Formalization of (Conflict-)Serializability and Strict Two-Phase Locking

Dmitriy Traytel

March 19, 2025

Abstract

Concurrency control is an essential component of any transactional database management system, which is responsible for providing isolation (the "I" in ACID) to transactions. Formally, concurrency control aims to achieve serializability: a way to rearrange the actions of concurrently executing transactions that eliminates concurrency while leaves the database modifications unchanged. In this small entry, we define serializability, a syntactic over-approximation called conflict-serializability, and characterize schedules generated by the frequently used concurrency control mechanism of strict two-phase locking (S2PL). We also prove two inclusions: S2PL implies conflict-serializability, which in turn implies serializability. The formalization is based on standard material from an advanced database systems course [1, Chapter 17].

1 Transactions

We work with a rather abstract model of transactions comprised of read/write actions.

Read/written values are natural numbers.

type-alias *val* = *nat*

Transactions *'xid* may read from/write two addresses *'addr*.

datatype (*'xid*, *'addr*) *action* = *isRead*: *Read* (*xid-of*: ‹*'xid*›) (*addr-of*: *'addr*)
| *isWrite*: *Write* (*xid-of*: ‹*'xid*›) (*addr-of*: *'addr*)

A schedule is a sequence of actions.

type-synonym (*'xid*, *'addr*) *schedule* = ‹(*'xid*, *'addr*) *action list*›

A database, which is being modified by the read/write actions, maps addresses to values.

type-synonym *'addr db* = ‹*'addr* ⇒ *val*›

Each transaction has a local state, which is represented as the list of previously read values (and the addresses they have been read from).

type-synonym *'addr xstate* = ‹(*'addr* × *val*) *list*›

The values written by a transaction are given by a higher-order parameter and may depend on the previously read values.

context fixes *write-logic* :: ‹*'xid* ⇒ *'addr xstate* ⇒ *'addr* ⇒ *val*› **begin**

Read values are recorded in the transaction's local state; writes modify the database.

fun *action-effect* :: ‹(*'xid*, *'addr*) *action* ⇒ (*'xid* ⇒ *'addr xstate*) × *'addr db* ⇒ (*'xid* ⇒ *'addr xstate*) × *'addr db*› **where**
‹*action-effect* (*Read xid addr*) (*xst*, *db*) = (*xst*(*xid* := (*addr*, *db addr*) # *xst xid*), *db*)›
| ‹*action-effect* (*Write xid addr*) (*xst*, *db*) = (*xst*, *db*(*addr* := *write-logic xid* (*xst xid*) *addr*))›

We are interested in how a schedule modifies the database (local state changes are discarded at the end).

definition *schedule-effect* :: ‹(*'xid*, *'addr*) *schedule* ⇒ *'addr db* ⇒*'addr db*› **where**
‹*schedule-effect s db* = *snd* (*fold action-effect s* (λ-. [], *db*))›

end

Actions that belong to the same transaction.

definition *eq-xid* **where**
‹*eq-xid a b* = (*xid-of a* = *xid-of b*)›

2 Serial and Serializable Schedules

declare *length-dropWhile-le*[*termination-simp*]

A serial schedule does not interleave actions of different transactions.

fun *serial* :: ‹(*'xid*, *'addr*) *schedule* ⇒ *bool*› **where**
‹*serial* [] = *True*›
| ‹*serial* (*a* # *as*) = (*let bs* = *dropWhile* (λ*b*. *eq-xid a b*) *as*
in serial bs ∧ *xid-of a* ∉ *xid-of* ' *set bs*)›

A schedule s can be rearranged into schedule t by a permutation π, which preserves the relative order of actions related by *eq*.

definition *permutes-upto* **where**
‹*permutes-upto eq* π *s t* =
(*bij-betw* π {..<*length s*} {..<*length t*} ∧
(∀ *i*<*length s*. *s* ! *i* = *t* ! π *i*) ∧
(∀ *i*<*length s*. ∀ *j*<*length s*. *i* < *j* ∧ *eq* (*s* ! *i*) (*s* ! *j*) ⟶ π *i* < π *j*))›

lemma *permutes-upto-Nil*[*simp*]: ‹*permutes-upto R π* [] []›
by (*auto simp*: *permutes-upto-def bij-betw-def*)

Two schedules are equivalent if one can be rearranged into another without rearranging the actions of each transaction and in addition they have the same effect on any database for any fixed write logic.

abbreviation *equivalent* :: ‹('*xid*, '*addr*) *schedule* ⇒ ('*xid*, '*addr*) *schedule* ⇒ *bool*›
where
‹*equivalent s t* ≡ (∃ π. *permutes-upto eq-xid π s t* ∧ (∀ *write-logic db*. *schedule-effect write-logic s db* = *schedule-effect write-logic t db*))›

A schedule is serializable if it is equivalent to some serial schedule. Serializable schedules thus provide isolation: even though actions of different transactions may be interleaved, the effect from the point of view of each transaction is as if the transaction was the only one executing in the system (as is the case in serial schedules).

definition *serializable* :: ‹('*xid*, '*addr*) *schedule* ⇒ *bool*› **where**
‹*serializable s* = (∃ *t*. *serial t* ∧ *equivalent s t*)›

3 Conflict Serializable Schedules

Two actions of different transactions are conflicting if they access the same address and at least one of them is a write.

definition *conflict* **where**
‹*conflict a b* = (*xid-of a* ≠ *xid-of b* ∧ *addr-of a* = *addr-of b* ∧ (*isWrite a* ∨ *isWrite b*))›

Two schedules are conflict-equivalent if one can be rearranged into another without rearranging conflicting actions or actions of one transaction. Note that unlike equivalence, the conflict-equivalence notion is purely syntactic, i.e., not talking about databases and action/schedule effects.

abbreviation *conflict-equivalent* :: ‹('*xid*, '*addr*) *schedule* ⇒ ('*xid*, '*addr*) *schedule* ⇒ *bool*› **where**
‹*conflict-equivalent s t* ≡ (∃ π. *permutes-upto* (*sup eq-xid conflict*) π *s t*)›

A schedule is conflict-serializable if it is conflict equivalent to some serial schedule.

definition *conflict-serializable* :: ‹('*xid*, '*addr*) *schedule* ⇒ *bool*› **where**
‹*conflict-serializable s* = (∃ *t*. *serial t* ∧ *conflict-equivalent s t*)›

4 Conflict-Serializability Implies Serializability

In the following, we prove that the syntactic notion implies the semantic one. The key obsevation is that swapping non-conflicting actions of different transactions preserves the overall effect on the database.

lemma *swap-actions*: ‹¬ *conflict a b* ⟹ ¬ *eq-xid a b* ⟹
action-effect wl a (*action-effect wl b st*) = *action-effect wl b* (*action-effect wl a st*)›
unfolding *conflict-def eq-xid-def*
by (*cases a*; *cases b*; *cases st*) (*auto simp add*: *fun-upd-twist insert-commute*)

lemma *swap-many-actions*: ‹∀ *i* < *length p*. ¬ *conflict a* (*p* ! *i*) ∧ ¬ *eq-xid a* (*p* ! *i*) ⟹
action-effect wl a (*fold* (*action-effect wl*) *p st*) = *fold* (*action-effect wl*) *p* (*action-effect wl a st*)›
proof (*induct p arbitrary*: *st*)
case (*Cons b p*)
from *Cons*(*2*) **show** *?case*
unfolding *fold-simps*
by (*subst Cons*(*1*)[*of* ‹*action-effect wl b st*›]) (*auto simp*: *swap-actions*[*of a b*])
qed *simp*

lemma *fold-action-effect-eq*:
assumes ‹*t* = *p* @ *a* # *u*›
shows
‹*fold* (*action-effect wl*) *s* (*action-effect wl a st*) =
fold (*action-effect wl*) (*p* @ *u*) (*action-effect wl a st*) ⟹
∀ *i* < *length p*. ¬ *conflict a* (*p* ! *i*) ∧ ¬ *eq-xid a* (*p* ! *i*) ⟹
fold (*action-effect wl*) (*a* # *s*) *st* = *fold* (*action-effect wl*) *t st*›
unfolding *schedule-effect-def fold-simps fold-append o-apply assms*
by (*subst swap-many-actions*) *auto*

definition *shift* **where**
‹*shift* π = (λ*i*. *if i* < π *0 then i else i* − *1*) *o* π *o Suc*›

lemma *bij-betw-remove*: ‹*bij-betw f A B* ⟹ *x* ∈ *A* ⟹ *bij-betw f* (*A* − {*x*}) (*B* − {*f x*})›
unfolding *bij-betw-def* **by** (*auto simp*: *inj-on-def*)

lemma *permutes-upto-shift*:
assumes ‹*permutes-upto eq* π (*a* # *s*) *t*›
shows ‹*permutes-upto eq* (*shift* π) *s* (*take* (π *0*) *t* @ *drop* (*Suc* (π *0*)) *t*)›
proof −
from *assms* **have** π: ‹*bij-betw* π {..< *Suc* (*length s*)} {..< *length t*}›
‹⋀*i*. *i* < *length s* + *Suc 0* ⟹ (*a* # *s*) ! *i* = *t* ! π *i*›
‹⋀*i j*. *i* < *j* ⟹ *i* < *length s* + *Suc 0* ⟹ *j* < *length s* + *Suc 0* ⟹
eq ((*a* # *s*) ! *i*) ((*a* # *s*) ! *j*) ⟹ π *i* < π *j*›
unfolding *permutes-upto-def* **by** *auto*
from π(*1*) **have** *distinct*: ‹π (*Suc i*) ≠ π *0*› **if** ‹*i* < *length s*› **for** *i*
using *that* **unfolding** *bij-betw-def* **by** (*auto dest*!: *inj-onD*[*of* π - ‹*Suc i*› *0*])
from π(*1*) **have** *le*: ‹π ‘ {..< *Suc* (*length s*)} ⊆ {..< *length t*}›
unfolding *bij-betw-def* **by** *auto*
then have ‹*bij-betw* (λ*i*. *if i* < π *0 then i else i* − *1*) ({..< *length t*} − {π *0*}) {..< *length t* − *1*}›

by (*cases t*) (*auto 0 3 simp*: *bij-betw-def inj-on-def image-iff not-less subset-eq Ball-def*)
moreover have ‹*bij-betw π* {*Suc 0* ..< *Suc* (*length s*)} ({..< *length t*} − {*π 0*})›
using *bij-betw-remove*[**where** *x* = *0*, *OF π*(*1*)]
by (*simp add*: *atLeast1-lessThan-eq-remove0*)
moreover have ‹*bij-betw Suc* {..< *length s*} {*Suc 0* ..< *Suc* (*length s*)}›
by (*auto simp*: *lessThan-atLeast0*)
ultimately have ‹*bij-betw* (*shift π*) {..< *length s*} {..< *length t* − *Suc 0*}›
unfolding *shift-def* **by** (*auto intro*: *bij-betw-trans*)
moreover have ‹*s* ! *i* = (*take* (*π 0*) *t* @ *drop* (*Suc* (*π 0*)) *t*) ! *shift π i*› **if** ‹*i* < *length s*› **for** *i*
using *that π*(*2*)[*of* ‹*Suc i*›] *le distinct*[*of i*]
by (*force simp*: *shift-def nth-append not-less subset-eq min-def*)
moreover have ‹*shift π i* < *shift π j*›
if ‹*i*<*length s*› ‹*j*<*length s*› ‹*i* < *j*› ‹*eq* (*s* ! *i*) (*s* ! *j*)› **for** *i j*
using *that π*(*3*)[*of* ‹*Suc i*› ‹*Suc j*›] *le distinct*[*of i*] *distinct*[*of j*]
by (*auto simp*: *shift-def not-less subset-eq*)
ultimately show *?thesis*
unfolding *permutes-upto-def* **using** *le* **by** (*auto simp*: *min-def*)
qed

lemma *permutes-upto-prefix-upto*:
assumes ‹*permutes-upto eq π* (*t* ! *π 0* # *s*) *t*› ‹*i* < *π 0*›
shows ‹¬ *eq* (*t* ! *π 0*) (*t* ! *i*)›
proof
assume ‹*eq* (*t* ! *π 0*) (*t* ! *i*)›
moreover
from *assms* **have** *π*: ‹*bij-betw π* {..< *Suc* (*length s*)} {..< *length t*}›
‹⋀*i*. *i* < *length s* + *Suc 0* ⟹ (*t* ! *π 0* # *s*) ! *i* = *t* ! *π i*›
‹⋀*i j*. *i* < *j* ⟹ *i* < *length s* + *Suc 0* ⟹ *j* < *length s* + *Suc 0* ⟹
eq ((*t* ! *π 0* # *s*) ! *i*) ((*t* ! *π 0* # *s*) ! *j*) ⟹ *π i* < *π j*›
unfolding *permutes-upto-def* **by** *auto*
define *k* **where** ‹*k* = *the-inv-into* {..<*Suc* (*length s*)} *π i*›
from *π*(*1*) *assms*(*2*) **have** ‹*i* < *length t*›
using *bij-betwE lessThan-Suc-eq-insert-0* **by** *fastforce*
with *π*(*1*) *assms*(*2*) **have** ‹*k* > *0*› ‹*π k* = *i*› ‹*k* < *Suc* (*length s*)›
using *f-the-inv-into-f*[*of π* ‹{..<*Suc* (*length s*)}› *i*]
the-inv-into-into[*of π* ‹{..<*Suc* (*length s*)}› *i*, *OF - - subset-refl*]
by (*fastforce simp*: *bij-betw-def set-eq-iff image-iff k-def*)+
ultimately have ‹*π 0* < *π k*›
using *π*(*2*)[*of k*] **by** (*intro π*(*3*)[*of 0 k*]) *auto*
with *assms*(*2*) ‹*π k* = *i*› **show** *False* **by** *auto*
qed

lemma *conflict-equivalent-imp-equivalent*:
assumes ‹*conflict-equivalent s t*›
shows ‹*equivalent s t*›
proof −
from *assms* **obtain** *π* **where** *π*: ‹*permutes-upto* (*sup eq-xid conflict*) *π s t*›

```
    by blast
  moreover from π have ‹fold (action-effect wl) s st = fold (action-effect wl) t
st› for wl st
  proof (induct s arbitrary: π t st)
    case Nil
    then show ?case
      by (force simp: permutes-upto-def bij-betw-def)
  next
    case (Cons a s)
    from Cons(2) have ‹π 0 < length t› and a: ‹a = t ! π 0›
      by (auto simp add: permutes-upto-def bij-betw-def)
    with Cons(2) show ?case
      by (intro fold-action-effect-eq)
        (auto simp only: length-take min-absorb2 less-imp-le nth-take
          intro!: id-take-nth-drop[of ‹π 0› t] Cons(1)[where π=‹shift π›]
          dest: permutes-upto-prefix-upto elim!: permutes-upto-shift)
  qed
  then have ‹schedule-effect wl s db = schedule-effect wl t db› for wl db
    unfolding schedule-effect-def by auto
  ultimately show ?thesis
    unfolding permutes-upto-def by blast
qed

theorem conflict-serializable-imp-serializable: ‹conflict-serializable s ⟹ serializ-
able s›
  unfolding conflict-serializable-def serializable-def
  using conflict-equivalent-imp-equivalent by blast
```

5 Schedules Generated by Strict Two-Phase Locking (S2PL).

To enforce conflict-serializability database management systems use locks. Locks come in two kinds: shared locks for reads and exclusive locks for writes. An address can be accessed in a reading fashion by multiple transactions, each holding a shared lock. If one transaction however holds an exclusive locks to write to an address, then no other transaction can hold a lock (neither shared nor exclusive) for the same address.

```
datatype 'addr lock = S (addr-of: 'addr) | X (addr-of: 'addr)

fun lock-for where
  ‹lock-for (Read - addr) = S addr›
| ‹lock-for (Write - addr) = X addr›

definition valid-locks where
  ‹valid-locks locks = (∀ addr xid1 xid2. X addr ∈ locks xid1 ⟶
    X addr ∈ locks xid2 ∨ S addr ∈ locks xid2 ⟶ xid1 = xid2)›
```

A frequently used lock strategy is strict two phase locking (S2PL) in which transactions attempt to acquire locks gradually (whenever they want to execute an action that needs a particular lock) and release them all at once at the end of each transaction.

The following predicate checks whether a schedule could have been generated using the S2PL strategy. To this end, the predicate checks for each action, whether the corresponding lock could have been acquired by the transaction executing the action. We also allow lock upgrades (from shared to exclusive), i.e., one transaction can hold both a shared and and exclusive lock

As in our model there is no explicit transaction end marker (commit), we treat each transaction as finished immediately when it has executed its last action in the given schedule. This is the moment, when the transaction's locks are released.

fun *s2pl* :: ‹('*xid* ⇒ '*addr lock set*) ⇒ ('*xid*, '*addr*) *schedule* ⇒ *bool*› **where**
  ‹*s2pl locks* [] = *True*›
| ‹*s2pl locks* (*a* # *s*) =
    (*let xid* = *xid-of a*; *addr* = *action.addr-of a*
     in if ∃ *xid'*. *xid'* ≠ *xid* ∧ (*X addr* ∈ *locks xid'* ∨ *isWrite a* ∧ *S addr* ∈ *locks xid'*)
     then False
     else s2pl (*locks*(*xid* := *if xid* ∉ *xid-of* ' *set s then* {} *else locks xid* ∪ {*lock-for a*})) *s*)›

We prove in the following that S2PL schedules are conflict-serializable (and thus also serializable). The proof proceeds by induction on the number of transactions in a schedule. To construct the conflict-equivalent serial schedule we always move the actions of the transaction that finished first in our S2PL schedule to the front. To do so we show that these actions are not conflicting with any preceding actions (due to the acquired/held locks).

lemma *conflict-equivalent-trans*:
  ‹*conflict-equivalent s t* ⟹ *conflict-equivalent t u* ⟹ *conflict-equivalent s u*›
proof (*elim exE*)
  fix π π'
  assume ‹*permutes-upto* (*sup eq-xid conflict*) π *s t*› ‹*permutes-upto* (*sup eq-xid conflict*) π' *t u*›
  then show ‹*conflict-equivalent s u*›
    by (*intro exI*[*of* - ‹π' ∘ π›])
      (*auto simp*: *permutes-upto-def bij-betw-trans dest*: *bij-betwE*)
qed

lemma *conflict-equivalent-append*: ‹*conflict-equivalent s t* ⟹ *conflict-equivalent* (*u* @ *s*) (*u* @ *t*)›
proof (*elim exE*)
  fix π
  assume π: ‹*permutes-upto* (*sup eq-xid conflict*) π *s t*›
  define π' **where** ‹π' *x* = (*if x* < *length u then x else* π (*x* − *length u*) + *length*

```
u)› for x
  define ππ' where ‹ππ' x = (if x < length u then x
    else the-inv-into {..<length s} π (x − length u) + length u)› for x
  from π have ‹bij-betw π' {..<length u + length s} {..<length u + length t}›
    unfolding bij-betw-iff-bijections
    by (auto simp: permutes-upto-def bij-betw-def π'-def ππ'-def
      the-inv-into-f-f f-the-inv-into-f split: if-splits
      intro!: exI[of - ππ'] the-inv-into-into[OF - - subset-refl, of - ‹{..<length s}›,
simplified])
  with π show ‹conflict-equivalent (u @ s) (u @ t)›
    by (intro exI[of - π']) (auto simp: permutes-upto-def nth-append π'-def)
qed

lemma conflict-equivalent-Cons: ‹conflict-equivalent s t ⟹ conflict-equivalent (a
# s) (a # t)›
  by (metis append-Cons append-Nil conflict-equivalent-append)

lemma conflict-equivalent-rearrange:
  assumes ‹⋀i j. xid-of (s ! i) = xid ⟹ j < i ⟹ i < length s ⟹ ¬ conflict (s
! j) (s ! i)›
  shows ‹conflict-equivalent s (filter ((=) xid ∘ xid-of) s @ filter (Not ∘ (=) xid ∘
xid-of) s)›
    (is ‹conflict-equivalent s (?filter s)›)
  using assms
proof (induct ‹length (filter ((=) xid ∘ xid-of) s)› arbitrary: s)
  case 0
  then show ?case
    by (auto simp: filter-empty-conv permutes-upto-def intro!: exI[of - id])
next
  case (Suc x)
  define i where ‹i = (LEAST i. i < length s ∧ xid-of (s ! i) = xid)›
  from Suc(2) have ‹∃i < length s. xid-of (s ! i) = xid›
    by (auto simp: Suc-length-conv filter-eq-Cons-iff nth-append nth-Cons')
  then have ‹i < length s ∧ xid-of (s ! i) = xid›
    unfolding i-def by (elim LeastI-ex)
  note i = conjunct1[OF this] conjunct2[OF this]
  then have lessi: ‹∀j < i. xid-of (s ! j) ≠ xid›
    using i-def less-trans not-less-Least by blast
  with i have filter-take: ‹filter ((=) xid ∘ xid-of) (take i s) = []›
   by (intro filter-False[of ‹take i s› ‹((=) xid ∘ xid-of)›]) (auto simp: nth-image[symmetric])
  with i have ∗: ‹filter ((=) xid ∘ xid-of) s = s ! i # filter ((=) xid ∘ xid-of)
(drop (Suc i) s)›
    by (subst id-take-nth-drop[of i s]) auto
  from i Suc(3) have ‹∀j < i. ¬ conflict (s ! j) (s ! i)› by blast
  with i lessi have ‹conflict-equivalent s (s ! i # take i s @ drop (Suc i) s)›
    (is ‹conflict-equivalent s (s ! i # ?s)›)
    by (intro exI[of - ‹λk. if k = i then 0 else if k < i then k + 1 else k›])
       (auto simp: permutes-upto-def min-def bij-betw-def inj-on-def nth-append
eq-xid-def
```

nth-Cons' dest!: *gr0-implies-Suc*)
also from *Suc*(*2*−) *i filter-take* **have** ‹*conflict-equivalent* (*s* ! *i* # *?s*) (*s* ! *i* # *?filter ?s*)›
by (*intro conflict-equivalent-Cons Suc*(*1*)) (*auto simp*: ∗ *nth-append*)
also (*conflict-equivalent-trans*) **from** *i lessi filter-take* **have** ‹*s* ! *i* # *?filter ?s* = *?filter s*›
unfolding ∗ **by** (*subst* (*8*) *id-take-nth-drop*[*of i s*]) *fastforce*+
finally show *?case* .
qed

lemma *serial-append*:
‹*serial s* ⟹ *serial t* ⟹ *xid-of* ' *set s* ∩ *xid-of* ' *set t* = {} ⟹ *serial* (*s* @ *t*)›
proof (*induct s rule*: *serial.induct*)
case (*2 a as*)
then show *?case*
using *dropWhile-eq-self-iff*[*THEN iffD2*, *of t* ‹*eq-xid a*›]
by (*fastforce simp*: *Let-def image-iff dropWhile-append eq-xid-def*[*symmetric*] *dest*: *set-dropWhileD*)
qed *simp*

lemma *serial-same-xid*: ‹∀ *x* ∈ *set s*. *xid-of x* = *xid* ⟹ *serial s*›
using *dropWhile-eq-Nil-conv*[*of* ‹(λ*x*. *xid* = *xid-of x*)› ‹*tl s*›]
by (*cases s*) (*auto simp*: *Let-def eq-xid-def*[*abs-def*] *equals0D simp del*: *dropWhile-eq-Nil-conv*)

lemma *conflict-equivalent-same-set*: ‹*conflict-equivalent s t* ⟹ *set s* = *set t*›
unfolding *permutes-upto-def bij-betw-def*
by (*auto simp*: *set-eq-iff in-set-conv-nth Bex-def image-iff*) *metis*+

lemma *s2pl-filter*:
‹*s2pl locks s* ⟹ *s2pl* (*locks*(*xid* := {})) (*filter* (*Not o* (=) *xid o xid-of*) *s*)›
by (*induct locks s rule*: *s2pl.induct*) (*auto simp*: *Let-def fun-upd-twist split*: *if-splits*)

lemma *valid-locks-grab*[*simp*]: ‹*valid-locks locks* ⟹
¬ (∃ *xid'*. *xid'* ≠ *xid-of a* ∧
(*X* (*action.addr-of a*) ∈ *locks xid'* ∨ *isWrite a* ∧ *S* (*action.addr-of a*) ∈ *locks xid'*)) ⟹
valid-locks (*locks*(*xid-of a* := *insert* (*lock-for a*) (*locks* (*xid-of a*))))›
by (*cases a*) (*auto simp*: *valid-locks-def*)

lemma *s2pl-suffix*: ‹*valid-locks locks* ⟹ *s2pl locks* (*s* @ *t*) ⟹
∀ *a* ∈ *set s*. ∃ *b* ∈ *set t*. *eq-xid a b* ⟹
∃ *locks'*. *valid-locks locks'* ∧ (∀ *xid*. *locks xid* ⊆ *locks' xid*) ∧ *s2pl locks' t*›
proof (*induct s arbitrary*: *locks*)
case (*Cons a s*)
from *Cons*(*1*)[*OF valid-locks-grab*[*of locks a*]] *Cons*(*2*−) **show** *?case*
by (*auto simp add*: *Let-def eq-xid-def fun-upd-def cong*: *if-cong split*: *if-splits*) *blast*+

qed *auto*

lemma *set-drop*: ‹$l \leq$ *length xs* $\Longrightarrow$ *set* (*drop l xs*) = *nth xs* ‘ {*l*..<*length xs*}›
by (*auto simp*: *set-conv-nth image-iff*) (*metis add.commute le-iff-add less-diff-conv*)

lemma *drop-eq-Cons*: ‹*i* < *length xs* $\Longrightarrow$ *drop i xs* = *xs* ! *i* # *drop* (*Suc i*) *xs*›
by (*subst id-take-nth-drop*[*of i xs*]) *auto*

theorem *s2pl-conflict-serializable*: ‹*s2pl* (λ-. {}) *s* $\Longrightarrow$ *conflict-serializable s*›
proof (*induct* ‹*card* (*xid-of* ‘ *set s*)› *arbitrary*: *s*)
case *0*
then show *?case*
by (*auto simp*: *conflict-serializable-def intro*!: *exI*[*of* - ‹[]›])
next
case (*Suc x*)
define *i* **where** ‹*i* = (*LEAST i*. *i* < *length s* ∧ (∀*j* ∈ {*i*+*1* ..< *length s*}. ¬ *eq-xid* (*s* ! *i*) (*s* ! *j*)))›
have ‹*i* < *length s* ∧ (∀*j* ∈ {*i*+*1* ..< *length s*}. ¬ *eq-xid* (*s* ! *i*) (*s* ! *j*))›
unfolding *i-def* **by** (*rule LeastI*[*of* - ‹*length s* − *1*›], *use Suc*(*2*) **in** ‹*cases s*›) *auto*
note *i* = *conjunct1*[*OF this*] *conjunct2*[*OF this*, *rule-format*]
define *xid* **where** ‹*xid* = *xid-of* (*s* ! *i*)›
with *i* **have** *xid*: ‹*xid* ∈ *xid-of* ‘ *set s*›
by (*auto simp*: *image-iff in-set-conv-nth Bex-def*)
from *i*(*1*) **have** ∗: ‹∃*k* ∈ {*i* ..< *length s*}. *eq-xid* (*s* ! *j*) (*s* ! *k*)› **if** ‹*j* < *i*› **for** *j*
proof (*cases* ‹*xid* = *xid-of* (*s* ! *j*)›)
case *False*
with *that i*(*1*) **show** *?thesis*
proof (*induct* ‹*i* − *j*› *arbitrary*: *j rule*: *less-induct*)
case *less*
from ‹*j* < *i*› *less-trans*[*OF* ‹*j* < *i*› ‹*i* < *length s*›]
obtain *j'* **where** ‹*j'* ∈ {*j*+*1* ..< *length s*}› ‹*eq-xid* (*s* ! *j*) (*s* ! *j'*)›
by (*subst* (*asm*) *i-def*) (*force dest*: *not-less-Least*)
with *less*(*1*)[*of j'*] *less*(*2*−) **show** *?case*
by (*cases* ‹*j'* ≥ *i*›) (*auto simp*: *diff-less-mono2 eq-xid-def*)
qed
qed (*auto simp*: *xid-def eq-xid-def Bex-def*)
have ‹*conflict-equivalent s* (*filter* ((=) *xid* ∘ *xid-of*) *s* @ *filter* (*Not* ∘ (=) *xid* ∘ *xid-of*) *s*)›
proof (*intro conflict-equivalent-rearrange notI*)
fix *k l*
let *?xid* = ‹*xid-of* (*s* ! *k*)›
assume *kl*: ‹*xid-of* (*s* ! *l*) = *xid*› ‹*k* < *l*› ‹*l* < *length s*› ‹*conflict* (*s* ! *k*) (*s* ! *l*)›
with *i*(*2*) **have** *li*: ‹*l* ≤ *i*›
unfolding *xid-def eq-xid-def*
by (*metis One-nat-def add.right-neutral add-Suc-right atLeastLessThan-iff not-less-eq-eq*)
from ‹*k* < *l*› **have** *drop-alt*: ‹*drop l s* = *drop* (*l* − *Suc k*) (*drop* (*Suc k*) *s*)›
by *auto*

```
  from li kl have take-drop-l: ‹∀ a∈set (take l s). ∃ b∈set (drop l s). eq-xid a b›
    by (force simp: nth-image[symmetric] set-drop Bex-def conj-commute
      dest!: ∗[OF less-le-trans])
  from li kl have ‹∀ a∈set (take k s). ∃ b∈set (drop k s). eq-xid a b›
    by (force simp: nth-image[symmetric] set-drop Bex-def conj-commute
      dest!: ∗[OF less-le-trans[OF less-trans[OF - ‹k < l›]]])
  with kl Suc(3) obtain locks where ‹valid-locks locks› ‹s2pl locks (drop k s)›
    using s2pl-suffix[of ‹λ-. {}› ‹take k s› ‹drop k s›] by (auto simp: valid-locks-def)
  with kl(2,3) Suc(3) ∗[of k]  ‹l ≤ i›
  have ‹valid-locks (locks(?xid := locks ?xid ∪ {lock-for (s ! k)})) ∧
    s2pl (locks(?xid := locks ?xid ∪ {lock-for (s ! k)})) (drop (Suc k) s)›
    by (subst (asm) drop-eq-Cons)
      (auto simp: Let-def image-iff eq-xid-def[symmetric] set-drop split: if-splits)
  with kl(2) li Suc(3) obtain locks' where
    ‹valid-locks locks'›
    ‹∀ xid. (locks(?xid := locks ?xid ∪ {lock-for (s ! k)})) xid ⊆ locks' xid›
    ‹s2pl locks' (drop l s)›
    using take-drop-l unfolding drop-alt
    by (atomize-elim, intro s2pl-suffix[of - ‹take (l − Suc k) (drop (Suc k) s)›])
      (auto simp del: drop-drop
        dest!: in-set-dropD[of - n ‹take (- + n) -› for n, folded take-drop])
  with kl show False
    by (subst (asm) drop-eq-Cons, simp)
      (cases ‹s ! k›; cases ‹s ! l›;
        auto simp: valid-locks-def xid-def Let-def conflict-def split: if-splits)
qed
moreover have ‹card (xid-of ‘ {x ∈ set s. xid ≠ xid-of x}) = card (xid-of ‘ set
s − {xid})›
  by (rule arg-cong) auto
with Suc(2−) have ‹conflict-serializable (filter (Not ∘ (=) xid ∘ xid-of) s)›
  using s2pl-filter[of ‹λ-. {}› s xid] xid
  by (intro Suc(1)) (auto simp: fun-upd-idem card-Diff-subset-Int)
then obtain t where t: ‹serial t› ‹conflict-equivalent (filter (Not ∘ (=) xid ∘
xid-of) s) t›
  unfolding conflict-serializable-def by blast
ultimately have ‹conflict-equivalent s (filter ((=) xid ∘ xid-of) s @ t)›
  by (auto elim!: conflict-equivalent-trans conflict-equivalent-append)
moreover from t have ‹serial (filter ((=) xid ∘ xid-of) s @ t)›
  by (intro serial-append)
    (auto dest!: conflict-equivalent-same-set intro: serial-same-xid[of - xid])
ultimately show ?case
  unfolding conflict-serializable-def by blast
qed

corollary s2pl-serializable: ‹s2pl (λ-. {}) s ⟹ serializable s›
  by (simp add: conflict-serializable-imp-serializable s2pl-conflict-serializable)
```

6 Example Executing S2PL

To make the S2PL check executable regardless of the transaction id type, we restrict the quantification to transaction ids that are occurring in the schedule.

fun *s2pl-code* :: ‹('*xid* ⇒ '*addr lock set*) ⇒ ('*xid*, '*addr*) *schedule* ⇒ *bool*› **where**
  ‹*s2pl-code locks* [] = *True*›
| ‹*s2pl-code locks* (*a* # *s*) =
    (*let xid* = *xid-of a*; *addr* = *action.addr-of a*
    in if ∃ *xid'* ∈ *xid-of* ‘ *set s*. *xid'* ≠ *xid* ∧ (*X addr* ∈ *locks xid'* ∨ *isWrite a* ∧ *S addr* ∈ *locks xid'*)
    then False
     else s2pl-code (*locks*(*xid* := *if xid* ∉ *xid-of* ‘ *set s then* {} *else locks xid* ∪ {*lock-for a*})) *s*)›

lemma *s2pl-code-cong*: (⋀*xid*. *xid* ∈ *xid-of* ‘ *set s* ⟹ *f xid* = *g xid*) ⟹
  (⋀*xid*. *xid* ∉ *xid-of* ‘ *set s* ⟹ *f xid* = {}) ⟹
  s2pl f s = *s2pl-code g s*
proof (*induct s arbitrary*: *f g*)
  case (*Cons a s*)
  from *Cons*(*2*−) **show** *?case*
    unfolding *s2pl.simps s2pl-code.simps Let-def*
    by (*intro if-cong*[*OF - refl Cons*(*1*)]) (*auto 8 2 simp*: *image-iff*)
qed *simp*

lemma *s2pl-code*[*code-unfold*]: *s2pl* (λ-. {}) *s* = *s2pl-code* (λ-. {}) *s*
  by (*simp add*: *s2pl-code-cong*)

definition *TB* = (*0* :: *nat*)
definition *TA* = (*1* :: *nat*)
definition *TC* = (*2* :: *nat*)
definition *AX* = (*0* :: *nat*)
definition *AY* = (*1* :: *nat*)
definition *AZ* = (*2* :: *nat*)

Good example involving a lock upgrade by *TA* and *TB*

lemma ‹*s2pl* (λ-. {})
  [*Write TB AZ*, *Read TA AX*, *Read TB AY*, *Read TC AX*, *Write TB AY*, *Write TC AY*, *Write TA AX*, *Write TA AY*]›
  by *eval*

Bad example: *TC* cannot acquire exclusive lock for *AY*, which is already held by *TA*

lemma ‹¬ *s2pl* (λ-. {})
  [*Read TA AX*, *Read TB AX*, *Read TC AX*, *Write TA AY*, *Write TC AY*, *Write TB AY*, *Write TA AZ*]›
  by *eval*

hide-const *TB TA TC AX AY AZ*
hide-fact *TB-def TA-def TC-def AX-def AY-def AZ-def*

References

[1] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 3rd edition, January 2003.