Mechanical Verification of Transaction Processing Systems

Dmitri Chkliaev[†] Jozef Hooman[‡] Peter van der Stok[†]

[†]Dept. of Computing Science
Eindhoven University of Technology
The Netherlands
e-mail: {dmitri,wsstok}@win.tue.nl

[‡]Computing Science Institute
University of Nijmegen
The Netherlands
e-mail: hooman@cs.kun.nl

Abstract

This paper concerns the formal specification and mechanical verification of transaction processing systems aimed at distributed databases. In such systems, a standard set of ACID properties must be ensured by a combination of concurrency control and recovery protocols. In the existing literature, these protocols are often studied in isolation, making strong assumptions about each other. The problem of combining them in a formal way is largely ignored. To study the formal verification of combined protocols, we specify a transaction processing system, integrating strict two-phase locking, undo/redo recovery and two-phase commit. In our method, the locking and undo/redo mechanism at distributed sites is defined by state machines, whereas the interaction between sites according to the two-phase commit protocol is specified by assertions. We proved with the interactive proof checker of PVS that our system satisfies atomicity, durability and serializability properties.

1. Introduction

A *transaction* is a logically indivisible sequence of actions performed on a database. Transaction processing systems (TPSs) are often very complex, because they have to deal with potential errors arising from two sources: concurrency and failures. To prevent such errors, a distributed TPS must support at least the following mechanisms: concurrency control, centralized recovery at a single site and distributed recovery.

When transactions interleave their access to the database, they can interfere. The interleaving of transactions is represented in a schedule, which is a sequence of actions such as reads and writes of data items, where each action belongs to some transaction. *Concurrency control* protocols must prevent interference by ensuring *serializability*. They should only accept a schedule if it is equivalent to some serial schedule. A serial schedule has no interleaving between actions of different transactions.

Centralized recovery protocols deal with two types of failures: transaction failures and memory failures. When a transaction is unable to finish its execution because of any reason (i.e. deadlock, concurrency control conflict), it is *aborted*, and the recovery protocol must erase its partial effects. It must also ensure that the results of *committed* transactions, i.e. those that complete successfully, are never lost. These two tasks are complicated by memory failures, which may erase portions of memory and also force some transactions to abort.

When a transaction updates data at multiple sites (called *participants* of this transaction), they all must agree on whether to commit or abort this transaction. The interaction between sites needed to reach a decision must be provided by *distributed recovery* protocols, usually in the form of *atomic (distributed) commitment* protocols.

The complete set of requirements for TPSs is usually summarized as ACID (Atomicity, Consistency, Isolation and Durability) [GA93].

Transaction processing systems provide the most important services of many modern applications (such as banking) and are often used in safety-critical applications. Therefore it is very important to guarantee their correctness. One method to increase the confidence in the correctness of TPSs is their formal verification. This, however, is quite challenging, because we have to prove a number of very different properties for a combination of several distributed fault-tolerant protocols. To study the verification of TPSs, we select a particular protocol as a case study, define a formal framework for its specification and verification and choose tool support. Therefore, in the remainder of this introduction we address the following questions: 1) What are the correctness properties for TPSs? 2) Which previous work has been done on their formal verification, and what are the weaknesses of this work? 3) What protocol is chosen for this particular case study, and why? 4) How to obtain mechanical support for its specification and verification? 5) What are the results of our verification efforts?

1) ACID properties. A TPS should satisfy the following four ACID properties:

- *Atomicity* means that the effect of a transaction is reflected in a database either completely or not at all, depending on whether the transaction is committed or aborted.
- *Consistency* requires some invariants (often called "integrity constraints") to be preserved by each individual transaction in each state of a database.
- *Isolation* means that different transactions do not interfere with each other. It is usually replaced by the more precise notion of serializability.
- *Durability* means that the effect of committed transactions must survive subsequent system failures. This property is closely related to atomicity, and they are usually analyzed together.

2) Previous work. Below we give our comments on some interesting papers related to the verification of TPSs, grouped by the property they concentrate on:

- [Spe99] contains interesting recent work on the verification of *consistency*, presenting the semi-automatic verification of integrity constraints with support of the Isabelle theorem prover. We don't consider the verification of consistency here, because we want to abstract from the details of database implementation.
- In [CHvdS99] we study the verification of *serializability*. A method to verify conflict serializability has been formulated in the verification system PVS and proved to be sound and complete with the interactive proof checker of this tool. The method has been used to verify the Two-Phase Locking protocol and the Timestamp Ordering protocol. We have defined a systematic way to extend these protocols with new actions and control information such that serializability is preserved. Failures are assumed not to occur and only actions of successful transactions are considered.
- *Atomicity* and *durability*. In [LMWF94] a very general formal definition of atomicity for abstract datatypes is given. A number of recovery algorithms for abstract datatypes and nested transactions has been modelled and verified in an I/O automata framework. In [Kuo96], a data manager responsible for single-site recovery has been verified. Its modelling is very much oriented at the ARIES system and it would be difficult to reuse the results for any other architectures. In both works distributed commitment is not considered and all (very complex) verification is performed manually.

  In [CHvdS00], we consider the non-blocking atomic commitment protocol of Babaoglu and Toueg [BT93], combined with our own termination protocol for recovered participants. A new method to specify such protocols has been developed. In this method, timed state machines are used to specify the processes, whereas the communication mechanism between processes is defined using assertions. All safety and liveness properties, including a new improved termination property, have been proved with the interactive proof checker of PVS. We also show that the original termination protocol of Babaoglu and Toueg has an error. Verification helped us to achieve deep insight in the protocol, but it was not immediately clear how to use our results for verification of a (more general) atomicity property.

In these papers, as well as in most textbooks (for instance [BHG87]), various protocols for concurrency control and recovery are studied in isolation, at the same time making strong assumptions about each other. For instance, in the classic serializability theory, as well as in [CHvdS99], it is assumed that the results of aborted transactions are successfully removed by an appropriate recovery mechanism. The situation with centralized recovery protocols is even more complicated. In [BHG87], an abstract model of database system is presented, which includes the transaction manager, the scheduler, the recovery manager and the cache manager. The recovery manager is solely responsible for recovery, and it is assumed that the scheduler invokes operations of the recovery manager in an order that produces a serializable and strict execution. [Kuo96] shares the same separation of the scheduler and the recovery manager. However, it is also admitted in [BHG87], that in any realistic software architecture these modules are more tightly integrated. It is therefore obvious that "the scheduler" is also prone to failures, and without appropriate measures would be unable to guarantee serializability. Also, the centralized undo/redo protocol in [BHG87] does not support distributed commitment, because it allows to arbitrarily abort uncommitted transactions as a result of a system failure, thus endangering the consistency of a decision to abort a transaction among multiple sites. It would be impossible to introduce distributed commitment into this protocol without significant changes.

3) Protocol studied. In this paper, we study these protocols in a more integrated fashion, with the aim of easier and more convincing verification than when combining isolated results on these protocols. This also makes verification easier to be reused for more realistic architectures. We present a framework of an experimental transaction processing system, closely integrating strict two-phase locking (2PL), undo/redo recovery and two-phase commit. Our integration focuses on the interaction between stable and volatile memory and on log management; the details of message exchange in two-phase commit are not considered. We also developed formal definitions of atomicity, durability and serializability for a fault-tolerant environment.

4) Mechanical support. Manual formal verification of a complex protocol is often at least as likely to be erroneous as the protocol itself. This is why in our situation some form of mechanical support is indispensable. Observe that the protocol we consider here highly depends on complex data structures and has complicated correctness properties. Hence completely automatic verification is not feasible. This is why we use a higher-order interactive theorem prover. We have chosen PVS [PVS], because we have an extensive experience with it, and to be able to reuse our method for proving conflict serializability, verified with PVS in [CHvdS99]. PVS has a convenient specification language and is relatively easy to learn and to use.

5) Verification results. The properties of atomicity, durability and serializability have been completely proved with the interactive proof checker of PVS. Atomicity and durability (very closely related in our model) required most time and effort. For serializability, we only had to adjust our definition from [CHvdS99] for a fault-tolerant environment, and then could do the verification in a few days by reusing our method for proving conflict serializability.

Complete verification with PVS was very complex and took 1,5 months to complete, but greatly improved our understanding of the protocol. PVS provided great help for managing and reusing the proofs. It is clear that without suitable tool support it would be almost impossible to complete a formal proof of such size and complexity, consisting of well over 10000 interactions with the proof checker. The preliminary version of the PVS specifications and proofs can be found at [URL].

Structure of the paper. In section 2, we define atomicity, durability and serializability. Section 3 describes the transaction processing techniques used in our protocol. Section 4 outlines the protocol itself. Our approach to specification of the protocol in PVS and verification of its correctness properties can be found in section 5. Section 6 contains concluding remarks.

2. Atomicity, durability and serializability

A database consists of a set of named *data items*, distributed over a set of *sites*. Each site has both volatile and stable memory. *Volatile* memory is fast, but limited in size due to its relatively high cost, and only part of the database can be kept in volatile memory. *Stable* memory is slow, but cheap and abundant. At any time, each data item has a *stable value* in stable memory, and it may also have a *volatile value* in volatile memory.

We consider protocols in which transactions perform atomic actions on data items. Each action is denoted by a pair consisting of its name and a site at which it is performed. The most important actions are read and write, which are the only actions of transactions that directly concern the values of the data items. Let $(Read(T,x,v),st)$ represent a read action of transaction T at site st on data item x obtaining value v, and $(Write(T,x,v),st)$ a write action of transaction T at site st assigning value v to data item x. Actions $(Commit(T),st)$ and $(Abort(T),st)$ indicate the successful and unsuccessful termination of transaction T at site st, respectively. Note that abort of a transaction may be initiated by the transaction itself or may be forced by the system, for instance after a memory failure. There are also actions for memory management. An infinite enumerated sequence of actions is called a *schedule*. $S[i]$ represents an action in a schedule S with index i.

Atomicity and durability. We say that T *commits at site st* in a schedule S, represented by $Commits(T,st,S)$, if S includes a $(Commit(T),st)$ action. We say that T *commits* in a schedule S, represented by $Commits(T,S)$, if there is a site st such that $Commits(T,st,S)$. Similarly, abort of T at site st and its "global" abort are represented by $Aborts(T,st,S)$ and $Aborts(T,S)$, respectively. We now represent atomicity and durability by the combination of two properties.

1) *Decision consistency.* The decisions to commit or abort a transaction are *consistent* in a schedule S, if

$$\forall T :\ Commits(T,S) \Longrightarrow \neg Aborts(T,S).$$

2) *Update in place.* The informal definition of atomicity given in the introduction uses the expression "reflected in a database", which should not be understood literally. Indeed, in our model stable and volatile values of each data item may be different at any time, and a read action may obtain either of them. Therefore it is more appropriate to define presence in a database only in terms of observable behaviour, i.e. the results of user's interaction with the database by means of transactions. In this paper the only actions of transactions that access the value of a data item are read and write, and therefore the formal definition of atomicity should relate the values obtained by read actions to the values produced by write actions. We consider only protocols based on the "update in place" approach. In such protocols, each read should obtain the "last committed value" of a data item, i.e. the last value written to a data item that is produced by a committed transaction.

To simplify the formal definition of update in place, we assume that each schedule starts with actions of a fictitious initial transaction $T0$, which writes the initial values of all data items, and commits at all sites after that (in chapters 4 and 5 $T0$ is treated slightly differently). We also consider only reads of committed transactions, because the values obtained by aborted transactions are of little interest. Here and in the rest of the paper we skip the "site" field of read and write actions when its value is not important, e.g., $(Write(T, x, v), st)$ is sometimes replaced by $Write(T, x, v)$.

We say that S *includes no non-aborted writes of* x *between indexes* i *and* j $(i < j)$, represented by $NoWrites(x,i,j,S)$, if for any k such that $i < k$ and $k < j$, T and v the following holds: if $S[k] = Write(T, x, v)$, then $Aborts(T, S)$. Using this abbreviation, we say that S satisfies the "update in place" strategy if the following holds: if $S[j] = Read(T2, x, v)$ and $Commits(T2, S)$, then there exist i and $T1$ such that $i < j$, $S[i] = Write(T1, x, v)$, $Commits(T1, S)$ and $NoWrites(x,i,j,S)$, and if $T1 \neq T2$, then $T1$ should commit *before* a read by $T2$.

Finally, we define S as *atomic and durable*, if it satisfies the decision consistency and the "update in place" properties.

Serializability. In our earlier work we developed the formal definition of conflict serializability for finite schedules, consisting only of actions of committed transactions. Here we first repeat some definitions from [CHvdS99] and next extend them to the more general model studied in this paper.

Let Sf, $Sf1$, $Sf2$ and $Sf0$ denote finite schedules consisting only of committed reads and writes. In such schedules each read obtains exactly the *last written value* of a data item. We say that actions $a1$ and $a2$ are *conflicting* in Sf, if they are performed on the same data item and at least one of them is write. $Sf1$ and $Sf2$ are *elementary equivalent* if $Sf1 = Sf3\ a1\ a2\ Sf4$, $Sf2 = Sf3\ a2\ a1\ Sf4$ and the actions $a1$ and $a2$ are not conflicting. $Sf1$ and $Sf2$ are *conflict equivalent* if there is a finite sequence of schedules $Sf_0,\ Sf_1, \ldots Sf_k,\ k \geq 0$, such that $Sf1 = Sf_0$, $Sf2 = Sf_k$ and for all $i < k$ the schedules Sf_i and Sf_{i+1} are elementary equivalent. A schedule $Sf0$ is *serial*, if it has no interleaving between actions of different transactions. Finally, we define a schedule Sf to be *conflict serializable*, denoted by *Conf_serializable(Sf)*, if there is a serial schedule $Sf0$ such that Sf and $Sf0$ are conflict equivalent.

For an infinite schedule S consisting of actions of both committed and aborted transactions, we denote by $C(S)$ its *committed projection*, i.e. a schedule obtained from S by removing all actions except for reads and writes of committed transactions. If atomicity is proved for S, then in $C(S)$ each read again obtains the last written value, and therefore the previous definition of conflicting actions is applicable. For a schedule S, we denote by *prefix(S, k)* its prefix of the length k. S is *fault-tolerant conflict serializable*, if $\forall k$: *Conf_serializable(prefix(C(S), k))*.

3. Transaction processing techniques

We describe the basic transaction processing methods informally, using many definitions from [BHG87].

Memory management. For performance reasons, transactions perform reads and writes on versions of data items located in volatile memory. Therefore a write does not immediately change the value of a data item in stable memory. The exchange of data between volatile and stable memory is realized by operations $Fetch$ and $Flush$. $Fetch(x)$ copies x from stable into volatile memory, whereas $Flush(x)$ moves x from volatile into stable memory. A flush operation is usually performed when there is no space in volatile memory for new data items. If a transaction attempts to read a data item existing only in stable memory, that item must be fetched to volatile memory and read after that.

There are two types of memory failures: system failures, when the entire contents of volatile memory are lost, and media failures, when portions of stable memory are lost. We consider only system failures in this paper, because media failures are relatively rare, and recovery from them is more difficult and sometimes even impossible.

Undo and Redo. We say that a recovery mechanism *requires undo*, if it allows to flush data items written by a transaction that has not committed yet. If a system failure occurs at this point, on recovery the stable database will contain effects of the uncommitted transactions, which usually must be *undone* to ensure atomicity of future reads. We say that a recovery mechanism *requires redo*, if it allows a transaction to commit before all the values it wrote have been flushed from volatile to stable database. Should a system failure occur at this point, on recovery the stable database will be missing some of the effects of the committed transactions, which must be *redone*. In this paper, we consider the protocol that requires both undo and redo. Such protocols allow to flush values from volatile memory whenever it is convenient for performance reasons, and therefore may be very efficient. However, they also tend to be rather complex.

To make undo and redo after a system failure possible, we must store some information about the values of data items written by transactions in stable memory in addition to the stable database itself. Such information is usually stored as a *log*. In the basic undo/redo algorithm in [BHG87] the log is represented as a sequence of entries of the form $[T,\ x,\ v]$, identifying the value v that transaction T wrote into data item x, and such entry is added each time the corresponding write occurs. The log also contains the sets of committed and aborted transactions, as well as other information. When volatile memory is lost in a system failure, the protocol examines the log, finds the last committed values of all data items that have ever been updated, and restores them as new volatile values of these data items.

In the next two paragraphs we explain two changes made in this paper to this basic algorithm. The first one is introduced for efficiency, the second for ease of verification.

We would not gain much by using volatile memory if each write had to access stable memory as well. Therefore it is important to minimize the number of values kept in the log. In our protocol most reads and writes are performed on volatile memory and don't change the log, and the transaction's writes are added to the log only shortly before it commits (to be precise, when it precommits, see the subsection on distributed commitment). This is obviously sufficient to ensure that the log always contains last committed value of each data item. These values are only used by restart and abort actions.

During restart after a system failure, the protocol from [BHG87] *scans* the log, i.e. a sequence of entries of the form $[T,\ x,\ v]$, in order to find the set of updated data items and their last committed values. In our protocol the set of updated data items is collected during normal processing, and only the *last* committed value is included in the log at any time. As a result, the implementation of restart and abort actions is remarkably easy compared to [BHG87] and [Kuo96], and a loop is never required. The absence of loops in the modelling of actions greatly simplifies the verification. There is also no need to specify garbage collection, because it is performed automatically.

Strict 2PL. In modern database systems, *Strict Two-Phase Locking* (2PL) is the most common method used to ensure that all executions are both strict and serializable. Strict 2PL allows a transaction to access a data item only if it is currently holding a lock on that item. The basic protocol, considered in this paper, has only two lock modes:

- **Shared**. If a transaction T has obtained a shared-mode lock on item x, then T can read, but cannot write, x. Any number of transactions can simultaneously hold a shared lock on a data item.

- **Exclusive**. If a transaction T has obtained an exclusive-mode lock on item x, then T can both read and write x. Only one transaction can hold an exclusive lock on a data item at any time. We also follow the usual requirement that a transaction can write to a data item at most once [Kuo96].

To access a data item, transaction T must first lock that item in the corresponding mode. If the data item is already locked in an incompatible mode, the request to lock this item is rejected (it is clear from the definition of locks given above that only shared locks are compatible with each other). The Strict 2PL protocol requires that each transaction locks and unlocks data items in two consequent phases:

- **Growing phase.** A transaction may obtain locks, but may not release any lock.

- **Shrinking phase.** A transaction commits or aborts, and subsequently releases all its locks.

Two-Phase Commit (2PC). The basic 2PC works as follows. One of the participants also acts as a *coordinator* to orchestrate the decision process about a transaction among participants. In Phase 1, after receiving an order from the coordinator to *vote*, each participant sends to the coordinator its *vote*: YES or NO, indicating its willingness to commit or abort the transaction, respectively. The coordinator collects all votes and makes a *decision*. If a YES vote was received from all participants, then the decision is **commit**; otherwise it is **abort**. In Phase 2, the coordinator disseminates the decision to all participants. If a participant receives the decision from the coordinator, it commits or aborts the transaction according to this decision. If no decision arrives due to the coordinator's failure, a participant should find some other way to reach a decision.

Every 2PC protocol should satisfy at least the following properties [BT93]:

AC1: All participants that decide reach the same decision.
AC2: If any participant decides **commit**, then all participants must have voted YES before.
AC3: If all participants vote YES and no failures occur, then all participants decide **commit**.
AC4: Each participant decides at most once (that is, a decision is irreversible).

In [BHG87], where distributed commitment is not supported by basic undo/redo protocol, all transactions executed at a particular site are aborted after a system failure at this site. Note that if a site votes YES in an execution of 2PC deciding on commit or abort of a transaction T, and a system failure happens at this site before it receives the coordinator's decision, then aborting T may contradict the decision made by other sites. To support the 2PC protocol, we made a non-trivial change to basic undo/redo by introducing a *precommit* action. $(Precommit(T), st)$ puts the values written by T into the log maintained at site *st* simultaneously with sending a YES vote by *st* in an execution of 2PC deciding on commit or abort of T. This allows to avoid aborting T if a system failure happens before the arrival of the coordinator's decision, because the lost updates of T may now be redone instead of undone, and T may continue its execution. Note that in Strict 2PL *locks* of T may be released only after it commits or aborts. This implies that locks of a precommitted transaction must survive a system failure as well. Therefore $(Precommit(T), st)$ also adds T's *locks* to the log of *st*, and they must be restored in a situation described above.

4. Outline of the protocol

Data structure. Let $DataI$ denote the set of names of data items located at one of the distributed sites st. Our aim is to describe the protocol performed by st. First we define the data structure of the protocol. Volatile memory contains 1) volatile values of some of the data items in $DataI$ and 2) locks, both exclusive ($xlocks$) and shared ($slocks$). In the definition given below, $Trans$ and $Values$ are uninterpreted data types representing transactions and values of data items in $DataI$, respectively. Since we prefer to use only totally defined functions, ε denotes the absence of a value.

Volatile Memory:

1) $vvalue:\ DataI \longrightarrow Values \cup \{\varepsilon\}$,
2) $locks$:
   a) $xlocks:\ Trans \longrightarrow setof[DataI]$,
   b) $slocks:\ Trans \longrightarrow setof[DataI]$

Stable memory contains stable values of all data items in $DataI$ and the log. The log includes a) the status of each transaction at st (inactive, active, precommitted, committed or aborted), b) last committed values of all data items in $DataI$, c) precommitted values of some of the data items in $DataI$ (i.e. values written by transactions that precommitted but have not committed or aborted yet), d) exclusive ($sxlocks$) and shared ($sslocks$) locks of precommitted transactions, e) the set of data items that have ever been updated (changed) in volatile memory, and f) a boolean variable indicating that recovery from a system failure is required.

Stable Memory:

1) $svalue:\ DataI \longrightarrow Values$,
2) log :
   a) $status$:
      $Trans \longrightarrow \{inact, act, pcom, com, ab\}$,
   b) $cvalue:\ DataI \longrightarrow Values$,
   c) $pcvalue:\ DataI \longrightarrow Values \cup \{\varepsilon\}$,
   d) $slocks$:
      1) $sxlocks:\ Trans \longrightarrow setof[DataI]$,
      2) $sslocks:\ Trans \longrightarrow setof[DataI]$,
   e) $updated:\ setof[DataI]$,
   f) $crashed:\ boolean$

In the initial state these values are as follows (recall that $T0$ is a fictitious initial transaction):

$\forall x, T$:

$vvalue(x) = \varepsilon \wedge xlocks(T) = \emptyset \wedge slocks(T) = \emptyset \wedge$
$svalue(x) = arbitrary\ value\ (produced\ by\ T0) \wedge$
$status(T) =$ if $(T = T0)$ then com else $inact \wedge$
$cvalue(x) = svalue(x) \wedge pcvalue(x) = \varepsilon \wedge$
$sxlocks(T) = \emptyset \wedge sslocks(T) = \emptyset \wedge$
$updated = \emptyset \wedge crashed = false$

The protocol is specified by a state machine with 8 atomic actions. Below we show the precondition and the effect of each of them (skipping the "site" field, because here it is equal to st for each action). The effect is given in an imperative style close to specifications of PVS. Variables that are not assigned to are not changed. For instance, $xlocks(T) := xlocks(T) \cup \{x\}$ means that T locks x in an exclusive mode, and exclusive locks of all other transactions are not changed. $Tset$ is a set of transactions.

Note that $Read(T, x, v)$ means that the value v has been read, and hence the precondition mentions that v is obtained properly. As explained in section 3, if the volatile value of x is absent, then it must be fetched from stable memory and read after that.

1) **Read(T, x, v)**

Precondition:
  $v =$ if $vvalue(x) \neq \varepsilon$ then $vvalue(x)$ else $svalue(x)$,
  $\forall T1:\ x \in xlocks(T1) \Longrightarrow T1 = T$,
  $status(T) = inact \vee status(T) = act$

Effect:
  $vvalue(x) := v$,
  $slocks(T) :=$ if $x \notin xlocks(T)$ then $slocks(T) \cup \{x\}$
    else $slocks(T)$,
  $status(T) := act$

2) **Write(T, x, v)**

Precondition:
  $\forall T1:\ x \notin xlocks(T1)$,
  $\forall T1:\ x \in slocks(T1) \Longrightarrow T1 = T$,
  $status(T) = inact \vee status(T) = act$

Effect:
  $vvalue(x) := v$,
  $xlocks(T) := xlocks(T) \cup \{x\}$,
  $slocks(T) := slocks(T) \setminus \{x\}$,
  $status(T) := act$,
  $updated := updated \cup \{x\}$

3) **Flush(x)**

Precondition: $vvalue(x) \neq \varepsilon$
Effect: $svalue(x) := vvalue(x)$

4) **Precommit(T)**

Precondition: $status(T) = act$
Effect:
  $status(T) := pcom$,
  $pcvalue := \forall x :$ if $x \in xlocks(T)$ then $vvalue(x)$
    else $pcvalue(x)$,
  $sxlocks(T) := xlocks(T),\ sslocks(T) := slocks(T)$

5) **Commit(T)**

Precondition: $status(T) = pcom$
Effect:

$status(T) := com,$
$cvalue := \forall x :$ if $x \in xlocks(T)$ then $pcvalue(x)$
else $cvalue(x),$
$pcvalue := \forall x :$ if $x \in xlocks(T)$ then ε
else $pcvalue(x),$
$sxlocks(T) := \emptyset,\ sslocks(T) := \emptyset,$
$xlocks(T) := \emptyset,\ slocks(T) := \emptyset$

6) **Abort(T)**
Precondition: $status(T) = act \vee status(T) = pcom$
Effect:
$vvalue := \forall x :$ if $x \in xlocks(T)$ then $cvalue(x)$
else $vvalue(x),$
$status(T) := ab,$
$pcvalue := \forall x :$ if $x \in xlocks(T)$ then ε
else $pcvalue(x),$
$sxlocks(T) := \emptyset,\ sslocks(T) := \emptyset,$
$xlocks(T) := \emptyset,\ slocks(T) := \emptyset$

7) **Crash**
Precondition: none
Effect:
$vvalue\ :=\ \forall x :\ \varepsilon,$
$xlocks\ :=\ \forall T :\ \emptyset,\ slocks\ :=\ \forall T :\ \emptyset,$
$crashed\ :=\ true$

8) **Restart(Tset)**
Precondition: $\forall T :\ T \in Tset \iff status(T) = act$
Effect:
$vvalue := \forall x :$ if $x \in updated$ then
(if $pcvalue(x) \neq \varepsilon$ then $pcvalue(x)$ else $cvalue(x)$)
else $\varepsilon,$
$xlocks\ :=\ \forall T :\ sxlocks(T),$
$slocks\ :=\ \forall T :\ sslocks(T),$
$status\ :=\ \forall T :$ if $status(T) = act$ then ab
else $status(T),$
$crashed\ :=\ false$

There is an additional precondition for all actions concerning $crashed$; a restart action can only be performed if $crashed\ =\ true$, and for all other actions we require $crashed\ =\ false$.

Comments. The precondition of a write action says that x is not locked in an exclusive mode by any transaction, *including* T. This ensures that T can write to x only once, because if it is already holding an exclusive lock on x, then it has already written to x before.

Restart action has a parameter $Tset$ indicating the set of transactions that are aborted during restart. This allows us to more rigorously define aborts of transactions such that this notion covers both "voluntary" aborts and "forced" aborts, i.e. aborts performed by a system during restart. We say now that T aborts at site st in a schedule S, represented by $Aborts(T, st, S)$, if S includes either an $(Abort(T), st)$ action or a $(Recover(Tset), st)$ action, where $T \in Tset$.

Implementation issues. Note that read and write actions update the status of transactions that become active. A write action also changes the set $updated$. Since the status of transactions and the set $updated$ are located in the log, read and write actions require accessing stable memory even if these variables are not really changed. Therefore in a reasonable implementation these variables must also be present in volatile memory, and we would access their values in the log only if necessary. We did not implement this, because it would not add much to the model and would only complicate the verification.

System executions. A global state is a function which assigns to each site its local state. Then a system execution is represented by an infinite sequence of the form $gs_0 \overset{(an_0,st_0)}{\rightarrow} \ldots \overset{(an_{i-1},st_{i-1})}{\rightarrow} gs_i \overset{(an_i,st_i)}{\rightarrow} gs_{i+1} \overset{(an_{i+1},st_{i+1})}{\rightarrow} \ldots$, where the gs_i are global states and (an_i, st_i) represents the action with name an_i executed at site st_i, for $i \in \mathbf{N}$. Such execution should be meaningful, and therefore if an_i is of the form $Read(T, x, v)$, $Write(T, x, v)$ or $Flush(x)$, then we require that x is located at st_i. Besides this, each $gs_i(st_i)$ must satisfy the precondition of (an_i, st_i) and every pair $(gs_i(st_i), gs_{i+1}(st_i))$ must correspond to the effect of (an_i, st_i).

Distributed commitment. Atomicity and serializability cannot be proved unless the properties AC1, AC2 and AC4 of the 2PC protocol are ensured (AC3 is only needed for liveness properties, which are not considered in this paper). In our model, AC4 can be easily proved for each site. AC1 and AC2 require exchange of messages between sites. We don't want to consider here the details of the communication mechanism, and this is why we specify AC1 and AC2 by global predicates on schedules.

Some additional abbreviations are needed to define these predicates. We say that T *decides* at site st in a schedule S, represented by $Decides(T, st, S)$, if either $Commits(T, st, S)$ or $Aborts(T, st, S)$. We say that T *is active* at site st in a schedule S, represented by $Active(T, st, S)$, if S includes either $(Read(T, x, v), st)$ or $(Write(T, x, v), st)$ action for some x and v. We say that T commits (precommits) at site *st at moment* i in a schedule S, represented by $Commits(T, st, i, S)$ ($Precommits(T, st, i, S)$) if $S[i]\ =\ (Commit(T), st)$ ($S[i] = (Precommit(T), st)$). As explained in section 4, a precommit action corresponds to voting YES in an execution of 2PC. If S is a schedule corresponding to a system execution, then AC1 and AC2 are defined for S as follows (where fn is a function from $Sites$ to the set of natural numbers):

$$AC1(S) = \forall st1, st2, T : \\ Decides(T, st1, S) \wedge Decides(T, st2, S) \Longrightarrow \\ (Commits(T, st1, S) \Longleftrightarrow Commits(T, st2, S))$$

$$AC2(S) = \forall st, T, i : Commits(T, st, i, S) \Longrightarrow \\ \exists fn : \forall st1 : Active(T, st1, S) \Longrightarrow \\ (fn(st1) < i \wedge Precommits(T, st1, fn(st1), S))$$

In the simplest (blocking) version of 2PC, a commit (abort) action always corresponds to receiving a commit (abort) decision from the coordinator, i.e., other ways to reach a decision (for instance, based on timeouts) are not allowed. It would be easy to include into our protocol the message exchange corresponding to this simplest 2PC, and no changes to the log management would be necessary. However, this would be more challenging for more realistic 2PC's (for instance, the protocol studied in [CHvdS00]), because they require changes to the log management.

5. Specification and verification in PVS

Modelling of values. In the PVS specification, some changes are introduced to the previous model in order to make verification easier. The proof of atomicity usually uses lemmas of the form: "if a transaction T aborts, no transaction can ever read values written by T". This becomes very difficult to prove, if we allow some committed transaction $T2$ to produce the same values as T. The problem may be solved if we require all values to be unique. In the PVS model, we easily implement this by using instead of "real" values the identifiers of transactions that produced these values. Therefore, $Write(T1, x, v)$ action is replaced by $Write(T1, x)$ action that assigns "$T1$" to x. $Read(T2, x, v)$ action, where v is produced by $T1$, is replaced by $Read(T2, x, T1)$ action. As a result of these changes, the definition of "update in place" becomes as follows: if $S[j] = Read(T2, x, T1)$ and $Commits(T2, S)$, then there exist i such that $i < j$, $S[i] = Write(T1, x)$, $Commits(T1, S)$ and $NoWrites(x, i, j, S)$.

Note that it is not unusual to use only "names" of values in definitions and proofs, and not the values themselves. For instance, the same approach is used for multiversion concurrency control in chapter 5 of [BHG87].

Verification of atomicity and durability. Decision consistency is easily proved using the assumption AC1. In the proof of "update in place", most lemmas are rather complicated invariants proved by induction. To present a typical example, we introduce the following notation: $sf(x)$ represents the site at which x is located, $AS(se)$ ($SS(se)$) is the sequence of actions (states) corresponding to a system execution se, and an abort action has an additional parameter i indicating the moment at which it happened. Suppose we want to prove that after $T1$ aborts at some site, no transaction reads the values written by $T1$ at this site. This lemma is presented as follows (in a mathematical form, close to its PVS implementation):

$$\forall se, T1, x, i : Aborts(T1, sf(x), AS(se), i) \Longrightarrow \\ \forall T2, j : j > i \Longrightarrow \\ AS(se)[j] \neq (Read(T2, x, T1), sf(x))$$

To prove this lemma, according to the definition of a read action it is sufficient to prove the following fact: after a transaction $T1$ aborts at some site, for every x that is located at this site, in every "non-crashed" state its volatile value is not produced by $T1$, and if this volatile value does not exist, then stable value of x is not produced by $T1$. This is expressed by the following invariant, which is proved by induction on k, using a number of additional invariants:

$$\forall se, T1, x, i : Aborts(T1, sf(x), AS(se), i) \Longrightarrow \\ \forall k : \neg crashed(SS(se)[i+1+k](sf(x))) \Longrightarrow \\ (\text{if } vvalue(SS(se)[i+1+k](sf(x)))(x) \neq \varepsilon \\ \text{then } vvalue(SS(se)[i+1+k](sf(x)))(x) \neq T1 \\ \text{else } svalue(SS(se)[i+1+k](sf(x)))(x) \neq T1)$$

Verification of serializability. Because our earlier PVS work on serializability uses *finite* schedules, it would not be convenient to work with committed projections of infinite schedules. Therefore we have adjusted our definition, such that it now uses only projections of finite schedules.

In [CHvdS99] we presented a method for the verification of conflict serializability, based on *conflict relations* and *conflict-preserving timestamps*. In this method, all finite schedules (consisting of committed actions) that have a conflict-preserving timestamp are proved to be conflict serializable. Here we extend this method to verify fault-tolerant conflict serializability. The new definition of a conflict relation for an *infinite* schedule S is as follows: a pair $(T1, T2)$ belongs to $Conflict(S)$ iff $T1 \neq T2$, $Commits(T1, S)$, $Commits(T2, S)$ and S includes conflicting actions $a1$ by $T1$ and $a2$ by $T2$ such that $a1$ precedes $a2$.

A timestamp TS, i.e. a function from $Transactions$ to the set of natural numbers, is called *injective* and *conflict-preserving* for S, if it assigns different values to all transactions that commit in S and the following holds:
$\forall T1, T2 : Conflict(S)(T1, T2) \Rightarrow TS(T1) < TS(T2)$

We define S to be *ordered* if there exists a timestamp TS which is injective and conflict-preserving with respect to S. It is not very difficult to prove that if S is ordered, then each finite prefix of its committed projection is also ordered as defined in [CHvdS99] (by the same timestamp). Therefore we were able to reuse our earlier results and prove that any ordered schedule is fault-tolerant conflict serializable.

We use this result to verify serializability for our protocol, and define a timestamp as follows: for a schedule S,

$TSp(S)$ assigns to each transaction committed in S the index of its first commit action. TSp is obviously injective. It is also easy to prove that it is conflict preserving. Indeed, suppose $Conflict(S)(T1, T2)$. Let's consider only the case when both conflicting actions are writes. Then $S[i] = (Write(T1, x), st)$ and $S[j] = (Write(T2, x), st)$ for some x, st, i and j such that $i < j$. After $S[i]$, x is locked by $T1$ in an exclusive mode. Right before $S[j]$, x is not locked in an exclusive mode by any transaction. This means that $T1$ must *unlock* x between i and j. We can prove that a transaction unlocks any data item only when it commits or aborts. $T1$ is committed, and hence it commits at some k such that $k < j$. The definition of TSp implies $TSp(S)(T1) \leq k$, and therefore $TSp(S)(T1) < j$. $T2$ precommits at st at some l such that $l > j$, because write actions precedes a precommit action at any site. AC2 now implies that $T2$ can commit at any site only after l, and using the definition of TSp we obtain $TSp(S)(T2) > l$, and finally $TSp(S)(T2) > j$. Therefore $TSp(S)(T1) < TSp(S)(T2)$.

6. Concluding remarks

We presented the formal specification of a transaction processing system, combining a few fundamental concurrency control and recovery protocols. We also formalized its correctness properties (ACID) and studied their relation to each other. Our modelling includes such interesting aspects of these protocols as distribution and different types of memory. At the same time, our approach does not lead to a very complicated model. As a result, we were able to completely verify the correctness properties with the interactive theorem prover of PVS. The verification of serializability was greatly simplified by the reuse of our earlier PVS specifications and proofs.

The close integration of concurrency control and recovery, achieved in our model, is very beneficial for both. By restoring some locks during restart we prevent arbitrary aborts of some transactions and therefore support distributed commitment. On the other hand, some information about locks is used to manage the log more efficiently. Indeed, when processing commit or abort of a transaction, it is important to know what data items this transaction has updated, and we can obtain this information by looking at the exclusive locks of this transaction.

Future work. Besides even closer integration of the protocols (e.g., adding the message exchange between sites), some improvements in the log management are possible. If most updates modify only small portions of data items, it would be more efficient to add to the log only the portion of each data item that was actually modified. This may be implemented by *partial logging algorithms*. It would be possible to extend our protocol with partial logging, but this would require fixing the physical or logical structure of data items such as pages, records or stacks. Another good possibility is the addition of *checkpointing*, which writes the last committed value of each data item to the *stable database*, thus reducing the amount of work needed to recover from a failure.

Acknowledgments. We would like to thank the anonymous reviewers for their useful and detailed comments.

References

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

[BT93] O. Babaoglu and S. Toueg. Non-blocking atomic commitment. In S. Mullender, editor, *Distributed Systems*, pages 147–168. Addison-Wesley Publishing Comp., 1993.

[CHvdS99] D. Chkliaev, J. Hooman, and P. van der Stok. Serializability preserving extensions of concurrency control protocols. In *Proc. of the A. Ershov Third Int'l Conf. Perspectives of System Informatics*, pages 180–193. LNCS 1755, 1999.

[CHvdS00] D. Chkliaev, J. Hooman, and P. van der Stok. Mechanical verification of a non-blocking atomic commitment protocol. In *Proc. of DSVV'2000 (International Workshop on Distributed System Validation and Verification), IEEE*, pages E96–E103, 2000.

[GA93] J.N. Gray and A.Reuter. *Transaction Processing Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[Kuo96] D. Kuo. Model and verification of a data manager based on ARIES. *ACM Transactions on Database Systems*, 21(4):427–479, 1996.

[LMWF94] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, Inc., 1994.

[PVS] *Prototype Verification System,* For more details, see http://pvs.csl.sri.com/.

[Spe99] David Spelt. *Verification Support for Object Database Design,* Ph.D. Thesis, University of Twente, http://wwwhome.cs.utwente.nl/∼spelt/thesis.pdf. 1999.

[URL] *PVS specifications and proofs,* See http://www.cs.kun.nl/∼hooman/TPS.html.