \( p \) with \( q^{-1} \), \( p^{-1} \) with \( q \) and \( p^{-1} \) with \( q^{-1} \) also commute. This property called perfectness which is discussed in Section 3.4 has a significant impact on the unified theory.

The following example shows that there exist semantically rich operations, which do not necessarily have the perfectness property:

**Example 2.** Let set \( S \) be a data object in the database with the following operations defined on it:

- **Insert**\((x)\) – Inserts element \( x \) into the set \( S \). If \( x \) was already in the set, the operation does nothing. It returns constant 1, if \( x \) was actually inserted by the operation. Otherwise, the operation returns constant 0.
- **Insert^{-1}(x)**, where \( x \) is either the value that was inserted by the corresponding forward operation, or \( x = 0 \) if the corresponding forward operation did not have to insert \( x \), because it was already there (we assume that the transaction manager maintains a log to be able to determine the argument to be passed to the backward operation). If \( x \neq 0 \), **Insert^{-1}(x)** deletes element \( x \) from the set, otherwise it does nothing. It always returns constant 0.
- **Delete**\((x)\) – Deletes element \( x \) from the set \( S \). If \( x \) was not in the set, the operation does nothing. It returns constant 1, if \( x \) was actually deleted by the operation. Otherwise, the operation returns constant 0.
- **Delete^{-1}(x)**, where \( x \) is either the value that was deleted by the corresponding forward operation, or \( x = 0 \), if the corresponding forward operation did not have to delete \( x \), because it was not already there. If \( x \neq 0 \), **Delete^{-1}(x)** inserts element \( x \) into the set, otherwise it does nothing. It always returns constant 0.
- **Test**\((x)\) – Returns constant 1 if element \( x \) is in a set, otherwise it returns constant 0.
- **Test^{-1}(x)** – A λ operation returning always null.

Fig. 2 shows the state-independent commutativity relation for the operations defined above. Below we illustrate some of the cases:
Fig. 2. State-independent commutativity relation.

- **Insert**(x) does not state-independently commute with itself. Indeed, it is easy to verify that return values in the sequences **Delete**(x) **Insert**<sup>v</sup>(x) **Insert**<sup>b</sup>(x) (indices v and b distinguish among different invocations of the same operation within a sequence) and **Delete**(x) **Insert**<sup>b</sup>(x) **Insert**<sup>v</sup>(x) are not the same. At the same time, the two operations state-dependently commute with respect to the sequence **z**<sub>0</sub> = **Insert**(x).

- **Insert**<sup>-1</sup>(x) state-independently commutes with itself. Consider a sequence of operations α **Insert**<sup>-1</sup>(x) **Insert**<sup>-1</sup>(x) β. At least one of the **Insert**<sup>-1</sup>(x) does not change the database. Consequently, return values of operations in β are the same regardless whether **Insert**<sup>-1</sup>(x) was executed first and followed by **Insert**<sup>-1</sup>(x) or vice versa.

It is important to mention that the commutativity of the above operations does not have the symmetric perfectness property. For example operation **Insert** does not commute with itself but two backward operations **Insert**<sup>-1</sup> with **Insert**<sup>-1</sup> commute with each other. The same is the case with **Delete**.

Note that if, instead of state-independent commutativity we consider a state-dependent commutativity, then we can easily show that the only pair of operations that state-dependently does not commute is **Insert**(x) and **Delete**(x).

Sequences α and β in our definition of commutativity can contain both forward and backward operations. A traditional definition of commutativity given in [21, 22] allowed only forward operations in sequences α and β. The following example demonstrates that this is an important distinction. Namely, the two operations that commute according to the definition in [21, 22] do not necessarily commute in our model.

**Example 3.** As in Example 2, let set S be a data object in the database with the following operations defined on it:

- **SI**(x) – Inserts element x into a set. If x was already in the set, the operation does nothing. It returns constant 1.
- **SI**<sup>-1</sup>(x), where x is either the value that was inserted by the corresponding forward operation, or x = 0, if the corresponding forward operation did not have to insert x (since it was already in the set). As in the previous example, we assume that the undo operations uses a log to determine what element was inserted by the corresponding
forward operation. If \( x! = 0 \), \( SIZ^{-1}(x) \) deletes element \( x \) from the set, otherwise it does nothing. It always returns constant 0.

- **Delete(x)** – Deletes element \( x \) from a set. If \( x \) was not in the set, the operation does nothing. It returns constant 1, if \( x \) was actually deleted by the operation. Otherwise, the operation returns constant 0.

- **Delete^{-1}(x)**, where \( x \) is the value that was deleted by the corresponding forward operation, or \( x = 0 \), otherwise. If \( x! = 0 \), \( Delete^{-1}(x) \) inserts element \( x \) into the set, otherwise it does nothing. It always returns constant 0.

- **Test(x)** – Returns 1 if element \( x \) is in a set, otherwise it returns 0.

- **Test^{-1}(x)** – A λ operation returning null.

1. Consider only the do operations: \( SI \) and **Delete** and **Test**. It is easy to show that two \( SI(x) \) operations commute both state dependently and state independently according to the traditional definition of commutativity [21,22].

2. Consider now all the operations: \( SI, SI^{-1}, Insert, Insert^{-1}, Delete, Sdelete^{-1}, Test \) and **Test^{-1}**. \( SI \) and \( SI \) do not commute state-independently. The operation **Test(x)** returns 1 in the sequence \( Delete(x) Sib(x) S&(x) SIZ;'(x) Test(x) \) and 0 in the sequence \( Delete(x) Sib(x) S&(x) SIZ;'(x) Test(x) \).

The results presented here are valid for both notions of commutativity. However, to simplify our presentation, in the rest of the paper we assume the state-independent commutativity.

### 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 \( \mathcal{F} \) is a partial order \( <_S \) of all operations of all transactions in \( \mathcal{F} \) such that for any transaction \( T_i \) in \( \mathcal{F} \), \( <_i \) 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 \( a(T_{i_1},\ldots,T_{i_k}) \), where \( T_{i_1},\ldots,T_{i_k} \) are from the transaction set \( \mathcal{F} \). 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 order of these aborts is irrelevant. Note that \( a(T_i) = a_i \).

Transaction \( T_i \) is said to be committed (aborted) in \( S \) if \( S \) contains \( c_i \) (\( a_i \) or \( a(\ldots,T_{i_1},\ldots) \)) 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).

**Definition 4.** Two operations \( p \) and \( q \) (state-dependently or state-independently) **conflict** in schedule \( S \) if and only if \( p \) and \( q \) do not (state-dependently or state-independently) commute and they belong to different transactions.
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. 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 (CSR) if its committed projection is conflict equivalent to a serial schedule.

2.4. Expanded schedules

The criterion of conflict serializability is only defined on the committed projection and does not capture aborted transactions. 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 $o \ o^{-1}$ are eliminated from the consideration (since they do not make any effect on a schedule) then the issues of serializability and atomicity are treated by such a scheduler in an uniform way. Assume that a scheduler has produced so far a schedule $(S, <_S)$. Assume that at this point a system failure has occurred. Then, after the system has recovered, the effects of all transactions that were either active or aborted in $S$ are eliminated from the database and the effects of all transactions that were committed in $S$ are restored in the database. Consequently, in order to generate a schedule that contains recovery actions explicitly, we would assume that every action of either aborted or active transactions in the original schedule must be undone by submitting a corresponding undo operation. Formalizing these ideas [1], for each schedule $(S, <_S)$, we define an expanded schedule $(\tilde{S}, <_{\tilde{S}})$ as follows.

**Definition 5 (Alonso et al. [1])**. 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 that 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 $\tilde{o}_i \in \tilde{S}$.

   (b) Active transactions are treated as aborted transactions, by adding a group abort $a(T_{i1} \ldots T_{ik})$ as a maximal element of $\tilde{S}$, where $T_{i1} \ldots T_{ik}$ 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 $a(T_{j1} \ldots T_{jn})$ is replaced with a sequence of $c_{i1}, \ldots, c_{in}$.

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 $\hat{S}$.

(d) If $o_n <_S a(T_{i_1}, \ldots, T_{i_k})$ and some undo operation $o_j^{-1}$ ($j \in \{i_1, \ldots, i_k\}$) conflicts
with $o_n$, then $o_n <_S o_j^{-1}$.

(e) If $a_i <_S a_j$ for some $i \neq j$ and $o_i^{-1}$ conflicts with $o_j^{-1}$, then $o_i^{-1} <_S o_j^{-1}$.

We say that schedule $S$ is reducible (RED) [19] if there exists at least one expanded
schedule $\hat{S}$ 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 $\hat{S}$ from different transac-
tions such that $o_1 <_S o_2$ and $o_1$ commutes with $o_2$ and there is no $p \in \hat{S}$ such
that $o_1 <_S p <_S o_2$, then the order $o_1 <_S o_2$ can be replaced by $o_2 <_S o_1$.

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

To illustrate, consider schedule $S_1$: $\text{Delete}_1(x) \text{Insert}_2(x) \text{Test}_3(x) c_2 a_3$. Its expansion
is $\hat{S}_1$: $\text{Delete}_1(x) \text{Insert}_2(x) \text{Test}_3(x) c_2 \text{Test}_3^{-1}(x) c_3 \text{Delete}_3^{-1}(x) c_1$ and it is not
reducible. On the other hand, schedule $S_2$: $\text{Delete}_1(x) \text{Insert}_2(x) \text{Test}_3(x) c_2 c_1 a_3$
with expansion $\hat{S}_2$: $\text{Delete}_1(x) \text{Insert}_2(x) \text{Test}_1(x) c_2 c_1 \text{Test}_3^{-1}(x) c_1$ is reducible.

Our goal is to obtain an expanded schedule for a given schedule and to design
the transaction manager in such a way that it generates schedules such that when it
is expanded explicitly with backward operations it still remains serializable after the
application of both commutativity and undo rules. That is the scheduler must generate
at least a reducible schedule to enable us to treat schedule serializability and failure
recovery in an uniform manner.

The transaction manager dynamically generates a schedule of executed transactions.
That means that at any time a schedule may contain operations of active transactions.
Therefore the transaction manager should not only ensure a serializability of committed
transactions, but also require that any prefix of the schedule would be also serializable
since we never can be sure whether a transaction will commit in the future. That
means the property of schedule reducibility should be prefix closed (i.e. if a schedule
is reducible every prefix should be also reducible).

Unfortunately, the class of reducible schedules is not prefix-closed and hence cannot
be used for online scheduling of transactions [19]. We resolve it by requiring the
schedule to be prefix reducible:

**Definition 6.** A schedule $S = (A, <)$ is prefix reducible (PRED) if every prefix of $S$
is reducible.

For example, schedule $S_2$ given above is reducible but not prefix reducible, while
schedule $S_3$: $\text{Delete}_1(x) \text{Insert}_2(x) \text{Test}_3(x) c_1 c_2 a_3$ is prefix reducible. Similar to [19]
we consider a class of prefix reducible schedules as a class of schedules that allow to unify the notions of transaction serializability and atomicity.

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 [1] 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 a practical design of schedulers. We therefore define subclasses of prefix reducible schedules possessing simpler characterizations 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 \((o_i, o_i')\) in an expanded schedule \(\tilde{S}\). If there are no other operations between \(o_i\) and \(o_i'\) in \(\tilde{S}\), then this pair can be eliminated using the undo rule. Assume now that there are some operations between \(o_i\) and \(o_i'\) in \(\tilde{S}\). Let \(o_1, \ldots, o_n\) be operations between \(o_i\) and \(o_i'\) such that each \(o_k\) conflicts with \(o_{k+1}\) \((k \in \{1, n-1\}\), \(o_i\) conflicts with \(o_1\) and \(o_n\) conflicts with \(o_n'\). Then, to eliminate the pair \((o_i, o_i')\) 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 \(o_k\) 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 \((o_i, o_i')\) in \(\tilde{S}\) whether it can be removed from the schedule.

Let \(S\) be a schedule and \(\tilde{S}\) its expansion. To characterize the 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 \(o_i'\) 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 \(o_i'\) in \(RG(\tilde{S})\).

**Proof.** Clearly, whenever there exists a path of pairwise conflicting operations from \(o_i\) to \(o_i'\) then \(o_i\) and \(o_i'\) cannot be moved together by use of the 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 $S$ nonreducible and exit. If such a pair does not exist and $\tilde{S}$ does not contain any backward operations, declare the schedule $S$ reducible and exit.
4. If such a pair does exist, remove it from $RG(\tilde{S})$ along with all edges incident 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 $S_3$: $Insert_1(x) Delete_2(x) Insert_3(x) a_1 a_2 a_3$. Its expansion is $\tilde{S}_3 = Insert_1(x) Delete_2(x) Insert_3(x) Insert_1^{-1}(x) Delete_2^{-1}(x) Insert_3^{-1}(x) c_1 c_2 c_3$. Operation $Insert_1(x)$ conflicts with $Delete_2(x)$, $Delete_2(x)$ conflicts with $Insert_3(x)$, $Insert_3(x)$ conflicts with $Insert_1^{-1}(x)$, $Insert_1^{-1}(x)$ conflicts with $Delete_2^{-1}(x)$ and $Delete_2^{-1}(x)$ conflicts with $Insert_3^{-1}(x)$. The reducibility graph for $\tilde{S}_3$ contains a path $(Insert_1(x), Delete_2(x), Insert_3(x), Insert_1^{-1}, Delete_2^{-1}, Insert_3^{-1})$ together with some additional edges. Thus, there is a path between any forward operation and its corresponding backward operation. Consequently, $S_3$ is not reducible.

Example 5. Consider schedule $S_4$: $Delete_1(x) Delete_2(x) Delete_3(x) a_1 a_2 a_3$. Its expansion is $\tilde{S}_4 = Delete_1(x) Delete_2(x) Delete_3(x) Delete_1^{-1}(x) Delete_2^{-1}(x) Delete_3^{-1}(x) c_1 c_2 c_3$. Operation $Delete_1(x)$ conflicts with $Delete_2(x)$, $Delete_2(x)$ conflicts with $Delete_3(x)$, $Delete_3(x)$ conflicts with $Delete_1^{-1}(x)$. The reducibility graph for $\tilde{S}_4$ does not contain any path between $Delete_3(x)$ and $Delete_3^{-1}(x)$ because $Delete^{-1}(x)$ operation commutes with itself. Consequently, $Delete_3(x)$ and $Delete_3^{-1}(x)$ can be eliminated from the graph. The remaining two pairs ($Delete_2(x), Delete_2^{-1}(x)$) and ($Delete_1(x), Delete_1^{-1}(x)$) can be eliminated from the graph in a similar way. 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 2–4 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 et al. [12, 13]. Their definition in our
model can be restated as follows.

**Definition 7 (Moss et al. [12, 13]).** 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 \( a(..., T_i, ..., T_j, ...) \in S \).

Schedule \( \text{Insert}_1(x) \text{Insert}_2(x) a_1 \) is both revokable and reducible. Not every re-
ducible schedule is revokable. Schedule \( \text{Insert}_1(x) \text{Insert}_2(x) a_1 a_2 \) is not revokable,
however, it is reducible. Furthermore, not every revokable schedule is reducible. Sched-
ule \( \text{Insert}_1(x) \text{Insert}_2(x) \text{Insert}_2(y) \text{Insert}_1(y) c_1 c_2 \) 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 applica-
tions of the commutativity and undo rules.

**Proof.** 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 \( \tilde{S} \) (if that is not the case we can repeat the
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 \( <_\tilde{S} \)-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 can be 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 $o_j^{-1} <_S o_i^{-1}$ which contradicts the $<_S$-minimality of $o_i^{-1}$. Thus the lemma is proven. \hfill \Box

**Proof of Theorem 1 (conclusion).** By Lemma 2 all operations of aborted transactions in $S$ can be eliminated from $\hat{S}$ using the commutativity and undo rules. Thus $\hat{S}$ would contain after such eliminations only operations of transactions committed in $S$. Since we assumed that $S$ is serializable, $\hat{S}$ is also serializable and consequently, $S$ is reducible.

Schedule $S$: $\text{Insert}_1(x) \text{Test}_2(x) a_1 a_2$ is reducible (even prefix reducible), but not revokable. Thus the containment is proper. \hfill \Box

**3.2. Prefix reducible schedules**

In [1] 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 [1] 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 $S$: $\text{Insert}_1(x) \text{Insert}_2(x) c_2 c_1$ is not prefix reducible. Indeed, consider $\text{Insert}_1(x) \text{Insert}_2(x) c_2$, which is a prefix of $S$. Its expansion $\text{Insert}_1(x) \text{Insert}_2(x) c_2 \text{Insert}_1^{-1}(x) c_1$ cannot be reduced since neither operations $\text{Insert}_2(x)$ and $\text{Insert}_1^{-1}(x)$ nor $\text{Insert}_1(x)$ and $\text{Insert}_2(x)$ 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 $\text{Insert}_1(x) \text{Delete}_2(x) a_1 a_2$ 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 [1]. 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 a SOT schedule.

**Theorem 2.** Every prefix reducible schedule is also serializable with ordered termination.
Proof. Assume to the contrary, that there exists $S \in \text{PRED} - \text{SOT}$. Let us consider the following cases:

1. Consider $S \in \text{PRED}$, 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 $c_i \not< S c_j$ or $c_j < S c_i$. In the first case from the definition of expanded schedules we derive that $o_i < S o_j < S o_i^{-1}$ for all $\tilde{S}$. Since $o_i$ and $o_j$ as well as $o_j$ and $o_i^{-1}$ are conflicting and $o_j$ belongs to a committed transaction, $\tilde{S}$ cannot be reducible, which contradicts the initial assumption that $S \in \text{PRED}$. In the second case we consider a prefix of $S$ containing $c_i$ but not $c_j$. Applying arguments similar to the first case, we again derive a contradiction with $S \in \text{PRED}$.

2. Consider $S \in \text{PRED}$, 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 $o_i^{-1}$ and $o_j^{-1}$ are conflicting and $T_i$ aborts in $S$, but either $a_i < S a_j$ or $T_j$ does not abort in $S$. Consider first the case where $a_i < S a_j$. From the definition of expanded schedules it follows that $o_i < S o_j < S o_i^{-1} < S o_j^{-1}$ holds in any $\tilde{S}$. Since $o_i$ and $o_j$ are conflicting, $o_j$ and $o_i^{-1}$ are conflicting and $o_j^{-1}$ and $o_i^{-1}$ are conflicting, $\tilde{S}$ cannot be reducible which contradicts the initial assumption that $S \in \text{PRED}$. 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. $\square$

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

Example 6. Let the database consist of positive integers with the following operations defined on them:

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

As in the previous examples we assume that operations invoked on different arguments commute and limit the commutativity considerations to only operations invoked on
identical arguments. Fig. 3 indicates (by placing +) which operation pairs commute. We illustrate several cases below:

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

- Operation \( \text{Incr}(x) \) does not commute with \( \text{Incr}^{-1}(x, y) \). Assuming that \( x > 0 \), we can verify that operation \( \text{Retrieve}(x) \) returns 0 in the sequence \( \text{Incr}(x) \text{Reset}(x) \text{Incr}^{-1}(x, y) \text{Incr}(x) \text{Retrieve}(x) \) and returns 1 in \( \text{Incr}(x) \text{Reset}(x) \text{Incr}(x) \text{Incr}^{-1}(x, y) \text{Retrieve}(x) \).

- Operation \( \text{Incr}(x) \) does not commute with \( \text{Decr}(x) \). Assuming that \( x = 1 \) initially, we can verify that operation \( \text{Incr}(x) \) returns 1 in \( \text{Incr}(x) \text{Decr}(x) \) and returns 0 in sequence \( \text{Decr}(x) \text{Incr}(x) \).

- Operation \( \text{Decr}(x) \) commutes with \( \text{Incr}^{-1}(x, y) \). Consider two arbitrary sequences \( \alpha, \beta \). Due to well-formedness, \( \alpha \) must contain \( \text{Incr}(x) \) in the sequences \( \alpha \text{Decr}(x) \text{Incr}^{-1}(x, y) \text{Decr}(x) \text{Decr}(x) \text{Incr}^{-1}(x, y) \text{Decr}(x) \text{Decr}(x) \). In both sequences \( \text{Decr} \) decrements \( x \) and returns 1. Assume that the value of \( x \) is \( k \) after executing \( \alpha \). If the return value of \( \text{Incr}(x) \) is 1, then \( \text{Incr}^{-1}(x, y) \) always decrements \( x \) and returns 0. Also the value of \( x \) is \( k - 2 \) at the end of both sequences and thus no \( \beta \) can distinguish the two sequences. The case when the return value of \( \text{Incr}(x) \) is 0 can be established in a similar way.

Consider now schedule \( S_3 = \text{Incr}(x) \text{Decr}(x) \text{Incr}(x) \text{Decr}(x) \text{Incr}(x) a_1 c_2 c_3 \). Its expansion is \( S_3' = \text{Incr}(x) \text{Decr}(x) \text{Incr}(x) \text{Incr}^{-1}(x, y) c_2 c_3 \). Operation \( \text{Incr}(x) \) conflicts with \( \text{Decr}(x) \), \( \text{Decr}(x) \) conflicts with \( \text{Incr}(x) \) and \( \text{Incr}(x) \) conflicts with \( \text{Incr}^{-1}(x, y) \). The reducibility graph for \( S_3 \) contains a path (\( \text{Incr}(x), \text{Decr}(x), \text{Incr}(x), \text{Incr}^{-1}(x, y) \)) and consequently, \( S_3 \) is not reducible. However, schedule \( S_3 \) satisfies the SOT

<table>
<thead>
<tr>
<th>Operation</th>
<th>I</th>
<th>R</th>
<th>Ret</th>
<th>I(^{-1})</th>
<th>R(^{-1})</th>
<th>Ret(^{-1})</th>
<th>D</th>
<th>D(^{-1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Incr} (I) )</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>( \text{Reset} (R) )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>( \text{Retrieve} (Ret) )</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>( \text{Incr}^{-1} (I^{-1}) )</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>( \text{Reset}^{-1} (R^{-1}) )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>( \text{Retrieve}^{-1} (Ret^{-1}) )</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>( \text{Decr} (D) )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>( \text{Decr}^{-1} (D^{-1}) )</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Fig. 3. Commutativity relation.
conditions, since \( \text{Incr}_1^{-1}(x, y) \) does not conflict with \( \text{Decr}_2(x) \) and \( \text{Incr}_1(x) \) does not conflict with \( \text{Incr}_3(x) \).

Thus, the requirement for a schedule to be \( \text{SOT} \) is not sufficient to ensure prefix reducibility. We know of only one way to check whether a schedule is \( \text{PRED} \). 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 could restrict the class of prefix reducible schedules or we could impose some restrictions on a commutativity relation. In the next section we study the first approach and in Section 3.4 we study 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 using 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 the \( \text{PRED} \) constructive characterization.

To restrict the \( \text{PRED} \) class to a class that can be effectively handled by a scheduler and/or a recovery manager, let us consider in more detail what happens when a backward operation is scheduled. The purpose of the backward operation as we stated earlier is to undo all visible effects of the corresponding forward operation. Consider the situation when 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 the 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 \( S_1 : \text{Incr}(x) \text{Decr}(x) c_2 a_1 \). Since \( \text{Decr}(x) \) conflicts with \( \text{Incr}(x) \), but \( \text{Incr}^{-1}(x, y) \) commutes with \( \text{Decr}(x) \), the schedule is backward safe, but not forward safe. On the other hand, schedule \( S_2 : \text{Incr}(z) \text{Decr}(z) c_2 a_1 \) is forward safe, but not backward safe, since \( \text{Incr}(z) \) commutes with itself, but not with \( \text{Incr}^{-1}(z, y) \). Schedule \( S_3 : \text{Incr}(x) \text{Decr}(x) c_2 a_3 \text{Incr}^3(y) \text{Decr}^3(y) c_4 a_3 \) is prefix 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. Assume, for example, a case in which transaction \( T_1 \) issues several select statements and transaction \( T_2 \) subsequently performs some conflicting update statements. If transaction \( T_1 \) aborts, then \( T_2 \) still can commit and guarantee 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 \( T_2 \) that 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 (e.g. both \( T_1 \) and \( T_2 \) may issue a single \( \text{Incr}(x) \) operation from Example 6). If transaction \( T_1 \) aborts, then transaction \( T_2 \) still can commit and guarantee 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 \( S : \text{Incr}(x) \text{Decr}(x) \text{Incr}(y) \text{Decr}(y) c_1 c_2 \) is backward safe, but not serializable. However, every forward safe schedule is serializable since it is a subclass of *commit ordered* (CO) schedules, which is, in turn, a subclass of serializable schedules [3, 17]. Every *rigorous* (RG) schedule [5] is also forward safe. On the other hand, schedule \( S : \text{Incr}(x) \text{Decr}(x) c_1 c_2 \) is forward safe, but not rigorous.

To compare the class of forward and backward safe schedules with *revokable schedules* [12, 13], we need to make the 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 \( o_i \prec o_j, a_i \) does not precede \( o_j \) in \( S \) and \( o_j^{-1} \) is in conflict with \( o_i \) the following is true:

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

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 $S: Insert_1(x) \ Test_2(x) \ a_1 \ a_2$. 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 [15, 16] 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 $S$. The class of strict schedules 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 $S: Insert_1(x) \ Delete_2(x) \ c_1 \ c_2$. 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 noncommitted in $S$ can be completely eliminated from some $\hat{S}$ 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 $<_S$-minimal and there are $k$ operations in between $o_i$ and $o_i^{-1}$. We show that this pair can be eliminated from some $\hat{S}$.

Let $o_j$ be an immediate $<_S$-successor of $o_i$ in $\hat{S}$. 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 $T_i$ cannot be committed in $S$ and $o_j^{-1} = \lambda$ (for if $o_j^{-1} \neq \lambda$ then $o_j^{-1} <_S o_i^{-1}$ in any $\hat{S}$ which contradicts the assumption of $o_i^{-1}$ being $<_S$-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. $\square$

**Proof of Theorem 3.** Let $S_1$ be an arbitrary prefix of $S$. We show that $S_1$ is reducible. Since forward safeness is a prefix-closed property, all operations of transactions noncommitted in $S_1$ can be eliminated from some $\hat{S}_1$ by Lemma 3. Since $S_1$ is forward safe, it is also commit ordered and thus it is also serializable [5, 17]. The only
remaining operations in \( \hat{S}_1 \) are operations of transactions committed in \( S_1 \). Thus, \( \hat{S}_1 \) is serializable and consequently \( S_1 \) is reducible. Hence, \( S \) is prefix reducible. \( \square \)

**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 noncommitted in \( S \) can be completely eliminated from some \( \hat{S} \) 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 a pair of operations \( o_i \) and \( o_i^{-1} \) where \( o_i^{-1} \) is \( <_S \)-minimal and there are \( k \) operations in between \( o_i \) and \( o_i^{-1} \). We show that this pair can be eliminated from some \( \hat{S} \).

Let \( o_j \) be an immediate \( <_S \)-predecessor of \( o_i^{-1} \) in \( \hat{S} \). 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 \( o_j^{-1} <_S o_i^{-1} \) in any \( \hat{S} \) which contradicts the assumption of \( o_i^{-1} \) being \( <_S \)-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. \( \square \)

**Proof of Theorem 4.** The rest of the proof of Theorem 4 is the same as in the proof of Theorem 3. \( \square \)

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 Fig. 4.

### 3.4. Normal and perfect commutativity relations

In this subsection we investigate the restrictions on the commutativity relation that lead to a simple characterization of prefix reducible schedules. The major problem we deal with in this paper is a consequence of the nonsymmetric commutativity behaviour of a forward operation and its related backward operation. In this section we study the regularity requirements on the commutativity behaviour of forward and backward operations, which are necessary for a simple characterization of prefix reducible schedules.

**Definition 11.** We call a commutativity relation normal if for every two operations \( p \) and \( q \) the following condition holds: if \( p \) does not commute with \( q \) and \( p^{-1} \) is not \( \lambda \), then \( p^{-1} \) does not commute with \( q \). If, in addition, \( q^{-1} \) is not \( \lambda \), then also \( p^{-1} \) does not commute with \( q^{-1} \).
Consider the set of operations from Example 6 after exclusion of operations \textit{Decr} and \textit{Decr}^{-1}. Then the restricted commutativity relation from Example 6 is normal. 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 every prefix reducible schedule is also an \textit{SOT} one. Thus it remains to show only that each \textit{SOT} schedule is also a prefix reducible. First, we prove an auxiliary lemma.

**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 \(<_S\)-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 $<_S$-successor of $o_i$ in $\tilde{S}$. If $o_j$ commutes with $o_i$, 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 $o_j^{-1}$ 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 $(o_i^{-1}, o_j) \in \text{CON}$ and $(o_i^{-1}, o_j^{-1}) \in \text{CON}$. Since $S$ is SOT, $T_j$ cannot commit in $S$ and consequently, for all expanded schedules $\tilde{S}$, $o_j <_S o_j^{-1} <_S o_i^{-1}$, which contradicts the $<_S$-minimality of $o_i^{-1}$. Thus the lemma is proven. 

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

**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 [5] and strict [16] schedules and a normality of the commutativity relation. Indeed, whenever a commutativity relation is normal then from $(o_i, o_j) \in \text{CON}$ it follows also that $(o_i^{-1}, o_j) \in \text{CON}$ and $(o_i^{-1}, o_j^{-1}) \in \text{CON}$. 

As we have seen, the commutativity relation of Example 2 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 the 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 Fig. 4 by assuming that $\text{PRED} = \text{SOT}$ and the class of rigorous schedules is a subset of strict serializable schedules. An obvious consequence of normal commutativity relations is that all the protocols defined in [1] 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 schedules 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 $S = \text{Incr}_1(y) \text{Incr}_2(y) \text{Incr}_{1}^{-1}(y, z_1) \text{Incr}_{2}^{-1}(y, z_2)$. It is easy to see that the commutativity relation in Example 6 (after exclusion of operation $\text{Decr}$ and its backward operation) is normal. $S$ is SOT, since $\text{Incr}_1(y)$ commutes with $\text{Incr}_2(y)$. On the other hand, the schedule is not backward safe, since $\text{Incr}_2(y)$ conflicts with $\text{Incr}_1^{-1}(y, z_1)$ but $\text{Incr}_1^{-1}(y, z_1)$ precedes $\text{Incr}_2^{-1}(y, z_2)$ in the schedule.

A special case of normal commutativity is perfect commutativity. In contrast to normality, perfectness requires that if some combination of backward and forward
operations does not commute, then all combinations of backward and forward operations do not commute. Formally,

**Definition 12.** We say that commutativity relation is perfect if for every two operations $p$ and $q$ either $p^\alpha$ commutes with $q^\beta$ for all possible combinations of $\alpha, \beta \in \{-1, 1\}$ or $p^\gamma$ does not commute with $q^\delta$ for all possible combinations of $\gamma, \delta \in \{-1, 1\}$ with the exception of $\lambda$ as a backward operation commuting with everything.

Perfectness is guaranteed in the read/write model because the undo of a write operation is another write. As we have already shown, in the more general model this property is not satisfied a priori. For instance, the commutativity relation in Example 2 is not normal and not perfect. The operation $\text{Insert}(x)$ does not commute with itself, but its backward operation $\text{Insert}^{-1}(x)$ does commute with itself. We have seen that the restricted commutativity relation in Example 6 is normal, but it is not perfect because operation $\text{Incr}(x)$ commutes with itself, but operations $\text{Incr}(x)$ and $\text{Incr}^{-1}(x, y)$ do not commute.

The main appeal of models with a perfect commutativity relation lies in their "isomorphism" to the read/write model. With perfectness, the classes of $\text{SOT}$ and $\text{BSF\&SR}$ 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 Definitions 8 and 9 and perfectness of the commutativity relation. Indeed, whenever a commutativity relation is perfect and $o_i$ and $o_j$ conflict it follows that $o_i^{-1}$ and $o_j$ as well as $o_i$ and $o_j^{-1}$ also conflict. □

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 $\text{SOT}$ and $\text{PRED}$). 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 Fig. 5.

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 $\alpha$, then $p$ and $q^{-1}$ or $p^{-1}$ and $q$ also state-dependently commute with respect to the sequences $\alpha q$ or $\alpha p$, respectively. Furthermore, $p^{-1}$ and $q^{-1}$ state-dependently commute with respect to the sequence $\alpha p q$.

Proof. We show that if $p$ state-dependently commutes with $q$ with respect to the sequence $\alpha_1$, 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 $\alpha_1$, 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 $\beta$.

Consider $\alpha_2 = \alpha_1 p$. Since $p p^{-1}$ is effect-free, the return values of $q$ and $\beta$ 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 $p p^{-1}$ is effect-free, we obtain that the return value sequences of $q$ and $\beta$ 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 0, we derive that the return values of $q$, $p^{-1}$ and $\beta$ are the same in sequences $\alpha_1 p q p^{-1} \beta$ and $\alpha_1 p p^{-1} q \beta$ and our claim holds. The other two cases use arguments similar to above and thus omitted. \qed
Note that the sequences of operations $\alpha q$, $\alpha p$ and $\alpha p q$ 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.

Any scheduler can execute commutative operations concurrently. In order to design 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 [2] 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.3 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 the operations conflict even if they do not according to our definition (while sacrificing the inter-operation concurrency). For the rest of the paper we assume that such a conflict detection method on operation invocations is imported into the transaction manager.

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 of not only $(o_i, o_j)$ conflicts (for serializability), but also $(o_i^{-1}, o_j)$ 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 in detail the forward safe pessimistic graph testing protocol and then show how other protocols generating forward safe schedules can be constructed. The forward safe pessimistic graph testing protocol uses serialization

---

3 In Examples 2 and 6 it was sufficient to determine whether the operation parameters are equal or not.
1. Operation $o_i$ (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 $o_i$ is rejected and $a_j$ is submitted instead. Otherwise, submit $o_i$ 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, $T$. Submit $a(T)$ for execution. After $a(T)$ is executed, all transactions from $T$ are removed from the serialization graph and from the commit queue.

To illustrate the forward safe pessimistic graph testing protocol consider that the scheduler receives the following sequence of operations: $S: \text{Insert}_1(x) \text{ Insert}_2(x) \text{ Insert}_3(y) \text{ Insert}_2(y) \text{ c}_2 \text{ c}_3 \text{ a}_1$. After receiving the prefix $\text{Insert}_1(x) \text{ Insert}_2(x) \text{ Insert}_3(y) \text{ Insert}_2(y)$ 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 $(T_3, T_2)$ are removed from the serialization graph. Nevertheless, $T_2$ still cannot commit, since its predecessor $T_1$ has 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 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 point 3 of the protocol. The removal of nodes corresponding to committed transactions in point 2 of the protocol does not lead to nonserializable schedules, since only the sinks of the serialization graph are removed [14]. The nodes removed from the graph in 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 the serialization graph in order 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 as well, 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 point 2 rather than in point 1 of the protocol. Similarly, a nonblocking version of the protocol can be obtained by the following modification of 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 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, then the serialization graph contains edge $(z, q)$. Similarly, whenever transaction $T_j$ waits for transaction $T_i$ to commit, the serialization graph contains edge $(z, r)$. Therefore, a cycle in the wait-for-graph is also a cycle in the serialization graph.

As the reader has probably noticed, 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 avoiding cascading aborts.

**Theorem 7.** The class of rigorous schedules is the $\subseteq$-maximal subclass of the class of forward safe schedules avoiding 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 rigorousness 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 $T_i$ while still retaining a relatively high degree of concurrency.

One possible way to limit the number of cascading aborts is to bound the length of every path in the serialization graph by a 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 in point 1: whenever a path longer than $n$ should appear in the serialization graph as a consequence of scheduling operation $o_j$ in 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 recognized by the modified protocol to the class of rigorous schedules. Whenever $n$ grows, the degree of concurrency grows and for large $n$ the class of schedules recognized by the modified protocol approximates the class of forward safe schedules. However, also the number of transactions that may need to be aborted as a result of abort of some transaction (and thus also the recovery costs) grow with $n$. 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 backward safeness by itself does not guarantee serializability, the backward safe protocols must therefore guarantee not only backward safeness, but also serializability. Serializability can be guaranteed by maintaining an acyclic serialization graph. In addition to that, the protocol must maintain also 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 $S$, $o_i < o_j$ such that $T_i$ does not abort before $o_j$ and $o_j^{-1}$ is in conflict with $o_j$, we add an oriented edge from $T_i$ to $T_j$. Clearly, whenever the graph contains edge $\langle T_i, T_j \rangle$ 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 contained a cycle $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$ then by backward safeness we derive that $c_1 < c_2 < \cdots < c_n < c_1$. A contradiction!).

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

To illustrate the protocol from Fig. 7 assume that the scheduler receives the following sequence of operations: $S : \text{Incr}_1(x) \text{Incr}_2(x) \text{Incr}_3(y) \text{Incr}_2(y) c_1 c_2 c_3$. Also assume that timestamps of the transactions are in the following order: $ts(T_1) < ts(T_2) < ts(T_3)$. If we assume that no transaction releases its locks until it submits its commit or
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. $cj$ 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 smaller timestamp than its sink. If not, reject $c_j$ and submit $a_j$ instead. Else, test if there are any edges coming to $T_j$. Is yes, reject $c_j$ and submit $a_j$ instead. Otherwise submit $c_j$ for execution.

3. $aj$ 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, $T$. Submit $a(T)$ for execution. After $a(T)$ is executed, all transactions from $T$ are removed from the serialization graph and from the commit queue.

Fig. 7. Backward safe hybrid protocol.

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 $c_1$ is received, it gets submitted for execution since $ts(T_1) < ts(T_2)$ and there are no edges coming to node $T_1$ in the termination graph. The edge $(T_1, T_2)$ together with the node $T_1$ are removed from the termination graph after $c_1$ is executed. When $c_2$ is received, it is rejected and $a_2$ is submitted instead, since there is an edge $(T_3, T_2)$ coming to node $T_2$ in the termination graph. $a_2$ is not expanded with abort of any additional transactions since $T_2$ is has no followers in the termination graph. Finally, $c_3$ is immediately scheduled for execution since the termination graph contains at that time only a single node $T_3$.

Similarly to the forward safe pessimistic graph testing protocol from Fig. 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 avoiding 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 $\text{Insert}(x)$ from Example 2. 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 $\text{Insert}(\ )$ and $\text{Delete}(\ )$ 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 $\text{Insert}(\ )$ or $\text{Delete}(\ )$ 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 $\text{Insert}(x)$ and $\text{Delete}(x)$. However, it does not commute with $\text{Test}(x)$. Thus the only cascading aborts are of those transactions that invoked $\text{Test}(x)$ after the forward operation has been issued.

5. Conclusion

In this paper we discussed an unified correctness criterion namely the class of prefix reducible schedules that guarantees both transaction serializability and atomicity within the framework of the general model with semantically rich operations. We have demonstrated that the class of schedules serializable with ordered termination introduced in [1], does not characterize all prefix reducible schedules in this general case. We found that the complexity of an exact characterization of prefix reducible schedules in the model with semantically rich operations stems from an arbitrary commutativity behavior of the operations and their undo operations. We identified the conditions on the model when such a characterization is exact. We have shown that with normal commutativity relations the class schedules of serializable 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 a 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 a 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. Since these operations in general are not only read/write accesses on pages, this postulates a need for investigation of single level models with an arbitrary set of operations. To prove correctness of execution of such transactions in a failure prone distributed database environment, a unified theory must be developed. Comparing to the classical theory, the unified theory should treat concurrency control and recovery uniformly.

The model of multilevel transactions [4, 23, 11] become recently widely accepted in modeling operations on abstract data types which can be correctly executed without requiring serializability of its read/write operations. It has been shown in [23] 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. However, concurrency control and recovery are still treated in the model of multilevel transactions as orthogonal problems. Therefore the multilevel transactions suffer from the same problem as in the classical flat model, i.e. the correctness criteria for concurrency control and recovery are incomparable and only the most restrictive criterion accounts for both. The unified theory for the read/write model [19] and the unified theory for semantically rich operations of a flat model described in this paper gives the basis for the extending the unified theory for the model of multilevel transactions [10].

Acknowledgements

The authors would like to thank Gerhard Weikum for numerous inspiring discussions, and many valuable comments and observations as well as an anonymous referee whose remarks led to a significant improvement of the presentation.

References