Serializability Preserving Extensions of Concurrency Control Protocols

Dmitri Chkliaev[1], Jozef Hooman[2], and Peter van der Stok[1]

[1] Dept. of Computing Science, Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: {dmitri,wsstok}@win.tue.nl
[2] Computing Science Institute, University of Nijmegen, The Netherlands
e-mail: hooman@cs.kun.nl

Abstract. The verification system PVS is used to obtain mechanized support for the formal specification and verification of concurrency control protocols, concentrating on database applications. A method to verify conflict serializability has been formulated in PVS and proved to be sound and complete with the interactive proof checker of this tool. The method has been used to verify a few basic protocols. Next we present a systematic way to extend these protocols with new actions and control information. We show that if such an extension satisfies a few simple correctness conditions, the new protocol is serializable by construction.

1 Introduction

Concurrency control protocols [SKS97,Ull88], when applied to databases, manage the concurrent access to a database by multiple users or processes. Access is performed by means of transactions, consisting of a number of actions (such as reads and writes of data items). This access has to be both correct, i.e. always leaving the database in a consistent state, and efficient, i.e. providing a good overall performance. One of the most important correctness notions is serializability of transactions, which is the main topic of this paper.

It is prominently difficult to achieve both correctness and efficiency at the same time. The most popular database protocol, the Two Phase Locking protocol (2PL), is simple and ensures serializability. Although it became a commercial standard in the seventies, it has been criticized for low performance and the possibility of deadlock (see, for instance, [Tho93]). A number of more efficient database protocols has been suggested, often on top of basic protocols such as 2PL. Newly developed protocols are becoming increasingly complex, and their correctness becomes difficult to ensure. Specification and reasoning are often very informal, which easily leads to ambiguous specifications. All these factors make the understanding and the use of these new protocols difficult and they increase the danger of incorrect protocols.

To address this problem, observe that many database protocols can be modeled as variations of a few basic concurrency control protocols. Although these variations can be obtained in different ways, they can often be considered as *extensions* of a basic protocol. An extension of a database protocol is a protocol, which includes more control information (such as timestamps and versions) and corresponding new actions.

The aim is to obtain the correctness of extensions from the correctness of a basic protocol. Here we focus on only one important correctness notion, namely serializability. An interleaved execution of a number of transactions is said to be *serializable*, if it has the same effect on a database as some *serial* execution of these transactions, i.e. an execution which has no interleaving between actions of different transactions. Deadlocks are assumed not to occur (as e.g. [Ull88]), since they do not influence serializability.

Given some set of basic concurrency control protocols, we propose to prove the correctness of extensions of these protocols, using the following strategy: a) Prove correctness (i.e. serializability) of the basic protocols. b) Derive the correctness of the extensions in a systematic way, using some assumptions on their construction. Ideally, this should be done in a structured way, using some mechanical support. The aim of our paper is to suggest a method to implement this strategy. Therefore, we address the following questions: 1) How to obtain mechanical support for specification and verification? 2) How to model concurrency control protocols? 3) How to formalize serializability? 4) How to verify serializability? 5) How to formalize protocol extensions and which conditions are needed to ensure their correctness?

1) Mechanical support. To get mechanical support, we use a higher-order interactive theorem prover, since notions like serializability are easily expressed in a property-oriented, assertional, way. To express general properties about these notions, that hold for all protocols, a higher-order logic is needed. Since we would like to use arbitrary data types and not restrict ourselves to finite state systems, completely automatic verification is not feasible. Although there are several verification systems that satisfy our requirements, we have chosen to use PVS [PVS], because it has a convenient specification language and is relatively easy to learn and to use.

The specification language of PVS is a strongly-typed higher-order logic. Specifications can be structured into a hierarchy of parameterized theories. There is a number of built-in theories and a mechanism for constructing abstract datatypes. The PVS system contains an interactive proof checker with, for instance, induction rules, automatic rewriting, and decision procedures for arithmetic. It allows users to construct proofs interactively, to discharge simple verification conditions automatically, and to check proofs mechanically.

2) Specification of protocols. To model a particular protocol in PVS, we define two types: 1) `Actions`, such as read and write, and possibly additional actions necessary for the adjustment of the control information 2) `States`, representing control information (locks, timestamps, etc.); and two predicates: 3) `Effect`, defining how a state is changed after applying a particular action 4) `Pre`, defining which actions are allowed in a particular state, and which are not.

3) Serializability notions. A *schedule* is a sequence of actions by transactions. Intuitively, a schedule is considered to be correct, if it is equivalent to some serial schedule. Serial schedules are those which have no interleaving between actions of different transactions. There are different ways to define equivalence of schedules. The most intuitively appealing one leads to the notions of view equivalence. Informally, two schedules are view equivalent iff each transaction in these schedules reads the values written by the same transaction. A schedule is said to be view serializable, if it is view equivalent to some serial schedule. Testing view serializability is NP-complete [Pap79], and therefore this notion is difficult to use in practice. Another form of schedule equivalence is conflict equivalence, leading to conflict serializability. Two schedules are conflict equivalent iff one of them can be transformed into the other by a sequence of swaps of non-conflicting actions. Testing conflict serializability has a quadratic complexity, and therefore the majority of existing database protocols ensures not just view serializability, but the stronger notion of conflict serializability.

We formalize the notions of conflict and view serializability, and prove, that any conflict serializable schedule is also view serializable. This relation is well known but has never been checked mechanically. In fact, there is no standard definition of view serializability in the literature. Here we combine the informal intuition of [SKS97] with the reads-from relation of [Vid91].

4) Method of verification. A traditional method for proving conflict serializability is based on *conflict graphs*. Our method is a modification of this traditional method and does not use any notions from graph theory. We believe that our method is logically more simple and straightforward, and therefore more appropriate for mechanical verification. This makes it possible to efficiently implement our method in PVS.

Our method is based on the notion of *conflict-preserving timestamps (CPT)*. We formulate a condition for schedules to be conflict serializable using an assignment of timestamps to transactions which orders conflicting transactions (two transactions are conflicting iff at least one of them contains a write to a common data item). We prove that this condition is necessary and sufficient. Hence, to show that a protocol ensures conflict serializability, we must prove that any schedule accepted by this protocol satisfies the condition. This implies then that the protocol indeed ensures conflict serializability, as well as the weaker notion of view serializability.

5) Extensions and correctness conditions. Suppose some basic protocol, for instance the 2PL protocol, has been proved correct. Adding more control information and more actions, we obtain various extensions of this protocol. We show that serializability of these extensions is ensured by four simple correctness conditions. The proof that these conditions lead to serializable protocols is far from trivial, but has to be done only once. By applying the resulting extension scheme, we easily obtain protocols that are serializable by construction.

As an example, we consider the 2PL protocol and several layered extensions. In the basic protocol, serializability is ensured by locking and unlocking data items. The first extension adds sequences of transactions, waiting for data items to become available (i.e., unlocked). The second extension gives priority to urgent transactions, resulting in a more realistic protocol. Since we formally verified the correctness of the 2PL protocol, a simple check of the four conditions leads to the correctness of these new protocols.

Structure of this paper. This paper is organized as follows. In section 2, we provide a general specification pattern and apply it to the specification of the 2PL protocol. In section 3, the notions of conflict and view serializability are formalized. We prove that conflict serializability implies view serializability. In section 4, our verification method is presented and its soundness and completeness are shown. The method has been applied to verify the 2PL protocol and the Timestamp Ordering protocol. In section 5, we formalize extensions of protocols and the restrictions on these extensions, needed to ensure their correctness. In section 6, we apply our method to specify and verify several layered extensions of the 2PL protocol. Section 7 contains some concluding remarks.

2 Specification of protocols

We consider protocols in which transactions perform atomic actions on certain data items. Two basic actions are common for such database protocols: read and write, which are the only actions that concern the values of the data items. Additionally, there are usually other actions, necessary for the concurrency control. The set of actions of a database protocol is defined by type `ActionNames`, containing at least read and write actions (denoted as `R` and `W`). In the PVS notation (henceforth written in `typewriter` style):

```
R, W : ActionNames
```

The set of data items is defined by uninterpreted type `Variables`; the set of transactions is defined by type `Transactions`, representing the names of transactions. Moreover, we define a type `Actions` consisting of records with three fields, called `act`, `tr`, and `vari`, expressing that a particular action is performed by a transaction on a data item.

```
Actions : TYPE = [# act  : ActionNames, tr : Transactions,
                    vari : Variables #]
```

E.g., `(W, T, x)` represents a write action by transaction `T` on variable `x`.

Concurrency control protocols maintain a control part to determine which actions on data items are allowed and which are not allowed in a particular state of a database. E.g., the control part for lock-based protocols determines which data items are locked and in which mode (shared or exclusive). The control part for timestamp-based protocols contains information about timestamps of data items. In PVS, the control part for database protocols is defined by type `States`.

Each action causes certain changes in the control part. For example, for lock-based protocols, it may lock or unlock some data items. For timestamp-based protocols, this concerns the adjustment of read- and write-timestamps of some data items. Therefore, we define the initial value of the control part, i.e. the initial state, and how the control part is changed after every possible action. We also have to define which actions are allowed in a particular state, and which are not. E.g., a transaction cannot lock a data item in an exclusive mode if it is already locked by another transaction. Consequently, a database protocol is defined by the following steps:

1. Define type `ActionNames`, containing the atomic actions `R` and `W` and possibly some other atomic actions, responsible for the adjustment of control information.
2. Define type `States`, containing all control information essential for the definition of the protocol, and define the initial state `is`.
3. Define how a particular state is changed after applying a particular (allowed) action (e.g., a read or write of a data item) by means of the `Effect` predicate; a function with three arguments of types `States, Actions, States`, resp., and result of type `bool`. For states `s1` and `s2` and an action `a1`, we have `Effect(s1, a1, s2) = TRUE` iff `s2` is obtained from `s1` by applying `a1`.
4. Define which actions are allowed in a particular state by the `Pre` predicate. For a state `s1` and an action `a1` we have `Pre(s1, a1) = TRUE` iff `a1` is allowed in `s1`.

A finite execution is represented by a sequence r of the form $s_0 \overset{a_0}{\rightarrow} s_1 \overset{a_1}{\rightarrow} ... s_n \overset{a_n}{\rightarrow} s_{n+1}$. Here s_i $(0 \leq i \leq n+1)$ are states, and a_i $(0 \leq i \leq n)$ are actions. Infinite executions are represented by all finite approximations. Sequence r is a correct execution or *run* iff s_0 is the initial state, subsequent states are related by the `Effect` predicate, and actions are enabled, as expressed by the `Pre` predicate.

In PVS, a run `r` is formalized as a record with two fields: `StateSeq(r)` is a finite sequence of states, and `ActionSeq(r)` is a finite sequence of actions, where `StateSeq(r)` has one more element then `ActionSeq(r)`. For the example run above, we have `StateSeq(r) =` $s_0 s_1 ... s_n s_{n+1}$ and `ActionSeq(r) =` $a_0 a_1 ... a_n$.

A finite sequence of actions is called a *schedule*. For instance, `(W, T1, x) (W, T2, y)(R, T1, y)` represents an execution where first transaction `T1` writes a data item `x`, then transaction `T2` writes a data item `y`, and next `T1` reads `y`.

For a protocol, represented by `States`, `is`, `Actions`, `Effect` and `Pre`, and a run r of this protocol, we say that `ActionSeq(r)` is *a schedule, allowed by this protocol*. Given a definition by the four points mentioned above, we identify a protocol with the set of allowed schedules.

```
protocol : setof[Schedules] =
   { S : Schedules | EXISTS (r : Runs) : S = ActionSeq(r) }
```

2.1 Example of the Two Phase Locking protocol

Informal description. The 2PL protocol (see, e.g., [SKS97]) requires that access to data items is done in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item. There are various modes in which a data item may be locked. The basic 2PL protocol, considered in this paper, has only two modes:

– **Shared**. If a transaction `T` has obtained a shared-mode lock on item `x`, then `T` can read, but cannot write, `x`.
– **Exclusive**. If a transaction `T` has obtained an exclusive-mode lock on item `x`, then `T` can both read and write `x`.

Let `A` and `B` represent arbitrary lock modes. Suppose that transaction `T2` requests a lock of mode `B` on item `x` on which transaction `T1` (`T1` $\neq$ `T2`) currently holds a lock of mode `A`. If `T2` can be granted a lock on `x` immediately, in spite of the presence of the mode `A` lock, then we say that mode `B` is *compatible* with mode `A`. In the 2PL protocol, shared mode is compatible with shared mode, but not with exclusive mode; exclusive mode is not compatible with both shared and exclusive modes.

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. The 2PL protocol requires that each transaction issues lock and unlock requests in two phases:

– **Growing phase.** A transaction may obtain locks, but may not release any lock.
– **Shrinking phase.** A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

PVS implementation. We specify this protocol, following the four steps mentioned above. The `Effect2PL` predicate and the `Pre2PL` predicate are not shown here.

– `ActionNames`. In our model, locking is incorporated in read and write actions, and hence does not require a separate action. We only add an `unlock` action to unlock a data item which is locked in a shared or exclusive mode and a `downgrade` action which changes the mode of the lock from exclusive to shared.

```
ActionNames2PL : TYPE = { R, W, unlock, downgrade }
```

– We define `States2PL` by a record with three fields. `xset` and `sset` map each transaction to a set of data items which it locks in an exclusive and shared mode, respectively. `shrinking` is a set of transactions which already entered the shrinking phase and therefore cannot issue any new locks.

```
States2PL : TYPE =
   [# xset  : [Transactions -> setof[Variables] ],
      sset  : [Transactions -> setof[Variables] ],
      shrinking  : setof[Transactions] #]
```

In the initial state, `is2PL`, all the data items are unlocked and no transaction is shrinking.

3 View and Conflict serializability

To define view serializability, we first define view equivalence between schedules, following [SKS97]. Consider two schedules S1 and S2, where the same set of transactions participates in both schedules.

Definition 1. *The schedules* S1 *and* S2 *are* view equivalent *if the following three conditions are met:*

1. *For each data item x, if transaction* T1 *reads the initial value of x in schedule* S1 *then, in schedule* S2*, transaction* T1 *must also read the initial value of x.*
2. *For each data item x, if transaction* T1 *reads a value of x in schedule* S1 *and the value was produced by transaction* T2 *then, in schedule* S2*, transaction* T1 *must also read the value of x that was produced by transaction* T2*.*
3. *For each data item x, the transaction* T1 *(if any) that performs the last write action on x in schedule* S1*, must also perform the last write action on x in schedule* S2*.*

Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state.

The definition of view equivalence can be presented in a more formal way using the notion of a *reads-from* relation [Vid91]. We associate with each schedule S a reads-from relation Reads_from(S), not shown here, relating a transaction that read a value of an item and the transaction that wrote this value. Then view equivalence can be defined as follows.

Definition 2. *(Equivalent to 1.) The schedules* S1 *and* S2 *are* view equivalent *if their reads-from relations are equal:*

```
view_equiv(S1, S2) : bool = (Reads_from(S1) = Reads_from(S2))
```

As we mentioned in introduction, a schedule is serial, in PVS represented by predicate serial(S), if it has no interleaving between actions of different transactions. For instance, schedule (W, T2, y)(W, T1, x)(R, T1, y) is serial, because an action by T2 precedes both actions by T1. Schedule (W, T1, x)(W, T2, y)(R, T1, y) is not serial, because two actions by T1 are interleaved by an action by T2.

A schedule S belongs to the set of view serializable schedules, denoted by View_serializable, iff it is view equivalent to a serial schedule.

```
View_serializable : setof[Schedules] =
   { S | EXISTS S0 : serial(S0) AND view_equiv(S, S0) }
```

Next we explain the notion of conflict equivalence. Suppose S includes two consecutive actions a1 = (A1, T1, x) and a2 = (A2, T2, y), where A1 and A2 belong to { R, W }. Thus S = S1 a1 a2 S2 for some subschedules S1 and S2. As explained in [SKS97], the order of a1 and a2 does not influence the result of computation if either x $\neq$ y or (x = y and A1 = A2 = R). If x = y and

(A1 = W or A2 = W), then the order of a1 and a2 matters, i.e. changes the result of computation. Observe that T1 = T2 is allowed, assuming that actions of a transaction are partially ordered rather than totally ordered as in [SKS97].

Definition 3. *The actions* (A1, T1, x) *and* (A2, T2, y) *are* conflicting *iff* x = y *and (*A1 = W *or* A2 = W*).*

Definition 4. *The schedules* S1 *and* S2 *are* elementary equivalent *iff* S1 = S3 a1 a2 S4*,* S2 = S3 a2 a1 S4 *and the actions* a1 *and* a2 *are not conflicting.*

Definition 5. *The schedules* S1 *and* S2 *are* conflict equivalent*, denoted* conf_equiv(S1,S2) *iff there is a finite sequence of schedules* S_0, S_1,...S_k*,* k >= 0*, such that* S1 = S_0*,* S2 = S_k *and for all* i < k *the schedules* S_i *and* S_(i + 1) *are elementary equivalent.*

A schedule S belongs to the set of conflict serializable schedules, denoted by Conf_serializable, iff it is conflict equivalent to a serial schedule.

```
Conf_serializable : setof[Schedules] =
   { S | EXISTS S0 : serial(S0) AND conf_equiv(S, S0) }
```

Since swaps of nonconflicting actions do not change the result of computation, we can expect that they do not change the reads-from relation as well. Indeed, we have proved in PVS theorem ConfView, expressing that conflict equivalent schedules S1 and S2 are also view equivalent:

```
ConfView : THEOREM  Conf_equiv(S1, S2) IMPLIES View_equiv(S1, S2)
```

4 Our method of verification

We present a general method for mechanical verification of conflict serializability. Our approach is a modification of a traditional method for proving conflict serializability based on *conflict graphs.* We do not use graphs, but do need a notion of conflicting transactions which is defined as a *conflict relation.*

Definition 6. *A conflict relation* Conflict(S) *of a schedule* S *is defined as follows: a pair* (T1, T2) *belongs to* Conflict(S) *iff* T1 $\neq$ T2 *and*

- S *includes actions* a1 *and* a2 *by* T1 *and* T2 *respectively*
- a1 *precedes* a2 *in* S
- a1 *and* a2 *are conflicting.*

It is well-known (although not mechanically verified) that a schedule S is conflict serializable iff the relation Conflict(S), considered as a graph in which nodes are transactions, is acyclic. Our method does not use graph theory, but assigns timestamps to transactions, using an irreflexive order on timestamps. A time domain Time is some domain with a transitive, irreflexive order. For instance, the set of natural, rational or real numbers with the conventional order. A timestamp TS is a function from Transactions to Time.

Our method is based on the notion of *conflict-preserving timestamps (CPT).*

```
CPT(S, TS) : bool = FORALL T1, T2: Conflict(S)(T1, T2) IMPLIES
                                        TS(T1) < TS(T2)
```

Definition 7. *A timestamp* `TS` *is a* conflict-preserving timestamp (CPT) *with respect to schedule* `S` *iff* `CPT(S, TS) = TRUE`*.*

If a schedule `S` has a CPT then the transitive closure of `Conflict(S)` is irreflexive, because `<` is an irreflexive order on `Time`.

A schedule `S` belongs to the set of *ordered* schedules `Ordered` iff there is a timestamp `TS` which is conflict-preserving with respect to `S`.

```
Ordered : setof[Schedules] = { S | EXISTS TS : CPT(S, TS) }
```

We proved that any ordered schedule is conflict serializable, and any conflict serializable schedule is ordered. The proof has been constructed by means of the interactive proof checker of PVS and is technically fairly complicated.

```
OrdSerializable : THEOREM Ordered = Conf_serializable
```

Theorem `OrdSerializable` provides a basis for a sound and complete method for proving serializability. Given a particular protocol, we prove that each schedule allowed by this protocol is ordered, i.e. has a conflict-preserving timestamp. Thus, for a particular `protocol`, the aim is to prove the following theorem.

```
ProtocolOrdered : THEOREM subset?(protocol, Ordered)
```

After that, theorem `OrdSerializable` implies that `protocol` indeed ensures conflict serializability:

```
ProtocolCS : THEOREM subset?(protocol, Conf_serializable)
```

We successfully applied our method to the machine-checked verification of the Timestamp Ordering protocol and the 2PL protocol.

5 Extensions of (serializable) protocols

Although we have formulated in the previous section a complete method to prove conflict serializability, it is not always easy to find a conflict-preserving timestamp function for any schedule (and to prove that it actually is one). Observing that many protocols can be seen as extensions of a basic protocol (such as Timestamp Ordering or 2PL), we investigate how we can obtain serializability of an extension from serializability of a basic protocol. First we define the notion of an extension more precisely.

We say that protocol `NewProt` is an *extension* of protocol `OldProt` iff

- `OldActionNames`, the set of atomic actions of `OldProt`, is a subset of `NewActionNames`, the set of atomic actions of `NewProt`.
- `NewStates`, the control part of `NewProt`, is obtained from `OldStates`, the control part of `OldProt`, by adding a record `ext` of type `Extension`, representing the added control information:

```
NewStates : TYPE = [# old  : OldStates, ext : Extension #]
```

Our goal is to prove that if `OldProt` ensures conflict serializability and extension `NewProt` satisfies certain conditions, then `NewProt` also ensures conflict serializability. Below we derive the required conditions during the construction of the proof.

Let `Conf_ser_Old` and `Conf_ser_New` be instantiations of set of schedules `Conf_serializable` for schedules from `OldProt` and `NewProt`, respectively. Our aim is to prove the following theorem.

```
MainTheorem : THEOREM subset?(OldProt, Conf_ser_Old) IMPLIES
                      subset?(NewProt, Conf_ser_New)
```

Proof: Suppose `OldProt` ensures conflict serializability and schedule `NewS` is accepted by `NewProt`. The proof that `NewS` is conflict serializable consists of two steps.

Step 1 We prove that `NewS` is a *refinement* of some schedule `OldS`, accepted by `OldProt`, i.e. it is obtained from `OldS` by adding some actions. To construct `OldS`, we simply remove from `NewS` all added actions, i.e. all actions that do not occur in `OldActionNames`. The result is formally defined by function `Extract(NewS)`. Note that we don't remove any read or write actions, because `R` and `W` belong to `OldActionNames`. The following theorem expresses that `Extract(NewS)` is accepted by `OldProt`.

```
ExtractOld : THEOREM NewProt(NewS) IMPLIES OldProt(Extract(NewS))
```

As we show below, the proof of this theorem reveals the required correctness conditions. Since `OldProt` is conflict serializable, theorem `ExtractOld` implies `Conf_ser_Old(Extract(NewS))`.

Step 2 If `Extract(NewS)` is conflict serializability, then also `NewS`:

```
ConfNewOld : THEOREM Conf_ser_Old(Extract(NewS)) IMPLIES
                       Conf_ser_New(NewS)
```

The proof of this theorem uses completeness of our verification method for conflict serializability. Since it implies `Conf_ser_New(NewS)`, this completes the proof of theorem `MainTheorem`. **End Proof**

It remains to prove theorem `ExtractOld` and to derive the required correctness conditions.

Proof of theorem `ExtractOld`
Assume `NewProt(NewS)`. Then there exists a run `NewR` $= s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} \ldots s_n \overset{a_n}{\to} s_{n+1}$ of `NewProt` such that `NewS = ActionSeq(NewR)`, i.e. `NewS` $= a_0 a_1 \ldots a_n$. Let $a'_0 a'_1 \ldots a'_k$ be the sequence obtained from $a_0 a_1 \ldots a_n$ by removing all actions that are not in `OldActionNames`, i.e., `Extract(NewS)` $= a'_0 a'_1 \ldots a'_k$.

To prove `OldProt(Extract(NewS))`, we construct a run $s'_0 \overset{a'_0}{\to} s'_1 \overset{a'_1}{\to} \ldots s'_k \overset{a'_k}{\to} s'_{k+1}$ of `OldProt`. This run is extracted from run `NewR` by the function `ExtractR` which removes from a run of `NewProt` any action that is not in `OldActionNames` and its successor state. Moreover, we take only the old part of the remaining states. Since `Extract` and `ExtractR` both remove the same actions (those with action names not in `OldActionNames`), observe that `ActionSeq(ExtractR(NewR))` = `Extract(ActionSeq(NewR))` = `Extract(NewS)` = $a'_0 a'_1 ... a'_k$. Hence, it remains to prove that `OldR = ExtractR(NewR)` is a run of `OldProt`.

For any run `r`, let `last(r)` denote the last state of `r`. Instead of proving, that `ExtractR(NewR)` is a run in `OldProt`, it is more convenient to prove the following, stronger statement, consisting of two parts:

(i) `ExtractR(NewR)` is a run of `OldProt` and

(ii) `last(ExtractR(NewR)) = old(last(NewR))`.

The proof proceeds by induction on the length of `ActionSeq(NewR)`.

Basic Step Let `length(ActionSeq(NewR)) = 0`. Then `NewR = NewInitState` and, by definition of `ExtractR`, `ExtractR(NewR) = old(NewInitState)`. Hence, `ExtractR(NewR)` is a run if `old(NewInitState)` is equal to the initial state of `OldProt`. Then also **(ii)** is satisfied. This leads to the first condition.

Condition 1 `old(NewInitState) = OldInitState`

Induction Step Let `length(ActionSeq(NewR)) = m + 1`. Then `NewR =` `NewR1` $\xrightarrow{a}$ `last(NewR)` for some run `NewR1`. We distinguish two cases.

act(a) $\notin$`OldActionNames` Then `ExtractR(NewR)` = `Extract(NewR1)`. For part **(i)**, recall that by the induction hypothesis `ExtractR(NewR1)`, and hence also `ExtractR(NewR)`, is a run of `OldProt`.

For **(ii)**, note that `last(ExtractR(NewR))` = `last(ExtractR(NewR1))` = `old(last(NewR1))`, using the induction hypothesis. To obtain `old(last(NewR1)) = old(last(NewR))`, we introduce a condition expressing that if we apply a newly added action `aa` to an extended state `es1`, then the `old` part of it should not change.

Condition 2 `NewEffect(es1, aa, es2) IMPLIES old(es1) = old(es2)`

act(a) $\in$`OldActionNames` By definition of `ExtractR`, we have in this case `ExtractR(NewR) = Extract(NewR1)` $\xrightarrow{a}$ `old(last(NewR))`.

By the induction hypothesis, part **(ii)**, we have

`last(Extract(NewR1)) = old(last(NewR1))`. (*)

To prove **(i)**, note that `ExtractR(NewR)` is a run of `OldProt` if the following two conditions are satisfied.

- a is allowed in the last state of `Extract(NewR1)`, that is, `OldPre(last(Extract(NewR1)),` a`) = TRUE`.
  By (*), it remains to prove `OldPre(old(last(NewR1)),` a`) = TRUE`. Since a is allowed in the last state of `NewR1`, we have that `NewPre(last(NewR1),` a`) = TRUE`. Hence it is sufficient to require that any old action `oa` which is allowed in an extended state `es` according to `NewPre`, is also allowed in the `old(es)` according to `OldPre`.

Condition 3 `NewPre(es, oa) IMPLIES OldPre(old(es), oa)`

– `old(last(NewR))` is obtained from `last(Extract(NewR1))` by applying a to it, i.e.
`OldEffect(last(Extract(NewR1)),` a`, old(last(NewR))) = TRUE`.
By (*), it remains to prove
`OldEffect(old(last(NewR1)),` a`,old(last(NewR))) = TRUE`.
Since `last(NewR)` is obtained from `last(NewR1)` by applying a to it, we have `NewEffect(last(NewR1),` a`,last(NewR)) = TRUE`. Hence it is sufficient to require, for any old action `oa`, that `NewEffect` must transform the `old` part of an extended state `es1` in the same way `OldEffect` does.

Condition 4 `NewEffect(es1, oa, es2) IMPLIES OldEffect(old(es1), oa, old(es2))`

This proves **(i)**. To prove **(ii)**, observe that by the definition of `ExtractR`, in this case `last(ExtractR(NewR))` = `old(last(NewR))`.

This completes the induction step and also the proof of `ExtractOld`. **End Proof**

To implement extensions in PVS, we define a general PVS theory `ProtExtend`. As parameters, it has all types and predicates that are needed to define `OldProt` and `NewProt`. Theorem `MainTheorem`, which establishes the main result, is proved in `ProtExtend`. The conditions 1 through 4 mentioned above are added to this theory by including them as four *assumptions*. If any theory imports `ProtExtend` then a proof of these assumptions is required.

Given a conflict serializable protocol `OldProt` we can prove serializability of an extension `NewProt`, by importing theory `ProtExtend`. This requires a proof of the four assumptions. Once they have been proved, we can use `MainTheorem`, and obtain conflict serializability of `NewProt`.

6 Two extensions of the 2PL protocol

We have applied our method to the basic 2PL protocol, described in section 2. This protocol is extended in two steps, leading to a realistic protocol which is serializable by construction.

First extension - adding a sequence of waiting transactions. In the first step, we associate with each data item a sequence of transactions that are waiting for the permission to read or write this data item. If a transaction is not allowed to read or write a data item `x` immediately (because it is currently locked in an incompatible mode), the corresponding action is inserted into the sequence of `x`. After `x` becomes available, a postponed action from the sequence of `x` may be executed.

The operation of inserting an action into a sequence is modeled by *read-request actions* (`Rrequest`) and a *write-request actions* (`Wrequest`). The extension of the state consists of a function that maps each data item to a finite sequence, consisting of read- and write-requests performed by certain transactions;

in an initial state all sequences are empty. A new effect predicate transforms the state in the same way `Effect2PL` does for old actions, it leaves the old part of the state unchanged for added actions, and includes an additional predicate to define how to insert and remove requests from the waiting sequences. A new precondition ensures that not only preconditions defined by `Pre2PL` are satisfied, but also some additional preconditions.

Second step - adding priorities to waiting transactions. We define a second-level extension of the 2PL protocol by extending the first-level extension above such that the processing of transactions depends on their *priorities*. A priority function `PR` assigns to each transaction `T` its priority `PR(T)` from the set of natural numbers.

We also introduce the notion of *urgent* transactions, which is important for real-time protocols. Assume given a natural number `U`. Transaction `T` is called *urgent with respect to* `U`, if `PR(T) >= U`. We define our protocol in a new theory, such that its set of parameters includes `PR` and `U`. Changing `PR` and `U`, we obtain different protocols. Therefore our theory actually defines a class of protocols.

This extension does not introduce any new control information or any new actions. Instead, it introduces some restrictions on the order, in which transactions are performed. The aim of these new restrictions is to ensure that "urgent" transactions obtain immediate access to data items, whereas that non-urgent transactions should be served on a first-in, first-out basis.

Suppose a data item `x` has a sequence `xs`. We define a predicate `urgent_exist`, which expresses that `xs` includes requests from urgent transactions. If `urgent_exist(xs) = TRUE`, then we must execute one of the urgent transactions with the highest priority `MaxPriority(xs)`. Otherwise, we may execute the first-inserted request of the waiting sequence.

Correctness of the obtained extensions. After importing the theory `ProtExtend` with corresponding parameters for both protocols, it turned out to be very easy to prove that our four assumptions are satisfied for both protocols. Therefore our extensions indeed ensure conflict serializability.

Note that one may satisfy the conflict serializability condition by not allowing any schedule. Therefore, we additionally show that for every valid schedule in the initial protocol there is a representative in the extended protocol. For the first extension of the 2PL protocol presented above, it is easy to see that for every schedule `S` in the 2PL protocol there is a representative `S'` in the extension, which consists of the same actions. Let `S'` be a schedule where a transaction never tries to read or write a data item if it is not immediately available; then all sequences of requests are always empty and `S'` is indeed accepted by the extension. The same holds for the second extension of the 2PL protocol.

7 Concluding remarks

We have presented a formal framework for the specification of concurrency control protocols and the verification of serializability, and successfully applied it to the verification of the 2PL protocol and the Timestamp Ordering protocol. Mechanical support has been obtained by formulating this framework in the language of the verification system PVS, and all proofs have been constructed by means of the interactive theorem prover of PVS.

Moreover, a systematic way to extend serializable concurrency control protocols has been developed. If such an extension satisfies four simple verification conditions, it is serializable by construction. This can be applied in a hierarchical way, thus complex protocols can be obtained by a sequence of extensions of a basic concurrency control protocol. An old, serializable, protocol can be extended to a new protocol by adding more control information to the state and introducing additional control actions. One has to define the new initial state, a new precondition for all actions and a new effect predicate which describes the state change after each action. Then the new protocol is serializable if the following conditions are satisfied.

1. Ignoring the added control part, the new initial state equals the old initial state.
2. A new action only affects the added part of the state; it does not change the original part of the state.
3. The new precondition of an old action implies its old precondition.
4. The new effect of an old action implies its old effect.

There are several directions for future work. We intend to investigate more protocols and develop more detailed strategies for their verification. We may also add timing, i.e. extend our method to real-time database protocols. Another possibility is to study not only serializability for databases, but also more general protocols and correctness notions such as atomicity of transactions.

References

[Pap79] C.H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[PVS] *PVS Specification and Verification System, http://pvs.csl.sri.com/.*

[SKS97] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts.* The McGraw-Hill Companies, Inc., 1997.

[Tho93] A. Thomasian. Two-phase locking performance and its thrashing behavior. *ACM Transactions on Database Systems*, 18(4):579–625, 1993.

[Ull88] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 1.* Computer Science Press, 1988.

[Vid91] K. Vidyasankar. Unified theory of database serializability. *Fundamenta Informaticae*, 14:147–183, 1991.