A Formal Analysis of the Deferred Update Technique*

Rodrigo Schmidt[1,2] and Fernando Pedone[2]

[1] École Polytechnique Fédérale de Lausanne, Switzerland
[2] University of Lugano, Switzerland

Abstract. The deferred update technique is a widely used approach for building replicated database systems. Its fame stems from the fact that read-only transactions can execute locally to any single database replica, providing good performance for workloads where transactions are mostly of this type. In this paper, we analyze the deferred update technique and show a number of characteristics and limitations common to any replication protocol based on it. Previous works on this replication method usually start from a protocol and then argue separately that it is based on the deferred update technique and satisfies serializability. Differently, ours starts from the abstract definition of a serializable database and gradually changes it into an abstract deferred update protocol. In doing that, we can formally characterize the deferred update technique and rigorously prove its properties. Moreover, our specification can be extended to create new protocols or used to prove existing ones correct.

1 Introduction

In the deferred update technique, a number of database replicas are used to implement a single serializable database interface. Its main idea consists in executing all operations of a transaction initially on a single replica. Transactions that do not change the database state can commit locally to the replica they executed, but other transactions must be globally certified and, if committed, have their update operations (those that change the database state) submitted to all database replicas. This technique is adopted by a number of database replication protocols in different contexts (e.g., [1,2,3,4,5]) for its good performance in general scenarios. The class of deferred update protocols is very heterogeneous, including algorithms that can optimistically apply updates of uncertified transactions [2], certify transactions locally to the database that executed them [1], execute all concurrent update transactions at the same database [3], reorder transactions during certification [4], and even cope with partial database replication [5]. However, all of them share the same basic structure, giving them some common characteristics and constraints.

Despite its wide use, we are not aware of any work that explored the inherent limitations and characteristics of deferred update database replication. Ours seems to be the first attempt in this direction. We specify a general abstract deferred update algorithm

* The work presented in this paper has been partially funded by the SNSF, Switzerland (project #200021-170824).

that embraces all the protocols we know of. This general specification allows us to isolate the properties of the *termination protocol* necessary to certify update transactions and propagate them to all database replicas. We show, for example, that the termination protocol must totally order globally committed transactions, a rather counter-intuitive result given that serializability itself allows transactions that operate on different parts of the database state to execute in any order. For example, according to serializability, if two transactions t_1 and t_2 update data items x and y, respectively, and have no other operations, it is correct to execute either t_1 before t_2 or t_2 before t_1. Therefore, one could expect that some databases would be allowed to execute t_1 followed by t_2 while others would execute t_2 followed by t_1. In deferred update protocols, however, all databases are obliged to execute t_1 and t_2 in exactly the same order, limiting concurrency.

Moreover, previous works considered that databases should satisfy a property called order-preserving serializability, which says that the commit order corresponds to a correct serialization of the committed transactions. This bears the question: Is order-preserving serializability necessary for deferred update replication? We show that databases can satisfy a weaker property, namely *active order-preserving serializability*, which we introduce. According to this property, found in some multiversion databases, the internal database serialization must satisfy the commit order only for transactions that change the database state, without further constraining read-only transactions.

In our approach, we start with a general serializable database and refine it to our abstract deferred update algorithm. Similarly, one can use our specification to ease designing and proving specific protocols. One can simply prove a protocol correct by showing that it implements ours through a refinement mapping [6]. Our specifications use atomic actions to define safety properties [7,8]. Due to the space limitations, we present only high-level specifications. Complete TLA$^+$ [9] specifications, which have been model checked for a finite subset of the possible execution scenarios, are given in our technical report [10].

2 A General Serializable Database

The consistency criterion for transactional systems in general is *Serializability*, which is defined in terms of the equivalence between the system's actual execution and a serial execution of the submitted transactions [11]. Traditional definitions of equivalence between two executions of transactions referred to the internal scheduling performed by the algorithms and their ordering of conflicting operations. This approach has led to different notions of equivalence and, therefore, different subclasses of Serializability [12]. In a distributed scenario, however, defining equivalence in terms of the internal execution of the scheduler is not straightforward since there is usually no central scheduler responsible for ordering transaction operations. To compare a serial centralized schedule with a general distributed one (e.g., in a replicated database), one has to create mappings between the operations performed in both schedules and extend the notion of conflicting operations to deal with sets of operations, since a single operation in the serial centralized schedule may be mapped to a set of operations executed on different sites in the distributed one [11]. This approach is highly dependent on the implemented protocol and, as explained in [13], does not generalize well.

Differently, we specify a general serializable database system, which responds to requests according to some internal serial execution of the submitted transactions. A database protocol satisfies serializability if it implements the general serializable database specification, that is, if its interface changes could be generated by the general serializable database. This sort of analysis is very common in distributed systems for its compromise between abstraction and rigorousness [8,9,13].

In our specification of serializability, we first define all valid state transitions for normal interactions between the clients and the database, without caring about the values returned as responses to issued operations, but rather storing them internally as part of the transaction state. The database is free to abort a transaction at any time during the execution of its operations. However, a transaction t can only be committed if its commit request was issued and there exists a sequential execution order for all committed transactions and t that corresponds to the results these transactions provided. We say the transaction is *decided* if the database has aborted or committed it. Operations issued for decided transactions get the final decision as its result.

We assume each transaction has a unique identifier and let Tid be the set of all identifiers. We call Op the set of all possible transaction operations, which execute over a database state in set $DBState$ and generate a result in set $Result$ and a new database state. We abstract the correct execution of an operation by the predicate $CorrectOp(op, res, dbst, newdbst)$, which is true iff operation op, when executed over database state $dbst$, may generate res as the operation result and $newdbst$ as the new database state. In this way, our specification is completely independent of the allowed operations, coping with operations based on predicates and even nondeterministic operations. As a simple example, one could define a database with two integer variables x and y with read and write operations for each variable. In this case, $DBstate$ corresponds to all possible combinations of values for x and y, Op is the combination of an identifier for x or y with a read tag or an integer (in case of a write), and $Result$ is the set of integers. $CorrectOp(op, res, dbst, newdbst)$ is satisfied iff $newdbst$ and res correspond to the results for the read or write operation op applied to $dbst$.

Two special requests, $Commit$ and $Abort$, both not present in Op, are used to terminate a transaction, that is, to force a decision to be taken. Two special responses, $Committed$ and $Aborted$, not present in $Result$, are used to tell the database user if the transaction has been committed or aborted. We also define $Decided$ to equal the set $\{Committed, Aborted\}$, $Request$ to equal $Op \cup \{Commit, Abort\}$, and $Reply$ to equal $Result \cup Decided$.

During a transaction execution, operations are issued and responses are given until the client issues a $Commit$ or $Abort$ request or the transaction is aborted by the database for some internal reason. We represent the history of a transaction execution by a sequence of elements in $Op \times Result$, corresponding to the sequence of operations executed on the transaction's behalf and their respective results. We say that a transaction history h is *atomically correct* with respect to initial database state $initst$ and final database state $finalst$ iff it satisfies the recursive predicate defined below, where $THist$ is the set of all possible transaction histories and $Head$ and $Tail$ are the usual operators for sequences. Moreover, for notation simplicity, we identify the first and second elements of a tuple t in $Op \times Result$ by $t.op$ and $t.res$ respectively.

```
$CorrectAtomicHist(h \in THist, initst, finalst \in DBState) \triangleq$
  $\mathbf{if}\ h = \langle\rangle\ \mathbf{then}\ initst = finalst$
      $\mathbf{else}\ \exists ist \in DBState : CorrectOp(Head(h).op, Head(h).res, initst, ist) \wedge$
                                $CorrectAtomicHist(Tail(h), ist, finalst)$
```

Intuitively, a transaction history is atomically correct with respect to $initst$ and $finalst$ iff there are intermediate database states so that all operations in the history can be executed in their correct order and generate their correct results.

During the system's execution, many transactions are started and terminated (possibly concurrently). We represent the current history of all transactions by a data structure called *history vector* (set $THistVector$) that maps each transaction to its current history. We say that a sequence seq of transactions and a history vector $thist$ correspond to a correct serialization with respect to initial state $initst$ and final state $finalst$ iff the recursive predicate below is satisfied, where $Seq(S)$ represents the set of all finite sequences of elements in set S.

```
$CorrectSerialization(seq \in Seq(Tid), thist \in THistVector, initst, finalst \in DBState) \triangleq$
  $\mathbf{if}\ seq = \langle\rangle\ \mathbf{then}\ initst = finalst$
      $\mathbf{else}\ \exists ist \in DBState : CorrectAtomicHist(thist(Head(seq)), initst, ist) \wedge$
                                $CorrectSerialization(Tail(seq), thist, ist, finalst)$
```

Intuitively, this predicate is satisfied iff there are intermediate database states so that all transactions in the sequence can be atomically executed in their correct order generating the correct results for their operations. We can now easily define a predicate $IsSerializable(S, thist, initst)$ for a finite set of transaction id's S, history vector $thist$, and database state $initst$, satisfied iff there is a sequence seq containing exactly one copy of each element in S and a final database state $finalst$ such that $Correct\ Serialization(seq, thist, initst, finalst)$ is satisfied. Predicate $IsSerializable$ indicates when a set of transactions can be serialized in some order, according to their execution history, so that every operation returns its correct result when the execution is started in a given database state.

We abstract the interface of our specification by the primitives $DBRequest(t, req)$, which represents the reception of a request req on behalf of transaction t, and $DBResponse(t, rep)$, which represents the database response to the last request on behalf of t with reply rep. The only restriction we make with respect to the database interface is that an operation cannot be submitted on behalf of transaction t if the last operation submitted for t has not been replied yet, which releases us from the burden of using unique identifiers for operations in order to match them with their results. Notice that the system still allows a high degree of concurrency since operations from different transactions can be submitted concurrently.

Our specification is based on the following internal variables:

$thist$**:** A history vector, initially mapping each transaction to an empty history.

$tdec$**:** A mapping from each transaction to its current decision status: $Unknown$, $Committed$, or $Aborted$. Initially, it maps each transaction to $Unknown$.

q**:** A mapping from each transaction to its current request or $NoReq$ if no request is being executed on behalf of that transaction. Initially, it maps each transaction to $NoReq$.

```
$ReceiveReq(t \in Tid, req \in Request)$
    Enabled iff:
        – $DBRequest(t, req)$
        – $q[t] = NoReq$
    Effect:
        – $q[t] \leftarrow req$

$ReplyReq(t \in Tid, rep \in Reply)$
    Enabled iff:
        – $q[t] \in Request$
        – **if** $tdec[t] \in Decided$
                **then** $rep = tdec[t]$
                **else** $q[t] \in Op \land rep \in Result$
    Effect:
        – $DBResponse(t, rep)$
        – $q[t] \leftarrow NoReq$
        – **if** $tdec[t] \notin Decided$ **then**
            $thist[t] \leftarrow thist[t] \circ \langle q[t], rep \rangle$

$DoAbort(t \in Tid)$
    Enabled iff:
        – $tdec[t] \notin Decided$
    Effect:
        – $tdec[t] \leftarrow Aborted$

$DoCommit(t \in Tid)$
    Enabled iff:
        – $tdec[t] \notin Decided$
        – $q[t] = Commit$
        – $IsSerializable(committedSet \cup \{t\},$
                      $thist, InitialDBState)$
    Effect:
        – $tdec[t] \leftarrow Committed$
```

Fig. 1. The atomic actions allowed in our specification of a serializable database

Figure 1 presents the atomic actions of our specification. Action $ReceiveReq(t, req)$ is responsible for receiving a request on behalf of transaction t. Action $ReplyRep(t, rep)$ replies to a received request. It is enabled only if the transaction has been decided and the reply is the final decision or the transaction has not been decided but the current request is an operation (neither $Commit$ nor $Abort$) and the reply is in $Result$. This means that responses given after the transaction has been decided carry the final decision and requests to commit or abort a transaction are only replied after the transaction has been decided. Action $ReplyReq$ is responsible for updating the transaction history if the transaction has not been decided. It does that by appending the pair $\langle q[t], rep \rangle$ to $thist[t]$ (we use $\circ$ to represent the standard *append* operation for sequences). Action $DoAbort(t)$ simply aborts a transaction if it has not been decided yet. Action $DoCommit(t)$ commits t only if a t's commit request was issued and the set of all committed transactions (represented by $committedSet$) together with t is serializable with respect to the initial database state, denoted by the constant $InitialDBState$.

3 The Deferred Update Technique

3.1 Preliminaries

As mentioned before, deferred update algorithms initially execute transactions on a single replica. Transactions that do not change the database state (hereinafter called *passive*) may commit locally only, but *active* transactions (as opposed to passive ones) must be globally certified and, if committed, have their updates propagated to all replicas (i.e., operations that make them active). In order to correctly characterize the technique, we need to formalize the concepts of active and passive operations and transactions. An operation op is passive iff its execution never changes the database state, that is, iff the following condition is satisfied.

$$\forall st1, st2 \in DBState, rep \in Result : CorrectOp(op, rep, st1, st2) \Rightarrow st1 = st2 \quad (1)$$

An operation that is not passive is called active. Similarly, we define a transaction history h to be passive iff the condition below is satisfied.

$$\forall st1, st2 \in \mathit{DBState} : \mathit{CorrectAtomicHist}(h, st1, st2) \Rightarrow st1 = st2 \tag{2}$$

Notice that a transaction history composed of passive operations is obviously passive, but the converse is not true. A transaction that adds and subtracts 1 to a variable is passive even though its operations are active.

The deferred update technique requires some extra assumptions about the system. Operations, for example, cannot generate new database states nondeterministically for this could lead different replicas to inconsistent states. The following assumption makes sure that operations do not change the database state nondeterministically but still allows nondeterministic results to be provided to the database user.

Assumption 1 (State-deterministic operations). *For every operation op, and database states st and* $st1$*, if there is a result* $res1$ *such that* $\mathit{CorrectOp}(op, res1, st, st1)$*, then there is no result* $res2$ *and database state* $st2$ *such that* $st1 \neq st2 \wedge \mathit{CorrectOp}(op, res2, st, st2)$.

As for the database replicas, one may wrongly think that simply assuming that they are serializable is enough to ensure global serializability. However, two replicas might serialize their transactions (local and global) differently, making the distributed execution non-serializable. Previous works on deferred update protocols assumed the notion of *order-preserving serializability*, originally introduced by Beeri et al. in the context of nested transactions [14]. In our model, order-preserving serializability ensures that the transactions' commit order represents a correct execution sequence, a condition satisfied by two-phase locking, for example. We show that this assumption can be relaxed since deferred update protocols can work with the weaker notion of *active order-preserving serializability* we introduce. Active order-preserving serializability ensures that there is an execution sequence of the committed transactions that generates their correct outputs and respects the commit order of all *active* transactions only. This notion is weaker than strict order-preserving serializability in that passive transactions do not have to provide results based on the latest committed state. Some multiversion concurrency control mechanisms [11] are active order-preserving but not strict order-preserving. Specifications of order-preserving and active order-preserving serializability can be derived from our specification in Figure 1 by just adding a variable $\mathit{serialSeq}$, initially equal to the empty sequence, and changing the $\mathit{DoCommit}$ action. We show the required changes in Figure 2. The strict case (a) is simple and only requires that $\mathit{serialSeq} \circ t$ be a correct sequential execution of all committed transactions. The action automatically extends $\mathit{serialSeq}$ with t. The active case (b) is a little more complicated to explain and requires some extra notation. Let $\mathit{Perm}(S)$ be the set of all permutations of elements in finite set S (all the possible orderings of elements in S), and let $\mathit{ActiveExtension}(seq, t)$ be seq if $\mathit{thist}[t]$ is a passive history or $seq \circ t$ otherwise. The action is enabled only if there exists a sequence containing all committed transactions such that it represents a correct sequential execution and $\mathit{ActiveExtension}(seq, t)$ is

```
$DoCommit(t \in Tid)$
    Enabled iff:
        – $tdec[t] \notin Decided$
        – $q[t] = Commit$
        – $\exists st \in DBState$ :
              $CorrectSerialization(serialSeq \circ t, thist, InitialDBState, st)$
    Effect:
        – $tdec[t] \leftarrow Committed$
        – $serialSeq \leftarrow serialSeq \circ t$
```

(a)

```
$DoCommit(t \in Tid)$
    Enabled iff:
        – $tdec[t] \notin Decided$
        – $q[t] = Commit$
        – $\exists seq \in Perm(committedSet \cup \{t\}), st \in DBState$ :
              $CorrectSerialization(seq, thist, InitialDBState, st) \wedge$
              $ActiveExtension(serialSeq, t)$ is a subsequence of $seq$
    Effect:
        – $tdec[t] \leftarrow Committed$
        – $serialSeq \leftarrow ActiveExtension(serialSeq, t)$
```

(b)

Fig. 2. *DoCommit* action for (a) strict and (b) active order-preserving serializability

a subsequence of it.[1] In this action, *serialSeq* is extended with t only if t is an active transaction.

3.2 Abstract Algorithm

We now present the specification of our abstract deferred update algorithm. It generalizes the ideas of a handful of deferred update protocols and makes it easy to think about sufficient and necessary requirements for them to work correctly. Our specification assumes a set *Database* of active order-preserving serializable databases, and we use the notation $DB(d)!Primitive(\_)$ to represent the execution of interface primitive *Primitive* (either *DBRequest* or *DBResponse*) of database d. Since transactions must initially execute on a single replica only, we let $DBof(t)$ represent the database responsible for the initial execution of transaction t. One important remark is that these internal databases receive transactions whose id set is $Tid \times \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. This is done so because a single transaction in the system might have to be submitted multiple times to a database replica in order to ensure that it commits locally. Recall that our definition of active order-preserving serializability does not force transactions to commit. Therefore, transactions that have been committed by the algorithm and submitted to the database replicas are not guaranteed to commit unless further assumptions are made. The only way around this is to submit these transactions multiple times (with different versions) until they commit. Besides the set of databases, we assume a concurrent termination protocol, fully explained in the next

[1] sequence *subseq* is a subsequence of sequence *seq* iff it can be obtained by removing zero or more elements of *seq*.

section, responsible for committing active transactions and propagating their active operations to all databases.

The algorithm we present in the following orchestrates the interactions between the global database interface and the individual internal databases. It is mainly based on the following internal variables:

$thist$, q**:** Essentially the same variables as in the specification of a serializable database.

$dreq$**:** A mapping from each transaction t to the operation that is currently being submitted for execution on $DBof(t)$, or $NoReq$ if no operation is being submitted. This variable is used to implement the asynchronous communication that tells $DBof(t)$ to execute an operation of t. Initially all transactions are mapped to $NoReq$.

$dreply$**:** Similar to $dreq$, but mapping each transaction t to the last response given by $DBof(t)$.

$dcnt$**:** A mapping from each database d and transaction t to an integer representing the number of operations that executed on d for t. It counts the number of operations $DBof(t)$ has executed for t during t's initial execution and, if t is active, the number of active operations the other databases (or $DBof(t)$ if it does not manage to commit t directly after it is globally committed) have executed for t after it is globally committed. It is initially 0 for all databases and transactions.

$pdec$**:** A mapping like $tdec$ in the specification of a serializable database, used to tell whether the transaction was decided without being proposed for global termination either because it was prematurely aborted during its initial execution or because it was a passive transaction that committed on its execution database.

$vers$**:** A mapping from each database d and transaction t to an integer representing the current version of t being submitted to d. It is initially 0 for all databases and transactions.

$dcom$**:** A mapping from each database d and transaction t to a boolean telling whether t has been committed on d. It is initially false for all databases and transactions.

When a $Commit$ request is issued for a transaction whose history has been active, a decision must be taken on whether to commit or abort this transaction with respect to active transactions executed on other databases. In our specification, this is done separately by a termination protocol. The reason why we isolated this part of the specification is twofold. First, the nature of the rest of the algorithm is essentially local to the database that is executing a given transaction and it seems interesting to separate it from the part of the specification responsible for synchronizing active transactions executed on different databases. Second, the properties of the termination protocol, when isolated, can be related to properties of other agreement problems in distributed computing, which helps understand and solve it. The interface variables of the termination protocol used in our general specification are the following:

$proposed$**:** This is an input variable that keeps the set of all proposed transactions. It is initially empty.

$gdec$**:** An output variable that keeps a mapping like $pdec$ above, but managed by the termination protocol only. It tells whether a proposed transaction has already been decided or not.

learnedSeq: Another output variable mapping each database d to a sequence of globally committed active transactions. This sequence tells database d the order in which these active transactions must be committed to make the whole execution serializable. Initially, it maps each database to the empty sequence.

Our specification implements a serializable database, which can be proved by a refinement mapping from its internal variables to those of a general serializable database. Actually, the only internal variable of our specification of a serializable database not directly implemented in our abstract algorithm is $tdec$, given by joining the values of $pdec$ and $gdec$ in the following way:

$$tdec[t] \triangleq \textbf{if } t \in proposed \textbf{ then } gdec[t] \textbf{ else } pdec[t] \tag{3}$$

For simplicity, we use this definition of $tdec$ in some parts of our specification. Another extra definition used in our algorithm is the $ActHist(t)$ operator that returns the subsequence of $thist[t]$ containing all its active operations. The atomic actions of our abstract algorithm, without the internal actions of the individual databases and the termination protocol, are shown in Figure 3.

Action $ReceiveReq$ treats the receipt of a transaction request. If the transaction responsible for the operation has been decided (either for $pdec$ or $gdec$ according to the definition of $tdec$ given above), then it only changes $q[t]$. Otherwise, it either proposes t for the termination protocol or sends the request to $DBof(t)$ through variable $dreq[t]$. Our complete specification allows passive transactions to be submitted for the termination protocol too and this is why we wrote “is active” between quotation marks. We allow that because sometimes it might not be possible to identify all passive transactions. Therefore, our specification also embraces algorithms that identify only a subset of the passive transactions as passive and conservatively propose the others for global termination.

Action $ReplyReq$ replies a transaction request. It is very similar to the original $ReplyReq$ action of our serializable database specification. The small differences only make sure that the value replied for a normal operation comes from $DBof(t)$ and, in this case, $dreq[t]$ is set back to $NoReq$ to wait for the next operation. Actions $Premature$ $Abort$ and $PassiveCommit$ abort or commit a transaction that has not been proposed for global termination. It can only be committed if a commit request was correctly replied by $DBof(t)$, which can only happen if t has a passive history.

Action $DBReq$ submits a request to a database. There are three conditions that enable this action. The first one represents a normal request during the transaction's initial execution or a commit request for a passive transaction. The second one represents an operation request for an active transaction that has been proposed to the termination protocol. Notice that operations of proposed transactions can be optimistically submitted to the database before they commit or appear in some $learnedSeq$. Some algorithms do that to save processing time after the transaction is committed, reducing the latency for propagating transactions to the replicas. The third condition that enables this action represents a commit request for a transaction that has been committed by the termination protocol. For that to happen, the transaction must be present in $learnedSeq[d]$ and all transactions previous to it in the sequence must have been committed on that database. Moreover, all active operations of that transaction must have been applied to

```
$ReceiveReq(t \in Tid, req \in Request)$
    Enabled iff:
      – $DBRequest(t, req)$
      – $q[t] = NoReq$
    Effect:
      – $q[t] \leftarrow req$
      – **if** $tdec[t] \notin Decided$ **then**
          **if** $req = Commit \wedge thist[t]$ “is active”
              **then** $proposed \leftarrow proposed \cup \{t\}$
              **else** $dreq[t] \leftarrow req$

$ReplyReq(t \in Tid, rep \in Reply)$
    Enabled iff:
      – $q[t] \in Request$
      – **if** $tdec[t] \in Decided$ **then**
          $rep = tdec[t]$
        **else**
          $q[t] \in Op \wedge rep \in Result \wedge$
          $dcnt[DBof(t)][t] > Len(thist[t]) \wedge$
          $rep = dreply[t]$
    Effect:
      – $DBResponse(t, rep)$
      – $q[t] \leftarrow NoReq$
      – **if** $tdec[t] \notin Decided$ **then**
          • $thist[t] \leftarrow thist[t] \circ \langle q[t], rep \rangle$
          • $dreq[t] \leftarrow NoReq$

$PrematureAbort(t \in Tid)$
    Enabled iff:
      – $t \notin proposed$
      – $pdec[t] \notin Decided$
    Effect:
      – $pdec[t] \leftarrow Aborted$

$PassiveCommit(t \in Tid)$
    Enabled iff:
      – $t \notin proposed$
      – $pdec[t] \notin Decided$
      – $dreply[t] = Committed$
    Effect:
      – $pdec[t] \leftarrow Committed$
```

```
$DBReq(d \in Database, t \in Tid, req \in Request)$
    Enabled iff **any** of the conditions below hold.
    Condition 1: (external operation request)
      – $d = DBof(t)$
      – $dreq[t] = req$
      – $dcnt[d][t] = Len(thist[t])$
    Condition 2: (operation after termination)
      – $t \in proposed$
      – $dcnt[d][t] < Len(ActHist(t))$
      – $req = ActHist(t)[dcnt[d][t] + 1].op$
    Condition 3: (commit after termination)
      – $req = Commit$
      – $\exists i \in 1..Len(learnedSeq[d])$ :
          $learnedSeq[d][i] = t \wedge$
          $\forall j \in 1..i$ : $dcom[d][learnedSeq[d][j]]$
      – **either** $d = DBof(t) \wedge vers[d][t] = 0$
        **or** $dcnt[d][t] = Len(ActHist(t))$

    Effect:
      – $DB(d)!DBRequest(\langle t, vers[d][t] \rangle, req)$

$DBRep(d \in Database, t \in Tid, rep \in Reply)$
    Enabled iff:
      – $DB(d)!DBResponse(\langle t, vers[d][t] \rangle, rep)$
    Effect:
      – **if** $d = DBof(t)$ **then** $dreply[t] \leftarrow rep$
      – **if** $rep = Aborted \wedge t \in proposed$ **then**
          • $vers[d][t] \leftarrow vers[d][t] + 1$
          • $dcnt[d][t] \leftarrow 0$
        **else**
          • $dcnt[d][t] \leftarrow dcnt[d][t] + 1$
          • $dcom[d][t] \leftarrow rep = Committed$
```

Fig. 3. The atomic actions allowed in our specification of a serializable database

the database already, which is true if the database is the one originally responsible for the transaction and it has not changed the transaction version or the operations counter $dcnt[d][t]$ equals the number of active operations in the transaction history. Recall that, by the definition of a serializable database, a request can only be submitted if there is no pending request for the same transaction. This is actually an implicit pre-condition for $DBReq$ given by the specification of a serializable database.

Action $DBRep$ treats the receipt of a response coming from a database. If the database is the one responsible for initially executing the transaction, it sets $drepy[t]$ to the value returned. If the transaction is aborted but it has been proposed for global termination, it changes the version of that transaction on that database and sets the operation counter to zero so that the transaction's operations can be resubmitted for its new version; otherwise, it just increments the operation counter and sets $dcom$ accordingly.

3.3 Termination Protocol

The termination protocol gives a final decision to proposed transactions and, if they are committed, forwards them to the database replicas. It "reads" from variables $\mathit{proposed}$ and thist (it relies on the transaction history to decide on whether to commit or abort it), and changes variables gdec and $\mathit{learnedSeq}$. As explained before, variable gdec simply assigns the final decision to a transaction; $\mathit{learnedSeq}$, however, represents the order in which each database should submit the active transactions committed by the termination protocol. These are the three safety properties the termination protocol must satisfy in order to ensure serializability:

Nontriviality. For any transaction t, t is decided ($\mathit{gdec}[t] \in \mathit{Decided}$) only if it was proposed.

Stability. For any transaction t, if t is decided at any time, then its decision does not change at any later time; and, for any database d, the value of $\mathit{learnedSeq}[d]$ at any time is a prefix of its value at all later times.

Consistency. There exists a sequence seq containing exactly one copy of every committed transaction (according to gdec) and a database state st such that $\mathit{Correct\ Serialization}(\mathit{seq}, \mathit{thist}, \mathit{InitialDBState}, \mathit{st})$ is true and, for every database d, $\mathit{learnedSeq}[d]$ is a prefix of seq.

The following theorem asserts that our complete abstract specification of a deferred update protocol is serializable. This result shows that every protocol that implements our specification automatically satisfies serializability. The proofs of our theorems can be found in [10].

Theorem 1. *Our abstract deferred update algorithm implements the specification of a serializable database given in Section 2.*

This theorem results in an interesting corollary, stated below. It shows that indeed databases are not required to be strict order-preserving serializable, an assumption that can be relaxed to our weaker definition of active order-preserving serializability.

Corollary 1. *Serializability is guaranteed by our specification if databases are active order-preserving serializable instead of strict order-preserving serializable.*

The three aforementioned safety properties are not strictly necessary to ensure serializability. Nontriviality can be relaxed so that non-proposed transactions may be aborted before they are proposed and Serializability is still guaranteed. However, we see no practical use of this since our algorithm already allows a transaction to be aborted at any point of the execution before it is proposed. Committing a transaction before proposing depends on making sure that the history of the transaction will not change and, in case it is active, on whether there are alternative sequences that ensure the Consistency properties if the transaction is committed or not, a rather complicated condition to be used in practice. Stability can be relaxed by allowing changes on suffixes of $\mathit{learnedSeq}[d]$ that have not been submitted to the database yet. However, keeping knowledge of what part of the sequence has already been submitted to the database and possibly changing the rest of it is equivalent to implementing our abstract algorithm with $\mathit{learnedSeq}[d]$ being the exact sequence locally submitted to the database. As a result, we see no practical advantage in relaxing Stability.

Consistency can be relaxed in a more complicated way. In fact, the different sequences $learnedSeq[d]$ can differ, as long as the set of intermediate states they generate (states in between transactions) are a subset of the intermediate states generated by some sequence seq containing all globally committed transactions and satisfying $CorrrectSerialization(seq, thist, InitialDBState, st)$ for some state st. Ensuring this property without forcing the $learnedSeq$ sequences to prefix a common sequence is hard and may lead to situations in which committed transactions cannot be added to a sequence $learnedSeq[d]$ for they would generate states that are not present in any sequence that could satisfy our consistency criterion.

One might think, for example, that the consistency property can be relaxed to allow commuting transactions that are not related (i.e., operate on disjunct parts of the database state) in the sequences $learnedSeq[d]$. For that, however, we have to make some assumptions about the database state in order to define what we mean by disjunct parts of the database state. For simplicity, let us assume our database state is a mapping from objects in a set $Object$ to values in a set $Value$ and operations can read or write a single object value. We define the objects of a transaction history h, represented by $Obj(h)$, to be the set of objects the operations in h read or write. A consistency property based on the commutativity of transactions that have no intersecting object sets can be intuitively defined as follows:

Alternative Consistency. There exists a sequence seq containing exactly one copy of every committed transaction (according to $gdec$) and a database state st such that $CorrectSerialization(seq, thist, InitialDBState, st)$ is true and, for every database d, $learnedSeq[d]$ contains exactly one copy of some committed transactions (according to $gdec$) and, for every transaction t in $learnedSeq[d]$, the following conditions are satisfied:
- Every transaction t' that precedes t in seq and shares some objects with t also precedes t in $learnedSeq[d]$, and
- Every transaction t' that precedes t in $learnedSeq[d]$ either precedes t in seq or shares no objects with t.

Although this new consistency condition seems a little complicated, it is weaker than our original property for it allows the sequences $learnedSeq[d]$ differ in their order for transactions that operate on different objects. The following theorem shows that this property is not enough to ensure Serializability in our abstract algorithm.

Theorem 2. *Our abstract deferred update algorithm with the Consistency property for termination changed for the Alternative Consistency property defined above does not implement the specification of a serializable database given in Section 2.*

This result basically means that one cannot profit much from using Generic Broadcast [15] algorithms to propagate committed transactions. Our properties as originally defined seem to be the weakest practical conditions for ensuring Serializability in deferred update protocols. In fact, we are not aware of any deferred update replication algorithm whose termination protocol does not satisfy the three properties above.

So far, we have not defined any liveness property for the termination protocol. Although we do not want to force protocols to commit transactions in any situation (since this might rule out some deferred update algorithms that conservatively abort transactions), we think that a termination protocol that does not update the sequences $learnedSeq[d]$ eventually, after having committed a transaction, is completely useless. Therefore, we add the following liveness property to our specification of the termination protocol:

Liveness. If t is committed at a given time, then $learnedSeq[d]$ eventually contains t.

As it happens with agreement problems like Consensus, this property must be revisited in failure-prone scenarios, since it cannot be guaranteed for databases that have crashed. Independently of that, one can easily spot some similarities between the properties we have defined and those of Sequence Agreement as explained in [16]. Briefly, in the sequence agreement problem, a set of processes agree on an ever-growing sequence of commands, built out of proposed ones. The problem is specified in terms of proposer processes that propose commands to be learned by learner processes, where $learned[l]$ represents the sequence of commands learned by learner l. Sequence Agreement is defined by the following properties:

Nontriviality. For any learner l, the value of $learned[l]$ is always a sequence of proposed commands.

Stability. For any learner l, the value of $learned[l]$ at any time is a prefix of its value at any later time.

Consistency. For any learners l_1 and l_2, it is always the case that one of the sequences $learned[l_1]$ and $learned[l_2]$ is a prefix of the other.

Liveness. If command V has been proposed, then eventually the sequence $learned[l]$ will contain V as an element.

This problem is a sequence-based specification of the celebrated atomic broadcast problem [17]. The exact relation between the termination protocol and Sequence Agreement is given by the following theorem.

Theorem 3. *The four properties Nontriviality, Stability, Consistency, and Liveness above satisfy the safety and liveness properties of Sequence Agreement for transactions that commit.*

One possible way of reading this theorem is that any implementation of the termination protocol is free to abort transactions, but it must implement Sequence Agreement for the transactions it commits. As a consequence, any lower bound or impossibility result for atomic broadcast and consensus applies to the termination protocol.

4 Conclusion

In this paper, we have formalized the deferred update technique for database replication and stated some intrinsic characteristics and limitations of it. Previous works have only considered new algorithms, with independent specifications, analysis, and correctness proofs. To the best of our knowledge, our work is first effort to formally characterize this family of algorithms and establish its requirements. Our general abstraction can be used to derive other general limitation results as well as to create new algorithms and prove existing ones correct. Some algorithms can be easily proved correct by a refinement mapping to ours. Others may require an additional effort due to the extra assumptions they make, but the task seems still easier than with previous formalisms. In our personal experience, we have successfully used our abstraction to obtain interesting protocols and correctness proofs, which will appear elsewhere.

Finally, to increase the confidence in our results, we have model checked our specifications using the TLA$^+$ model checker (TLC). Our specifications have been extensively checked for consistency problems besides type safety and deadlocks. For that we used a database containing a small vector of integers with operations that could read and write the vector's elements. Our model considered a limited number of transactions (up to 10), each one containing a few operations. The automatic checking confirmed our results and allowed us to find a number of small mistakes in the TLA$^+$ translation of our ideas. We strongly believe these specifications can be extended or directly used in future works in this area.

References

1. Kemme, B., Alonso, G.: A new approach to developing and implementing eager database replication protocols. ACM Transactions on Database Systems 25(3), 333–379 (2000)
2. Patino-Martínez, M., Jiménez-Peris, R., Kemme, B., Alonso, G.: Scalable replication in database clusters. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, Springer, Heidelberg (2000)
3. Pedone, F., Frølund, S.: Pronto: A fast failover protocol for off-the-shelf commercial databases. In: SRDS 2000. Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems, pp. 176–185. IEEE Computer Society Press, Los Alamitos (2000)
4. Pedone, F., Guerraoui, R., Schiper, A.: The database state machine approach. Journal of Distributed and Parallel Databases and Technology 14(1), 71–98 (2003)
5. Schiper, N., Schmidt, R., Pedone, F.: Optimistic algorithms for partial database replication. In: Shvartsman, A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 81–93. Springer, Heidelberg (2006)
6. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science 82(2), 253–284 (1991)
7. Lamport, L.: A simple approach to specifying concurrent systems. Communications of the ACM 32(1), 32–45 (1989)
8. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers, Inc., San Mateo, CA, USA (1996)
9. Lamport, L. (ed.): Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
10. Schmidt, R., Pedone, F.: A formal analysis of the deferred update technique. Technical report, EPFL (2007)
11. Bernstein, P., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
12. Papadimitriou, C.H.: The serializability of concurrent database updates. Journal of the ACM 26(4), 631–653 (1979)

13. Lynch, N., Merrit, M., Weihl, W., Fekete, A.: Atomic Transactions. Morgan Kaufmann Publishers, Inc., San Mateo, CA, USA (1994)
14. Beeri, C., Bernstein, P.A., Goodman, N.: A model for concurrency in nested transaction systems. Journal of the ACM 36(2), 230–269 (1989)
15. Pedone, F., Schiper, A.: Handling message semantics with generic broadcast protocols. Distributed Computing 15(2), 97–107 (2002)
16. Lamport, L.: Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research (2004)
17. Hadzilacos, V., Toueg, S.: Fault-tolerant broadcasts and related problems. In: Distributed systems, 2nd edn., pp. 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1993)