A Formal Analysis of the Deferred Update Technique

Rodrigo Schmidt[⋆,†] Fernando Pedone[†]

[⋆]École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland
Phone: +41 21 693 6668 Fax: +41 21 693 6490
E-mail: rodrigo.schmidt@epfl.ch

[†]University of Lugano (USI), CH-6904 Lugano, Switzerland
Phone: +41 91 912 4695 Fax: +41 91 913 8536
E-mail: fernando.pedone@unisi.ch

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.

Keywords: Database replication, deferred update technique, distributed transactions, group communication.

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 database. 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., [5, 12, 13, 14, 16]) 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 [12], certify transactions locally to the database that executed them [5], execute all concurrent update transactions at the same database [13], reorder transactions during certification [14], and even cope with partial database replication [16]. 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 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 such a 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, respectively, update data items x and y inside the database 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 by, for example, a refinement mapping [1]. Our specifications use atomic actions to define safety properties [6, 9]. Due to the space limitations, we present only high-level specifications in the main text. Complete TLA$^+$ [7] specifications, which have been model checked for a finite subset of the possible execution scenarios, are given in the appendix. Moreover, in order to help the reader cope with our notation, a glossary/index also appears at the appendix.

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 [3]. 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 [11]. 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 [3]. This approach is highly dependent on the implemented protocol and, as explained in [10], 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 [7, 9, 10].

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.

$$
\begin{array}{l}
CorrectAtomicHist(h \in THist, initst \in DBState, finalst \in DBState) \triangleq \\
\quad \textbf{if} \quad h = \langle\rangle \quad \textbf{then} \quad initst = finalst \\
\qquad\qquad\quad \textbf{else} \quad \exists ist \in DBState : \quad CorrectOp(Head(h).op, Head(h).res, initst, ist) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad CorrectAtomicHist(Tail(h), ist, finalst)
\end{array}
$$

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.

$$\begin{array}{l} \mathit{CorrectSerialization}(\mathit{seq} \in \mathit{Seq}(\mathit{Tid}), \mathit{thist} \in \mathit{THistVector}, \mathit{initst} \in \mathit{DBState}, \mathit{finalst} \in \mathit{DBState}) \triangleq \\ \quad \textbf{if} \quad \mathit{seq} = \langle\rangle \quad \textbf{then} \quad \mathit{initst} = \mathit{finalst} \\ \qquad\qquad\quad \textbf{else} \quad \exists \mathit{ist} \in \mathit{DBState} : \quad \mathit{CorrectAtomicHist}(\mathit{thist}(\mathit{Head}(\mathit{seq})), \mathit{initst}, \mathit{ist}) \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{CorrectSerialization}(\mathit{Tail}(\mathit{seq}), \mathit{thist}, \mathit{ist}, \mathit{finalst}) \end{array}$$

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 $\mathit{IsSerializable}(S, \mathit{thist}, \mathit{initst})$ 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 $\mathit{CorrectSerialization}(\mathit{seq}, \mathit{thist}, \mathit{initst}, \mathit{finalst})$ 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 $\mathit{DBRequest}(t, \mathit{req})$, which represents the reception of a request *req* on behalf of transaction t, and $\mathit{DBResponse}(t, \mathit{rep})$, 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*.

Figure 1 presents the atomic actions of our specification. Action $\mathit{ReceiveReq}(t, \mathit{req})$ is responsible for receiving a request on behalf of transaction t. Action $\mathit{ReplyRep}(t, \mathit{rep})$ 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 $\langle q[t], \mathit{rep}\rangle$ to $\mathit{thist}[t]$ (we use $\circ$ to represent the standard *append* operation for sequences). Action $\mathit{DoAbort}(t)$ simply aborts a transaction if it has not been decided yet. Action $\mathit{DoCommit}(t)$ 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*.

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 $q[t] \in Op \wedge rep \in Result$

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:
- $tdec[t] \leftarrow Committed$

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

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.

$$\forall st1, st2 \in DBState : CorrectAtomicHist(h, st1, st2) \Rightarrow st1 = st2 \quad (2)$$

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* $CorrectOp(op, res1, st, st1)$*, then there is no result* $res2$ *and database state* $st2$ *such that* $st1 \neq st2 \wedge CorrectOp(op, res2, st, st2)$*.*

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 [2]. 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 [3] 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 *serialSeq*, initially equal to the empty sequence, and changing the *DoCommit* action. We show the required changes in Figure 2 below. The strict case (a) is simple and only requires that $serialSeq \circ t$ be a correct sequential execution of all committed transactions. The action automatically extends *serialSeq* with t. The active case (b) is a little more complicated to explain and requires some extra notation. Let $Perm(S)$ be the set of all permutations of elements in finite set S (all the possible orderings of elements in S), and let $ActiveExtension(seq, t)$ be seq if $thist[t]$ 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 $ActiveExtension(seq, t)$ is a subsequence of it.[1] In this action, *serialSeq* is extended with t only if t is an active transaction.

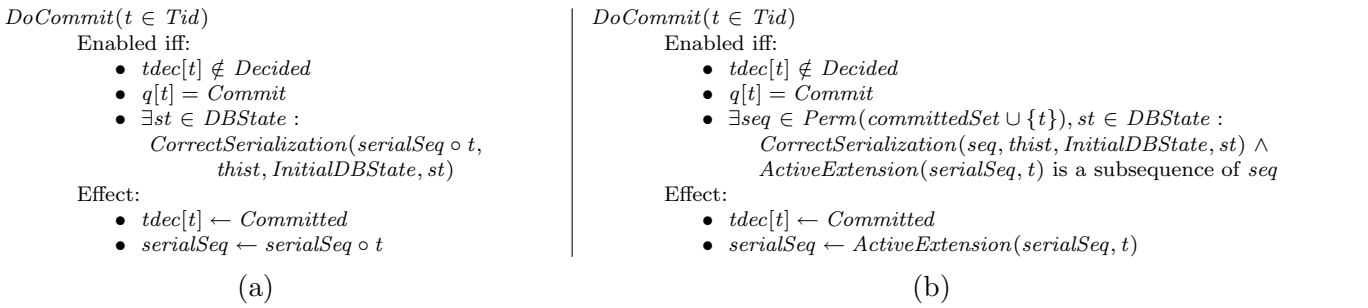

Figure 2: *DoCommit* action for (a) strict and (b) active order-preserving serializability.

3.2 The 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

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

assume a concurrent termination protocol, fully explained in the next 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 committing or aborting 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, disconsidering 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 *PrematureAbort* 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 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 The termination protocol and its implications

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 *proposed* and *thist* (it relies on the transaction history to decide on whether commit or abort it), and changes variables *gdec* and *learnedSeq*. As explained

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
  - $dcnt[d][t] \leftarrow dcnt[d][t] + 1$
  - $dcom[d][t] \leftarrow rep = Committed$

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

before, variable *gdec* simply assigns the final decision to a transaction; *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 ($gdec[t] \in Decided$) 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 $learnedSeq[d]$ 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 $CorrectSerialization(seq, thist, InitialDBState, st)$ is true and, for every database d, $learnedSeq[d]$ is a prefix of *seq*.

The following theorem asserts that our complete abstract specification of a a deferred update protocol is serializable. This result shows that every protocol that implements our specification automatically satisfies serializability. The proofs of our theorems are given in the appendix.

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. Non-triviality 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 sufixes of $learnedSeq[d]$ 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 $learnedSeq[d]$ 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 intermmediate states generated by a sequence seq corresponding to a permutation of all globally committed transactions that satisfies $CorrrectSerialization(seq, thist, InitialDBState, st)$ 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 [8]. 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 [4]. 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] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] C. Beeri, P. A. Bernstein, and N. Goodman. A model for concurrency in nested transaction systems. *Journal of the ACM*, 36(2):230–269, Apr. 1989.

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

[4] V. Hadzilacos and S. Toueg. *Fault-tolerant broadcasts and related problems*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2 edition, 1993.

[5] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, 2000.

[6] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, Jan. 1989.

[7] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[8] L. Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2004.

[9] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, USA, 1996.

[10] N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, USA, 1994.

[11] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979.

[12] M. Patino-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proc. of the* 14^{th} *International Symposium on Distributed Computing (DISC'00)*, 2000.

[13] F. Pedone and S. Frølund. Pronto: A fast failover protocol for off-the-shelf commercial databases. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'2000)*, pages 176–185, 2000.

[14] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Journal of Distributed and Parallel Databases and Technology*, 14(1):71–98, 2003.

[15] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, April 2002.

[16] N. Schiper, R. Schmidt, and F. Pedone. Optimistic algorithms for partial database replication. In *Proceedings of the 10th International Conference on Principles of Distributed Systems (OPODIS'2006)*, pages 81–93, 2006.

Appendix – Glossary

A Proof of Theorem 1

A.1 The main invariants

In order to prove Theorem 1, we state three invariants satisfied by our abstract algorithm. The invariants are very intuitive, given the algorithm's expected behavior. However, a rigorous proof that the algorithm actually satisfies them is given in Section A.3. Before presenting these invariants, though, we introduce some auxiliary notation. We let $Commit > tedAt(d)$ be the set of all transactions that have been committed at database d under any version number. That is,

$$CommittedAt(d) \triangleq \{t \in Tid : \exists v \in \mathbb{N} : DB(d)!tdec[\langle t, v\rangle] = Committed\}.$$

Moreover, we let $NVserialSeq(d)$ be the standard projection of sequence $DB(d)!serialSeq$ without the transactions' version numbers.

Our first invariant relates the databases' internal states to the global variables *thist* and *learnedSeq*. It is mainly based on the fact that databases are active order-preserving serializable and transactions proposed to the termination protocol (which includes all active ones) have their *Commit* requests submitted to database d according to the order specified by $learnedSeq[d]$.

Database Invariant For every database d, there exists a sequence $seq \in Perm(CommittedAt(d))$, and a database state st such that all conditions below hold:

1. $CorrectSerialization(seq, thist, InitialDBState, st)$,
2. $NVserialSeq(d)$ is the subsequence of seq containing all its active transactions, and
3. $NVserialSeq(d)$ is the subsequence of a prefix of $learnedSeq[d]$ that contains all its active transactions.

Besides the invariant above, our proof uses the following two auxiliary invariants.

tdec **Invariant** For every transaction t, if $t \notin proposed$ and $pdec[t] = Committed$, then $thist[t]$ is passive and $t \in CommittedAt(DBof(t))$.

dreply **Invariant** For every transaction t, if $dreply[t] = Committed$ and $t \notin proposed$, then $thist[t]$ is passive and t belongs to $CommittedAt(DBof(t))$.

A.2 The theorem proof

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

PROOF: The proof is by a refinement mapping where *thist* and q are implemented by the variables with the same name and *tdec* is implemented according to the definition in terms of *proposed*, *pdec*, and *gdec* given in the explanation of our abstract deferred update algorithm. Below, we show that each action executed by the abstract algorithms implements an action of our serializable database specification.

1. Action *ReceiveReq* implements the action with the same name.
   PROOF SKETCH: Pre- and post-conditions on variables q and *thist* are exactly the same. The action may change variable *proposed*, influencing *tdec*. However, t is proposed only if $tdec[t] \notin Decided$ and the Nontriviality property of the termination protocol ensures that *tdec* remains the same.
2. Action *ReplyReq* implements the action with the same name.
   PROOF SKETCH: The actions' pre- and post-conditions are obviously stricter than those of the original action.

3. Action *PrematureAbort* implements action *DoAbort*
   PROOF SKETCH: The action changes *pdec* iff its changes reflect the changes on *tdec* performed by action *DoAbort*.

We now skip action *PassiveCommit* for it deserves a slightly more complicated analysis. It is explained right after the simple actions below.

4. Actions *DBReq*, *DBRep*, and internal actions performed by any database represent stuttering steps in our specification of a serializable database
   PROOF SKETCH: Such actions do not change variables q, *thist*, *proposed*, *pdec* and *gdec*, not influencing the mapping.
5. Changes on *learnedSeq* performed by the termination protocol also implement stuttering steps
   PROOF SKETCH: Such changes have no influence on variables q, *thist*, *proposed*, *pdec* and *gdec*.

The two cases below deserve a special analysis and a higher degree of rigorousness.

6. Action *PassiveCommit* implements action *DoCommit*
   PROOF: Let *committedSet* be the set of all committed transactions (according to the definition of *tdec* in terms of *pdec* and *gdec*). We must show that $IsSerializable(committedSet \cup \{t\}, thist, InitialDBState)$ is true before the execution of *PassiveCommit*. We prove that in the following proof steps.
   6.1. Choose a sequence *gseq* that contains exactly one copy of every transaction mapped to *Committed* in *gdec* and satisfies the two conditions below
      1. $\exists st \in DBState : CorrectSerialization(gseq, thist, InitialDBState, st)$
      2. $\forall d \in Database : learnedSeq[d]$ is a prefix of *gseq*

      PROOF: This sequence exists for the Consistency property of the termination protocol.

   LET: *subgseq* be the subsequence of *gseq* containing all its active transactions.

   6.2. For every database d, $NVSerialSeq(d)$ is a prefix of *subgseq*.
      PROOF: By step 6.1 and the third item of the Database Invariant.

   6.3. For any transaction t contained by *subgseq*, let $stgen(t)$ be the unique database state *st* such that $CorrectSerialization(preft, thist, InitialDBState, st)$ is satisfied for the prefix *preft* of *subgseq* limited by (and containing) t.
      PROOF: Such state exists by the step $PC1$ and the definition of *subgseq*, and it is unique by Assumption 1 (State-deterministic Operations).

   6.4. For every passive transaction t that belongs to $CommittedAt(DBof(t))$, that is, every transaction committed with some version at its delegate database, either one of the two conditions below is satisfied:
      - $CorrectAtomicHist(thist[t], InitialDBState, InitialDBSTAte)$, or
      - $\exists t_w \in Tid : t_w$ appears in *subgseq* and $CorrectAtomicHist(thist[t], stget(t_w), stget(t_w))$.

      PROOF: By the Database Invariant and the fact that t belongs to $CommittedAt(DBof(t))$, there exists a sequence *seq* such that:
      1. *seq* contains t,
      2. there exists a database state *st* such that $CorrectSerialization(seq, thist, InitialDBState, st)$,
      3. $NVserialSeq(DBof(t))$ is the subsequence of *seq* containing all its active transactions,
      4. $NVserialSeq(DBof(t))$ is the subsequence of a prefix of $learnedSeq[d]$ containing all its active transactions.

      Let *strippedseq* be the subsequence of *seq* containing all its active transactions and t, only. Since only passive transactions are taken out, by the definition of a correct serialization, *strippedseq* also represents a correct serialization with respect to *thist*, *InitialDBState*, and *st*. Now take the longest prefix of *strippedseq* that does not contain t and let us call it *strippedpref*. If *strippedpref* is empty, then the definition of a correct serialization and the fact that t is passive imply that $thist[t]$ is

atomically correct with respect to *InitialDBState* (first condition of step 6.4). Otherwise, *strippedpref* is a prefix of *subgseq*, and Assumption 1 (State-deterministic Operations) implies that $thist[t]$ is atomically correct with respect to $stgen(t_w)$, where t_w is the transaction immediately before t in *strippedseq*, which satisfies the second condition of step 6.4.

6.5. Q.E.D.
PROOF: By the Consistency and Nontriviality properties of the termination protocol, *gseq* contains every transaction t such that t is proposed and $gdeq[t]$ equals *Committed*. We first extend *gseq* with all other committed transactions. By our mapping of *tdec*, these are the transactions not in proposed but mapped to *Committed* by *pdec*. However, the *tdec* Invariant tells us that these transactions are passive and internally committed at their delegate databases. Step 6.4 tells us that they can be inserted at some position of *gseq* and still generate a correct serialization, by the definition of *CorrectSerialization*. Last, the *dreply* Invariant and the pre-condition of action *PassiveCommit* also imply that t is passive and internally committed at $DBof(t)$. Therefore, by step 6.4, t can also be inserted at some position of *gseq* and generate a correct serialization with initial state *InitialDBState*.

7. Changes on $gdec[t]$ performed by the termination protocol implement either $DoCommit(t)$ or $DoAbort(t)$
PROOF SKETCH: Here we assume the termination protocol changes only one entry of *gdec* at a time. An implementation that does not do that can be easily proved equivalent to this behavior by the creation of "dummy" states that change one entry at a time with the introduction of prophecy variables [1]. If $gdec[t]$ is changed to *Aborted*, the Nontriviality and Stability properties automatically imply the pre- and post-conditions of $DoAbort(t)$. Otherwise, we must follow basically the same steps as in step 6. The only two (small) differences are the following:
   - Step 1 should be based on the consistency property guaranteed after *gdec* is changed, producing a sequence *gseq* that already contains t.
   - The Q.E.D. step does not have to add t to the built sequence since it is originally in the *gseq* sequence initially created.

A.3 Proving the basic invariants

In order to prove the basic invariants, we have to define a number of auxiliary invariants. We divided the auxiliary invariants into two types: transaction invariants and database-transaction invariants. The first group refers to invariants that are based on transactions only. The second group refers to invariants that relate transactions and databases.

The only extra notation we introduce in these auxiliary invarints is the definition of an operator $Substr(seq, begin, end)$ for a sequence *seq* and naturals *begin* and *end*, used in invariant DTI5. This operator returns the substring of *seq* from index *begin* until index *end*. If $end < begin$, it is assumed to return an empty sequence.

Transaction Invariants (TI) For every transaction t:

1. $(tdec[t] \notin Decided \wedge q[t] \in Op) \Rightarrow$
   (a) $\forall d \in Database : t \notin CommittedAt(d)$ and
   (b) $t \notin proposed$
2. $(tdec[t] \notin Decided \wedge q[t] = NoReq) \Rightarrow$
   (a) $\forall d \in Database : t \notin CommittedAt(d)$,
   (b) $t \notin proposed$,

(c) $vers[DBof(t)][t] = 0$,

(d) $thist[t] = DB(DBof(t))!thist[\langle t, 0\rangle]$,

(e) $dcnt[DBof(t)][t] = Len(thist[t])$,

(f) $\forall d \in Database : DB(d)!q[\langle t, vers[d][t]\rangle] = NoReq$, and

(g) $dreq[t] = NoReq$

3. $dcnt[DBof(t)][t] > Len(thist[t]) \wedge t \notin proposed \Rightarrow$

   (a) $\forall d \in Database : DB(d)!q[\langle t, vers[d][t]\rangle] = NoReq$,

   (b) $vers[DBof(t)][t] = 0$,

   (c) $thist[t] \circ \langle dreq[t], dreply[t]\rangle = DB(DBof(t))!thist[\langle t, 0\rangle]$, and

   (d) $dcnt[DBof(t)][t] = Len(thist[t]) + 1$

4. $t \in proposed \Rightarrow$

   (a) $thist[t] = DB(DBof(t))!thist[\langle t, 0\rangle]$,

   (b) $tdec[t] \in Decided \vee q[t] = Commit$,

   (c) $dcnt[DBof(t)][t] \geq Len(thist[t]) \vee vers[DBof(t)][t] > 0$, and

   (d) $dreq[t] = NoReq$

5. $dreq[t] \in Request \wedge dcnt[d][t] = Len(thist[t]) \Rightarrow$

   (a) $t \notin proposed$,

   (b) $vers[DBof(t)][t] = 0$, and

   (c) $thist[t] = DB(DBof(t))!thist[\langle t, 0\rangle]$

6. $(tdec[t] \notin Decided \wedge t \notin proposed) \Rightarrow dreq[t] = q[t]$

7. $dreq[t] = Commit \Rightarrow$

   (a) $thist[t]$ is passive and

   (b) $t \notin proposed$

Database-Transaction Invariants (DTI) For every database d and transaction t:

1. $DB(d)!q[\langle t, v\rangle] \neq NoReq \Rightarrow v = vers[d][t]$

2. $\forall v \neq vers[d][t] : DB(d)!tdec[\langle t, v\rangle] \neq Committed$

3. If $DB(d)!q[\langle t, vers[d][t]\rangle] = Commit$, then either

   (a) $thist[t] = DB(d)!thist[\langle t, vers[d][t]]$ or

   (b) the projection of the operations in $DB(d)!thist[\langle t, vers[d][t]\rangle]$ equals the projection of the operations in $ActHist(t)$.

4. If $DB(d)!tdec[\langle t, vers[d][t]\rangle] = Committed$ and $thist[t]$ is active, then for all database states $st1$, $st2$, and $st3$:
   $\wedge\ CorrectAtomicHist(DB(d)!thist[\langle t, vers[d][t]], st1, st2)$
   $\wedge\ CorrectAtomicHist(thist[t], st1, st3)$
   $\Rightarrow st2 = st3$

5. If $\neg(d = DBof(t) \wedge vers[d][t] = 0)$, then the projection of the operations in $DB(d)!thist[\langle t, vers[d][t]\rangle]$ equals the projection of the operations in $Substr(ActHist(t), 1, dcnt[d][t])$.

6. If $DB(d)!tdec[\langle t, vers[d][t]\rangle] = Committed$ and t is proposed, then t appears in $learnedSeq[d]$ and every transaction t' that precedes t in $learnedSeq[d]$ satisfies $dcom[d][t']$.

7. $dcom[d][t] \Rightarrow t \in CommittedAt(d)$

8. If $DB(d)!q[\langle t, vers[d][t]\rangle] = Commit$, then either $tdec[t] \in Decided$ or $q[t] = Commit$.

9. If $DB(d)!q[\langle t, vers[d][t]\rangle] = Commit$ and t is proposed, then t appears in $learnedSeq[d]$ and every transaction t' that precedes t in $learnedSeq[d]$ satisfies $dcom[d][t']$.

10. $DB(d)!q[\langle t, vers[d][t]\rangle] \in Request \wedge t \notin proposed \Rightarrow$

    (a) $d = DBof(t)$,

    (b) $dcnt[d][t] = Len(thist[t])$,

    (c) $DB(d)!q[\langle t, vers[d][t]\rangle] = dreq[t]$,

    (d) $vers[DBof(t)][t] = 0$, and

    (e) $thist[t] = DB(DBof(t))!thist[\langle t, 0\rangle]$

11. $DB(d)!q[\langle t, vers[d][t]\rangle] \in Op \wedge t \in proposed \Rightarrow$

    (a) $d \neq DBof(t) \vee vers[d][t] \neq 0$ and

    (b) $DB(d)!q[\langle t, vers[d][t]\rangle] = ActHist(t)[dcnt[d][t] + 1].op$

12. $\forall v > vers[d][t] : DB(d)!thist[\langle t, v\rangle] = \langle\rangle$

The intuition of the proof is quite simple. It is relatively easy to check the invariants for the initial state of the abstract algorithm. We then assume that they are true and show that they remain true after the execution of each of the algorithm's atomic actions no matter what was the state upon which the action was executed (as long as the invariants were satisfied on it). In the following we analyze action by action and sketch the proof for each of the invariants we have previously defined.

Action *ReceiveReq*

Database Invariant This action does not change any of the variables involved in the Database Invariant.

tdec **Invariant** With respect to the *tdec* Invariant, this action can only propose a transaction, which does not invalidate the invariant.

dreply **Invariant** As in the previous case, this action can only propose a transaction, which does not invalidate the invariant.

TI1 This action sets $q[t]$ to a request and may add t to *proposed*. Invariant TI2(a,b) and the fact that t is added to *proposed* only if $q[t]$ is set to *Commit*, which is not in *Op*, imply that TI1 is preserved.

TI2 The action sets $q[t]$ to a request (different from *NoReq*), which preserves the invariant, since it invalidates the implication condition for transaction t.

TI3 Invariant TI2(e) and the action's pre-condition invalidate the implication condition of this invariant for transaction t.

TI4 If t is proposed, then TI2(d) ensures TI4(a). TI4(b) is ensured because t is proposed only if $q[t]$ is set to *Commit*, TI4(c) is ensured by TI2(e), and TI4(d) is ensured by TI2(g).

TI5 If $dreq[t]$ is set to a value in *Request*, the invariant is ensured by TI2(b-d).

TI6 The action sets $q[t]$ to *req*. It does nothing else if $tdec \in Decided$, but this condition invalidates the invariant's implication condition. Otherwise, the action either proposes t, which also invalidates the invariant's implication condition, or it makes $dreq[t]$ equal to $q[t]$. In all the cases, the invariant is preserved.

TI7 Condition (a) is easily verified. Condition (b) is ensured in case the action sets $dreq[t]$ to *Commit* by TI2(b).

DTI1-5,7 Automatically preserved.

DTI6 Transaction t is proposed only if $tdec[t] \notin Decided$ and the action's pre-condition together with invariant TI2(a) implies that t has not been committed at any database under any version, which invalidates this invariant's implication condition.

DTI8 For the sake of contradiction, assume there is a database d such that $DB(d)!q[\langle t, vers[d][t]\rangle] = Commit$, $tdec[t] \notin Decided$, and $q[t]$ is set to *Commit* by this action. Then, invariant TI2(f) with these assumptions and the action's pre-condition imply that $DB(d)!q[\langle t, vers[d][t]\rangle] = NoReq$, a contradiction with our first assumption.

DTI9 Transaction t is proposed only if $tdec[t] \notin Decided$ and the action's pre-condition together with invariant TI2(f) implies that $DB(d)!q[\langle tvers[d][t]\rangle] = NoReq$, which invalidates this invariant's implication condition.

DTI10 For the sake of contradiction, assume there is a database d such that $DB(d)!q[\langle t, vers[d][t]\rangle] \in Request$, $t \notin proposed$, and $dreq[t]$ is set to a value different from $DB(d)!q[\langle t, vers[d][t]\rangle]$ (we concentrate on condition (c) since the verification of the other conditions and the implication itself are simple). Then, invariant TI2(f) with these assumptions and the fact that $dreq[t]$ is only changed if $tdec[t] \notin Decided$ imply that $DB(d)!q[\langle t, vers[d][t]\rangle] = NoReq$, a contradiction with our first assumption.

DTI11 If t is proposed by this action, then $tdec[t] \notin Decided$ and invariant TI2(f) invalidates this invariant's implication condition.

DTI12 Automatically preserved.

Action *ReplyReq*

Database Invariant The action changes *thist*, which could affect the Database Invariant. However, TI1(a) implies that t has not been committed at any database, preserving the Database Invariant.

tdec **Invariant** The action only changes $thist[t]$ if $tdec[t] \notin Decided$, which is not true if $t \notin proposed$ and $pdec[t] = Committed$, by the definition of *tdec*. Therefore, the invariant is preserved.

dreply **Invariant** According to the action's pre-condition, $thist[t]$ is only changed if $rep \in Result$ and $rep = dreply[t]$, which implies that $dreply[t] \neq Committed$ and automatically preserves the invariant.

TI1 The action changes $q[t]$ to *NoReq*, which automatically preserves this invariant.

TI2 The action's pre-condition implies that it is executed for a transaction t such that $tdec[t] \notin Decided$ only if $q[t] \in Op$. Invariant $TI1$ implies that no database has committed t in this case and t has not been proposed. Conditions (c-f) are ensured by TI3(a-d) and condition (g) is ensured by the action definition.

TI3 By invariant TI3 and the action's definition, if $thist[t]$ changes, its length will equal $dcnt[DBof(t)][t]$, which just invalidates TI3's implication condition.

TI4 As for conditions (a), (c), and (d), the action only changes $thist[t]$ and $dreq[t]$ if $tdec[t] \notin Decided$ and $q[t] \in Op$. Invariant TI1(b) implies that $t \notin proposed$, which contradicts the invariant's implication condition. As for condition (b), assume $t \in proposed$, $tdec[t] \notin Decided$ and $q[t]$ is changed from *Commit* to *NoReq* by this action. Such assumptions conflict with the action definition since $q[t]$ must be in Op for it to be enabled when $tdec[t] \notin Decided$.

TI5 The action only changes $thist[t]$ if it sets $dreq[t]$ to *NoReq*, which automatically preserves the invariant.

TI6 Easily verified.

TI7 If the action changes $thist[t]$, it also sets $dreq[t]$ to *NoReq*, preserving the invariant.

DTI1-2 Automatically preserved.

DTI3 The only variable related to the invariant that is changed by the action is *thist*. However, $thist[t]$ is only changed if $tdec[t] \notin Decided$, $q[t] \in Op$, and $dcnt[DBof(t)][t] > Len(thist[t])$. Invariant TI1(b) validates the implication condition of TI3 for t and TI3(a) automatically invalidates the implication condition of DTI3, preserving the invariant.

DTI4 Again, the only variable of interest is *thist*, and it is changed only if $tdec[t] \notin Decided$ and $q[t] \in Op$. In this case, invariant TI1(a) invalidates the implication condition of DTI4, preserving the invariant.

DTI5 The action may only extend $thist[t]$, which automatically preserves this invariant.

DTI6-7 Automatically preserved.

DTI8 Assume, for the sake of contradiction, that $DB(d)!q[\langle t, vers[d][t]\rangle] = Commit$, $tdec[t] \notin Decided$ and $q[t]$ is changed from *Commit* to *NoReq* by this action. However, the action is only enabled when $tdec[t] \notin Decided$ if $q[t] \in Op$, which contradicts the fact that $q[t]$ equals *Commit* before the action is executed.

DTI9 Automatically preserved.

DTI10 The action only changes $thist[t]$ and $dreq[t]$ if $tdec[t] \notin Decided$ and, in this case, the action's pre-condition implies that $dcnt[d][t] > Len(thist[t])$, which conflicts with the invariant's condition (b) and contradicts its validity before the action execution, unless the implication condition is not satisfied. Since the action does not change the variables involved in the implication condition, the invariant is preserved.

DTI11 This action only changes $thist[t]$ if $tdec[t] \notin Decided$ and invariant TI2(f) invalidates DTI11's implication condition, preserving the invariant.

DTI12 Automatically preserved.

Action *PrematureAbort*

Database Invariant Automatically preserved since this invariant does not involve *pdec*.

tdec **Invariant** The invariant preserved since it involves only transactions t such that $t \notin proposed$ and $pdec[t] = Committed$. *PrematureAbort* executes for a transaction t only if $t \notin proposed$ and $pdec[t] \notin Decided$ and it changes $pdec[t]$ to *Aborted*, not interfering with the invariant condition.

dreply **Invariant** Automatically preserved.

TI1-2,6 This action can only change $tdec[t]$ from *Unknown* to *Aborted*, which preserves these invariants since $Aborted \in Decided$.

TI3 Automatically preserved.

TI4 Conditions (a), (c) , and (d) are automatically preserved. As for condition (b), this action can only change $tdec[t]$ to a value in *Decided* (*Aborted*), preserving the invariant as well.

TI5,7 Automatically preserved

DTI1-7,9-12 Automatically preserved.

DTI8 Easily verified.

Action *PassiveCommit*

Database Invariant Automatically preserved.

tdec **Invariant** If $tdec[t]$ is changed to *Committed*, the action's pre-condition implies that $dreply[t]$ equals *Committed* and the *dreply* Invariant ensures that $thist[t]$ is passive and t belongs to $CommittedAt(DBof(t))$.

dreply **Invariant** Automatically preserved.

TI1-2,6 This action can only change $tdec[t]$ from *Unknown* to *Committed*, which preserves these invariants since $Committed \in Decided$.

TI3 Automatically preserved.

TI4 Condition (a), (c), and (d) are automatically preserved. As for condition (b), this action can only change $tdec[t]$ to a value in *Decided* (*Committed*), preserving the invariant as well.

TI5,7 Automatically preserved

DTI1-7,9-12 Automatically preserved.

DTI8 Easily verified.

Action *DBReq* **joint with** $DB(d)!ReceiveReq$

Database Invariant Automatically preserved.

tdec **Invariant** Automatically preserved.

dreply **Invariant** Automatically preserved.

TI1 Automatically preserved.

TI2 This action could break condition (f) of invariant TI2 for some transaction t. If the action is enabled by its first condition, invariants TI5(a) and TI6 imply that $q[t] \in Request$, contradicting TI2's implication condition. If the action is enabled by its second or third condition, then the termination properties ensure that $t \in proposed$ and invariant TI4(b) contradict TI2's implication condition.

TI3 This action sets $DB(d)!q[\langle t, vers[d][t]\rangle]$ to a value different from *NoReq* and could break TI3(a) for t. However, condition 1 requires that $dcnt[DBof(t)][t] = Len(thist[t])$, contradicting TI3's implication condition. Conditions 2 and 3 (with the Nontriviality property of the termination protocol) imply that t has been proposed, also contradicting TI3's implication condition.

TI4-7 Automatically preserved.

DTI1 Obviously preserved.

DTI2 Automatically preserved.

DTI3 This action can only set $Db(d)!q[\langle t, vers[d][t]\rangle]$ to *Commit* by conditions 1 or 3. In the first case, the invariant is guaranteed by invariant TI5(b-c). If condition 3 enables this action, there are two cases to consider.

$d = DBof(t) \wedge vers[d][t] = 0$**:** The Nontriviality and Consistency properties of the termination protocol imply that $t \in proposed$ and invariant TI4(a) ensures that DTI3 is preserved.

$dcnt[d][t] = Len(ActHist(t))$**:** In this case, DTI3 is ensured by DTI5.

DTI4-7 Automatically preserved.

DTI8 If the action is triggered by condition 1 and sets $DB(d)!q[\langle t, vers[d][t]\rangle]$ to *Commit*, then $dreq[t] = Commit$. Invariant TI7 implies that $t \notin proposed$ and invariant TI6 ensures DTI8. If the action is triggered by condition 2, it cannot set $DB(d)!q[\langle t, vers[d][t]\rangle]$ to *Commit*. Finally, if the action is triggered by condition 3, then the Consistency and Nontriviality properties of the termination protocol ensure that $tdec[t] = Committed$.

DTI9 If the action is triggered by condition 1, then $t \notin proposed$ by TI7(b). If the action is triggered by condition 3, this condition itself ensures DTI9.

DTI10 The only enabling condition that could interfere with this invariant for this action is condition 1. However, it can be easily verified that it ensures DTI10(c).

DTI11 The only enabling condition that could interfere with this invariant for this action is condition 2. TI4(c) ensures DTI11(a) and DTI11(b) is easily verified.

DTI12 Automatically preserved.

Action $DB(d)!DoAbort$

Database, *tdec***, and** *dreply* **Invariants, and TI1-2** The action can only change $DB(d)!tdec$ by internally abort a transaction, which does not change $CommittedAt(d)$.

TI3-7 Automatically preserved.

DTI1 Automatically preserved.

DTI2 Easily verified since it changes $DB(d)!tdec[\langle t, v\rangle]$ from *Unknown* to *Aborted*.

DTI3 Automatically preserved.

DTI4 Easily verified since it changes $DB(d)!tdec[\langle t, v\rangle]$ from *Unknown* to *Aborted*.

DTI5 Automatically preserved.

DTI6 Easily verified since it changes $DB(d)!tdec[\langle t, v\rangle]$ from *Unknown* to *Aborted*.

DTI7-12 Automatically preserved.

Action $DB(d)!DoCommit$

Database Invariant There are two cases to consider.

$t \in proposed$ Take the sequence *seq* of the Database Invariant before the action is executed. Invariants DTI9 and DTI7 imply that all transactions previous to t in $learnedSeq[d]$ are already present in *seq*. Invariants DTI6 and DTI7 imply that all proposed transactions committed at d appear before t in $learnedSeq[d]$, otherwise t would have already been committed at d and the action would not be enabled. This fact and conditions 2 and 3 of the Database Invariant imply that the sequence of states generated by *seq* is the same as the one generated by the longest prefix not including

t of the sequence defined in the Consistency property for termination. Let st be the last state generated by this sequence. The Consistency property ensures that there is a state $st2$ such that $CorrectAtomicHist(thist[t], st, st2)$ is true. As a result, we can add t to the end of seq and satisfy condition 1 of the Database Invariant after the action is executed. By DTI3, Assumption 1, the definition of $ActHist$, if $thist[t]$ is active, so is $DB(d)!thist[\langle t, vers[d][t]\rangle]$ and t should be added to $DB(d)!serialSeq$, satisfying condition 2 of the Database Invariant. Condition 3 is satisfied because, as we pointed out in the very beginning of this case's analysis, a proposed transaction is committed at d iff it appears before t in $learnedSeq[d]$.

$t \notin proposed$ As before, take sequence seq of the Database Invariant before the action is executed. A simple induction on the size of $DB(d)!serialSeq$ and $NVerialSeq(d)$ taking into consideration the Database Invariant as well as DTI4 shows that the sequence of different states generated by these two sequences with respect to $DB(d)!thist$ and $thist$, respectively, is exactly the the same. DTI10(c), TI7 and the definition of action $DoCommit$ imply that t is passive and it can be atomically executed after some of the states mentioned in the previous step. We can place t exactly after that state is generated in seq, satisfying condition 1 of the Database Invariant after the action is executed. Conditions 2 and 3 are automatically satisfied since t is passive.

tdec **Invariant** Easily preserved, since $CommittedAt(d)$ can only be increased.

dreply **Invariant** Easily preserved, since $CommittedAt(d)$ can only be increased.

TI1-2 Easily preserved, given DTI7.

TI3-7 Automatically preserved.

DTI1 Automatically preserved.

DTI2 Easily verified given DTI1.

DTI3,5 Automatically preserved.

DTI4 By DTI3.

DTI6 By DTI9.

DTI7 Obviously preserved, since $CommittedAt(d)$ can only be increased.

DTI8-12 Automatically preserved.

Action *DBRep* **joint with** $DB(d)!ReplyReq$

Database and *tdec* **Invariants** Automatically preserved.

dreply **Invariant** $dreply[t]$ is set to *Committed* only if $DB(d)!q[\langle t, vers[d][t]\rangle]$ equals *Commit*. Invariants DTI10(c) and $TI7$(a) imply that $thist[t]$ is passive, and invariant DTI10(a) with the definition of action $DB(d)!ReplyReq$ ensures that t will belong to $CommittedAt(DBof(t))$.

TI1 Automatically preserved.

TI2 The fact that $t \in proposed$ contradicts TI2(b) and imply that the implication condition of TI2 is false. Therefore, we have to consider only the case in which $t \notin proposed$. In this case, DTI10(c) implies that $dreq[t]$ is different from $NoReq$ and TI6 implies that so is $q[t]$, a contradiction with the implication condition of TI2.

TI3 Easily verified by DTI10 and the action definition.

TI4 DTI11(a) implies that TI4(a) is preserved. TI4(b) is automatically preserved, and TI4(c) is easily verified by TI4(c) itself and the action definition. TI4(d) is also automatically preserved.

TI5 If $t \in proposed$, TI4(d) implies that $dreq[t] = NoReq$, contradicting the implication condition and preserving the invariant. If $t \notin proposed$, DTI10(b) and the action definition imply that $dcnt[d][t]$ is set to $Len(thist[t]) + 1$, also contradicting the implication condition and preserving the invariant.

TI6-7 Automatically preserved.

DTI1 The action sets $DB(d)!q[\langle t, vers[d][t]\rangle]$ to $NoReq$, preserving the invariant.

DTI2 $vers[d][t]$ is increased only if $DB(d)!tdec[\langle t, v\rangle]$ equals $Aborted$.

DTI3 Easily verified by DTI1 and the action definition since $DB(d)!q[\langle t, vers[d][t]\rangle]$ is set to $NoReq$.

DTI4 If $vers[d][t]$ is changed, DTI2 preserves DTI4. Otherwise, if $DB(d)!tdec[\langle t, vers[d][t]\rangle] = Committed$, $DB(d)!thist$ is not changed and the invariant is preserved.

DTI5 If $vers[d][t]$ is changed, then it is increased and $dcnt[d][t]$ is set to 0. In this case, invariant DTI12 preserves DTI5. If $vers[d][t]$ is not changed, there are to cases to analyze. If $t \notin proposed$ the invariant's implication condition is invalidated by DTI10(a,d); If $t \in proposed$, then DTI11(b) and the action definition preserve DTI5.

DTI6 Easily verified by DTI2 in case $vers[d][t]$ changes.

DTI7 Easily verified by the action definition.

DTI8-9 Easily verified in case $vers[d][t]$ changes by DTI1.

DTI10-11 The action sets $DB(d)!q[\langle t, vers[d][t]\rangle]$ to $NoReq$, invalidating these invariants' implication condition.

DTI12 By DTI12 and the fact that $vers[d][t]$ can only be increased.

Termination action changing $gdec[t]$ **from** *Unknown* **to a value in** *Decided*

Database, *tdec*, **and** *dreply* **Invariants** Automatically preserved.

TI1-2 Easily verified since this action can only change $tdec[t]$ to a value in *Decided*, invalidating these invariants' implication condition.

TI3 Automatically preserved.

TI4 Conditions (a) and (c-d) are automatically preserved. Condition (b) is easily verified since this action changes $tdec[t]$ to a value in *Decided*.

TI5,7 Automatically preserved.

TI6 Easily verified since this action can only change $tdec[t]$ to a value in *Decided*, invalidating the invariant's implication condition.

DTI1-7,9-12 Automatically preserved.

DTI8 Easily verified since this action can only change $tdec[t]$ to a value in *Decided*, invalidating the invariant's implication condition.

Termination action changing $learnedSeq[d]$ — Recall that this action can only extend $learnedSeq[d]$ by the Stability property of the termination protocol.

Database Invariant Easily verified since $learnedSeq[d]$ is only extended.

tdec **and** *dreply* **Invariants** Automatically preserved.

TI1-7 Automatically preserved.

DTI1-5,7-8,10-12 Automatically preserved.

DTI6,9 Easily verified since $learnedSeq[d]$ is only extended.

B Proof of Theorem 2

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.*

PROOF SKETCH: To understand why, consider the case with two active transactions t_1 and t_2 that write distinct database objects, x and y, respectively, and do not read anything. Transaction t_1 can execute on database d_1 and transaction t_2 can execute on database d_2. Both transactions are free to commit and can be proposed to the termination protocol. Executing either t_1 before t_2 or t_2 before t_1 leads to the same final state and they both can be committed in *gdec*. Assume, then, that database d_1 follows the ordering $\langle t_1, t_2 \rangle$ and executes and commits t_1 first. Database d_2 follows the ordering $\langle t_2, t_1 \rangle$, executing and committing t_2 first. At this point, if a passive transaction reads the whole state of database d_1, it will see the execution of t_1 but not the execution of t_2, which implies that t_1 must be serialized before t_2. If a passive transaction reads d_2, it will imply that t_2 must be serialized before t_1. Since passive transactions are free to execute completely at the databases responsible for them, all these transactions are free to commit locally and this scenario would break the global serializability.

C Proof of Theorem 3

Theorem 3 *The four properties Nontriviality, Stability, Consistency, and Liveness of our Termination Protocol specification satisfy the Nontriviality, Stability, Consistency, and Liveness properties of Sequence Agreement for transactions that commit where commands are transactions and learnedSeq implements learned.*

PROOF SKETCH: For any execution of the Termination Protocol, consider only the set of proposed transactions that eventually commit (*gdec*[t] is set to *Committed*) as the set of proposed transactions in an execution of Sequence Agreement. We show that all properties of Sequence Agreement are guaranteed in the following:

Nontriviality Guaranteed by Consistency and Nontriviality of Termination.

Stability Trivially guaranteed by Stability of Termination.

Consistency By the Consistency property of Termination, all *learnedSeq* sequences are prefixes of a common sequence *seq* of committed (proposed, for Sequence Agreement) transactions, which guarantees that, for every two of them, one is a prefix of the other.

Liveness By the Liveness property of Termination.

D TLA Specifications

D.1 Module *DatabaseConstants*

This module contains general database definitions.

MODULE *DatabaseConstants*

EXTENDS *Sequences*, *FiniteSets*, *Naturals*

The specification is based on the following constants:
- *Tid*: Set of transaction ids, where each id identifies a single transaction.
- *Op*: Set of possible transaction operations different from *Commit* or *Abort*.
- *Commit*, *Abort*: Special operations for committing/aborting a transaction.
- *Result*: Set of operation results.

- *Committed*, *Aborted*: Special results returned when a transaction is committed/aborted.
- *CorrectOp*(*op*, *res*, *dbstate*, *newdbstate*): Predicate that tells if operation *op*, when executed upon database state *dbstate*, may give *res* as a result and generate new database state *newdbstate*.
- *DBState*: Set of database states.
- *InitialDBState*: Initial database state.
- *FSeq*: A substitute for *Seq* – just a trap for the model checker.
- *Universe*: A set to bound unbounded CHOOSE statements – another trap for the model checker.

CONSTANTS Tid, Op, $Commit$, $Abort$, $Result$, $Committed$, $Aborted$, $CorrectOp(\_, \_, \_, \_)$,
$DBState$, $InitialDBState$, $FSeq(\_)$, $Universe$

We define *Unknown* as a transaction status in which the transaction has been neither committed nor aborted.

$Decided \triangleq \{Committed, Aborted\}$
$Unknown \triangleq \text{CHOOSE } v \in Universe : v \notin Decided$

Request is the set of all possible requests and *NoReq* is defined to be something that is not a (valid) request.

$Request \triangleq Op \cup \{Commit, Abort\}$
$NoReq \triangleq \text{CHOOSE } noreq \in Universe : noreq \notin Request$

Reply is the set of all possible replies for a request.

$Reply \triangleq Result \cup Decided$

Assumptions

The values used as transaction decisions (Committed and *Aborted*) must be different from operation results because we assume the decision is given as the response for operations issued after the transaction has been committed or aborted, so that the client is told that the operation was not performed because the transaction has been decided. If *Committed* or *Aborted* corresponds to a correct operation result, the client cannot tell if the operation executed or the transaction terminated.

ASSUME $Committed \notin Result$
ASSUME $Aborted \notin Result \cup \{Committed\}$

We must also assume that *Commit* and *Abort* requests are different and not present in *Op*.

ASSUME $Commit \notin Op$
ASSUME $Abort \notin Op \cup \{Commit\}$

InitialDBState must belong to *DBState*

ASSUME $InitialDBState \in DBState$

CorrectOp must be a correct predicate on *op*, *res*, *dbstate*, and *newdbstate*

ASSUME $\forall\, op \in Op, res \in Result,$
$dbstate \in DBState, newdbstate \in DBState :$
$CorrectOp(op, res, dbstate, newdbstate) \in \text{BOOLEAN}$

Auxiliar Expressions

OpRec represents a tuple in $Op \times Result$ as a record with two fields: *op* and *res*.

$OpRec \triangleq [op : Op, res : Result]$

THist is the set of all possible transaction histories.

$THist \triangleq FSeq(OpRec)$

THistVector is the set of all possible history vectors.

$$THistVector \triangleq [Tid \to THist]$$

CorrectAtomicHist verifies if the operations in transaction history h, when sequentially applied to the database state *initst*, can provide the same results they provided in h and generate the final database state *finalst*. It is defined as a recursive function that tests operation by operation, in order, with a simple tail recursion. *CorrectAtomicHist* is defined so that even nondeterministic operations are allowed. A single operation can provide nondeterministic results or change the database nondeterministically.

$$\begin{array}{l} CorrectAtomicHist[h \in THist,\ initst \in DBState,\ finalst \in DBState] \triangleq \\ \quad \text{IF } h = \langle\rangle \\ \qquad \text{THEN } initst = finalst \\ \qquad \text{ELSE } \exists\, ist \in DBState : \wedge\ CorrectOp(Head(h).op,\ Head(h).res,\ initst,\ ist) \\ \qquad\qquad\qquad\qquad\qquad\ \wedge\ CorrectAtomicHist[Tail(h),\ ist,\ finalst] \end{array}$$

$Perm(S)$ represents all sequences containing exactly one copy of each element in set S. It represents all the possible orderings of elements in S. The name *Perm* comes from permutations although a permutation is a function from S to S, and not a sequence derived from S. For want of a better name, we kept *Perm*.

$$\begin{array}{l} Perm(S) \triangleq \text{LET } N \triangleq Cardinality(S) \\ \qquad\qquad \text{IN } \quad \{s \in [1\,..\,N \to S] : \{s[i] : i \in 1\,..\,N\} = S\} \end{array}$$

CorrectSerialization verifies if sequence *seq* of transaction ids represents a correct serial execution of its transactions with respect to their histories in history vector *thist*, initial database state *initst*, and final database state *finalst*. It is defined as as a recursive function, like *CorrrectAtomicHist*, that verifies transaction by transaction with a simple tail recursion.

$$\begin{array}{l} CorrectSerialization[seq \in FSeq(Tid),\ thist \in THistVector,\ initst \in DBState,\ finalst \in DBState] \triangleq \\ \quad \text{IF } seq = \langle\rangle \\ \qquad \text{THEN } initst = finalst \\ \qquad \text{ELSE } \exists\, ist \in DBState : \wedge\ CorrectAtomicHist[thist[Head(seq)],\ initst,\ ist] \\ \qquad\qquad\qquad\qquad\qquad\ \wedge\ CorrectSerialization[Tail(seq),\ thist,\ ist,\ finalst] \end{array}$$

IsSerializable(S, *thist*, *initst*) verifies if set S can have a sequence containing each of its elements exactly once such that its execution is serializable with respect to history vector *thist* and initial database state *initst*.

$$\begin{array}{l} IsSerializable(S,\ thist,\ initst) \triangleq \\ \quad \exists\, seq \in Perm(S),\ db \in DBState : CorrectSerialization[seq,\ thist,\ initst,\ db] \end{array}$$

PassiveOp(op) is satisfied iff operation op is passive.

$$\begin{array}{l} PassiveOp(op) \triangleq \\ \quad \forall\, st1,\ st2 \in DBState,\ res \in Result : \\ \qquad CorrectOp(op,\ res,\ st1,\ st2) \Rightarrow st1 = st2 \end{array}$$

PassiveHist(h) is satisfied iff history h is passive.

$$\begin{array}{l} PassiveHist(h) \triangleq \\ \quad \forall\, st1,\ st2 \in DBState : \\ \qquad CorrectAtomicHist[h,\ st1,\ st2] \Rightarrow st1 = st2 \end{array}$$

D.2 Module *SerializableDB*

This module presents a TLA$^+$ version of our serializable database specification. It extends module *DatabaseInterface* that defines interface operators *DBRequest* and *DBResponse* in terms of an interface variable *DBinter*. Our specifications are practically oblivious to how these operators are defined as long as their definitions are disjoint. For the sake of simplicity, we do not present our specification of module *DatabaseInterface*.

MODULE *SerializableDB*

EXTENDS *DatabaseConstants*, *DBInterface*

VARIABLES *thist*, *tdec*, *q* and *DBinter* from *DBInterface*

$$thistType \triangleq THistVector$$
$$tdecType \triangleq [Tid \to Decided \cup \{Unknown\}]$$
$$qType \triangleq [Tid \to Request \cup \{NoReq\}]$$

Set of all committed transactions

$$committedSet \triangleq \{t \in Tid : tdec[t] = Committed\}$$

ReceiveReq(t, *req*) deals with the receipt of a transaction request. It stores the received request in $q[t]$ for it to be processed later.

$$\begin{aligned} ReceiveReq(t, req) \triangleq\ & \land DBRequest(t, req) \\ & \land q[t] = NoReq \\ & \land q' = [q \text{ EXCEPT } ![t] = req] \\ & \land \text{UNCHANGED } \langle thist, tdec\rangle \end{aligned}$$

ReplyReq(t, *rep*) deals with the response of an executing request. It checks whether transaction t has already been decided; if so, the response to t's executing request is its final decision. If t has not been decided yet, then the action is enabled only if *op* is in *Op* and *rep* is in *Result*. In such a case, the operation and its result are enqueued in t's history.

$$\begin{aligned} ReplyReq(t, rep) \triangleq\ & \land q[t] \in Request \\ & \land DBResponse(t, rep) \\ & \land q' = [q \text{ EXCEPT } ![t] = NoReq] \\ & \land \text{IF } tdec[t] \in Decided \\ & \quad \text{THEN} \land rep = tdec[t] \\ & \qquad\quad \land \text{UNCHANGED } \langle thist, tdec\rangle \\ & \quad \text{ELSE} \land q[t] \in Op \\ & \qquad\quad \land rep \in Result \\ & \qquad\quad \land thist' = [thist \text{ EXCEPT } ![t] = Append(@, [op \mapsto q[t], \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad res \mapsto rep])] \\ & \qquad\quad \land \text{UNCHANGED } \langle tdec\rangle \end{aligned}$$

Action *DoAbort*(t) aborts t by setting $tdec[t]$ to *Aborted*. This can be done at any time as long as t has not been decided yet.

$$\begin{aligned} DoAbort(t) \triangleq\ & \land tdec[t] \notin Decided \\ & \land tdec' = [tdec \text{ EXCEPT } ![t] = Aborted] \\ & \land \text{UNCHANGED } \langle thist, q, DBinter\rangle \end{aligned}$$

Action *DoCommit*(t) commits t by setting $tdec[t]$ to *Committed*, which is done only if t has not been decided and t's commit request has been issued.

$$\begin{aligned} DoCommit(t) \triangleq\ & \land tdec[t] \notin Decided \\ & \land q[t] = Commit \\ & \land tdec' = [tdec \text{ EXCEPT } ![t] = Committed] \\ & \land IsSerializable(committedSet', thist, InitialDBState) \\ & \land \text{UNCHANGED } \langle thist, q, DBinter\rangle \end{aligned}$$

Specification

Initialization.

$$
\begin{aligned}
Init \triangleq\ & \land\ InitInterface \\
& \land\ thist = [i \in Tid \mapsto \langle\rangle] \\
& \land\ tdec\ = [i \in Tid \mapsto Unknown] \\
& \land\ q = [t \in Tid \mapsto NoReq]
\end{aligned}
$$

Next defines the possible "next" steps in a correct execution.

$$
\begin{aligned}
Next \triangleq\ & \exists\, t \in Tid : \\
& \lor\ \lor \exists\, req \in Request : ReceiveReq(t,\ req) \\
& \quad \lor \exists\, rep \in Reply\ \ \ : ReplyReq(t,\ rep) \\
& \lor\ DoCommit(t) \\
& \lor\ DoAbort(t)
\end{aligned}
$$

Final specification.

$$Spec \triangleq Init \land \Box[Next]_{\langle thist,\ tdec,\ q,\ DBinter\rangle}$$

D.3 Module *OPSerializableDB*

This module presents our specification of an order-preserving serializable database.

MODULE *OPSerializableDB*

EXTENDS *SerializableDB*

serialSeq keeps the commit order

VARIABLES *serialSeq*

$$serialSeqType \triangleq \{s \in FSeq(Tid) : \forall\, i,\ j \in \text{DOMAIN } s : i \neq j \Rightarrow s[i] \neq s[j]\}$$

$$
\begin{aligned}
OPDoCommit(t) \triangleq\ & \land\ tdec[t] = Unknown \\
& \land\ q[t] = Commit \\
& \land\ tdec' = [tdec \text{ EXCEPT } ![t] = Committed] \\
& \land\ serialSeq' = Append(serialSeq,\ t) \\
& \land \exists\, st \in DBState : CorrectSerialization[serialSeq',\ thist,\ InitialDBState,\ st] \\
& \land \text{UNCHANGED } \langle thist,\ q,\ DBinter\rangle
\end{aligned}
$$

Specification

Initialization.

$$
\begin{aligned}
OPInit \triangleq\ & \land\ Init \\
& \land\ serialSeq = \langle\rangle
\end{aligned}
$$

Next defines the possible "next" steps in a correct execution.

$$
\begin{aligned}
OPNext \triangleq\ & \exists\, t \in Tid : \\
& \lor\ \land\ \lor \exists\, req \in Request : ReceiveReq(t,\ req) \\
& \qquad \lor \exists\, rep \in Reply\ \ \ : ReplyReq(t,\ rep) \\
& \qquad \lor\ DoAbort(t) \\
& \quad \land \text{UNCHANGED } serialSeq \\
& \lor\ OPDoCommit(t)
\end{aligned}
$$

Final specification.

$OPSpec \triangleq OPInit \land \Box[OPNext]_{\langle thist,\, tdec,\, q,\, DBinter,\, serialSeq\rangle}$

D.4 Module *AOPSerializableDB*

This module presents our specification of an active order-preserving serializable database.

MODULE *AOPSerializableDB*

EXTENDS *SerializableDB*

serialSeq keeps the commit order

VARIABLES *serialSeq*

$serialSeqType \triangleq \{s \in FSeq(Tid) : \forall\, i, j \in \text{DOMAIN } s : i \neq j \Rightarrow s[i] \neq s[j]\}$

Function *IsSubSeq* below verifies if *smallseq* is a subsequence of *bigseq*.

$$
\begin{array}{l}
IsSubSeq[smallseq \in FSeq(Tid),\, bigseq \in FSeq(Tid)] \triangleq \\
\quad (smallseq \neq \langle\rangle) \Rightarrow \\
\qquad \exists\, i \in 1 \,..\, Len(bigseq) : \\
\qquad\quad \land\, bigseq[i] = Head(smallseq) \\
\qquad\quad \land\, \forall\, j \in 1 \,..\, Len(bigseq) : \\
\qquad\qquad bigseq[j] = Head(smallseq) \Rightarrow j \geq i \\
\qquad\quad \land\, IsSubSeq[Tail(smallseq),\, SubSeq(bigseq,\, i+1,\, Len(bigseq))]
\end{array}
$$

$$
\begin{array}{l}
AOPDoCommit(t) \triangleq \land\, tdec[t] = Unknown \\
\qquad\qquad\qquad\quad \land\, q[t] = Commit \\
\qquad\qquad\qquad\quad \land\, tdec' = [tdec \text{ EXCEPT } ![t] = Committed] \\
\qquad\qquad\qquad\quad \land \text{ IF } PassiveHist(thist[t]) \\
\qquad\qquad\qquad\qquad\quad \text{THEN UNCHANGED } serialSeq \\
\qquad\qquad\qquad\qquad\quad \text{ELSE } serialSeq' = Append(serialSeq,\, t) \\
\qquad\qquad\qquad\quad \land \exists\, seq \in Perm(committedSet'),\, st \in DBState : \\
\qquad\qquad\qquad\qquad \land\, CorrectSerialization[seq,\, thist,\, InitialDBState,\, st] \\
\qquad\qquad\qquad\qquad \land\, IsSubSeq[serialSeq',\, seq] \\
\qquad\qquad\qquad\quad \land \text{ UNCHANGED } \langle thist,\, q,\, DBinter\rangle
\end{array}
$$

Specification

Initialization.

$$
\begin{array}{l}
AOPInit \triangleq \land\, Init \\
\qquad\qquad\quad \land\, serialSeq = \langle\rangle
\end{array}
$$

Next defines the possible "next" steps in a correct execution.

$$
\begin{array}{l}
AOPNext \triangleq \exists\, t \in Tid : \\
\qquad \lor\, \land\, \lor \exists\, req \in Request : ReceiveReq(t,\, req) \\
\qquad\qquad\quad \lor \exists\, rep \in Reply \quad : ReplyReq(t,\, rep) \\
\qquad\qquad\quad \lor\, DoAbort(t) \\
\qquad\qquad \land \text{ UNCHANGED } serialSeq \\
\qquad \lor\, AOPDoCommit(t)
\end{array}
$$

Final specification.

$AOPSpec \triangleq AOPInit \wedge \Box[AOPNext]_{\langle thist,\, tdec,\, q,\, DBinter,\, serialSeq\rangle}$

D.5 Module *GeneralDeferredUpdate*

This is the TLA$^+$ specification of our abstract deferred update algorithm.

MODULE *GeneralDeferredUpdate*

EXTENDS *DatabaseConstants*, *DBInterface*

CONSTANTS *Database*, $DBof(\_)$, $StripPassive(\_)$

VARIABLES *thist*, *q*, *dreq*, *pdec*, Client variables
dreply, *dcnt*, *vers*, *dcom*, Database variables
ldinter, *dthist*, *dtdec*, *dq*, *dserialSeq*, Internal database variables
proposed, *learnedSeq*, *gdec* Termination variables

ASSUME $\forall\, t \in Tid : DBof(t) \in Database$

ASSUME $\forall\, hist \in THist,\, st1,\, st2 \in DBState :$
$\wedge\ StripPassive(hist) \in THist$
$\wedge\ CorrectAtomicHist[hist,\, st1,\, st2] \Rightarrow CorrectAtomicHist[StripPassive(hist),\, st1,\, st2]$

Definition of *tdec* based on *pdec* e *gdec*

$$tdec \triangleq [t \in Tid \mapsto \text{IF } t \in proposed \text{ THEN } gdec[t] \text{ ELSE } pdec[t]]$$

Each database accepts transactions with ids in the form $\langle tid,\, version\rangle$ where *tid* is an element of *Tid* and version is a Natural. This allow "a single" transaction to be submitted to a database multiple times.

$LocalTid \triangleq Tid \times Nat$

The definition below instantiates each local database used by the general algorithm.

$$DBS(d) \triangleq \text{INSTANCE } AOPSerializableDB \text{ WITH } \begin{array}{ll} Tid & \leftarrow LocalTid, \\ DBinter & \leftarrow ldinter[d], \\ thist & \leftarrow dthist[d], \\ tdec & \leftarrow dtdec[d], \\ q & \leftarrow dq[d], \\ serialSeq & \leftarrow dserialSeq[d] \end{array}$$

NoRep is defined to be some value that is not a valid reply.

$NoRep \triangleq \text{CHOOSE } v : v \notin Reply$

The definition below creates an instance of the termination protocol specification.

$GT \triangleq \text{INSTANCE } GeneralTermination$

Auxiliary definitions to help dealing with the declared variables.

$cvars \triangleq \langle thist,\ q,\ dreq,\ pdec \rangle$
$ldvars \triangleq \langle ldinter,\ dthist,\ dtdec,\ dq,\ dserialSeq \rangle$
$gdvars \triangleq \langle dreply,\ dcnt,\ vers,\ dcom \rangle$
$dvars \triangleq \langle gdvars,\ ldvars \rangle$
$tvars \triangleq \langle proposed,\ learnedSeq,\ gdec \rangle$

ActHist(t) returns the current history of transaction t with some of its passive operations taken out of the sequence (according to operator *StripPassive*).

$ActHist(t) \triangleq StripPassive(thist[t])$

DBvars(d) returns the internal variables of database d.

$DBvars(d) \triangleq \langle ldinter[d],\ dthist[d],\ dtdec[d],\ dq[d],\ dserialSeq[d] \rangle$

OtherDBsStutter(d) is an action that forces all databases but d to execute a stuttering step, that is, a step in which their internal variables do not change values. For simplicity, our specification does not allow interleaving of database actions. In fact, as we explain in the following, it does not allow interleaving at all.

$OtherDBsStutter(d) \triangleq$
  LET $dbfn \triangleq [nd \in (Database \setminus \{d\}) \mapsto DBvars(nd)]$
  IN $dbfn' = dbfn$

Here are the atomic actions of the general deferred update technique, not including the internal database actions and the internal actions of the termination protocol. In order to model check this specification, we had to make it noninterleaving, that is, we had to specify it in terms of actions that cannot occur concurrently (even considering that they are executed by different specification components). This prevented us from using the *DBRequest* and *DBResponse* primitives to interact with the internal databases. Instead, we used the *ReceiveReq* and *ReplyReq* actions directly to submit an operation and get a response from a database.

The *ReceiveReq* action as explained in the paper.

$ReceiveReq(t,\ req) \triangleq$
  $\land\ DBRequest(t,\ req)$
  $\land\ q[t] \notin Request$
  $\land\ q' = [q \text{ EXCEPT } ![t] = req]$
  $\land$ IF $tdec[t] \notin Decided$
    THEN $\lor\ \land\ req = Commit$
            $\land\ GT!Propose(t)$
            $\land$ UNCHANGED $\langle thist,\ dreq,\ pdec,\ dvars \rangle$
         $\lor\ \land\ req = Commit \Rightarrow PassiveHist(thist[t])$
            $\land\ dreq' = [dreq \text{ EXCEPT } ![t] = req]$
            $\land$ UNCHANGED $\langle thist,\ pdec,\ dvars,\ tvars \rangle$
    ELSE UNCHANGED $\langle thist,\ dreq,\ pdec,\ dvars,\ tvars \rangle$

The *ReplyReq* action.

$ReplyReq(t,\ rep) \triangleq$
  $\land\ q[t] \in Request$
  $\land\ DBResponse(t,\ rep)$
  $\land\ q' = [q \text{ EXCEPT } ![t] = NoReq]$
  $\land$ IF $tdec[t] \in Decided$
    THEN $\land\ rep = tdec[t]$

$\land$ UNCHANGED $\langle thist,\ dreq,\ pdec,\ dvars,\ tvars\rangle$

ELSE $\land\ q[t] \in Op$
$\land\ rep \in Result$
$\land\ dcnt[DBof(t)][t] > Len(thist[t])$
$\land\ rep = dreply[t]$
$\land\ thist' = [thist$ EXCEPT $![t] = Append(@, [op \mapsto q[t],$
$res \mapsto rep])]$
$\land\ dreq' = [dreq$ EXCEPT $![t] = NoReq]$
$\land$ UNCHANGED $\langle pdec,\ dvars,\ tvars\rangle$

The *PrematureAbort* action.

$PrematureAbort(t) \triangleq$ $\land\ t \notin proposed$
$\land\ pdec[t] \notin Decided$
$\land\ pdec' = [pdec$ EXCEPT $![t] = Aborted]$
$\land$ UNCHANGED $\langle thist,\ q,\ dreq,\ dvars,\ tvars,\ DBinter\rangle$

The *PassiveCommit* action.

$PassiveCommit(t) \triangleq$ $\land\ t \notin proposed$
$\land\ pdec[t] \notin Decided$
$\land\ dreply[t] = Committed$
$\land\ pdec' = [pdec$ EXCEPT $![t] = Committed]$
$\land$ UNCHANGED $\langle thist,\ q,\ dreq,\ dvars,\ tvars,\ DBinter\rangle$

The *DBReq* action with its three enabling conditions.

$DBReq(d,\ t,\ req) \triangleq$
$\land$ $\lor\ \land\ d = DBof(t)$ Condition 1
$\land\ dreq[t] = req$
$\land\ dcnt[d][t] = Len(thist[t])$
$\lor\ \land\ t \in proposed$ Condition 2
$\land\ dcnt[d][t] < Len(ActHist(t))$
$\land\ req = ActHist(t)[dcnt[d][t] + 1].op$
$\lor\ \land\ req = Commit$ Condition 3
$\land\ \exists\, i \in 1\,..\,Len(learnedSeq[d]) :$
$\land\ learnedSeq[d][i] = t$
$\land\ \forall\, j \in 1\,..\,i : dcom[d][learnedSeq[d][j]]$
$\land\ \lor\ d = DBof(t) \land vers[d][t] = 0$
$\lor\ dcnt[d][t] = Len(ActHist(t))$
$\land$ $DBS(d)!ReceiveReq(\langle t,\ vers[d][t]\rangle,\ req)$
$\land$ $OtherDBsStutter(d)$
$\land$ UNCHANGED $\langle cvars,\ gdvars,\ tvars,\ DBinter\rangle$

The *DBRep* action.

$DBRep(d,\ t,\ rep) \triangleq$
$\land$ $DBS(d)!ReplyReq(\langle t,\ vers[d][t]\rangle,\ rep)$
$\land$ $OtherDBsStutter(d)$
$\land$ IF $d = DBof(t)$ THEN $dreply' = [dreply$ EXCEPT $![t] = rep]$
ELSE UNCHANGED $dreply$
$\land$ IF $rep = Aborted \land t \in proposed$
THEN $\land\ vers' = [vers$ EXCEPT $![d][t] = @ + 1]$

$\land\ dcnt' = [dcnt \text{ EXCEPT } ![d][t] = 0]$
$\land$ UNCHANGED $dcom$
ELSE $\land\ dcnt' = [dcnt \text{ EXCEPT } ![d][t] = @ + 1]$
$\land\ dcom' = [dcom \text{ EXCEPT } ![d][t] = (rep = Committed)]$
$\land$ UNCHANGED $vers$
$\land$ UNCHANGED $\langle cvars, tvars, DBinter \rangle$

Initialization.

$Init \triangleq \land InitInterface$
$\land\ q = [t \in Tid \mapsto NoReq]$
$\land\ dreq = [t \in Tid \mapsto NoReq]$
$\land\ dreply = [t \in Tid \mapsto NoRep]$
$\land\ pdec = [t \in Tid \mapsto Unknown]$
$\land\ vers = [d \in Database \mapsto [t \in Tid \mapsto 0]]$
$\land\ dcom = [d \in Database \mapsto [t \in Tid \mapsto \text{FALSE}]]$
$\land\ dcnt = [d \in Database \mapsto [t \in Tid \mapsto 0]]$
$\land \forall\, d \in Database : DBS(d)!AOPInit$
$\land\ GT!Init$ includes *thist*

The next-state action in terms of noninterleaving actions.

$Next \triangleq \lor \exists\, t \in Tid : \lor \exists\, req \in Request : ReceiveReq(t, req)$
$\lor \exists\, rep \in Reply : ReplyReq(t, rep)$
$\lor PrematureAbort(t)$
$\lor PassiveCommit(t)$
$\lor \exists\, d \in Database : \lor \exists\, t \in Tid : \lor \exists\, req \in Request : DBReq(d, t, req)$
$\lor \exists\, rep \in Reply : DBRep(d, t, rep)$
$\lor\ \land$ UNCHANGED $ldinter[d] \land DBS(d)!AOPNext$
$\land\ OtherDBsStutter(d)$
$\land$ UNCHANGED $\langle cvars, gdvars, tvars, DBinter \rangle$
$\lor\ \land$ UNCHANGED $\langle cvars, dvars, DBinter \rangle$
$\land\ GT!TNext$

The final specification, including the liveness condition of the termination protocol.

$Spec \triangleq Init \land \Box[Next]_{\langle cvars, dvars, tvars, DBinter \rangle} \land GT!Liveness$

D.6 Module *GeneralTermination*

This module presents our specification of the Termination Protocol.

MODULE *GeneralTermination*

EXTENDS *DatabaseConstants*

CONSTANTS *Database*

VARIABLES *proposed*, *learnedSeq*, *gdec*, *thist*

pdec stands for premature/passive decision (client decision)
gdec stands for global decision (for global transactions)

$vars \triangleq \langle proposed,\ learnedSeq,\ gdec \rangle$

$committedSet \triangleq \{t \in Tid : gdec[t] = Committed\}$

IsPrefix(*smallseq*, *bigseq*) verifies if *smallseq* is a prefix of *bigseq*.

$IsPrefix(smallseq,\ bigseq) \triangleq \exists\, n \in 0\,..\,Len(bigseq) : smallseq = SubSeq(bigseq,\ 1,\ n)$

The consistency property in TLA+ The other properties are automatically guaranteed by the atomic actions below.

$Consistency \triangleq$
$\quad \exists\, seq \in Perm(committedSet),\ st \in DBState :$
$\qquad \land\ CorrectSerialization[seq,\ thist,\ InitialDBState,\ st]$
$\qquad \land\ \forall\, d \in Database : IsPrefix(learnedSeq[d],\ seq)$

Propose(t) proposes a transaction t for termination.

$Propose(t) \triangleq$
$\quad \land\ t \notin proposed$
$\quad \land\ proposed' = proposed \cup \{t\}$
$\quad \land$ UNCHANGED $\langle learnedSeq,\ gdec \rangle$

Decide(t) makes a final decision (Committed or *Aborted*) about proposed transaction t.

$Decide(t) \triangleq$
$\quad \land\ t \in proposed$
$\quad \land\ gdec[t] = Unknown$
$\quad \land\ \exists\, v \in Decided : gdec' = [gdec$ EXCEPT $![t] = v]$
$\quad \land$ UNCHANGED $\langle proposed,\ learnedSeq \rangle$
$\quad \land\ Consistency'$

Learn(d, *seq*) extends *learnedSeq*[d], but only if the new value ensures consistency.

$Learn(d,\ seq) \triangleq$
$\quad \land\ \ IsPrefix(learnedSeq[d],\ seq) \land Len(seq) > Len(learnedSeq[d])$
$\quad \land\ \ learnedSeq' = [learnedSeq$ EXCEPT $![d] = seq]$
$\quad \land\ \ $ UNCHANGED $\langle proposed,\ gdec \rangle$
$\quad \land\ \ Consistency'$

The following two definitions have to do with our weak liveness requirement for termination.

$LivenessDatabase(t,\ d) \triangleq$
$\quad gdec[t] = Aborted \ \Rightarrow$
$\qquad \Diamond(t \in \{learnedSeq[d][i] : i \in$ DOMAIN $learnedSeq[d]\})$

$Liveness \triangleq$
$\quad \Box(\forall\, t \in Tid,\ d \in Database : LivenessDatabase(t,\ d))$

The following action simply helps the model checking. It changes the transactions' history vector.

$ChangeHist(t) \triangleq$
$\quad \land\ t \notin proposed$
$\quad \land\ \exists\, o \in Op,\ r \in Result :$
$\qquad thist' = [thist$ EXCEPT $![t] = Append(@,\ [op \mapsto o,$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad res \mapsto r])]$
$\quad \land$ UNCHANGED $vars$

TNext allows any action but $Propose(v)$. It is used in the specification of our general deferred update protocol.

$$
\begin{array}{l}
TNext \triangleq \\
\quad \lor \exists\, t \in Tid : Decide(t) \\
\quad \lor \exists\, d \in Database,\ seq \in FSeq(Tid) : Learn(d,\ seq)
\end{array}
$$

Next allows all the actions and is used by to model check termination in an isolated way.

$$
\begin{array}{l}
Next \triangleq \\
\quad \lor\ TNext \land \text{UNCHANGED}\ thist \\
\quad \lor \exists\, t \in Tid :\ \lor Propose(t) \land \text{UNCHANGED}\ thist \\
\qquad\qquad\qquad\ \lor ChangeHist(t)
\end{array}
$$

Initialization.

$$
\begin{array}{l}
Init \triangleq \\
\quad \land proposed \quad = \{\} \\
\quad \land learnedSeq = [d \in Database \mapsto \langle\rangle] \\
\quad \land gdec \qquad = [t \in Tid \mapsto Unknown] \\
\quad \land thist \qquad = [t \in Tid \mapsto \langle\rangle]
\end{array}
$$

Final specification.

$$Spec \triangleq Init \land \Box[Next]_{\langle vars,\ thist\rangle} \land Liveness$$