A Formal Model for the Deferred Update Replication Technique

Andrea Corradini[1], Leila Ribeiro[2], Fernando Dotti[3], and Odorico Mendizabal[3]

[1] Universitá di Pisa, Dipartimento di Informatica
Pisa, Italy
andrea@di.unipi.it
[2] Universidade Federal do Rio Grande do Sul, Instituto de Informática,
Porto Alegre, Brazil,
leila@inf.ufrgs.br
[3] Pontifícia Universidade Católica do Rio Grande do Sul, Faculdade de Informática,
Porto Alegre, Brazil,
fernando.dotti@pucrs.br; omendizabal@gmail.com

Abstract. Database replication is a technique employed to enhance both performance and availability of database systems. The Deferred Update Replication (DUR) technique offers strong consistency (i.e. serializability) and uses an optimistic concurrency control with a lazy replication strategy relying on atomic broadcast communication. Due to its good performance, DUR has been used in the construction of several database replication protocols and is often chosen as a basic technique for several extensions considering modern environments. The correctness of the DUR technique, i.e. if DUR accepted histories are serializable, has been discussed by different authors in the literature. However, a more comprehensive discussion involving the completeness of DUR w.r.t. serializability was lacking. As a first contribution, this paper provides an operational semantics of the DUR technique which serves as foundation to reason about DUR and its derivatives. Second, using this model the correctness of DUR w.r.t. serializability is shown. Finally, we discuss the completeness of DUR w.r.t. serializability and show that for any serializable history there is an equivalent history accepted by DUR. Moreover, we show that transactions aborted by DUR could not be accepted without changing the order of already committed transactions.

1 Introduction

Since several decades, database management systems are of paramount importance to safely keep users data. A database system is typically manipulated by concurrent transactions from several users. The common correctness criterion used to validate consistency in databases is *serializability* (or *strong consistency*) [3,11]. Roughly, an interleaved execution of several concurrent transactions is *serializable* if it has the same effect on a database as some serial execution of these transactions. The preservation of database consistency is one key aspect which is usually ensured by schedulers responsible for implementation of concurrency control mechanisms. While early approaches to assuring database integrity where based on some type of locking, hindering concurrency and thus transaction throughput, optimistic methods for concurrency control emerged

to take advantage both from the low conflict rate among transactions and the hardware architectures allowing better performance [9]. According to such methods, a transaction executes under the optimistic assumption that no other transaction will conflict with it. At the end of the execution, a validation phase takes place using read and write sets of the transactions to decide commit or to invalidate the transaction [4].

Further seeking to enhance performance, replication techniques for database systems have been thoroughly studied in past years [3]. Several replication techniques emerged in the database community which can be classified according to two basic characteristics: where updates take place (primary copy vs. update anywhere) and kind of interaction for update synchronization (eager vs. lazy) [7]. While database replication is aimed primarily at performance, in distributed systems replication has as major concern high availability. Replication approaches for distributed systems were developed in parallel such as the primary-backup [5] and state machine replication [15] approaches.

The search for highly available and high performance databases leads to consider the combination of replication and optimistic concurrency control. Since with optimistic concurrency control transactions progress independently and are validated at the end of the execution, and since in a distributed setting communication costs and delays are to be avoided, a natural configuration to consider is the primary copy approach with lazy uptate. The Deferred Update Replication (DUR) approach [12] uses these ideas with multiple primary copies. According to DUR, a group of servers fully replicate the database while clients choose any server to submit a transaction. Transaction processing at the server takes place without coordinating with other servers. Upon transaction termination issued by the client, a certification test is performed to assure database consistency. The finalization protocol is based on atomic broadcast to submit the modifications of a local transaction to all other servers in total delivery order. Each server will receive the same finalization requests in the same order, apply the same certification tests, leading to same sequence of coherent modifications in each server.

The DUR technique is currently being extended in different ways, such as to support byzantine faults, to enhance throughput of update transactions and to support in-memory transaction execution [13,17,16]. The correctness of the basic technique, i.e. if DUR accepted histories are serializable, is discussed by different authors [6,8,12,13]. In [14] the authors use TLA$^+$ to validate serializability of DUR based protocols. A similar approach is adopted by [1], where the authors present a formal specification and correctness proof for replicated database systems using I/O automata.

In this paper we contribute to the analysis of the DUR technique in three main aspects: (i) We provide an operational semantics of the DUR technique, which serves as a foundation to discuss its correctness and completeness, as well as solid contribution for future extensions to represent variations of DUR as mentioned before; (ii) Based on this model the correctness of DUR is shown w.r.t. serializability; (iii) Furthermore we show the completeness of DUR w.r.t. serializable histories: for any serializable history there is an equivalent history accepted by DUR; moreover it is shown that if a transaction would be aborted by the execution of DUR, then the history obtained by including this transaction in a history recasting exactly the execution of the algorithm up to this point would not be serializable in the strict sense, that is, without changing the order of already committed transactions. While (ii) is, to some extent, closely related to previous

works, contributions (i) and (iii) bring new elements to the discussion of DUR and the theory of concurrency control.

This paper is structured as follows: in Section 2 we present several concepts from [3] on serializability theory; in Section 3 we present the DUR technique based on [17]; in Section 4 the operational semantics of DUR is presented; in Section 5 both correctness and completeness of DUR as mentioned above are discussed; finally in Section 6 we review the results achieved and discuss possible directions for future work.

2 Serializability Theory

In this section we review some of the main definitions of serializability theory [3].

Definition 1 (Database actions). *Given a set of variables X, a **database action** may be $\mathbf{r}[\mathbf{x}]$, a $\mathbf{read}$ of variable $x \in X$; $\mathbf{w}[\mathbf{x}]$, a $\mathbf{write}$ on variable $x \in X$; $\mathbf{c}$, a commit; or $\mathbf{a}$, an abort. The set of actions over variables in X will be denoted by $\mathbf{Act}(X)$. We say that two actions are **conflicting** is they operate on the same variable and at least one of them is a $\mathbf{write}$.*

Definition 2 (Transaction). *A **transaction** is a pair $\langle T, \leq_T \rangle$ where $T : \mathbf{Act}(X) \to \mathbb{N}$ is a multiset of actions, mapping each action to the number of its occurrences, and $\leq_T \subseteq T \times T$ is a finite partial order relation than satisfies*

1. *either $\mathbf{c} \in T$ or $\mathbf{a} \in T$;*
2. *T has at least one $\mathbf{read}$ or $\mathbf{write}$ action;*
3. *$\mathbf{c}$ and $\mathbf{a}$ must be maximal wrt $\leq_T$;*
4. *if $\mathbf{w}[\mathbf{x}] \in T$ then for any other action $\mathbf{o}[\mathbf{x}]$ that reads or writes variable x, either $\mathbf{o}[\mathbf{x}] \leq_T \mathbf{w}[\mathbf{x}]$ or $\mathbf{w}[\mathbf{x}] \leq_T \mathbf{o}[\mathbf{x}]$*

We denote often transaction $\langle T, \leq_T \rangle$ simply by T, leaving the partial order understood. Here and in the following we often confuse the multiset T with its extension $\{\mathbf{a}^{\mathbf{i}}_{\mathbf{T}}[\mathbf{x}] \mid a \in \{r, w\}, x \in X, 0 < i \leq T(\mathbf{a}[\mathbf{x}])\}$. We denote by $vars(T)$ the set of variables used in T, namely $vars(T) = \{x \in X \mid T(\mathbf{r}[\mathbf{x}]) + T(\mathbf{w}[\mathbf{x}]) > 0\}$.

We also introduce the following auxiliary notations:

- *$T[x] = T \cap \{\mathbf{r}^{\mathbf{i}}_{\mathbf{T}}[\mathbf{x}], \mathbf{w}^{\mathbf{i}}_{\mathbf{T}}[\mathbf{x}]\}$ is the subset of actions of T involving variable x;*
- *$\leq^x_T = \leq_T \cap (T[x] \times T[x])$;*

We consider three subsets of $vars(T)$.

*The set of **local variables** is defined as $\mathbf{loc}(T) = \{x \mid \exists i \,.\, \mathbf{w}^{\mathbf{i}}_{\mathbf{T}}[\mathbf{x}] = min(\leq^x_T)\}$. Intuitively, if the first action over x in T is a write (often called a "blind write"), then x is handled as a local variable and its value before T is irrelevant. By the condition on $\leq_T$, if the minimum of $\leq^x_T$ is a $\mathbf{write}$ then it is unique.*

*The **readset** of T is defined as $\mathbf{rs}(T) = \{\, x \mid \exists i \,.\, \mathbf{r}^{\mathbf{i}}_{\mathbf{T}}[\mathbf{x}] \in min(\leq^x_T)\}$. Thus the readset of T consists of all the variables of the transaction whose original value was read outside T. Note that $\mathbf{rs}(T)$ and $\mathbf{loc}(T)$ form a partition of $vars(T)$.*

*The **writeset** of T is defined as $\mathbf{ws}(T) = \{x \mid T(\mathbf{w}[\mathbf{x}]) > 0\}$, i.e. the set of variables modified by T. A transaction with an empty writeset is called a **read-only transaction**; we shall use the predicate $\mathbf{ro}$ defined as $\mathbf{ro}(T) \equiv (\mathbf{ws}(T) = \varnothing)$. Note that $\mathbf{loc}(T) \subseteq \mathbf{ws}(T)$.*

Histories represent concurrent executions of transactions. Actions of different transactions can never be the same, but they may act on shared data (variables). Therefore, a history must carry information about the order in which conflicting actions shall occur. Moreover, it may define also relationships among other (non-conflicting) actions.

Definition 3 (History). *Given a set of transactions $S = \{T_1, \ldots, T_n\}$, a **complete history** over S is a partial order $\langle H, \leq \rangle$ where*

1. $H = \bigcup_{i=1}^{n} T_i$;
2. $\leq \supseteq \bigcup_{i=1}^{n} \leq_i$;
3. *for any conflicting actions o_1 and o_2 in H, either $o_1 \leq o_2$ or $o_2 \leq o_1$.*

*A **history** is a prefix of a complete history. A transaction T_i is **committed/aborted** in a history H if $\mathbf{c_{T_i}} \in H$ / $\mathbf{a_{T_i}} \in H$. If a transaction is neither committed nor aborted, it is **active** in a history. The **committed projection** of a history H is denoted by $C(H)$ and is obtained removing from H all actions that belong to active or aborted transactions. The **non-aborted projection** of a history H is denoted by $NA(H)$ and is obtained removing from H all actions of aborted transactions.*

*The set of variables used (written or read) is a history H is denoted by $vars(H)$. Given a history $(H, \leq_H)$, the **induced dependency relation** on transactions $\leq_H^T$ is defined as: $T_1 \leq_H^T T_2$ if there are actions $o_1 \in T_1$ and $o_2 \in T_2$ such that $o_1 \leq_H o_2$. A history is **strict** if whenever there are actions $\mathbf{w_{T_j}[x]} \leq_H \mathbf{o_{T_i}[x]}$, with $T_i, T_j \in H$ and $i \neq j$, either $\mathbf{a_{T_j}} \leq_H \mathbf{o_{T_i}[x]}$ or $\mathbf{c_{T_j}} \leq_H \mathbf{o_{T_i}[x]}$.*

Strictness implies that no data item may be read by a transaction if another transaction that is writing to it has not terminated. This is commonly required in applications since histories that are not strict allow serializations that are rather counterintuitive (that would "undo the past"), and also implies recoverability and avoids cascading aborts.

The following equivalence notion on histories is known as conflict-equivalence, since it is based on the compatibility between conflicting items of histories. In this paper we will stick to this kind of equivalence.

Definition 4 (History Equivalence). *Given two histories $(H_1, \leq_{H_1})$ and $(H_2, \leq_{H_2})$, we say that they are **equivalent** if*

1. *$H_1 = H_2$ (they have the same actions);*
2. *they order conflicting actions of non-aborted transactions in the same way: for all $o_1, o_2 \in NA(H_1)$ such that o_1 and o_2 are conflicting, $o_1 \leq_{H_1} o_2$ if and only if $o_1 \leq_{H_2} o_2$.*

Note that this definition of equivalence poses no restriction on the dependencies of non-conflicting actions.

Definition 5 (Serial History). *A complete history $(H, \leq_H)$ is **serial** if for every two transactions $T_i, T_j \in H$, either for all $o_i \in T_i$ we have $end_j \leq_H o_i$, or for all $o_j \in T_j$ it holds $end_i \leq_H o_j$, where end_j (end_i) is the commit or abort action of T_j (T_i).*

Note that the induced dependency relation on transactions of a serial history is a total order, even if $\leq_H$ does not need to be total.

Definition 6 (Serializable History). *A history $(H, \leq_H)$ is (conflict) **serializable** if there is a serial history $(H_s, \leq_{Hs})$ that is equivalent to the committed projection $C(H)$.*

For a proof of the following theorem, see [3].

Theorem 1. *Let $(H, \leq_H)$ be a history and $G(H)$ be the graph whose nodes are the transactions of H and arrows denote the relationship between the conflicting actions in $\leq_H$, lifted to the corresponding transactions. A history is serializable iff $G(H)$ is acyclic.*

3 The Deferred Update Replication

The Deferred Update Replication (DUR) technique coordinates transaction processing on a group of servers that fully replicate the database. It provides fault-tolerance due to replication while offering good performance. Clients submit transactions to any server. Servers execute transactions locally, without coordination with other servers, in a so-called execution phase. The concurrency control strategy is optimistic. When the client requests the transaction's commit, it broadcasts its local updates to all servers which then have to certify that the transaction can be serialized with other commited transactions that executed concurrently. Note that the termination phase uses an atomic broadcast protocol that ensures that all servers receive the same termination messages in the same order. Since all execute the same certification procedures, they decide homogeneously, accepting (committing and updating the local states) or not (aborting) the transaction, and thus progress over same (replicated) states. The good performance is achieved since: update transactions progress independently in each server during the execution phase; read-only transactions can be certified locally; communication is restricted to the dissemination of updates for certification at the end of the transaction (lazy approach) which implies a lower overhead if compared to propagation of updates during the transaction (eager approach).

Since in the following sections we provide an operational semantics for DUR, we adopt and explain here the main DUR algorithms at client and server sides from [17]. Each transaction t has a unique identifier id, a readset rs keeping the variables read by t; a writeset ws keeping both the variables and the values updated by t; and a snapshot id st that records the snapshot of the database when t started reading values.

According to the client algorithm of [17] (see Fig. 1, left), after a transaction t is initiated (lines 1 to 4), the client may request a read, a write or a commit operation. For brevity, it is not shown in the algorithm the case in which the client executes an abort operation. A read operation by transaction t on variable k will add k to $t.rs$. If k belongs to $t.ws$ it means that k has been previously written and the read operation returns the local value (lines 7, 8). Otherwise the value has to be read from one server s from the set of servers S (lines 10, 11). If this was the first read of this transaction, the snapshot id returned by the server is stored in $t.st$. A write operation simply adds locally to $t.ws$ the value written on the variable. A commit of a read-only transaction is decided locally. In case of an update transaction, a request for certification of transaction t is atomically broadcast to all servers and an answer from one of the servers is awaited.

The transaction certification decision is based on the readset of the transaction and on the writesets of the already committed concurrent transactions.

Notice that the algorithm does not distinguish between local and non-local variables: when a variable is read it is added to the readset even if it was initialized with a blind write.

DUR, Client c's code

```
 1: begin(t):
 2: t.rs ← ∅
 3: t.ws ← ∅
 4: t.st ← ⊥
 5: read(t, k):
 6: t.rs ← t.rs ∪ k
 7: if (k, ∗) ∈ t.ws then
 8:     return v s.t. (k, v) ∈ t.ws
 9: else
10:     send(read, k, t.st) to some s ∈ S
11:     wait until receive(k, v, st) from s
12:     if t.st = ⊥ then t.st = st
13:     return v
14: write(t, k, v):
15: t.ws ← t.ws ∪ k, v
16: commit(t):
17: if t.ws = ∅ then
18:     return commit
19: else
20:     abcast(c, t)
21:     wait until receive(outcome) from s
22:     return outcome
```

DUR, Server s's code

```
 1: initialization:
 2: SC ← 0
 3: WS[..] ← ∅
 4: when receive(read, k, st) from c
 5: if st = ⊥ then st ← SC
 6: retrieve(k, v, st) from database
 7: send(k, v, st) to c
 8: when adeliver(c, t)
 9: outcome ← certify(t)
10: if outcome = commit then
11:     apply t.ws to database
12: send(outcome) to c
13: function certify(t)
14: for i = t.st to SC do
15:     if WS[i] ∩ t.rs ≠ ∅ then
16:         return abort
17: SC ← SC + 1
18: WS[SC] ← items(t.ws)
19: return commit
```

Fig. 1. DUR Algorithms

According to the server algorithm, a server has a snapshot counter SC recording the number of committed transactions (line 17) and a writeset for each snapshot $WS[i]$ that records the writeset of the transaction committed at snapshot i. When a read request arrives, if it is the first read of a transaction (line 5) then the server assigns to the transaction's snapshot id st the current value of the snapshot counter, that the client will keep for future reads. Then the server retrieves from the database the most recent value v for variable k before or at the same snapshot identified by st (line 6). When client c requests a commit (using the atomic broadcast primitive $adeliver$ - line 8) all servers will run the certification test (line 9) and in case the outcome is $commit$ the writeset is used to update the database (lines 10, 11). Any outcome is sent to client c (line 12). The certification of t verifies if any committed transaction concurrent with t (line 14) updated variables of the readset of t (line 15). In such case t has to be aborted. Otherwise t is committed, a new snapshot is generated at the server (line 17) and the server keeps track of the variables updated in the last commit (line 18).

As one can observe, the certification of a transaction t depends only on whether the values read ($t.rs$) are valid upon termination, i.e. if no committed concurrent transaction has written on a value after it has been read by t.

Certification of read-only transactions is straightforward since all read values are consistent with the snapshot st of the first read, assuring that no update happened on the variables until after st - the transaction is considered to happen atomically at st, reading a consistent state even if snapshot st is in the past.

We can observe that since updates are deferred to the moment of termination, and actually updates and commit of one transaction are performed atomically, no updates and commits of different transactions interleave. The database progresses over states where updates are from a single transaction.

4 The DUR algorithm, formally

We present here a formalization of the behaviour of the server, as described in the previous section, using a transition system. This will be exploited for a proof of correctness and completeness of the **DUR Server** algorithm.

We assume that there is a fixed set of transactions $\mathcal{T}$ which includes all transactions that will ever interact with the server. Since at most a finite number of transactions can be terminated in each state of the server's evolution, to make the formalization easier we assume $\mathcal{T}$ to be finite. Notice that the readset and the writeset of a transaction are built in the client's code, and are used by the server only when the transaction is completed. Therefore we consider them as statically known, and denote them as $\mathbf{rs}(T)$ and $\mathbf{ws}(T)$, respectively. Note however that $\mathbf{rs}(T)$ denotes the readset as defined in Def. 2, while the readset of a transaction according to the algorithm of Fig. 1 also includes the local variables, and could be denoted as $\mathbf{rs}(T) \cup \mathbf{loc}(T)$: we will discuss later the consequences of this assumption.

Instead the snapshot id, identifying the snapshot of the database that a transaction accesses, is assigned dynamically by the server at the first read operation. Therefore the state of a server includes, besides the snapshot counter SC and a vector of committed writesets WS (see lines 2-3 of the algorithm), a function ST defined on $\mathcal{T}$ returning the snapshot id, if defined, and $\perp$ otherwise. The state also includes an additional function on transactions, the *commit index* CI, which is defined only on terminated transactions and records whether the transaction is aborted or not, and in case it is not read-only the index of its committed writeset in vector WS.

The components of a well-formed server state have to satisfy several constraints listed in the next definition, most of which are pretty obvious: as shown in Thm. 2, well-formedness will guarantee reachability. We just stress that the commit index of a read-only transaction is set, quite arbitrarily, to its snapshot id plus 0.5. In this way it is smaller than the commit index of any transaction that could modify the variables in the readset after being accessed.

Definition 7 (server state). *A **server state** over a finite set of transactions $\mathcal{T}$ is a four-tuple $D = \langle SC, WS, ST, CI \rangle$, where*

- *SC is an integer, the* snapshot counter*;*

- *WS is a vector of (committed) writesets $WS : \{0, \ldots, SC\} \to \mathcal{P}(X)$;*
- *$ST : \mathcal{T} \to \{0, \ldots, SC\} \cup \{\bot\}$ maps each transaction in S to its* snapshot id*, i.e. an integer $ST(T) \leq SC$ which is the index of the writeset from which the first value of T is read, if any, and $\bot$ otherwise;*
- *$CI : \mathcal{T} \to \{0.5, 1, 1.5, \ldots, SC, SC + 0.5\} \cup \{\bot, abort\}$ maps each transaction to its* commit index*. A transaction T is* aborted *if $CI(T) = abort$; it is* active *if $CI(T) = \bot$ and $ST(T) \neq \bot$; and it is* committed *if $CI(T) \in \mathbb{Z}$. We shall denote by $Comm(D)$ the set of transactions committed in D, and by $RO(D)$ its subset of read-only transactions.*

We will denote by $\bot$ the constant function mapping all transactions to $\bot$.
*A server state is **well-formed** if the following conditions are satisfied:*

1. *$WS(0) = X$, that is, the first writeset of WS includes all variables (it represents their initial values);*
2. *every other writeset in WS corresponds to exactly one committed, non-read-only transaction; formally, CI restricts to a bijection $Comm(D) \setminus RO(D) \leftrightarrow [1, SC]$, and $\forall T \in Comm(D) \setminus RO(D)\,.\, WS(CI(T)) = \mathbf{ws}(T)$;*
3. *if the snapshot id of a transaction is defined then its readset is not empty: $ST(T) \neq \bot \Rightarrow \mathbf{rs}(T) \neq \varnothing$;*
4. *an aborted transaction has a defined snapshot id and its readset has been modified by a committed transaction: $CI(T) = abort \Rightarrow ST(T) \neq \bot \wedge (\exists i \in [ST(T)+1, SC]\,.\, WS(i) \cap \mathbf{rs}(T) \neq \varnothing)$;*
5. *the snapshot id of a non-read-only committed transaction, if defined, is less than the commit index, and in this case its readset was not modified before committing: $\neg\mathbf{ro}(T) \wedge CI(T) \notin \{abort, \bot\} \wedge ST(T) \neq \bot \Rightarrow$ $0 \leq ST(T) < CI(T) \wedge (\forall i \in [ST(T)+1, CI(T)-1]\,.\, WS(i) \cap \mathbf{rs}(T) = \varnothing)$;*
6. *a read-only transaction cannot abort, and if committed its commit index is one half more than its snapshot id: $\mathbf{ro}(T) \Rightarrow$ $(CI(T) \neq abort) \wedge (CI(T) \neq \bot \Rightarrow ST(T) \neq \bot \wedge CI(T) = ST(T) + 0.5)$.*

The behaviour of the server can be represented as a transition systems where transitions are triggered by the interactions with the clients. Every client, while executing a transaction $T \in \mathcal{T}$, interacts with the server to read the values of the variables in its readset and to deliver the values of the variables in its writeset upon completion. From the server's side, as described in algorithm **DUR Server** (Fig. 1, right), this corresponds to receiving a sequence of $receive$ (briefly rec) requests, followed by one $adeliver$ ($adel$) request, which depending on the situation can cause the transaction to commit or to abort. The first rec request is handled in a special way, as it will fix the snapshot of the data repository which is relevant for transaction T.

For our goals, the concrete values of the variables are irrelevant, as it is the name of the variable read with a rec request, assuming that it belongs to the readset of the transaction. We will therefore disregard this information. assuming that rec requests will have only the transaction issuing the request as argument, exactly as the $adel$ request.

Definition 8 (server as transition system). *A server $\mathcal{S}$ over a set of transactions $\mathcal{T}$ is a transition system having as states the well-formed server states of Def. 7, as initial*

state the state $D_0 = \langle SC_0 = 0, WS_0 = [0 \mapsto X], ST_0 = \bot, CI_0 = \bot]\rangle$*, and where transitions are generated by the following inference rules (for the sake of readability, the components of states that are not changed in a rule are represented by an undescore):*

$$\textit{[read-}\bot\textit{]}\ \frac{ST(T) = \bot}{\langle \_, \_, ST, \_\rangle \xrightarrow{rec(T)} \langle \_, \_, ST[T \mapsto SC], \_\rangle}$$

$$\textit{[read]}\ \frac{ST(T) \neq \bot}{\langle \_, \_, \_, \_\rangle \xrightarrow{rec(T)} \langle \_, \_, \_, \_\rangle}$$

[commit]

$$\frac{CI(T) = \bot, \neg\mathbf{ro}(T), \neg\,(\exists x \in \mathbf{rs}(T), i \in \{ST(T)+1, \ldots, SC\}\,.\,x \in WS(i))}{\langle SC,\, WS, \_,\, CI\rangle \xrightarrow{adel(T)} \langle SC+1,\, WS[SC+1 \mapsto \mathbf{ws}(T)], \_,\, CI[T \mapsto SC+1]\rangle}$$

$$\textit{[commit-RO]}\ \frac{CI(T) = \bot, \mathbf{ro}(T)}{\langle \_, \_, \_, CI\rangle \xrightarrow{adel(T)} \langle \_, \_, \_, CI[T \mapsto ST(T) + 0.5]\rangle}$$

$$\textit{[abort]}\ \frac{CI(T) = \bot, \neg\mathbf{ro}(T), (\exists x \in \mathbf{rs}(T), i \in \{ST(T)+1, \ldots, SC\}\,.\,x \in WS(i))}{\langle \_, \_, \_, CI\rangle \xrightarrow{adel(T)} \langle \_, \_, \_, CI[T \mapsto abort]\rangle}$$

Rules *[read-⊥]* and *[read]* encode lines 4-7 of algorithm **DUR Server** (Fig. 1): since we abstract from variables values, and variable names are recorded by functions $\mathbf{rs}$ and $\mathbf{ws}$, the only visible effect in the server state is the assignment of a snapshot id to a transaction, if missing. Rules *[commit]* and *[abort]* encode lines 8-19 of the algorithm. Note that in the premises of these rules we used the set $\mathbf{rs}(T)$ instead of the larger set $\mathbf{rs}(T) \cup \mathbf{loc}(T)$, as used in the algorithm in Fig. 1. This means that our model has actually less abort transitions than the algorithm would have, and we will show in Thm. 4 that our definition characterises exactly the histories that should be aborted because would lead to non-serializable histories. Rule *[commit-RO]* records the completion of a read-only transaction, that has no visible effect for the server in the DUR algorithm, but is necessary in our encoding to keep function CI up to date.

Note that rules are well defined, even if $ST(T)$ might be undefined. The snapshot id of T is undefined if and only if the execution of T never generates a $rec(T)$ transition, i.e. if the readset $\mathbf{rs}(T)$ is empty. In this case, T is not read-only (because transactions without any action are forbidden by Def. 2) and thus the second premise of rule *[commit]* is vacuously satisfied, while the premise of the *[abort]* rule is vacuously false.

We shall often represent transitions by labeling them with both the request of the client (over the arrow) and with the applied rule (under), as in $D \xrightarrow[\textit{[read-}\bot\textit{]}]{rec(T)} D'$. Furthermore, we write $D \Rightarrow D'$ if there is a transition from D to D' using the rules of Def. 8.

To conclude this section, let us show that the well-formedness of a server state guarantees its reachability. The lenghty proof is in the appendix.

Theorem 2 (well-formed server states are reachable). *A server state over a set of transactions $\mathcal{T}$ is reachable if and only if it is well-formed.*

5 Correctness and Completeness of Deferred Update Replication

We show now that the server as previously specified guarantees the serializability of the transactions that commit. This is a pretty straigthforward correctness result. More interestingly, thanks to the rigorous formalization we are also able to prove that the server is "complete", in the sense that it never happens that a transaction aborts if it was serializable.

Since by Thm. 2 all and only the well-formed states are reachable in an execution of the server, the correctness of the server can be proved by showing that in any well-formed state, the dependencies among committed transactions that are recorded in the state are compatible with a history including them *only if* the history is serializable. In other words, a non-serializable history could not be executed by the server.

Therefore let us define when a complete history containing a set of committed transactions is consistent with a well-formed state of the server.

Definition 9 (history-state consistency). *Let $D = \langle SC, WS, ST, CI\rangle$ be a well-formed server state over a set of transactions $\mathcal{T}$, and $\langle H, \leq_H\rangle$ be a complete history. Then H and D are **consistent** if*

1. *$\langle H, \leq_H\rangle$ is a history over the transactions of $\mathcal{T}$ which committed in D ($Comm(D)$);*
2. *for each pair $T \neq T' \in Comm(D)$, for each pair of conflicting actions $\mathbf{a} \in T$ and $\mathbf{b} \in T'$, we have:*
   (a) $\mathbf{a} <_H \mathbf{b}$ implies $CI(T) < CI(T')$;
   (b) if $x \in \mathbf{rs}(T')$ and $\mathbf{b} = \mathbf{r_{T'}[x]}$ (and thus $\mathbf{a} = \mathbf{w_T[x]}$), then $\mathbf{w_T[x]} <_H \mathbf{r_{T'}[x]}$ if and only if $CI(T) \leq ST(T')$.

Condition *2(a)* states that the causality among conflicting actions belonging to distinct transactions in H is consistent with the commit ordering of transactions. Condition *2(b)* guarantees that the history correctly records the values read by a transaction for the variables in its readset, imposing that such values are those available at the database snapshot $ST(T)$. In fact, each read action in the readset must depend on all and only the write actions for the same variable in transactions that committed not later than the snapshot id. Note that since all pairs of conflicting events have to be causally related in a history, it follows that $\mathbf{r_{T'}[x]} <_H \mathbf{w_T[x]}$ if and only if $ST(T') < CI(T)$. In this case, since by *2(a)* we also know that $CI(T') < CI(T)$, we can conclude that $ST(T') \leq CI(T') < CI(T)$, as $ST(T') \leq CI(T')$ because D is well-formed.

The next result states the correctness of DUR algorithm.

Proposition 1 (consistent histories are serializable). *Let $\langle H, \leq\rangle$ be a complete history consistent with a well-formed server state $D = \langle SC, WS, ST, CI\rangle$. Then H is serializable.*

Proof. Let $\sqsubseteq'_D$ be the commit ordering on $Comm(D)$, i.e. $T \sqsubseteq'_D T'$ if $CI(T) \leq CI(T')$; it is a partial order because two read-only transactions may have the same commit index. Let $\sqsubseteq_D$ be any total order compatible with $\sqsubseteq'_D$, ordering such read-only transactions in an arbitrary way. Then by condition *2(a)* above for each pair of conflicting actions $\mathbf{a_T[x]} \leq_H \mathbf{b_{T'}[x]}$ in H with $T \neq T'$, we have $T \sqsubseteq_D T'$. Therefore H is serializable, because it is equivalent to a serial history where all actions of a transaction T are caused by the commit action of a transaction T' if and only if $T' \sqsubseteq_D T$. □

Viceversa, completeness can be proved by showing that any serializable history is consistent with a well-formed server state.

Theorem 3 (Serializable histories and consistent states). *Let H be a complete serializable history without aborted transactions. Then there is a well-formed server state D that is consistent with H.*

Proof. The proof is by induction on the number of committed transactions in H.

For the base case, the initial state of Def. 8 is clearly consistent with the empty history since there are no aborted/active transactions in H.

Now suppose we have a complete serializable history H_{n+1} with $n+1$ transactions. Let H_n be obtained by removing one transaction, say T, that is maximal with respect to the transaction order induced by $\leq_{H+1}$. H_n is a complete serializable history because H_{n+1} is. By induction hypothesis there is a well-formed server state $D_n = \langle SC, WS, ST, CI\rangle$ that is compatible with H_n. A well-formed server state D_{n+1} can be obtained in the following way. If T is not read-only and $\mathbf{rs}(T) \neq \varnothing$

$$D_{n+1} = \langle SC+1, WS[T \mapsto \mathbf{ws}(T)], ST[T \mapsto i], CI[T \mapsto SC+1]\rangle$$

where $i \in [SC_U, SC]$ and SC_U is the last snapshot in which some variable in $\mathbf{rs}(T)$ was updated ($SC_U = max\{CI(T_i) \mid WS(T_i) \cap \mathbf{rs}(T) \neq \varnothing\}$). This state clearly satisfies all conditions of Def. 7. If the transaction does not read any value, ST maps T to $\perp$, and also in this case all conditions are satisfied. If T is read-only, we define the server state as

$$D_{n+1} = \langle SC, WS, ST[T \mapsto i], CI[T \mapsto i+0.5]\rangle$$

Again, this state satisfies all conditions of Def. 7. In particular, in this case T must read some value by point 2 of Def. 2. □

To conclude, let us exploit the proposed formalization to show that the server causes a transaction to abort only when allowing it to commit would result in a non-serializable history. Interestingly, this property would not hold if in the premise of rule *[abort]* of Def. 8 we would have used as readset $\mathbf{rs}(T) \cup \mathbf{loc}(T)$ instead of $\mathbf{rs}(T)$.

For a given server state, it is possible to define a history corresponding to the execution of the transactions within this state. This history contains dependencies that enforce an order on the committed transactions according to the CI order, and otherwise would relate conflicting events of active/committed transactions according to the way they are related in ST and CI. Note that there can not be any dependency between active transactions because such transactions only have read actions in D (all write actions of a transaction occur together with the commit).

Definition 10 (Execution history). *Let D be a well-formed server state over a set of transactions $\mathcal{T}$ without aborted transactions. Then we define the* ***execution history consistent with*** *D, denoted by $execHist(D)$, as $\langle H, \leq_H\rangle$ where*

- *H contains all actions of committed transactions of D, and only the minimal read actions from variables in $\mathbf{rs}(T)$ of active transactions of D;*

– $\leq_H$ *is the transitive closure of the relation containing all dependencies of transactions in* $\mathcal{T}$ *plus the pairs (we consider* CI *and* ST *whenever they are defined):*
$\langle \mathbf{c_{T_i}}, \mathbf{c_{T_j}} \rangle$*, if* $CI(T_i) < CI(T_j)$*;*
$\langle \mathbf{c_{T_i}}, \mathbf{o_{T_j}}[\mathbf{x}] \rangle$*, if* $CI(T_i) < CI(T_j)$ *and* $o_{T_j}[x]$ *conflicts with an action* $w_{T_i}[x]$*;*
$\langle \mathbf{w_{T_i}}[\mathbf{x}], \mathbf{r_{T_j}}[\mathbf{x}] \rangle$ *if* $CI(T_i) \leq ST(T_j)$*; and*
$\langle \mathbf{r_{T_i}}[\mathbf{x}], \mathbf{w_{T_j}}[\mathbf{x}] \rangle$ *if* $ST(T_i) < CI(T_j)$.

The conditions on well-formed states (Def. 7) assure that $\leq_H$ is a partial order. By construction, if D has no active/aborted transactions, $execHist(D)$ is strict. The following theorem states that if a transaction T will be aborted at server state D, then the corresponding history, that is, the history that contains all committed and active transactions until that moment plus the writeset and commit of T would not be serializable.

Theorem 4 (abort is necessary). *Given a well-formed server state* D *without aborted transactions, the corresponding execution history* $execHist(D) = \langle EH, \leq_{EH} \rangle$ *and an active transaction* T *from* D*. Let* $\langle H, \leq_H \rangle$ *be defined as*

– $H = EH \cup T$*;*
– $\leq_H$ *is the transitive closure of the relation containing* $\leq_{EH}$*,* $\leq_T$ *plus the pairs* $\langle \mathbf{c_i}, \mathbf{o_T} \rangle$*, if an action of transaction* T *conflicts with some action of transaction* $T_i \in EH$.

If rule [abort] is enabled for a transaction T *then* H *is not serializable.*

Proof. If rule *[abort]* is enabled then T is not read-only and its read-set is not empty. Moreover, there is at least one transaction that committed in D, say at snapshot $i \leq SC$, that updated a variable, say by action $\mathbf{w_i}[\mathbf{x}]$, that was read by T with an action $\mathbf{r_T}[\mathbf{x}]$ and $TS(T) < i$. This means that actions $\mathbf{w_i}[\mathbf{x}]$ and $\mathbf{r_T}[\mathbf{x}]$ are in EH and $\mathbf{r_T}[\mathbf{x}] \leq_{EH} \mathbf{w_i}[\mathbf{x}]$ by Def. 10. By definition of H, we must have that $\mathbf{c_i} \leq_H \mathbf{r_T}[\mathbf{x}]$ and thus $\mathbf{w_i}[\mathbf{x}] \leq_H \mathbf{r_T}[\mathbf{x}]$ (because all actions of a transaction are related to the commit of the transaction and $\leq_H$ is transitive). Therefore, since $\leq_H$ includes $\leq_{EH}$, $\leq_H$ induces a cycle $T \leq_H^T Ti$ and $Ti \leq_H^T T$ and is therefore not serializable (actually $\leq_H$ is not even a history, since $\leq_H$ is not a partial order). □

The proof of the last result is crucially based on the fact that each variable x in the readset of T has its initial value set by an action $\mathbf{r_T}[\mathbf{x}]$: if this value is overwritten by a concurrent transaction that commits before T, then T has to abort because its addition to the current history would cause a cycle of dependencies. If in the precondition of rule *[abort]* we would have used $\mathbf{rs}(T) \cup \mathbf{loc}(T)$ (as in the algorithm of Fig. 1) instead of $\mathbf{rs}(T)$, the result would not hold. In fact, if the variable overwritten by a concurrent transaction that commits before T is a local one, i.e. it is initialized in T by a blind write $\mathbf{w_T}[\mathbf{x}]$, then the corresponding action $\mathbf{r_T}[\mathbf{x}]$ would not belong to the execution history EH, and in the resulting history H it would be larger than any conflicting event in EH, giving rise to a serializable history. Thus in this situation the algorithm of Fig. 1 would cause an unnecessary failure of the transaction, that is avoided in our model thanks to a careful definition of readset.

6 Discussion

In this paper we have analyzed the Deferred Update Replication (DUR) technique, providing a formal model as a transition system that described the behaviour of the algorithm. In the construction of this transition system we used a slightly more permissive premise for committing transactions than the algorithm presented in [17], allowing transactions to commit even if some update was performed in some of its read variables, as long as the first action on this variable in the committing transaction was a write (thus, the variable was considered to be local). We showed that for this model all reachable states correspond to serialisable histories involving the corresponding transactions. Moreover, we showed that for all serialisable histories, it is possible that the DUR algorithm generates an execution containing all these transactions. But note that, given a server state D and a transaction T that is trying to commit, if the algorithm suggest the abortion, this does not mean that there is no serialisable history containing all committed transaction plus T, what it means is that it is not possible to find a serialisation without changing the order of some already committed transaction. This was stated as a theorem relating abort transitions and strict histories.

Besides being used to show the correctness and completeness of DUR, the formal model can be used as a basis to reason about other extensions of DUR that have been proposed recently in the literature, for example, considering replicated database partitions to enhance overall throughput [17], byzantine fault tolerance [13], and in-memory transaction execution [16]. Such extensions are very much attractive to modern computational environments (cloud computing; open systems and untrusted parties; modern architectures) and a formal analysis involving correctness and completeness is highly desired. More than the directly related family of DUR protocols, the contribution is relevant to many other existing systems using replication and optimistic concurrency control, a frequent combination.

Another interesting line of research would be to check to which extent the theory of concurrency can be applied in this setting. The serialisability theory was very well-studied mainly in the 80s and 90s, and to a great extent results are based on very basic definitions of switching transactions to obtain equivalence notions over histories. To handle more complex scenarios, like the ones arising e.g. from unreliable systems, cloud computing or adaptive systems, it might be necessary to reason using more abstract notions of histories and equivalences. The use of concurrency models explicitly handling causality like event structures [10] or asymmetric event structures [2] may allow to reason about database updates in such settings.

References

1. Armendáriz-Iñigo, J.E., González, D.M., Garitagoitia, J.R., Muñoz-escoí, Francesc, D.: Correctness proof of a database replication protocol under the perspective of the I/O automaton model. Acta Informatica 46(4) (2009)
2. Baldan, P., Corradini, A., Montanari, U.: Contextual petri nets, asymmetric event structures and processes. Information and Computation 171(1), 1–49 (2001)
3. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)

4. Bhargava, B.: Concurrency control in database systems. IEEE Transactions on knowledge and Data Engineering pp. 3–16 (1999)
5. Budhiraja, N., Marzullo, K., Schneider, F.B., Toueg, S.: The primary-backup approach. Distributed systems 2, 199–216 (1993)
6. Garcia, R., Rodrigues, R., Preguiça, N.: Efficient middleware for byzantine fault tolerant database replication. In: 6th Conference on Computer systems. EuroSys '11, ACM (2011)
7. Gray, J., Helland, P.: The dangers of replication and a solution. In: In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data. pp. 173–182 (1996)
8. Kemme, B., Alonso, G.: A new approach to developing and implementing eager database replication protocols. ACM Trans. Database Syst. 25(3) (Sep 2000)
9. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Transactions on Database Systems (TODS) 6(2), 213–226 (1981)
10. Nielsen, M., Plotkin, G., Winskel, G.: Petri nets, event structures and domains, part I. Theoretical Computer Science 13(1), 85 – 108 (1981)
11. Papadimitriou, C.H.: The serializability of concurrent database updates. Journal of the ACM (JACM) 26(4), 631–653 (1979)
12. Pedone, F., Guerraoui, R., Schiper, A.: Transaction Reordering in Replicated Databases. In: 16th IEEE Symposium on Reliable Distributed Systems (1997)
13. Pedone, F., Schiper, N.: Byzantine fault-tolerant deferred update replication. Journal of the Brazilian Computer Society 18 (2012)
14. Schmidt, R., Pedone, F.: A formal analysis of the deferred update technique. In: 11th International Conference on Principles of distributed systems. OPODIS'07, Springer-Verlag, Berlin, Heidelberg (2007)
15. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys (CSUR) 22(4), 299–319 (1990)
16. Sciascia, D., Pedone, F.: RAM-DUR: In-memory deferred update replication. In: Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on. pp. 81–90. IEEE (2012)
17. Sciascia, D., Pedone, F., Junqueira, F.: Scalable deferred update replication. In: Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on. pp. 1–12. IEEE (2012)

A Appendix

We present here the proof of Thm. 2.

Theorem 2 (well-formed server states are reachable). *A server state over a set of transactions $\mathcal{T}$ is reachable if and only if it is well-formed.*

Proof. **Only if part** The fact that every server state reachable from the initial state D_0 of Def. 8 is well-formed can be proved easily by checking for all the inference rules of Def. 8 that the target state of a transition is well-formed if so is the source state and if the application conditions are satisfied.

If part Let $D = \langle SC_D, WS_D, ST_D, CI_D \rangle$ be a well-formed server state over $\mathcal{T}$. We proceed by induction on the cardinality of $Comm(D)$.

If $|Comm(D)| = 0$, no transaction of $\mathcal{T}$ committed, and therefore we must have $SC_D = 0$, $WS_D = [0 \mapsto X]$, and $CI_D = \perp$, as no transaction could have aborted either, by condition 4 of Def. 7. Furthermore, $ST_D(T) \in \{\perp, 0\}$ for all $T \in \mathcal{T}$, i.e. some transactions may already have 0 as snapshot id. Let us show that D is reachable, i.e. $D_0 \Rightarrow^* D$ where D_0 is as in Def. 8. In fact, if $\{T_i\}_{1 \leq i \leq k} = \{T \mid ST_D(T) = 0\}$,

by condition 3 of Def. 7 we know that $\mathbf{rs}(T_i)$ is not empty for $1 \leq i \leq k$, and thus there exists a sequence of transitions $D_0 \xrightarrow[\textit{[read-}\bot\textit{]}]{rec(T_1)} D_0^1 \cdots D_0^{k-1} \xrightarrow[\textit{[read-}\bot\textit{]}]{rec(T_k)} D_0^k = D$, where for all $i \in [1,k]$ it holds $ST_{D_0^i}(T) = 0 \iff T \in \{T_1, \ldots, T_i\}$.

Suppose now that $|Comm(D)| = n + 1$. We first show that, without loss of generality, we may assume that no transaction aborted yet in D. In fact, if $\{T_i\}_{1 \leq i \leq k} = \{T \mid CI_D(T) = \mathit{aborted}\}$, then D is reachable from a state D' where those transactions are still active (i.e. $CI_{D'}(T_i) = \bot$), with a sequence of k *[abort]* transitions, one for each element of $\{T_i\}_{1 \leq i \leq k}$. The preconditions of such *[abort]* transitions are satisfied by condition 4 of Def. 7.

Now, assuming that D has no aborted transactions, let T be one of the transactions in $Comm(D)$ with maximal commit index. We have two cases: either T is read-only or not.

If T is read-only, by conditions 6 and 2 of Def. 7 we have $CI_D(T) = ST_D(T) + 0.5 = SC + 0.5$. Consider the server state $D' = \langle SC, WS, ST', CI' \rangle$ where

$$ST'(x) = \begin{cases} ST(x) & \text{if } x \neq T \\ \bot & \text{if } x = T \end{cases} \qquad CI'(x) = \begin{cases} CI(x) & \text{if } x \neq T \\ \bot & \text{if } x = T \end{cases}$$

State D' represents a snapshot of the system where all transactions but T are as in state D, while T did not start yet (its snapshot id is $\bot$). It is easily shown that D' is well-formed, therefore by inductive hypothesis D' is reachable from D_0.

It remains to show that D is reachable from D' by accepting all the requests generated by the execution of T, i.e. $D' \xrightarrow[\textit{[read-}\bot\textit{]}]{rec(T)} D'' \xrightarrow[\textit{[read]}]{rec(T)} D'' \cdots D'' \xrightarrow[\textit{[commit-RO]}]{adel(T)} D$; in fact the first transition sets $ST(T)$ to SC, and the last one sets $CI(T)$ to $SC + 0.5$.

If T is not read-only, by condition 2 of Def. 7 we have that $CI_D(T) = SC_D$. Let us additionally assume that $ST_D(T) = CI_D(T) - 1$. The idea, as in the case just seen, is to remove T from D obtaining a state D' with less committed transactions. But if there are transactions with $ST_D(T') = SC_D = CI_D(T)$, the resulting state would not be well-formed because $ST_D(T') > SC_{D'} = SC_D - 1$.

Therefore let us consider state D' obtained from D by setting $ST_{D'}(T) = \bot$ for all transactions in $\{T\}_{1 \leq i \leq k} = \{T \mid ST_D(T) = CI_D\}$. We clearly have $D' \Rightarrow^* D$ with a sequence of transitions $D' \xrightarrow[\textit{[read-}\bot\textit{]}]{rec(T_1)} D'_1 \cdots D'_{k-1} \xrightarrow[\textit{[read-}\bot\textit{]}]{rec(T_k)} D'_k = D$, which are possible by condition 3 of Def. 7.

Consider now the server state $D'' = \langle SC'', WS'', ST'', CI'' \rangle$ where

$$SC'' = SC' - 1, \qquad WS''(x) = \begin{cases} WS'(x) & \text{if } x \neq SC' \\ \bot & \text{if } x = SC' \end{cases}$$

$$ST''(x) = \begin{cases} ST'(x) & \text{if } x \neq T \\ \bot & \text{if } x = T \end{cases} \qquad CI''(x) = \begin{cases} CI'(x) & \text{if } x \neq T \\ \bot & \text{if } x = T \end{cases}$$

State D'' is the server state before transaction T has started, and it is easily shown to be well-formed. Therefore by induction hypothesis D'' is reachable from D_0. To show that $D'' \Rightarrow^* D'$, we consider two cases, depending on the readset of T.

1. $\mathbf{rs}(T) = \varnothing$: In this case, the premise of *[commit]* is satisfied because T is not read-only, thus $D'' \xrightarrow[\textit{[commit]}]{adel(T)} \hat{D}$. The resulting state is

$$\hat{D} = \langle SC'' + 1,\, WS''[SC'' + 1 \mapsto \mathbf{ws}(T)],\, ST'',\, CI''[T \mapsto SC'' + 1]\rangle$$

and using $SC'' = SC' - 1$, $CI'(T) = SC'$, $\mathbf{rs}(T) = \varnothing$ we conclude that

$$\hat{D} = \langle SC',\, WS''[CI'(T) \mapsto \mathbf{ws}(T)],\, ST',\, CI''[T \mapsto SC']\rangle$$

and thus $D' = \hat{D}$ is reachable.

2. $\mathbf{rs}(T) \neq \varnothing$: Here, analogously to the case of read-only transactions, we may start by an application of rule *[read-$\perp$]* followed by some applications of rule *[read]* until all variables in $\mathbf{rs}(T)$ are read, leading to state $\hat{D}$. Since state D was well-formed, it is easy to check that rule *[commit]* is enabled for T in $\hat{D}$, and that its application yields state D'.

It remains to consider the last case, where the transaction with highest commit index in D, say T, is not read-only and where $ST_D(T) < CI_D(T) - 1$. We argue as follows. Let D' be exactly like D, but with $ST_{D'}(T) = CI_D(T) - 1$. By the argument just presented we know that D' is reachable from D_0, i.e. there is a sequence of transitions $D_0 \Rightarrow D_1 \cdots D_{n-1} \Rightarrow D_n = D'$. In this sequence, the transition $\cdot \xrightarrow[\textit{[read-$\perp$]}]{rec(T)} \cdot$, that sets the value of $ST(T)$, must occurr after transition $\cdot \xrightarrow[\textit{[commit]}]{adel(T')} \cdot$, which sets $CI(T') = CI(T) - 1$. Between the two transitions, there could be other *[read]*, *[read-$\perp$]* and *[commit-RO]* transitions only. Now, it is easy to show that $\cdot \xrightarrow[\textit{[read-$\perp$]}]{rec(T)} \cdot$ can be anticipated by switching it with all these transitions, without affecting the well-formedness of the states and without changing the final state. Finally, when we have the consecutive transitions $\cdot \xrightarrow[\textit{[commit]}]{adel(T')} \cdot \xrightarrow[\textit{[read-$\perp$]}]{rec(T)} \cdot$, we can switch them by obtaining $\cdot \xrightarrow[\textit{[read-$\perp$]}]{rec(T)} \cdot \xrightarrow[\textit{[commit]}]{adel(T')} \cdot$. This is possible, again, because the well-formedness of state D ensures that $\mathbf{ws}(T') \cap \mathbf{rs}(T) = \varnothing$. In the resulting final state only the value of $ST(T)$ is changed, and it is $CI(T') = CI(T) - 2$. By iterating this transformation of the sequence of transitions we can show that the original state D is reachable. □