GrokRxiv:2026.10.machine-checked-batch-landing [cs.LO; cs.DC] 20 Oct 2026

Machine-Checked Batch Landing: Merge Resolution and Serializability in a Transactional Model of Source Control

Part IV of *Source Code as a Transactional Abstraction*

Matthew Long
The YonedaAI Collaboration, YonedaAI Research Collective
Chicago, IL
matthew@yonedaai.com · https://yonedaai.com

October 2026

Abstract

Source code and source control can be treated as transactional abstractions: a repository is a database, a change is a transaction with a read set and a write set, and landing a change is a commit under optimistic concurrency. Part I of this series gave the transaction model behind that view and proved its properties by hand. This paper makes the core of that model machine-checked. A Lean 4 development of 180 theorems, with no unproved steps and no axioms beyond propositional extensionality and quotient soundness, formalizes paths and their overlap, tree states, transactions with read and write sets, the log and its snapshots, validation, and three-way merge at file and line granularity. The headline result concerns batch landing. Suppose every member of a batch validates as a fast-forward, a rebase or a conflict-free merge, no member reads or writes what an earlier member wrote, and every merged member is patch-compatible with the head. Then the batch installs the serial execution of its members in ticket order, and the declared-access serialization graph is acyclic. Patch compatibility is exact: a conflict-free merged landing equals running its transaction on the head if and only if the condition holds. Checked counterexamples show that neither hypothesis can be dropped, and that a blind write that never read its merged path breaks serial equivalence. The formalization also corrects Part I in four places; for example, Part I's merge rule makes identical changes conflict. The theorems concern the model, not the implementation.

1 Introduction

Source code and source control are now transactional abstractions. A repository is a database, a change is a transaction that reads part of the source and writes another part, landing a change is a commit under optimistic concurrency control, and a version is a snapshot. Part I of this series gave the transaction model behind that view and proved its properties by hand [10]. Part II treated derived data as views maintained from the log [11], and Part III realized the model on cloud-portable primitives and measured it [12]. This paper makes the core of Part I's model machine-checked. The definitions and proofs form a Lean 4 development [6], and we present its statements here, together with the counterexamples that show where each hypothesis is needed and the corrections that the formalization forced on Part I.

We argue that code is taking a step that data took first. Data lakes began as files on object stores; when concurrent writers and derived computation broke file-level semantics, the field moved to tables

with a transaction log, snapshots and optimistic concurrency [3]. Agents put source control under the same pressure by multiplying concurrent writers. Scion's published design responds in the same way, with server-side changesets recorded in a write-ahead log and a ref update by compare-and-swap on one database row [2]. A transactional foundation is useful only if its guarantees hold, and the guarantees that carry the most weight under many concurrent writers are those about batched landing.

Batching is also where hand proofs are most exposed. A combiner that lands many changes in one call to the serialized authority validates each member against the committed log, but not against the other members. Part I showed that this admits write skew (Part I, Theorem 7.3) and that a pairwise rule removes it for members validated as fast-forwards or rebases (Part I, Proposition 7.4). Members validated as Merge reach the head through a three-way merge, and Part I left them outside that proposition with only a description of the merged state (Part I, Proposition 6.5). The synthesis paper then uses the pairwise rule to treat batch size as a capacity parameter with no cost in isolation [13]. That conclusion depends on the rule being correct, and its reach depends on whether the rule extends to merged members.

The headline theorem settles both questions in the model. Take a log in which no landing was a Conflict, and a batch accepted at its head: every member validated as FastForward, as Rebase, or as a Merge whose three-way merge with the head has no conflict path, and no member reads or writes a location that an earlier member wrote. Suppose every Merge member is patch-compatible with the head, which means that on its write set, running its transaction on the head gives its own diff applied to the head. Then the batch installs exactly the serial execution of its members in ticket order, and the serialization graph over declared reads and writes is acyclic (Theorem 5.1). Neither extra hypothesis can be dropped. Without the pairwise rule, the write-skew batch is accepted and equals neither serial order (Proposition 5.3). Patch compatibility is the exact condition for one merged member (Theorem 4.3), and a blind write that never read the path it merged already violates it (Proposition 4.5).

The development proves the results the headline rests on as well. These are the frame lemma, serializability of logs whose landings are fast-forwards and rebases, the order-insensitivity of independent changes, and acyclicity of the serialization graph. It proves the laws of the three-way merge: its specification, uniqueness, commutativity, identities and idempotence. It shows that a file-level conflict between edits to disjoint lines becomes a clean merge when lines are modelled as child locations with stable keys (Theorem 4.2). And it asks what survives when writers read more than they declare. No landed update is lost, serializability can fail, and a sufficient condition for serializability cannot be weakened in general (Section 6). Concrete instances show that every main hypothesis can hold nontrivially (Section 7).

Formalizing changed four statements. The merge rule as Part I describes it makes two identical changes conflict, so idempotence fails. The frame lemma holds only because agreement and the write frame use different regions of the location tree. A merged landing is a serial step exactly under patch compatibility, and the two conditions one would guess first are not enough. And the premise that undeclared reads violate is coverage of true reads by declared ones, not Part I's sound-refinement assumption. Section 8 states each as a correction to Part I.

Every result is about the model. The implementation, YonedaRepo, is first-party work by the author, and it is linked to the model only by the code-evidence appendices of Parts I and III, which rest on reading code at named commits. We therefore call the theorems machine-checked and do not call the implementation verified. The Lean sources are public in the directory `lean/` of the series repository [14], and Appendix A maps every listing in this paper to a file and line.

The paper makes five contributions.

1. A Lean 4 formalization of the concurrency core of Part I: paths and overlap, tree states and agreement, transactions with a frame condition, the log and its snapshots, validation, and file-level and line-level three-way merge (Section 2).

2. A machine-checked batch landing theorem that covers members validated as Merge, under a hypothesis shown to be exact, with counterexamples for each hypothesis (Sections 4 and 5).

3. Machine-checked proofs of Part I's frame lemma, serializability theorem, merged-writes proposition, independence theorem and coarsening proposition, in the adapted forms stated in Section 3.

4. Results on under-declared reads: no lost update, a counterexample to serializability, a sufficient condition and its tightness (Section 6).

5. Four corrections to Part I found by formalizing (Section 8), and a statement of what the formalization leaves out (Section 9).

2 The formal model

The development uses core Lean 4 at version 4.35.0, with no library beyond the standard prelude. Each definition below is stated in the notation of Part I and then shown as the Lean source it abbreviates. Listings quote statements and definitions. Proofs are omitted, including the proof fields of structure instances such as `Txn.changeset`, except where a short proof is an argument of a definition; the documentation comments of the source are dropped. Where the model departs from Part I we say so where the definition appears, and Section 9 collects the departures.

2.1 Paths and overlap

Definition 2.1 (Paths and overlap)**.** A *path* is a finite sequence of string components. For paths a, b, write $a \sqsubseteq b$ when $b = a \cdot t$ for some sequence t. Two paths *overlap*, $a \bowtie b$, when $a \sqsubseteq b$ or $b \sqsubseteq a$. An *access set* is a finite list of paths. For access sets X, Y, $X \bowtie Y$ holds when some $x \in X$ and $y \in Y$ overlap, and $X \not\bowtie Y$ otherwise. The set *covered* by X is $\mathrm{cov}(X) = \{p : x \sqsubseteq p \text{ for some } x \in X\}$, and the *overlap region* of X is $\mathrm{reg}(X) = \{p : x \bowtie p \text{ for some } x \in X\}$.

This is Part I's location tree (Part I, Definition 4.4) realized as paths. Directories, files and symbols are all nodes: a symbol entry such as `src/lib.rs#Policy::backoff` becomes the path with components `src`, `lib.rs`, `Policy` and `backoff`. Ancestry is component-wise prefix, so `src/a` is not an ancestor of `src/ab`, as in the implementation's component-boundary test. The covered set $\mathrm{cov}(X)$ is Part I's $\bigcup_{x \in X} \mathrm{sub}(x)$. The overlap region $\mathrm{reg}(X)$ adds the proper ancestors of each element of X.

```
abbrev Path := List String

def anc : Path → Path → Bool
  | [], _ => true
  | _ :: _, [] => false
  | x :: xs, y :: ys => decide (x = y) && anc xs ys

def meets (a b : Path) : Bool := anc a b || anc b a

abbrev Access := List Path

def meetsSet (X Y : Access) : Bool := X.any fun x => Y.any fun y => meets x y
```

```
def covered (X : Access) (p : Path) : Bool := X.any fun x => anc x p

def region (X : Access) (p : Path) : Bool := X.any fun x => meets x p
```

Listing 1: Paths, ancestry, overlap, coverage and region (`YonedaTx/Paths.lean`)

Lemma 2.2 (Overlap; Part I, Lemma 4.5)**.** *$a \bowtie b$ if and only if some path x satisfies $a \sqsubseteq x$ and $b \sqsubseteq x$. Overlap is reflexive and symmetric, and if $a' \sqsubseteq a$ and $a \bowtie b$ then $a' \bowtie b$.*

```
theorem overlap_iff_common_descendant {a b : Path} :
    meets a b = true ↔ ∃ x, anc a x = true ∧ anc b x = true

theorem overlap_refl (a : Path) : meets a a = true

theorem overlap_symm (a b : Path) : meets a b = meets b a

theorem overlap_coarsen {a a' b : Path} (h : anc a' a = true) (hab : meets a b = true) :
    meets a' b = true
```

Listing 2: The overlap lemma (`YonedaTx/Paths.lean`)

Proof. If $a \sqsubseteq b$, then b is a common descendant, and symmetrically. Conversely, two ancestors of one path are comparable: by induction on a, two prefixes of the same sequence agree on their common length, so one extends the other. Reflexivity and symmetry are immediate from the definition. For the last claim, if $a \sqsubseteq b$ then $a' \sqsubseteq b$ by transitivity, and if $b \sqsubseteq a$ then a' and b are both ancestors of a and hence comparable. □

2.2 States and agreement

Definition 2.3 (States, agreement and confinement)**.** A *state* is a function s from paths to $\mathbb{N} \cup \{\bot\}$, where $\bot$ means absent and a natural number stands for a blob. Two states *agree* on an access set X, written $s \equiv_X t$, when $s(p) = t(p)$ for every $p \in \mathrm{reg}(X)$. A state t *changes s only within W* when $t(p) = s(p)$ for every $p \notin \mathrm{cov}(W)$.

Agreement in the model compares more locations than Part I's relation, because every node of the tree carries its own content. Part I gives directories no content and compares the file and symbol locations in $\bigcup_{x \in X} \mathrm{sub}(x)$ (Part I, Section 4.2). The Lean relation compares every path in $\mathrm{reg}(X)$, including the proper ancestors of each x, and the content of a node is independent of the content of its children. When the ancestors of the accessed locations carry no content, as directories do in Part I, the two relations coincide; this is an informal observation, not a Lean theorem. For an access below a file, such as a symbol or a line, the Lean relation also compares the file node, whose content the model does not derive from its children. The mappings of Part I's results that conclude agreement are therefore adapted rather than literal.

Agreement and confinement use different regions on purpose: agreement inspects $\mathrm{reg}(X)$, while a change is confined to the smaller set $\mathrm{cov}(\mathsf{W})$. Section 8 shows that making the two regions equal breaks the frame lemma. Blobs are natural numbers so that concrete examples can be evaluated; any type with decidable equality would serve.

```
abbrev Blob := Nat

abbrev State := Path → Option Blob
```

```
def Agree (X : Access) (s t : State) : Prop := ∀ p, region X p = true → s p = t p

def ChangesWithin (W : Access) (s t : State) : Prop := ∀ p, covered W p = false → t p = s p
```

Listing 3: States, agreement and the write frame (`YonedaTx/State.lean`)

2.3 Transactions

Definition 2.4 (Transaction)**.** A *transaction* $c = (\mathsf{R}, \mathsf{W}, f)$ consists of access sets R and W and a function f from states to states such that

1. (frame) $f(s)(p) = s(p)$ for every state s and every $p \notin \mathrm{cov}(\mathsf{W})$;
2. (reads) if $s \equiv_{\mathsf{R} \cup \mathsf{W}} t$, then $f(s)(p) = f(t)(p)$ for every $p \in \mathrm{cov}(\mathsf{W})$.

The frame condition is Part I's sound-refinement assumption (Part I, Assumption 6.2) for one step at path granularity. The reads condition lets written values depend on W as well as on R, which admits read-modify-write edits such as patching one line of a file. It is a weaker demand on transactions than dependence on R alone, so every theorem below is at least as strong as its counterpart with that stricter condition. The set R bounds dependence from above: nothing requires f to depend on every location of R.

```
structure Txn where
  R : Access
  W : Access
  f : State → State
  frame : ∀ s p, covered W p = false → f s p = s p
  reads : ∀ s t, Agree (R ++ W) s t → ∀ p, covered W p = true → f s p = f t p
```

Listing 4: Transactions (`YonedaTx/State.lean`)

Definition 2.5 (Changeset)**.** For access sets R, W and a state Φ of new content, the *changeset* $\chi(\mathsf{R}, \mathsf{W}, \Phi)$ is the transaction with function $s \mapsto s \lhd_{\mathsf{W}} \Phi$, where $(s \lhd_{\mathsf{W}} \Phi)(p) = \Phi(p)$ for $p \in \mathrm{cov}(\mathsf{W})$ and $s(p)$ otherwise.

This is Part I's changeset (Part I, Definition 8.1) as a transaction. It is a blind write: its result on W does not depend on the state it runs on, so the reads condition holds for any R.

```
def overlay (s t : State) (W : Access) : State := fun p => if covered W p then t p else s p

def Txn.changeset (R W : Access) (Φ : State) : Txn where
  R := R
  W := W
  f s := overlay s Φ W
```

Listing 5: Changesets as transactions (`YonedaTx/State.lean`)

2.4 The log and its snapshots

Definition 2.6 (Log and snapshots)**.** A *log entry* $e = (c, b)$ is a transaction and a base version. A *log* on one ref is a finite sequence $\mathcal{L} = (e_1, \ldots, e_n)$ with $e_v = (c_v, b_v)$ and $c_v = (\mathsf{R}_v, \mathsf{W}_v, f_v)$. Its *snapshots* from an initial state s_0 are

$$S_0 = s_0, \qquad S_v = \mathrm{land}(S_{b'},\ S_{v-1},\ f_v(S_{b'})) \quad \text{with } b' = \min(b_v, v-1),$$

where land is the landing merge of Definition 2.8. The *later write sets* of a base b at head n are $\mathrm{Later}(b, n) = (\mathsf{W}_v : b < v \le n)$.

Version v is list position $v - 1$ in Lean. A range of versions $b < u \le v$ is therefore the range of positions $b \le i < v$, which is the form the Lean statements of the frame lemma and of later results use. The base is capped at the head, as the authority caps it at landing (Part I, Definition 4.16). Each change is computed on the snapshot at its own base, so Part I's consistent-base assumption (Part I, Assumption 6.1) holds by construction, as it does for changesets (Part I, Proposition 8.3(2)). Commits and first parents do not appear in this log, and neither do refs other than the one it describes.

```
structure Entry where
  txn : Txn
  base : Nat

abbrev Log := List Entry

def snap (init : State) (log : Log) : Nat → State
  | 0 => init
  | n + 1 =>
    match log[n]? with
    | none => snap init log n
    | some e =>
      landOn (snap init log (min e.base n)) (snap init log n)
        (e.txn.f (snap init log (min e.base n)))
```

Listing 6: Log entries and snapshots (`YonedaTx/Log.lean`)

```
def laterWrites (log : Log) (b n : Nat) : List Access := ((log.take n).drop b).map (·.txn.W)
```

Listing 7: Later write sets (`YonedaTx/Log.lean`)

2.5 Validation

Definition 2.7 (Validation; Part I, Definition 4.15, adapted)**.** For a list ℓ of write sets and access sets R, W, validation returns $\mathrm{val}(\ell, \mathsf{R}, \mathsf{W}) = \mathrm{Conflict}$ if $\mathsf{R} \bowtie \mathsf{W}'$ for some W' in ℓ; otherwise FastForward if ℓ is empty; otherwise Merge if $\mathsf{W} \bowtie \mathsf{W}'$ for some W' in ℓ; and otherwise Rebase. Entry e_v is validated against $\mathrm{Later}(b_v, v - 1)$. A log has *no Conflict* when every entry satisfies $b_v \le v - 1$ and its outcome is not Conflict; it is *clean* when every outcome is FastForward or Rebase.

The definition is adapted. Part I's stale-read test uses the later entries' refined write sets W^ρ, its overlap test uses the unrefined ones, and it treats opaque entries as conflicts. Here one list of write sets serves both tests and no entry is opaque. Part I's symbol-granularity corollary (Part I, Corollary 6.8), which depends on the two granularities, is therefore not formalized.

```
inductive Outcome where
  | conflict
  | merge
  | rebase
  | fastForward
  deriving DecidableEq, Repr

def validate (later : List Access) (R W : Access) : Outcome :=
  if later.any (fun Wv => meetsSet R Wv) then .conflict
  else if later.isEmpty then .fastForward
```

```
  else if later.any (fun Wv => meetsSet W Wv) then .merge
  else .rebase

def validateAt (log : Log) (n : Nat) (R W : Access) (b : Nat) : Outcome :=
  validate (laterWrites log b n) R W
```

Listing 8: Validation outcomes (`YonedaTx/Validation.lean`)

```
def NoConflict (log : Log) : Prop :=
  ∀ i e, log[i]? = some e →
    e.base ≤ i ∧ validateAt log i e.txn.R e.txn.W e.base ≠ .conflict

def Validated (log : Log) : Prop :=
  ∀ i e, log[i]? = some e →
    e.base ≤ i ∧ (validateAt log i e.txn.R e.txn.W e.base).clean = true
```

Listing 9: Validated logs (`YonedaTx/Validation.lean`)

2.6 Three-way merge

Definition 2.8 (Three-way merge and landing)**.** For values b, o, t, let $m_3(b, o, t)$ be t if $o = b$; else o if $t = b$; else o if $o = t$; and a conflict otherwise. For states B, O, T, the merge at p is $m_3(B(p), O(p), T(p))$, and p is a *conflict path* when $O(p) \neq B(p)$, $T(p) \neq B(p)$ and $O(p) \neq T(p)$. The merged state $\mathrm{M}(B, O, T)$ takes the merged value at each path, with $O(p)$ at a conflict path. The landing of a change C built on base B onto head H is $\mathrm{land}(B, H, C)$: the merge with H as ours and C as theirs, with $C(p)$ at a conflict path.

The third clause, under which two sides that made the same change do not conflict, is the rule of `diff3` and Git; Section 8 shows what fails without it. The fallback at a conflict path keeps the log total. The two fallbacks, $O(p)$ for M and the change's value for land, never affect a result: a conflict path differs from the base on the change's side, so it lies in the change's write set, and every theorem about M assumes that no path is a conflict path. Part I's authority refuses such landings, and no theorem below depends on the fallback beyond one property, that a landing changes the head only within the change's write set: results about validated logs exclude conflict paths either by hypothesis or because a clean outcome has none.

```
def merge3Cell (b o t : α) : Cell α :=
  if o = b then .ok t else if t = b then .ok o else if o = t then .ok o else .conflict

def merge3 (b o t : State) (p : Path) : Cell (Option Blob) := merge3Cell (b p) (o p) (t p)

def ConflictAt (b o t : State) (p : Path) : Prop := o p ≠ b p ∧ t p ≠ b p ∧ o p ≠ t p

def merge3State (b o t : State) : State := fun p => (merge3 b o t p).getD (o p)

def landOn (b h c : State) : State := fun p => (merge3 b h c p).getD (c p)
```

Listing 10: Three-way merge and the landing state (`YonedaTx/Merge.lean`)

Definition 2.9 (Diff)**.** For states B, C, H, the diff of C against B applied to H is $\delta(B, C, H)(p) = C(p)$ if $C(p) \neq B(p)$, and $H(p)$ otherwise.

Lemma 2.10 (Merge as a diff)**.** *If no path is a conflict path of* (B, H, C)*, then* $\mathrm{M}(B, H, C) = \delta(B, C, H)$.

```
def applyDiff (b c h : State) : State := fun p => if c p = b p then h p else c p

theorem merge3State_eq_applyDiff {b h c : State} (hnc : ∀ p, ¬ ConflictAt b h c p) :
    merge3State b h c = applyDiff b c h
```

Listing 11: Diffs, and the merge as a diff (`YonedaTx/Merge.lean`)

Proof. At each path, split on whether $H(p) = B(p)$, $C(p) = B(p)$ and $H(p) = C(p)$. If $C(p) = B(p)$, both sides give $H(p)$. If $C(p) \neq B(p)$ and $H(p) = B(p)$, both give $C(p)$. If both differ from $B(p)$, the absence of a conflict forces $H(p) = C(p)$, and both give that value. □

2.7 Lines with stable keys

Definition 2.11 (Lines)**.** For a file path f and a line key ℓ, the line ℓ of f is the child location $f \cdot \ell$, and the line map of f in s is $\ell \mapsto s(f \cdot \ell)$. A *file-granular conflict* on f between O and T over B holds when the line map of f in O differs from that in B, the map in T differs from that in B, and the maps in O and T differ.

A file is thus a location whose children are its lines. Declared at file granularity, two edits to different lines of one file overlap; declared at line granularity, they do not. The keys are stable: inserting a line does not renumber the lines after it. Positional `diff3`, in which insertions and deletions shift line numbers and hunks must be aligned, is not modelled.

```
def lines (s : State) (f : Path) : String → Option Blob := fun ℓ => s (f ++ [ℓ])

def FileConflict (b o t : State) (f : Path) : Prop :=
  lines o f ≠ lines b f ∧ lines t f ≠ lines b f ∧ lines o f ≠ lines t f
```

Listing 12: Lines as child locations (`YonedaTx/Merge.lean`)

2.8 Batches

Definition 2.12 (Batch entry)**.** A *member* $m = (c, B)$ is a transaction and the state it was built on. Its merged change against a head H is $\mu_H(m) = \mathrm{land}(B, H, f(B))$. The *octopus merge* of states $M_1, \ldots, M_k$ over H is $\mathrm{oct}(H;\,) = H$ and

$$\mathrm{oct}(H; M_1, \ldots, M_k)(p) = \begin{cases} M_1(p) & \text{if } M_1(p) \neq H(p), \\ \mathrm{oct}(H; M_2, \ldots, M_k)(p) & \text{otherwise.} \end{cases}$$

The state installed by a batch $m_1, \ldots, m_k$ at head H is $\mathrm{oct}(H; \mu_H(m_1), \ldots, \mu_H(m_k))$.

This is the adapter's octopus merge (Part I, Definition 7.1): each member is merged against the head, and at each path the first member whose merged change moved it wins. When the merged changes move disjoint paths, the order does not matter.

```
structure Member where
  txn : Txn
  base : State

def octopus (H : State) : List State → State
  | [] => H
  | M :: Ms => fun p => if M p ≠ H p then M p else octopus H Ms p
```

```
def mergedChange (H : State) (m : Member) : State := landOn m.base H (m.txn.f m.base)

def landBatch (H : State) (ms : List Member) : State := octopus H (ms.map (mergedChange H))
```

Listing 13: Batch members and the batch entry (`YonedaTx/Batch.lean`)

Definition 2.13 (Serial forms and the pairwise rule)**.** For members $m_1, \dots, m_k$ with $m_i = (c_i, B_i)$ and a head H, *sequential landing* folds $s \mapsto \mathrm{land}(B_i, s, f_i(B_i))$ from H; *serial execution* is $f_k(\cdots f_1(H)\cdots)$; and the *serial merge* folds $s \mapsto \mathrm{M}(B_i, s, f_i(B_i))$ from H. The members satisfy the *pairwise rule* when $(\mathsf{R}_j \cup \mathsf{W}_j) \not\bowtie \mathsf{W}_i$ for all $i < j$.

The pairwise rule is the condition of Part I, Proposition 7.4. A member built on a log entry (c, b) is (c, S_b).

```
def seqLand (H : State) (ms : List Member) : State :=
  ms.foldl (fun s m => landOn m.base s (m.txn.f m.base)) H

def serialTxns (H : State) (cs : List Txn) : State := cs.foldl (fun s c => c.f s) H

def serialMerge (H : State) (ms : List Member) : State :=
  ms.foldl (fun s m => merge3State m.base s (m.txn.f m.base)) H

def toMember (init : State) (log : Log) (e : Entry) : Member := ⟨e.txn, snap init log e.base⟩

def BatchRule (ci cj : Txn) : Prop := meetsSet (cj.R ++ cj.W) ci.W = false
```

Listing 14: Sequential landing, serial execution and the pairwise rule (`YonedaTx/Batch.lean`)

Definition 2.14 (Patch compatibility)**.** A member $m = (c, B)$ is *patch-compatible* with a state H, written $\mathrm{PC}(m, H)$, when $f(H)(p) = \delta(B, f(B), H)(p)$ for every $p \in \mathrm{cov}(\mathsf{W})$.

Patch compatibility compares two readings of a change. The transactional reading runs c on the current head. The patch reading takes the diff that c made against its own base and applies it to the head. The condition says the readings agree on the write set; off the write set they agree for every transaction (Theorem 4.3).

```
def PatchCompatible (m : Member) (H : State) : Prop :=
  ∀ p, covered m.txn.W p = true → m.txn.f H p = applyDiff m.base (m.txn.f m.base) H p
```

Listing 15: Patch compatibility (`YonedaTx/Batch.lean`)

Definition 2.15 (Accepted batch)**.** Let $\mathcal{L}$ have n entries and head $H = S_n$. An entry $e = (c, b)$ is *accepted* at H when $b \le n$ and $\mathrm{val}(\mathrm{Later}(b, n), \mathsf{R}, \mathsf{W})$ is FastForward or Rebase, or is Merge and no path is a conflict path of $(S_b, H, f(S_b))$. A sequence of entries $e_1, \dots, e_k$ is an *accepted batch* when every e_i is accepted and the pairwise rule holds in that order, which is ticket order.

Acceptance allows FastForward members, which Part I's combiner lands alone; allowing them makes the theorems more general. A Merge member must merge without a conflict path at file granularity. Under the line model of Definition 2.11, that includes two edits to different lines of one file (Theorem 4.2).

```
def MemberAccepted (init : State) (log : Log) (e : Entry) : Prop :=
  e.base ≤ log.length ∧
    match validateAt log log.length e.txn.R e.txn.W e.base with
```

```
    | .fastForward | .rebase => True
    | .merge => ∀ p, ¬ ConflictAt (snap init log e.base) (snap init log log.length)
        (e.txn.f (snap init log e.base)) p
    | .conflict => False

structure BatchAccepted (init : State) (log : Log) (ms : List Entry) : Prop where
  member : ∀ e ∈ ms, MemberAccepted init log e
  rule : ms.Pairwise fun ei ej => BatchRule ei.txn ej.txn
```

Listing 16: Accepted members and batches (`YonedaTx/Soundness.lean`)

2.9 Serialization graph

Definition 2.16 (Declared-access serialization graph)**.** For a log, place the reads of entry v at version b_v and its writes at version v. There is an edge $u \to v$, for entries $u \neq v$, when $u < v$ and $\mathsf{W}_u \bowtie \mathsf{W}_v$ (write-write); or $u \leq b_v$ and $\mathsf{W}_u \bowtie \mathsf{R}_v$ (write-read); or $b_u < v$ and $\mathsf{R}_u \bowtie \mathsf{W}_v$ (read-write).

This is the graph of the proof of Part I, Theorem 6.4, with operations that conflict when their accesses overlap and one is a write. Its edges come from declared sets only. A Merge landing also depends on the head at the paths it merges, and no edge records that dependence; Theorem 5.1 takes it into account through patch compatibility instead. The Lean definition indexes entries by list position, so versions u and v appear as positions $i = u - 1$ and $j = v - 1$: the write-read condition $u \leq b_v$ is $i + 1 \leq b_v$, written `i+1 ≤ ej.base`, and the read-write condition $b_u < v$ is $b_u \leq j$, written `ei.base ≤ j`.

```
def SEdge (log : Log) (i j : Nat) : Prop :=
  ∃ ei ej, log[i]? = some ei ∧ log[j]? = some ej ∧ i ≠ j ∧
    ((i < j ∧ meetsSet ei.txn.W ej.txn.W = true) ∨
     (i + 1 ≤ ej.base ∧ meetsSet ei.txn.W ej.txn.R = true) ∨
     (ei.base ≤ j ∧ meetsSet ei.txn.R ej.txn.W = true))

inductive SReach (log : Log) : Nat → Nat → Prop where
  | edge {i j : Nat} : SEdge log i j → SReach log i j
  | tail {i j k : Nat} : SReach log i j → SEdge log j k → SReach log i k
```

Listing 17: The declared-access serialization graph (`YonedaTx/Graph.lean`)

3 Single landings

The results of this section are Part I's results about one landing at a time, in the adapted forms the model supports. Each is used in the batch theorem.

Lemma 3.1 (Frame; Part I, Lemma 6.3)**.** *Let $b \leq v$. If $X \not\bowtie \mathsf{W}_u$ for every entry e_u with $b < u \leq v$, then $S_b \equiv_X S_v$.*

```
theorem frame_lemma {X : Access} {b v : Nat} (hbv : b ≤ v)
    (h : ∀ i e, b ≤ i → i < v → log[i]? = some e → meetsSet X e.txn.W = false) :
    Agree X (snap init log b) (snap init log v)
```

Listing 18: The frame lemma (`YonedaTx/Log.lean`)

Proof. A landing changes the head only within the change's write set: where $f(B)(p) = B(p)$, the merge returns the head's value, and the frame condition gives $f(B)(p) = B(p)$ for $p \notin \mathrm{cov}(\mathsf{W})$.

By induction on v, a path that no W_u with $b < u \le v$ covers keeps its value from b to v. Now let $p \in \mathrm{reg}(X)$ and suppose some W_u covers p, with $x \bowtie p$ for $x \in X$ and $w \sqsubseteq p$ for $w \in \mathsf{W}_u$. If $x \sqsubseteq p$, then x and w are both ancestors of p and are comparable; if $p \sqsubseteq x$, then $w \sqsubseteq x$. Either way $X \bowtie \mathsf{W}_u$, against the hypothesis, so p is covered by no later write set and keeps its value. □

Theorem 3.2 (Landing outcomes; Part I, Theorem 6.4, parts 1 to 3)**.** *Let entry $e_{n+1} = (c, b)$ have $b \le n$. If its outcome is not* Conflict*, then $S_b \equiv_{\mathsf{R}} S_n$. If its outcome is* FastForward*, then $b = n$ and $S_{n+1} = f(S_n)$. If its outcome is* Rebase*, then $S_{n+1}(p) = f(S_b)(p)$ for $p \in \mathrm{cov}(\mathsf{W})$ and $S_{n+1}(p) = S_n(p)$ otherwise.*

```
theorem reads_current {n : Nat} {R W : Access} {b : Nat} (hb : b ≤ n)
    (h : validateAt log n R W b ≠ .conflict) :
    Agree R (snap init log b) (snap init log n)

theorem fastForward_landing {n : Nat} {e : Entry} (he : log[n]? = some e) (hb : e.base ≤ n)
    (hff : validateAt log n e.txn.R e.txn.W e.base = .fastForward) :
    e.base = n ∧ snap init log (n + 1) = e.txn.f (snap init log n)

theorem rebase_landing {n : Nat} {e : Entry} (he : log[n]? = some e) (hb : e.base ≤ n)
    (hr : validateAt log n e.txn.R e.txn.W e.base = .rebase) (p : Path) :
    snap init log (n + 1) p =
      if covered e.txn.W p = true then e.txn.f (snap init log e.base) p
      else snap init log n p
```

Listing 19: The three parts of Part I, Theorem 6.4 (`YonedaTx/Serializability.lean`)

Proof. An outcome other than Conflict means R meets no set in $\mathrm{Later}(b, n)$, and Lemma 3.1 with $X = \mathsf{R}$ gives the first claim. FastForward means $\mathrm{Later}(b, n)$ is empty; its length is $n - b$, so $b = n$, and a landing on its own base installs the change, because the merge with $O = B$ returns T. Rebase means $\mathsf{R} \cup \mathsf{W}$ meets no later write set, so $S_b \equiv_{\mathsf{W}} S_n$ by the frame lemma. On $\mathrm{cov}(\mathsf{W})$ the head therefore equals the base and the merge takes the change. Off $\mathrm{cov}(\mathsf{W})$ the change equals the base and the merge takes the head. □

Part 2 differs from Part I's statement, which concludes $S_{v-1}(r) = \pi_1(c)$: commits do not appear in the model, so the statement is about versions and states instead. FastForward is defined, as in Part I, by the absence of later entries; once the base is capped, that is equivalent to a base equal to the head, which is the form Part I's proof uses.

Theorem 3.3 (Serializability relative to declared reads; Part I, Theorem 6.4)**.** *Let $\mathcal{L}$ be clean. Then for every v, S_v equals the serial execution $f_v(\cdots f_1(s_0)\cdots)$ of the first v transactions in log order; and for every entry e_v, S_{b_v} agrees on R_v with the serial execution of the first $v - 1$ transactions.*

```
def serial (init : State) (log : Log) (n : Nat) : State :=
  (log.take n).foldl (fun s e => e.txn.f s) init

theorem serializability (hv : Validated log) :
    (∀ n, snap init log n = serial init log n) ∧
    (∀ i e, log[i]? = some e →
      Agree e.txn.R (snap init log e.base) (serial init log i))
```

Listing 20: Serializability of clean logs (`YonedaTx/Serializability.lean`)

Proof. The step is a lemma: if $B \equiv_{\mathsf{R}\cup\mathsf{W}} H$, then $\mathrm{land}(B, H, f(B)) = f(H)$. On $\mathrm{cov}(\mathsf{W})$, $H(p) = B(p)$, so the merge takes $f(B)(p)$, which equals $f(H)(p)$ by the reads condition. Off $\mathrm{cov}(\mathsf{W})$, $f(B)(p) = B(p)$, so the merge takes $H(p)$, which equals $f(H)(p)$ by the frame condition. A clean outcome means $\mathsf{R}_v \cup \mathsf{W}_v$ meets no later write set, and the frame lemma gives $S_{b_v} \equiv_{\mathsf{R}_v\cup\mathsf{W}_v} S_{v-1}$. Induction on v then gives the first claim. For the second, the frame lemma with $X = \mathsf{R}_v$ gives $S_{b_v} \equiv_{\mathsf{R}_v} S_{v-1}$, and S_{v-1} is the serial state. □

The Lean proof goes through a weaker hypothesis than cleanness: every entry's $\mathsf{R} \cup \mathsf{W}$ meets no write set landed after its base. Validation is one way to obtain it; Section 6 uses another. The theorem is a state equality. Part I proves conflict-serializability by the serializability theorem of Bernstein, Hadzilacos and Goodman [5, Theorem 2.1]; the model proves the semantic statement directly and proves separately the direction of that theorem the argument uses (Proposition 3.7).

Proposition 3.4 (Merged writes; Part I, Proposition 6.5, at file level)**.** *For any entry $e_{n+1} = (c, b)$ with $b \le n$, write $B = S_b$, $H = S_n$ and $C = f(B)$. Then $S_{n+1}(p) = H(p)$ for $p \notin \mathrm{cov}(\mathsf{W})$; $S_{n+1}(p) = C(p)$ for $p \in \mathrm{cov}(\mathsf{W})$ that no set in $\mathrm{Later}(b, n)$ covers; and the merge of (B, H, C) at every path that is not a conflict path is the value $S_{n+1}(p)$.*

```
theorem merged_writes {n : Nat} {e : Entry} (he : log[n]? = some e) (hb : e.base ≤ n)
    (p : Path) :
    let B := snap init log e.base
    let H := snap init log n
    let C := e.txn.f B
    (covered e.txn.W p = false → snap init log (n + 1) p = H p) ∧
    (covered e.txn.W p = true → (∀ W' ∈ laterWrites log e.base n, covered W' p = false) →
      snap init log (n + 1) p = C p) ∧
    (¬ ConflictAt B H C p → merge3 B H C p = .ok (snap init log (n + 1) p))
```

Listing 21: Merged writes at file level (`YonedaTx/Serializability.lean`)

Proof. The first claim is the confinement of a landing to its write set. For the second, the pointwise frame argument gives $H(p) = B(p)$, so the merge takes $C(p)$. For the third, the merge at a non-conflict path is a value, and the landing installs that value. □

The proposition holds for every landing, not only for those validated as Merge. At file granularity it is the whole of Part I's Proposition 6.5. The case that Part I's proposition exists for, two edits to different lines of one shared file, is Theorem 4.2.

Theorem 3.5 (Independence; Part I, Theorem 6.7)**.** *Changes c_i, c_j satisfy $\mathsf{W}_i \not\bowtie (\mathsf{R}_j \cup \mathsf{W}_j)$ and $\mathsf{W}_j \not\bowtie (\mathsf{R}_i \cup \mathsf{W}_i)$ if and only if $\mathrm{val}((\mathsf{W}_i), \mathsf{R}_j, \mathsf{W}_j) = \mathrm{Rebase}$ and $\mathrm{val}((\mathsf{W}_j), \mathsf{R}_i, \mathsf{W}_i) = \mathrm{Rebase}$. If they do, and both are built on base b of a log with exactly b entries, then landing them in either order validates the second as* Rebase*, both orders give the same state, and that state is $f_j(f_i(S_b)) = f_i(f_j(S_b))$.*

```
def Independent (Ri Wi Rj Wj : Access) : Prop :=
  meetsSet Wi (Rj ++ Wj) = false ∧ meetsSet Wj (Ri ++ Wi) = false

theorem independence_iff_rebase (Ri Wi Rj Wj : Access) :
    Independent Ri Wi Rj Wj ↔ validate [Wi] Rj Wj = .rebase ∧ validate [Wj] Ri Wi = .rebase
```

Listing 22: Independence at validation (`YonedaTx/Validation.lean`)

```
theorem independence {pre : Log} {b : Nat} {ci cj : Txn} (hpre : pre.length = b)
    (hind : Independent ci.R ci.W cj.R cj.W) :
    validateAt (pre ++ [⟨ci, b⟩, ⟨cj, b⟩]) (b + 1) cj.R cj.W b = .rebase ∧
    validateAt (pre ++ [⟨cj, b⟩, ⟨ci, b⟩]) (b + 1) ci.R ci.W b = .rebase ∧
    snap init (pre ++ [⟨ci, b⟩, ⟨cj, b⟩]) (b + 2) =
      snap init (pre ++ [⟨cj, b⟩, ⟨ci, b⟩]) (b + 2) ∧
    snap init (pre ++ [⟨ci, b⟩, ⟨cj, b⟩]) (b + 2) = cj.f (ci.f (snap init pre b)) ∧
    snap init (pre ++ [⟨cj, b⟩, ⟨ci, b⟩]) (b + 2) = ci.f (cj.f (snap init pre b))
```

Listing 23: Independence in the log (`YonedaTx/Serializability.lean`)

Proof. With one later entry writing W_i, validation of c_j returns Rebase exactly when $\mathsf{R}_j \not\bowtie \mathsf{W}_i$ and $\mathsf{W}_j \not\bowtie \mathsf{W}_i$. The conjunction over both orders is the independence condition, because overlap is symmetric. For the states, c_i changes only $\mathrm{cov}(\mathsf{W}_i)$, which meets neither R_j nor W_j, so S_b and $f_i(S_b)$ agree on $\mathsf{R}_j \cup \mathsf{W}_j$, and the landing step of Theorem 3.3 installs $f_j(f_i(S_b))$. The two serial orders are equal path by path: a path covered by neither write set is unchanged in both; a path covered by one is written from agreeing inputs in both orders; and no path is covered by both, since then $\mathsf{W}_i \bowtie \mathsf{W}_j$. □

The equivalence is proved at the level of validation, as Part I states it. The state equality is the forward direction only, and the hypothesis that no other entry lands in between is expressed by the length of the log.

Proposition 3.6 (Coarsening; Part I, Proposition 6.10)**.** *If every access of* R, W *and of each later write set is replaced by an ancestor, every* Conflict *outcome stays* Conflict*, and every* Merge *outcome becomes* Merge *or* Conflict*.*

```
theorem coarsening_sound {R R' W W' : Access} {later later' : List Access}
    (hR : Coarsens R' R) (hW : Coarsens W' W) (hl : CoarsensAll later' later) :
    (validate later R W = .conflict → validate later' R' W' = .conflict) ∧
    (validate later R W = .merge →
      validate later' R' W' = .conflict ∨ validate later' R' W' = .merge)
```

Listing 24: Coarsening is sound (`YonedaTx/Validation.lean`)

Proof. By the last claim of Lemma 2.2, every overlap between the original sets survives the coarsening, and each outcome test is monotone in overlap. Coarsening also preserves the emptiness of the later list, so a Merge cannot become FastForward. □

The development also proves that independence after coarsening implies independence before, and that the frame condition survives coarsening of W, which are the remaining claims of Part I's proposition.

Proposition 3.7 (Acyclic serialization graph)**.** *If a log has no Conflict landing, every edge* $u \to v$ *of its serialization graph satisfies* $u < v$*. Hence the graph has no cycle, and log order is a topological order of it.*

```
theorem sgraph_edge_forward {log : Log} (h : NoConflict log) {i j : Nat}
    (he : SEdge log i j) : i < j

theorem sgraph_acyclic {log : Log} (h : NoConflict log) (i : Nat) : ¬ SReach log i i
```

Listing 25: Acyclicity of the serialization graph (`YonedaTx/Graph.lean`)

Proof. Write-write edges point forward by definition. A write-read edge has $u \le b_v < v$. For a read-write edge with $v < u$, the bounds $b_u < v < u$ place W_v in $\mathrm{Later}(b_u, u-1)$, and $\mathsf{R}_u \bowtie \mathsf{W}_v$ would make entry u a Conflict. A path in the graph is therefore strictly increasing, and no position reaches itself. □

This is the direction of Theorem 2.1 of Bernstein, Hadzilacos and Goodman that Part I uses [5]; the generic theorem is not formalized. The proposition needs only the absence of Conflict, so it holds for logs with Merge landings. For those, it is a statement about declared accesses and nothing more.

4 Merge resolution

4.1 Laws of the three-way merge

Proposition 4.1 (Merge laws)**.** *For states B, O, T:*

1. *a path is a conflict path if and only if the merge reports a conflict there;*
2. *if no path is a conflict path, $\mathrm{M}(B,O,T)$ equals O where O differs from B, equals T where T differs from B, and equals B elsewhere;*
3. *any state with the property of part 2 equals $\mathrm{M}(B,O,T)$;*
4. *the merge of (B,O,T) equals the merge of (B,T,O), conflicts included;*
5. *merging (B,B,T) gives T, merging (B,O,B) gives O, and merging (B,X,X) gives X;*
6. *if O changes B only within W_o, T changes B only within W_t, and $\mathsf{W}_o \not\bowtie \mathsf{W}_t$, then no path is a conflict path.*

```
def MergeSpec (b o t m : State) : Prop :=
  ∀ p, (o p ≠ b p → m p = o p) ∧ (t p ≠ b p → m p = t p) ∧
    (o p = b p → t p = b p → m p = b p)

theorem merge3_conflict_iff (b o t : State) (p : Path) :
    merge3 b o t p = .conflict ↔ ConflictAt b o t p

theorem merge3_spec {b o t : State} (h : ∀ p, ¬ ConflictAt b o t p) :
    MergeSpec b o t (merge3State b o t)

theorem merge3_unique {b o t m : State} (hm : MergeSpec b o t m) : m = merge3State b o t

theorem merge3_comm (b o t : State) : merge3 b o t = merge3 b t o

theorem merge3_base_left (b t : State) : merge3 b b t = fun p => .ok (t p)

theorem merge3_base_right (b o : State) : merge3 b o b = fun p => .ok (o p)

theorem merge3_idem (b x : State) : merge3 b x x = fun p => .ok (x p)

theorem merge_no_conflict_of_disjoint {b o t : State} {Wo Wt : Access}
    (ho : ChangesWithin Wo b o) (ht : ChangesWithin Wt b t) (hd : meetsSet Wo Wt = false)
    (p : Path) : merge3 b o t p ≠ .conflict
```

Listing 26: Laws of the three-way merge (`YonedaTx/Merge.lean`)

Proof. Parts 1, 2, 4 and 5 are case analyses on the three equalities among $B(p)$, $O(p)$ and $T(p)$ at each path. For part 3, a state with the property cannot exist when some path is a conflict path, since it would equal both $O(p)$ and $T(p)$ there; so no hypothesis is needed, and elsewhere the property fixes every value. For part 6, a conflict path differs from B on both sides, so it is covered by both W_o and W_t, and two sets that cover one path meet. □

Uniqueness is the sense in which Part I's description of the merged state determines it. Commutativity and idempotence hold here because a location is merged as a single value. They do not hold for positional `diff3` in general: Khanna, Kunal and Pierce show that idempotence and stability fail, and that the obvious formulations of the intuition that edits to well-separated regions never conflict are wrong [8]. The laws are therefore laws of the model's merge, and they transfer to an implementation only as far as its merge agrees with the model's on the inputs it sees.

4.2 Line-level refinement

Theorem 4.2 (Line-level refinement)**.** *Let f be a file path, and suppose that for every line key ℓ, $O(f \cdot \ell) = B(f \cdot \ell)$ or $T(f \cdot \ell) = B(f \cdot \ell)$. Then for every ℓ, the path $f \cdot \ell$ is not a conflict path, the merge there is $\delta(B,T,O)(f \cdot \ell)$, and $\delta(B,T,O)(f \cdot \ell) = \delta(B,O,T)(f \cdot \ell)$.*

```
theorem line_merge_refines {b o t : State} {f : Path}
    (hdisj : ∀ ℓ, lines o f ℓ = lines b f ℓ ∨ lines t f ℓ = lines b f ℓ) (ℓ : String) :
    ¬ ConflictAt b o t (f ++ [ℓ]) ∧
    merge3 b o t (f ++ [ℓ]) = .ok (applyDiff b t o (f ++ [ℓ])) ∧
    applyDiff b t o (f ++ [ℓ]) = applyDiff b o t (f ++ [ℓ])
```

Listing 27: Line-level refinement (`YonedaTx/Merge.lean`)

Proof. At each line one side equals the base, so the line is not a conflict path. If O equals the base there, the merge takes T, which is also what $\delta(B,T,O)$ and $\delta(B,O,T)$ give. If T equals the base, all three give O. □

No hypothesis excludes a file-granular conflict, and Section 7 gives a case with both: the whole-file contents differ on both sides, but each line changed on at most one side. A pair of edits declared at file granularity is then validated as Merge, and treating the file as a single location would report a conflict. With lines as child locations, the merge is clean and keeps both edits, which is Part I's Proposition 6.5 in the case that motivates it. The theorem requires stable keys, so it says nothing about insertions that shift later lines.

4.3 Patch compatibility

Theorem 4.3 (Merged landing as a serial step)**.** *Let $m = (c, B)$ be a member and H a state such that no path is a conflict path of $(B, H, f(B))$. Then* $\mathrm{M}(B, H, f(B)) = f(H)$ *if and only if* $\mathrm{PC}(m, H)$.

```
theorem merge_serial_iff (m : Member) (H : State)
    (hnc : ∀ p, ¬ ConflictAt m.base H (m.txn.f m.base) p) :
    merge3State m.base H (m.txn.f m.base) = m.txn.f H ↔ PatchCompatible m H
```

Listing 28: The exact condition for a merged landing (`YonedaTx/Batch.lean`)

Proof. By Lemma 2.10, $\mathrm{M}(B, H, f(B)) = \delta(B, f(B), H)$. Off $\mathrm{cov}(\mathsf{W})$, the frame condition gives $f(B)(p) = B(p)$, so the diff returns $H(p)$, and it gives $f(H)(p) = H(p)$; the two sides agree for every transaction. On $\mathrm{cov}(\mathsf{W})$, equality of the two sides is the definition of $\mathrm{PC}(m, H)$. □

The theorem makes patch compatibility exact for a single member: a conflict-free merged landing is a serial step of its transaction precisely when the condition holds. Members validated as FastForward or Rebase never need it, and for changesets it has a direct reading.

Proposition 4.4 (Clean members and changesets)**.** *If $B \equiv_{\mathsf{R}\cup\mathsf{W}} H$, then $\mathrm{PC}((c, B), H)$. For a changeset, $\mathrm{PC}((\chi(\mathsf{R}, \mathsf{W}, \Phi), B), H)$ holds if and only if $H(p) = B(p)$ for every $p \in \mathrm{cov}(\mathsf{W})$ with $\Phi(p) = B(p)$.*

```
theorem patchCompatible_of_agree {m : Member} {H : State}
    (h : Agree (m.txn.R ++ m.txn.W) m.base H) : PatchCompatible m H

theorem changeset_patchCompatible_iff {R W : Access} {Φ B H : State} :
    PatchCompatible ⟨Txn.changeset R W Φ, B⟩ H ↔
      ∀ p, covered W p = true → Φ p = B p → H p = B p
```

Listing 29: Patch compatibility for clean members and for changesets (`YonedaTx/Batch.lean`)

Proof. For the first claim, take $p \in \mathrm{cov}(\mathsf{W})$. Agreement gives $H(p) = B(p)$, and the reads condition gives $f(H)(p) = f(B)(p)$. The diff returns $f(B)(p)$ when the change moved p, and $H(p) = B(p) = f(B)(p)$ when it did not. For the second, the changeset gives $f(H)(p) = \Phi(p)$ on $\mathrm{cov}(\mathsf{W})$, and the diff gives $\Phi(p)$ where $\Phi(p) \neq B(p)$ and $H(p)$ where $\Phi(p) = B(p)$. The two agree exactly when $\Phi(p) = B(p)$ implies $H(p) = \Phi(p) = B(p)$. □

A changeset is patch-compatible when it never writes a stale base value back over a path the head moved. A changeset carries content for every path of its write set. Where that content equals the base, the change did not touch the path: the merge then keeps whatever the head holds, while running the changeset writes the base value back. The condition says these two readings never diverge. A direct consequence follows from the second claim, by a one-line argument outside Lean: if every path covered by W receives content different from its base content, then every conflict-free merged changeset is patch-compatible. Part I's touched paths contain the changed paths (Part I, Definition 4.2), but equality of the two sets is not established there, so we do not claim the consequence for the implementation.

Proposition 4.5 (Patch compatibility cannot be dropped)**.** *There are an initial state, a one-entry log and a changeset m with $\mathsf{R} = \varnothing$ such that m is validated as* Merge *with no conflict path, the one-member batch (m) is accepted, the batch state differs from running m on the head, and m is not patch-compatible with the head.*

```
def cx : Txn := Txn.changeset [] [pX] (setAt s0 pX (some 1))

def m : Txn := Txn.changeset [] [pX, pY] (setAt s0 pY (some 1))

def mlog : Log := [⟨cx, 0⟩]

theorem merge_not_serial :
    let ms : List Entry := [⟨m, 0⟩]
    let H := snap s0 MergeCE.mlog 1
    validateAt MergeCE.mlog 1 m.R m.W 0 = .merge ∧ m.R = [] ∧
```

```
    BatchAccepted s0 MergeCE.mlog ms ∧
    landBatch H (ms.map (toMember s0 MergeCE.mlog)) ≠ serialTxns H [m] ∧
    ¬ PatchCompatible (toMember s0 MergeCE.mlog ⟨m, 0⟩) H
```

Listing 30: Patch compatibility cannot be dropped (`YonedaTx/Soundness.lean`)

Proof. Let $x = y = 0$ initially. The entry c_x sets x to 1 as a fast-forward, so the head has $x = 1$ and $y = 0$. The changeset m, built on version 0, declares $\mathsf{W} = \{x, y\}$ and $\mathsf{R} = \varnothing$, and carries $x = 0$ and $y = 1$. Its write set meets the later write $\{x\}$ and its empty read set meets nothing, so validation returns Merge. The change differs from its base only at y, which the head did not move, so there is no conflict path, and a single member satisfies the pairwise rule. The merge keeps the head's $x = 1$ and sets $y = 1$. Running m on the head gives $x = 0$ and $y = 1$. Patch compatibility fails at x, where $\Phi(x) = B(x) = 0$ but $H(x) = 1$. Lean evaluates each claim. □

The example matters because it defeats the two conditions one would try first. One might expect a blind write to be safe, since it reads nothing that could be stale, and one might expect a merged path outside the read set to be safe, since no read of it can be invalidated. The member m is both, and its merged landing is still not the serial execution of m on the head. The landed state is arguably the one the writer wanted: it keeps c_x's update to x, which m never meant to change, and no update is lost. But it is not the state of serial execution in landing order (c_x, then m), because the transaction m says that x becomes 0. It is the state of the other serial order, m then c_x: the merge places m's stale write of x before c_x's, as Thomas's write rule does for obsolete writes [5, Section 4.5]. The theorem shows only that the merged landing differs from serial execution in ticket order; it does not show that no serial order explains it. A serializability statement in ticket order about merged landings must therefore either describe them as patches, as the serial merge of Definition 2.13 does, or assume patch compatibility.

5 Batch landing soundness

Theorem 5.1 (Batch landing soundness)**.** *Let $\mathcal{L}$ be a log of n entries with no Conflict landing, let $H = S_n$, and let $e_1, \ldots, e_k$ be an accepted batch at H with $e_i = (c_i, b_i)$ and members $m_i = (c_i, S_{b_i})$. Suppose every member validated as* Merge *is patch-compatible with H. Then:*

1. *the batch state* $\mathrm{oct}(H; \mu_H(m_1), \ldots, \mu_H(m_k))$ *equals the serial execution $f_k(\cdots f_1(H)\cdots)$ of the members in ticket order;*
2. *the final snapshot of the log $\mathcal{L}$ followed by $e_1, \ldots, e_k$ equals the batch state;*
3. *the declared-access serialization graph of that log has no cycle.*

```
theorem batch_landing_sound (hlog : NoConflict log) (hacc : BatchAccepted init log ms)
    (hpatch : ∀ e ∈ ms, validateAt log log.length e.txn.R e.txn.W e.base = .merge →
      PatchCompatible (toMember init log e) (snap init log log.length)) :
    let H := snap init log log.length
    let batch := landBatch H (ms.map (toMember init log))
    batch = serialTxns H (ms.map (·.txn)) ∧
    snap init (log ++ ms) (log.length + ms.length) = batch ∧
    ∀ i, ¬ SReach (log ++ ms) i i
```

Listing 31: Batch landing soundness (`YonedaTx/Soundness.lean`)

Proof. The proof has four steps, each a lemma of the development.

Disjoint writes and no conflict paths. The write half of the pairwise rule gives $\mathsf{W}_i \not\bowtie \mathsf{W}_j$ for $i \neq j$. Every accepted member has no conflict path against H: a Merge member by acceptance, and a FastForward or Rebase member because its base and H agree on W (Theorem 3.2), so the head never moved a path the change wrote.

The octopus merge is sequential landing. With pairwise disjoint write sets, the batch state equals landing the members one at a time in ticket order. The proof is by induction on the members, at each path p. If the first member does not cover p, its merged change leaves $H(p)$, and every later member sees the same head value at p. If it covers p, no later member does, so their merged changes keep the head at p, and both sides take the first member's value.

Sequential landing is the serial merge. Each sequential landing is conflict-free, so it installs the three-way merge. At a path that a later member writes, the earlier members left the head unchanged, so a conflict path against the intermediate state would already be one against H.

The serial merge is serial execution. By induction, with patch compatibility for every member: Merge members by hypothesis, and the others by Proposition 4.4. The first member's merge equals $f_1(H)$ by Theorem 4.3. For a later member m_j, the rule gives $(\mathsf{R}_j \cup \mathsf{W}_j) \not\bowtie \mathsf{W}_1$, and f_1 changes only $\mathrm{cov}(\mathsf{W}_1)$, so H and $f_1(H)$ agree on $\mathsf{R}_j \cup \mathsf{W}_j$. Hence m_j has no conflict path against $f_1(H)$, and patch compatibility transfers: on $\mathrm{cov}(\mathsf{W}_j)$, $f_j(f_1(H))(p) = f_j(H)(p) = \delta(B_j, f_j(B_j), H)(p) = \delta(B_j, f_j(B_j), f_1(H))(p)$, using the reads condition, patch compatibility at H, and $f_1(H)(p) = H(p)$.

For part 2, the snapshot after j members of the extended log is the sequential landing of the first j members on H, since each member's base lies in $\mathcal{L}$. For part 3, the extended log has no Conflict landing: entries of $\mathcal{L}$ keep their later write sets, and member j is validated against $\mathrm{Later}_{\mathcal{L}}(b_j, n)$ followed by the write sets of members $1, \ldots, j-1$, none of which meets R_j by the rule. Proposition 3.7 then gives acyclicity. □

```
theorem landBatch_eq_seqLand (ms : List Member) (hW : WritesDisjoint ms) (H : State) :
    landBatch H ms = seqLand H ms

theorem seqLand_eq_serialMerge (ms : List Member) (hW : WritesDisjoint ms) (H : State)
    (hnc : ∀ m ∈ ms, ∀ p, ¬ ConflictAt m.base H (m.txn.f m.base) p) :
    seqLand H ms = serialMerge H ms

theorem serialMerge_eq_serialTxns (ms : List Member)
    (hrule : (ms.map (·.txn)).Pairwise BatchRule) (H : State)
    (hnc : ∀ m ∈ ms, ∀ p, ¬ ConflictAt m.base H (m.txn.f m.base) p)
    (hpc : ∀ m ∈ ms, PatchCompatible m H) :
    serialMerge H ms = serialTxns H (ms.map (·.txn))
```

Listing 32: The three steps of the headline proof (`YonedaTx/Batch.lean`)

```
theorem snap_append_members (hb : ∀ e ∈ ms, e.base ≤ log.length) :
    ∀ j, j ≤ ms.length → snap init (log ++ ms) (log.length + j) =
      seqLand (snap init log log.length) ((ms.take j).map (toMember init log))

theorem noConflict_append_batch (hlog : NoConflict log) (hacc : BatchAccepted init log ms) :
    NoConflict (log ++ ms)
```

Listing 33: The batch as a log (`YonedaTx/Soundness.lean`)

Part I appends a batch as one entry, whose read set is the union of the members' read sets and whose base is the smallest member base (Part I, Definition 4.16). The model's entries carry one transaction each, so it represents the batch by the members appended individually in ticket order,

as Part I's proof of Proposition 7.4(2) does, and part 2 shows that the final snapshot of that log is the state of the single batch entry. Part 3 is a statement about declared accesses. For a member validated as Merge, the dependence on the head that its three-way merge introduces is not an edge, and the serializability content for that member is part 1 together with patch compatibility.

Corollary 5.2 (Merge form and clean batches)**.** *Under the hypotheses of Theorem 5.1 without patch compatibility, the batch state equals the serial merge of the members over H in ticket order, and parts 2 and 3 hold. If every member validated as* FastForward *or* Rebase*, the batch state equals serial execution in ticket order, which is Part I, Proposition 7.4.*

```
theorem batch_landing_merged (hlog : NoConflict log) (hacc : BatchAccepted init log ms) :
    let H := snap init log log.length
    let batch := landBatch H (ms.map (toMember init log))
    batch = serialMerge H (ms.map (toMember init log)) ∧
    snap init (log ++ ms) (log.length + ms.length) = batch ∧
    ∀ i, ¬ SReach (log ++ ms) i i

theorem batch_landing_serial (hacc : BatchAccepted init log ms)
    (hclean : ∀ e ∈ ms, (validateAt log log.length e.txn.R e.txn.W e.base).clean = true) :
    landBatch (snap init log log.length) (ms.map (toMember init log)) =
      serialTxns (snap init log log.length) (ms.map (·.txn))
```

Listing 34: Batch landing in merge form, and for clean members (`YonedaTx/Soundness.lean`)

The merge form is a statement about the installed state, not by itself a serializability statement. Its first two steps use only disjoint write sets and the absence of conflict paths, so the octopus merge equals the serial merge for the write-skew batch below as well. The serial merge applies each member's change, computed on that member's own base, to the state so far. Patch compatibility and the read half of the rule are what turn it into serial execution of the transactions. The state equality in merge form needs only the write half of the rule; the read half is needed for serial execution and for acyclicity.

Proposition 5.3 (The pairwise rule is needed; Part I, Theorem 7.3)**.** *There is a log and a two-member batch, both members accepted and validated as* Rebase*, that violates the pairwise rule, whose batch state equals neither serial order of its members, and whose extended log has a cycle in its serialization graph.*

```
def s0 : State := fun p => if p = pX then some 0 else if p = pY then some 0 else none

def c0 : Txn := Txn.write1 [] pZ (fun _ => some 1) (fun _ _ _ => rfl)

def ca : Txn := Txn.write1 [pX] pY (fun s => some ((s pX).getD 0 + 1))
  (fun s t h => by rw [h pX (by decide)])

def cb : Txn := Txn.write1 [pY] pX (fun s => some ((s pY).getD 0 + 1))
  (fun s t h => by rw [h pY (by decide)])

def log : Log := [⟨c0, 0⟩]
```

Listing 35: The write-skew history (`YonedaTx/Batch.lean`)

```
theorem batch_write_skew :
    let H := snap s0 log 1
    let B := snap s0 log 0
```

```
    let batch := landBatch H [⟨ca, B⟩, ⟨cb, B⟩]
    validateAt log 1 ca.R ca.W 0 = .rebase ∧
    validateAt log 1 cb.R cb.W 0 = .rebase ∧
    meetsSet ca.W cb.W = false ∧
    meetsSet ca.R cb.W = true ∧ meetsSet cb.R ca.W = true ∧
    batch ≠ serialTxns H [ca, cb] ∧
    batch ≠ serialTxns H [cb, ca] ∧
    ca.f H pY ≠ B pY ∧ cb.f H pX ≠ B pX ∧
    validate (laterWrites log 0 1 ++ [ca.W]) cb.R cb.W = .conflict ∧
    validate (laterWrites log 0 1 ++ [cb.W]) ca.R ca.W = .conflict
```

Listing 36: Part I, Theorem 7.3, in the model (`YonedaTx/Batch.lean`)

```
theorem batch_rule_needed :
    let ms : List Entry := [⟨ca, 0⟩, ⟨cb, 0⟩]
    let H := snap s0 WriteSkew.log 1
    let batch := landBatch H (ms.map (toMember s0 WriteSkew.log))
    (∀ e ∈ ms, MemberAccepted s0 WriteSkew.log e) ∧
    ¬ ms.Pairwise (fun ei ej => BatchRule ei.txn ej.txn) ∧
    batch ≠ serialTxns H [ca, cb] ∧ batch ≠ serialTxns H [cb, ca] ∧
    SReach (WriteSkew.log ++ ms) 1 1
```

Listing 37: The pairwise rule cannot be dropped (`YonedaTx/Soundness.lean`)

Proof. Initially $x = y = 0$. The entry c_0 writes only z and lands as a fast-forward. Members c_a (reads x, sets $y := x + 1$) and c_b (reads y, sets $x := y + 1$) are built on version 0. Each is validated against Later $= (\{z\})$, which meets neither its read set nor its write set, so both are Rebase and accepted. The rule fails because $\mathsf{R}_b = \{y\}$ meets $\mathsf{W}_a = \{y\}$. The batch merges each change against the head and installs $x = 1$ and $y = 1$. Serial execution in the order c_a, c_b gives $y = 1$ and $x = 2$; in the order c_b, c_a it gives $x = 1$ and $y = 2$. In the extended log, c_a read x at version 0 before c_b wrote it at version 3, and c_b read y at version 0 before c_a wrote it at version 2, so there are read-write edges in both directions. □

The cycle consists of two read-write anti-dependency edges, which is phenomenon G2-item in Adya's classification [1, Section 3.2.4], and the history has the shape of write skew A5B [4]. Fekete and colleagues show that every non-serializable history under snapshot isolation contains two consecutive anti-dependency edges between concurrent transactions [7, Theorem 2.1]; this pair is the smallest instance. The development also proves Part I's Theorem 7.3 itself for this history, including the facts that landing the two members one at a time would refuse the second as a Conflict. The history is built in the model. It is not a statement about the code at the revision Part I names, and the adapter's rule of deferring members with overlapping merged paths is modelled as disjoint declared write sets.

6 Under-declared reads

Part I's guarantees hold relative to declared reads. A writer that reads a location without declaring it gets no protection for that read. This section makes precise what survives. The transaction's own set R now stands for its true read set R^*, and validation uses a separately declared set D_v for each entry, which may miss part of R^*.

Definition 6.1 (Declared reads and the write check)**.** A log is *validated with declared sets* D when every entry has $b_v \le v - 1$ and $\mathrm{val}(\mathrm{Later}(b_v, v - 1), D_v, \mathsf{W}_v)$ is FastForward or Rebase. A log passes

the *write check* when every entry has $b_v \le v-1$ and no conflict path of $(S_{b_v}, S_{v-1}, f_v(S_{b_v}))$. The *undeclared part* of a true read set is $\mathsf{R}^* \setminus D$.

Validation with any declared sets implies the write check, because a clean outcome means that the head agrees with the base on the write set.

```
abbrev Declared := Nat → Access

def ValidatedDecl (log : Log) (D : Declared) : Prop :=
  ∀ i e, log[i]? = some e → e.base ≤ i ∧ (validateAt log i (D i) e.txn.W e.base).clean = true

def WriteChecked (init : State) (log : Log) : Prop :=
  ∀ j e, log[j]? = some e → e.base ≤ j ∧
    ∀ p, ¬ ConflictAt (snap init log e.base) (snap init log j) (e.txn.f (snap init log e.base)) p

def undeclared (Rstar R : Access) : Access := Rstar.filter fun x => decide (x ∉ R)
```

Listing 38: Declared reads and the write check (`YonedaTx/UnderDeclared.lean`)

Theorem 6.2 (No lost update)**.** *Under the write check, a landing that changes the head at p writes p from a base that held the head's value at p. Hence, for $i \le v$, either $S_v(p) = S_i(p)$, or some entry u with $i < u \le v$ covers p, has $S_{b_u}(p) = S_{u-1}(p)$, and changes p.*

```
theorem change_seen (hw : WriteChecked init log) {j : Nat} {e : Entry} (he : log[j]? = some e)
    {p : Path} (hch : snap init log (j + 1) p ≠ snap init log j p) :
    covered e.txn.W p = true ∧ snap init log e.base p = snap init log j p

theorem under_declared_no_lost_update (hw : WriteChecked init log) {i : Nat} {p : Path} :
    ∀ v, i ≤ v → snap init log v p = snap init log i p ∨
      ∃ j e, i ≤ j ∧ j < v ∧ log[j]? = some e ∧ covered e.txn.W p = true ∧
        snap init log e.base p = snap init log j p ∧
        snap init log (j + 1) p ≠ snap init log j p
```

Listing 39: No lost update under the write check (`YonedaTx/UnderDeclared.lean`)

Proof. If the landing at u changes p, the change must differ from its base at p, since otherwise the merge keeps the head; the frame condition then puts p in $\mathrm{cov}(\mathsf{W}_u)$. If the head had also moved p away from the base, the absence of a conflict path would force the head and the change to agree at p, and the merge would again keep the head. So the base held the head's value. The second claim follows by induction on v. □

No read set appears in the hypothesis. The file-level write check alone ensures that a value is replaced only by a landing whose base saw it, which is the absence of lost updates in the sense of phenomenon P4 [4]. The comparison is by value, not by version, so a history in which a path returns to an earlier value between the base and the head counts as seen.

Proposition 6.3 (Serializability can fail)**.** *There is a two-entry log, validated with declared sets and passing the write check, whose undeclared reads meet an intervening write and whose final state equals neither serial order. Validated with its true read set, the second entry would be a Conflict.*

```
def ulog : Log := [⟨cb, 0⟩, ⟨ca, 0⟩]

def D : Declared := fun i => if i = 0 then [pY] else []
```

```
theorem under_declared_counterexample :
    ValidatedDecl ulog D ∧ WriteChecked s0 ulog ∧
    meetsSet (undeclared ca.R (D 1)) cb.W = true ∧
    snap s0 ulog 2 ≠ serialTxns s0 [cb, ca] ∧
    snap s0 ulog 2 ≠ serialTxns s0 [ca, cb] ∧
    validateAt ulog 1 ca.R ca.W 0 = .conflict
```

Listing 40: Serializability fails under under-declared reads (`YonedaTx/UnderDeclared.lean`)

Proof. Use the write-skew transactions. The entry c_b (declares and reads y, sets $x := y + 1$) lands first, then c_a (reads x, declares nothing, sets $y := x + 1$), both built on version 0. Validation of c_a against Later $= (\{x\})$ with $D = \varnothing$ returns Rebase. The final state is $x = 1$, $y = 1$, while the serial orders give $y = 2$ or $x = 2$. With $\mathsf{R} = \{x\}$, validation of c_a returns Conflict. □

Theorem 6.4 (A sufficient condition)**.** *Let a log be validated with declared sets D. If, for every entry e_v, the undeclared part $\mathsf{R}^*_v \setminus D_v$ meets no set in* Later$(b_v, v-1)$*, then the conclusions of Theorem 3.3 hold for the true read sets.*

```
theorem under_declared_safe {D : Declared} (hv : ValidatedDecl log D)
    (hsafe : ∀ i e, log[i]? = some e →
      ∀ W ∈ laterWrites log e.base i, meetsSet (undeclared e.txn.R (D i)) W = false) :
    (∀ n, snap init log n = serial init log n) ∧
    (∀ i e, log[i]? = some e → Agree e.txn.R (snap init log e.base) (serial init log i))
```

Listing 41: A sufficient condition (`YonedaTx/UnderDeclared.lean`)

Proof. It suffices that each entry's $\mathsf{R}^*_v \cup \mathsf{W}_v$ meets no later write set, the hypothesis through which the proof of Theorem 3.3 goes. The write set meets none because the outcome is clean. A true read x that meets a later write set is either declared, which validation refuses, or undeclared, which the hypothesis excludes. □

Theorem 6.5 (Tightness)**.** *Let $\mathsf{R}^*, D, \mathsf{W}_i, \mathsf{W}_j$ be access sets with W_j non-empty, such that $\mathsf{R}^* \setminus D$ meets W_i while D and W_j do not. Then there are transactions c_i with write set W_i and c_j with true read set R^* and write set W_j such that the log $(c_i, 0), (c_j, 0)$ is validated with declared sets $(\varnothing, D)$ and its final state equals neither serial order.*

```
theorem under_declared_tight {Rstar D Wi Wj : Access}
    (hmeet : meetsSet (undeclared Rstar D) Wi = true)
    (hD : meetsSet D Wi = false) (hW : meetsSet Wj Wi = false) (hne : Wj ≠ []) :
    ∃ ci cj : Txn, ci.W = Wi ∧ cj.R = Rstar ∧ cj.W = Wj ∧
      ValidatedDecl [⟨ci, 0⟩, ⟨cj, 0⟩] (fun i => if i = 0 then [] else D) ∧
      snap emptyState [⟨ci, 0⟩, ⟨cj, 0⟩] 2 ≠ serialTxns emptyState [ci, cj] ∧
      snap emptyState [⟨ci, 0⟩, ⟨cj, 0⟩] 2 ≠ serialTxns emptyState [cj, ci]
```

Listing 42: Tightness of the sufficient condition (`YonedaTx/UnderDeclared.lean`)

Proof. Choose $p \in \mathrm{cov}(\mathsf{W}_i) \cap \mathrm{reg}(\mathsf{R}^*)$, which exists because an undeclared read meets W_i, and let q be the first element of W_j. Since $\mathsf{W}_j \not\bowtie \mathsf{W}_i$, W_j does not cover p. Let c_i set $p := q + 1$ and c_j set $q := p + 1$, starting from the empty state. Validation of c_j against (W_i) with declared set D returns Rebase, the landed state has $p = 1$ and $q = 1$, and the serial orders give $q = 2$ and $p = 2$ respectively. □

The tightness result is existential. Whenever the sufficient condition fails in the stated way, some transactions with exactly those sets break serializability. It does not say that every such transaction does: a particular transaction whose true read is irrelevant to its output may remain serializable, because the read set bounds dependence from above. Theorem 6.4 therefore gives the weakest sufficient condition on access sets alone. It does not characterize the serializable histories of particular transactions.

7 Non-vacuity

A theorem whose hypotheses can never hold is true and says nothing. The risk is real here. With a file modelled as one location, two different edits to one file always conflict, so a conflict-free Merge member could only be degenerate: identical changes on both sides, or declared sets that overlap while the changes touch different locations. An earlier version of the development had no constructed accepted batch at all, and the adversarial review of that version (Section 10) observed that the headline's hypotheses were satisfiable only on paper. The witnesses below are closed Lean proofs that the hypotheses of the main positive theorems hold in nontrivial instances, with non-empty logs, bases below the head, and a Merge member that combines two edits to one file.

The line-level witness has a file F with lines $F/1 = 10$ and $F/2 = 20$, and a second file G. The entry c_0 sets $F/1 := 11$ and declares the whole file F; it lands as a fast-forward. The member m_1, built on version 0, sets $F/2 := 21$ and also declares F. Validated against c_0, it is a Merge. The file-granular merge of F conflicts, since both sides changed F to different contents, but no path is a conflict path, each line is unchanged on one side, and the landing keeps both edits. Declared at line granularity, the same pair would be a Rebase.

```
def s0 : State := fun p => if p = fL1 then some 10 else if p = fL2 then some 20 else none

def c0 : Txn := Txn.patch [] [fF] [fL1] (fun _ => some 11) coarsens_F1

def m1 : Txn := Txn.patch [] [fF] [fL2] (fun _ => some 21) coarsens_F2

def m2 : Txn := Txn.write1 [gG] gG (fun s => some ((s gG).getD 0 + 1))
  (fun s t h => by rw [h gG (by decide)])

def log : Log := [⟨c0, 0⟩]

def ms : List Entry := [⟨m1, 0⟩, ⟨m2, 0⟩]

theorem line_merge_witness :
    let H := snap s0 log 1
    let C := m1.f s0
    validateAt log 1 m1.R m1.W 0 = .merge ∧
    validate [[fL1]] [] [fL2] = .rebase ∧
    FileConflict s0 H C fF ∧
    (∀ p, ¬ ConflictAt s0 H C p) ∧
    (∀ ℓ, lines H fF ℓ = lines s0 fF ℓ ∨ lines C fF ℓ = lines s0 fF ℓ) ∧
    landOn s0 H C fL1 = some 11 ∧ landOn s0 H C fL2 = some 21
```

Listing 43: The line-level merge witness (`YonedaTx/Witnesses.lean`)

The batch witness adds a member m_2 that reads G and increments it. The batch (m_1, m_2) is accepted at the head of the log (c_0): m_1 is a conflict-free Merge, and m_2, which neither reads nor writes F, validates cleanly. The Merge member is patch-compatible, because the patch never sets a line to its base content. Theorem 5.1 applies, and the batch state equals serial execution, with

$F/1 = 11$, $F/2 = 21$ and $G = 1$.

```
theorem batch_witness :
    let H := snap s0 log 1
    log ≠ [] ∧ NoConflict log ∧ BatchAccepted s0 log ms ∧
    validateAt log 1 m1.R m1.W 0 = .merge ∧
    PatchCompatible (toMember s0 log ⟨m1, 0⟩) H ∧
    landBatch H (ms.map (toMember s0 log)) = serialTxns H [m1, m2] ∧
    serialTxns H [m1, m2] fL1 = some 11 ∧ serialTxns H [m1, m2] fL2 = some 21 ∧
    serialTxns H [m1, m2] gG = some 1
```

Listing 44: The batch witness (`YonedaTx/Witnesses.lean`)

Three smaller witnesses use the log (c_0, m_2) in which m_2 is built on version 0 and lands at version 2 as a Rebase. The log is clean, so Theorem 3.3 applies with a base below its version. Between versions 0 and 1 the file F changes while the snapshots agree on G; the proof obtains the agreement from the frame lemma, because G meets no write set in between. And with m_2's read of G undeclared, the log validated with empty declared sets equals serial execution; the proof obtains this from Theorem 6.4, after checking that the undeclared read meets no later write. In both cases the premise is established inside the proof, and the statement records the conclusion.

```
def log2 : Log := [⟨c0, 0⟩, ⟨m2, 0⟩]

theorem serializability_witness :
    Validated log2 ∧ validateAt log2 1 m2.R m2.W 0 = .rebase ∧
    snap s0 log2 2 = serial s0 log2 2 ∧ serial s0 log2 2 gG = some 1 ∧
    serial s0 log2 2 fL1 = some 11

theorem frame_witness :
    snap s0 log2 1 fL1 ≠ snap s0 log2 0 fL1 ∧ Agree [gG] (snap s0 log2 0) (snap s0 log2 1)

theorem under_declared_safe_witness :
    ValidatedDecl log2 (fun _ => []) ∧ undeclared m2.R [] = [gG] ∧
    snap s0 log2 2 = serial s0 log2 2
```

Listing 45: Witnesses for serializability, frame and under-declared safety (`YonedaTx/Witnesses.lean`)

The counterexamples of Propositions 4.5, 5.3 and 6.3 are concrete histories as well. Each is evaluated by the Lean kernel, and each establishes its hypotheses, such as acceptance and validation outcomes, as part of its statement.

8 Findings from the formalization

Four statements changed when the model was written down in Lean. Each corrects or sharpens a statement of Part I.

8.1 The merge rule and idempotence

Part I's proof of its Theorem 6.4 describes the three-way merge as taking, at each path, the side that differs from the base. The first formal definition followed that description: take theirs if ours equals the base, ours if theirs equals the base, and a conflict otherwise. Under that rule, merging two identical changes is a conflict at every path where they differ from the base, so the merge of (B, X, X) is not X and idempotence fails. The corrected rule adds the clause of `diff3` and Git:

when both sides made the same change, the result is that change. With it, a path is a conflict exactly when both sides changed it to different values (Proposition 4.1, part 1), which is how `diff3` and Git characterize a conflict. The failure under the first rule is a one-line computation and is not a Lean theorem; the development contains only the corrected rule. Part I's results do not depend on the missing clause, because in the case its proof treats only one side differs from the base. The description in that proof should nevertheless read: the side that differs from the base, or the common value when both sides made the same change. Two agents that apply the same fix to one file would otherwise be told they conflict.

8.2 The frame lemma and its two regions

Part I states the frame lemma with agreement on $\bigcup_{x \in X} \mathrm{sub}(x)$ and confines each landing's changes to $\bigcup_{w \in \mathsf{W}} \mathrm{sub}(w)$ (Part I, Lemma 6.3 and Assumption 6.2). The formalization shows that the two regions cannot both be the overlap region. Take $X = \{a/b\}$ and $\mathsf{W} = \{a/c\}$, which do not overlap. If a write with this W were allowed to change any path in $\mathrm{reg}(\mathsf{W})$, it could change the ancestor a. But $a \in \mathrm{reg}(X)$, so the snapshots before and after would disagree on X although X meets no later write set, and the lemma would be false. The model therefore takes agreement on $\mathrm{reg}(X)$, which a model that gives content to every node needs, and confines writes to $\mathrm{cov}(\mathsf{W})$. This is again a direct argument, not a Lean theorem; the Lean frame lemma is the corrected statement.

Part I's lemma remains correct as stated, because its agreement excludes ancestors and its directories carry no content. The finding is that this choice carries weight. In Git, a directory has content of its own, the identifier of its tree, and that identifier changes whenever anything below it changes. A model that compared directory identifiers on reads, for instance to record that a writer listed a directory, could not keep the frame lemma with writes confined to the subtrees of the write set. Such a read must be declared as the directory prefix itself, whose subtree contains every write below it, as Part I's access sets already do.

8.3 Merged landings and patch compatibility

Part I's Proposition 7.4 excludes members validated as Merge, and its Proposition 6.5 describes the merged state without a serializability claim. The formalization extends the batch result to Merge members and shows what the extension costs. Two hypotheses suggest themselves: that the merged member is a blind write, so no read can be stale, and that its merged paths lie outside its read set. Proposition 4.5 satisfies both and is not serial in ticket order. The exact condition for one member is patch compatibility (Theorem 4.3), and under it the batch theorem covers Merge members (Theorem 5.1).

The correction to Part I concerns changesets. Part I's Definition 8.1 appends a changeset validated as Merge with the effective change produced by the three-way merge, and its Proposition 8.3(3) states that, for FastForward and Rebase, the new branch state is the previous one with Φ applied. Theorem 4.3 and Proposition 4.4 show that the effective change of a merged changeset equals Φ on W exactly when no written path whose content equals its base content was moved by the head. Statements about merged changesets that speak of applying Φ need that condition.

8.4 The premise behind undeclared reads

The question of what survives under-declared reads was first posed as dropping Part I's sound-refinement assumption (Part I, Assumption 6.2). That assumption concerns refined write sets: it requires each landing to change only locations that its refined writes cover. A writer that reads more than it declares violates a different premise, that declared reads cover true reads, which Part I

uses implicitly in the qualifier "relative to declared reads" of its Theorem 6.4. Section 6 drops that premise and keeps Assumption 6.2. Without it, no landed update is lost, serializability can fail, and serializability holds when the undeclared reads meet no later write, a condition that cannot be weakened in general.

9 Limits of the formalization

The positional three-way merge is not modelled. Lines are child locations with stable keys, so an insertion that shifts later lines, a deletion, and the alignment of hunks are outside the model. Theorem 4.2 and the merge laws hold for this abstraction. Khanna, Kunal and Pierce show that the corresponding laws fail for positional `diff3` [8], so they cannot be transferred to it without further work.

Validation works at one granularity. Refined write sets W^{ρ}, which Part I's stale-read test uses, and opaque entries, which Part I's validation treats as conflicts, are not modelled; Part I's Corollary 6.8 on symbol granularity is not formalized. The log has one ref, so entries on other refs, and Part I's restriction of Later to entries on the same ref, do not appear.

Agreement differs from Part I's, as Section 2 describes. The Lean relation compares every path in the overlap region, including ancestors and directory nodes, where Part I compares only file and symbol locations in the subtrees. Content is assigned to each node independently, so a file whose content is determined by its symbols or lines is not represented: a write declared at symbol granularity changes the symbol node and not the file node. The mappings of Part I's Lemma 6.3 and Theorem 6.4 to Lean theorems are therefore adapted, not literal, and the correspondence with Part I's relation when ancestors carry no content is an informal observation.

The log is simpler than Part I's in ways that remove failure cases rather than add guarantees. A conflicting landing falls back to the change's value instead of being refused, so the log is total. Each change is computed on the snapshot at its base, so the consistent-base assumption cannot fail. And the log starts at version 0 rather than at an arbitrary version.

The ref and projection results take as hypotheses what Part I proves of the code or assumes of the hash. Part I's append discipline (Part I, Lemma 5.3) is a hypothesis of the ref results, and so is the ordering of commit generations. In the Git projection, a commit is identified with the data it encodes, which models content addressing with no collision.

Several of Part I's results are not formalized: the rewrite cost (Proposition 4.3), replay determinism and checkpoints (Theorem 5.1), tamper evidence (Proposition 5.2), properties 1 to 3 and 5 of combiner safety (Theorem 7.2), the symbol-granularity corollary (Corollary 6.8), repairs (Definition 7.5), and the correspondence with Kung and Robinson [9] as a theorem. The generic serializability theorem of Bernstein, Hadzilacos and Goodman [5] is not formalized either; the model proves the direction Part I uses and proves serializability as a state equality.

The code is not verified. The theorems are about the model. The implementation is connected to the model by the code-evidence appendices of Parts I and III, which rest on reading code at named commits. Part I reports that the implementation's three-way merge agrees with `git merge-file` on a differential test, which is evidence about the merge and not a proof. The check that enforces the pairwise rule in the implementation is described in Part I (Part I, Section 7.3). Whether the implementation's merged landings satisfy patch compatibility is not established by this paper or by any earlier Part.

10 Trusted computing base

The development is checked by Lean 4 at version 4.35.0, fixed in the project's toolchain file, and uses core Lean only: no Mathlib and no package dependencies. It contains 180 theorem declarations, helpers included. No proof uses `sorry` or `admit`, no `axiom` is declared, and no definition is `unsafe` or `partial`.

A reader must trust three things: the Lean kernel, the axioms the proofs use, and the claim that the definitions of Section 2 mean what the prose says. The kernel rechecks every proof term, including those produced by tactics and by evaluation of decidable propositions, so tactics are outside the trusted base. The third item cannot be checked by Lean, which is why this paper shows the core definitions as listings. A few helpers appear in the listings without one: `Outcome.clean` holds for FastForward and Rebase, `Coarsens` and `CoarsensAll` say that every access has an ancestor in the coarser set, `WritesDisjoint` says that members' write sets are pairwise disjoint, and `Txn.write1`, `Txn.assign` and `Txn.patch` build transactions that set one location, or a set of locations below the declared write set, from a function of the declared reads.

The development was reviewed adversarially in two rounds. The first round asked whether each main theorem's hypotheses could hold, whether the definitions model Part I faithfully, and whether the counterexamples exhibit what they claim. It found three major problems. Validation had been described as exactly Part I's Definition 4.15, although it uses one granularity and has no opaque entries. With a file as a single location, an accepted Merge member could only be degenerate, so the headline did not reach Part I's line-level merge case. And the sufficient condition for under-declared reads had been described as an exact boundary. It also found that no concrete accepted batch had been constructed. All of these were fixed: the adapted status of validation is stated (Section 2), lines became child locations with stable keys (Theorem 4.2), the tightness result replaced the claim of exactness (Theorem 6.5), and the witnesses of Section 7 were added. The second round confirmed the fixes and raised two minor points, a miscounted number of theorems and a stale source comment about line merging, which were also fixed.

The file `AxiomsReport.lean` prints the axioms of 72 main theorems. Every one depends on a subset of `propext` and `Quot.sound`: 6 on `propext` alone and 66 on both. None depends on `Classical.choice` or on `sorryAx`. `Quot.sound` enters through function extensionality, which every equality of states uses, and through lemmas of the core simplifier. Table 1 lists the report.

Table 1: Axioms of the main theorems, as printed by `AxiomsReport.lean`, grouped by file.

File	Count	Theorems	Axioms
Paths	4	`overlap_iff_common_descendant`, `overlap_refl`, `overlap_coarsen`, `meetsSet_coarsen`	`propext`, `Quot.sound`
Paths	1	`overlap_symm`	`propext`
Merge	10	`merge3_unique`, `merge3_spec_iff`, `merge3_comm`, `merge3State_comm`, `merge3_base_left`, `merge3_base_right`, `merge3_idem`, `merge_conflict_characterization`, `merge_no_conflict_of_disjoint`, `merge3State_eq_applyDiff`	`propext`, `Quot.sound`
Merge	2	`merge3_spec`, `line_merge_refines`	`propext`
Log	3	`frame_point`, `frame_lemma`, `snap_append`	`propext`, `Quot.sound`

File	Count	Theorems	Axioms
Validation	4	`independence_iff_rebase`, `coarsening_sound`, `coarsening_preserves_frame`, `independent_of_coarsened`	`propext`, `Quot.sound`
Serializability	9	`reads_current`, `fastForward_landing`, `rebase_landing`, `serializability`, `changeset_landing`, `merged_writes`, `independence`, `serializability_of_laterDisjoint`, `no_conflict_path_of_clean`	`propext`, `Quot.sound`
Batch	12	`combiner_member_reads`, `batch_write_skew`, `batch_serializable`, `batch_sequential_validates`, `batch_serializable_log`, `landBatch_eq_seqLand`, `seqLand_eq_serialMerge`, `batch_merge_sound`, `merge_serial_iff`, `patchCompatible_of_agree`, `serialMerge_eq_serialTxns`, `patch_patchCompatible_iff`	`propext`, `Quot.sound`
Batch	1	`changeset_patchCompatible_iff`	`propext`
Graph	8	`sgraph_edge_forward`, `sgraph_acyclic`, `refs_advance`, `refs_unchanged_no_entry`, `no_lost_update`, `projection_deterministic`, `projection_stable`, `projection_chain`	`propext`, `Quot.sound`
Graph	2	`projection_first_parent`, `projection_tree`	`propext`
Soundness	6	`noConflict_append_batch`, `batch_landing_merged`, `batch_landing_serial`, `batch_rule_needed`, `batch_landing_sound`, `merge_not_serial`	`propext`, `Quot.sound`
UnderDeclared	5	`change_seen`, `under_declared_no_lost_update`, `under_declared_safe`, `under_declared_counterexample`, `under_declared_tight`	`propext`, `Quot.sound`
Witnesses	5	`Witnesses.line_merge_witness`, `Witnesses.batch_witness`, `Witnesses.serializability_witness`, `Witnesses.frame_witness`, `Witnesses.under_declared_safe_witness`	`propext`, `Quot.sound`

The build is two commands, run in the directory `lean/`.

```
lake build                          # toolchain leanprover/lean4:v4.35.0
lake env lean AxiomsReport.lean     # prints the axioms of each main theorem
```

Listing 46: Building the development and printing the axioms report

A clean build takes about 3 s of wall-clock time on the author's machine and reports no warnings.

11 Relation to Parts I to III and the synthesis

Table 2 maps each result of the series that the development touches to its Lean theorems. The status *proved* means the Lean statement is the paper's statement in the model's terms. *Adapted* means it is proved for a changed definition, such as one granularity or the stronger agreement relation. *Proved with extra hypothesis* means a hypothesis the paper does not state, or states as a property of the code, appears in the Lean statement. *Not formalized* means no Lean statement exists.

Table 2: Results of the series and the Lean theorems that formalize them.

Paper result	Lean theorems	File	Status
Part I, Definition 4.4	`anc`, `meets`, `meetsSet`, `covered`	Paths	Adapted (symbols as path components)
Part I, Lemma 4.5	`overlap_iff_common_descendant`, `overlap_refl`, `overlap_symm`, `overlap_coarsen`	Paths	Proved
Part I, Definition 4.7 (Eq. 1)	`Independent`	Validation	Proved
Part I, Definitions 4.8 to 4.11	`Entry`, `Log`, `snap`	Log	Adapted (tree states of one ref)
Part I, Definition 4.15	`validate`, `validateAt`	Validation	Adapted (one granularity, no opaque entries)
Part I, Theorem 5.4	`refs_advance`, `refs_unchanged_no_entry`	Graph	Proved with extra hypothesis (append discipline, generations)
Part I, Corollary 5.5	`no_lost_update`	Graph	Proved with extra hypothesis (append discipline)
Part I, Theorem 5.6	`snap_append`, `snap_of_take_eq`	Log	Adapted (tree states)
Part I, Assumption 6.2	`Txn.frame`, `ChangesWithin`, `snap_step_changesWithin`	State, Log	Adapted (one step, path granularity)
Part I, Lemma 6.3	`frame_lemma`, `frame_point`, `frame_of_laterWrites`	Log	Adapted (agreement)
Part I, Theorem 6.4, parts 1 to 3	`reads_current`, `fastForward_landing`, `rebase_landing`	Serializability	Adapted (agreement; no commits)
Part I, Theorem 6.4, serializability	`serializability`	Serializability	Adapted (agreement; state equality)
Part I, Proposition 6.5	`merge3_spec`, `merge3_unique`, `merged_writes`, `line_merge_refines`	Merge, Serializability	Adapted (per location; lines with stable keys)
Part I, Theorem 6.7	`independence_iff_rebase`, `independence`	Validation, Serializability	Proved (state equality in the forward direction)
Part I, Corollary 6.8	None	None	Not formalized
Part I, Proposition 6.10	`coarsening_sound`, `coarsening_preserves_frame`, `independent_of_coarsened`	Validation	Proved
Part I, Theorem 7.2, property 4	`combiner_member_reads`	Batch	Adapted (agreement)
Part I, Theorem 7.2, properties 1 to 3 and 5	None	None	Not formalized
Part I, Theorem 7.3	`batch_write_skew`, `batch_rule_needed`	Batch, Soundness	Adapted (history built in the model)
Part I, Proposition 7.4	`batch_serializable`, `batch_serializable_log`, `batch_sequential_validates`, `batch_landing_serial`	Batch, Soundness	Proved

Paper result	Lean theorems	File	Status
Part I, Proposition 7.4 with Merge members	`batch_landing_sound`, `batch_landing_merged`	Soundness	Proved with extra hypothesis (patch compatibility)
Part I, Proposition 8.3(3)	`changeset_landing`	Serializability	Proved
Part I, Proposition 8.3(5)	`projection_chain`	Graph	Proved
Part I, Assumption 8.2 and Part II, Lemma 13.2	`projection_deterministic`, `projection_stable`, `projection_first_parent`, `projection_tree`	Graph	Proved (hashing abstracted)
Part I, Theorem 5.1, Propositions 4.3 and 5.2, Definition 7.5	None	None	Not formalized
Part III, capacity model and realization	None	None	Not formalized
Synthesis, Theorem 4.4	`batch_write_skew`, `batch_landing_serial`	Batch, Soundness	Adapted (as Part I, Theorem 7.3 and Proposition 7.4)
Synthesis, Proposition 5.3, isolation part	`serializability`, `batch_landing_sound`	Serializability, Soundness	Proved for the model (throughput bound not formalized)

The results without a counterpart in Parts I to III are the exact condition for merged landings and its counterexample, the changeset criterion, line-level refinement, the merge laws other than the specification, the under-declared results and the witnesses.

Part I's results about refs, its Theorem 5.4 and Corollary 5.5, are formalized over an abstract ref log. A ref log entry names a ref and the commit it installs, and parents are a function from commits to lists of commits. Under the append discipline, which Part I proves of the code as its Lemma 5.3 and which the model takes as a hypothesis, a ref's later values reach its earlier value by first parents. If, in addition, every parent has a smaller generation, a ref with equal values at two versions is named by no entry between them. And a commit landed on a ref stays an ancestor of the ref's value.

```
def AppendDiscipline (parents : CommitId → List CommitId) (log : List RefEntry) : Prop :=
  ∀ v e, log[v]? = some e →
    refSnap log v e.ref = none ∨ fp parents e.new = refSnap log v e.ref

theorem refs_advance (hd : AppendDiscipline parents log) {u : Nat} {r : RefName} {x : CommitId}
    (hx : refSnap log u r = some x) :
    ∀ v, u ≤ v → ∃ y, refSnap log v r = some y ∧ FPChain parents y x

theorem refs_unchanged_no_entry {gen : CommitId → Nat} (hd : AppendDiscipline parents log)
    (hg : GenOk parents gen) {u v : Nat} {r : RefName}
    (heq : refSnap log u r = refSnap log v r) :
    ∀ i e, u ≤ i → i < v → log[i]? = some e → e.ref ≠ r

theorem no_lost_update (hd : AppendDiscipline parents log) {i : Nat} {e : RefEntry}
    {c : CommitId} (he : log[i]? = some e) (hc : e.new = c ∨ c ∈ parents e.new) :
    ∀ v, i + 1 ≤ v → ∃ y, refSnap log v e.ref = some y ∧ Ancestor parents y c
```

Listing 47: Refs advance along first-parent chains (`YonedaTx/Graph.lean`)

The Git projection of Part II is formalized as a function of the log prefix. Part II defines the projected commit of main at version j from the commit at version $j-1$ and the j-th changeset entry (Part II, Definition 13.1) and proves that it depends only on the prefix (Part II, Lemma 13.2). The Lean statement is that two logs with equal prefixes of length j give the same projected commit. The projected commit at j has the commit at $j-1$ as first parent when entry j exists, which discharges Part I's Assumption 8.2, and the commit at u lies on the first-parent chain of the commit at every later v, which is Part I's Proposition 8.3(5).

```
theorem projection_deterministic {log log' : List ChangesetEntry} {j : Nat}
    (hpre : log.take j = log'.take j) :
    project treeOf h log j = project treeOf h log' j

theorem projection_first_parent {log : List ChangesetEntry} {j : Nat} {e : ChangesetEntry}
    (he : log[j]? = some e) :
    (project treeOf h log (j + 1)).firstParent = some (project treeOf h log j)

theorem projection_chain {log : List ChangesetEntry} {u : Nat} :
    ∀ v, u ≤ v → GitChain (project treeOf h log v) (project treeOf h log u)
```

Listing 48: The Git projection (`YonedaTx/Graph.lean`)

For the synthesis, the theorem that matters is Proposition 5.3, which treats batch size as a capacity parameter under either of two hypotheses [13]. Under the first, members are appended one at a time, each validated against a log that contains the earlier members; that is a clean log, and Theorem 3.3 applies. Under the second, the command defers any member whose reads or writes meet an earlier member's writes; that is the pairwise rule, and Theorem 5.1 applies to the accepted members, including members validated as Merge when they are patch-compatible. The throughput part of the proposition, which rests on Part III's capacity model, is not formalized.

12 Related work

The serializability arguments are classical. Kung and Robinson's serial validation is the landing step [9]. The serialization graph and the theorem that an acyclic graph gives serializability come from Bernstein, Hadzilacos and Goodman [5]. The batch anomaly is phenomenon G2-item in Adya's classification [1], with the shape of write skew in the terms of Berenson and colleagues [4], and Fekete and colleagues characterize when histories under snapshot isolation fail to be serializable [7]. The present development instantiates these arguments for one model and checks them mechanically, with the state equality of serial execution as the conclusion rather than graph acyclicity alone.

The write-write case is where source control differs from a database, and the three-way merge is its mechanism. Khanna, Kunal and Pierce give a formal investigation of `diff3` and show that several properties one would expect of it fail in general [8]. The model here sidesteps their difficulties by merging per location and treating lines as locations with stable keys. It is therefore a statement about an idealized merge, and its laws should be read alongside their negative results. The transaction log itself follows the lakehouse formats, of which Delta Lake is the model for this series [3]. Lean 4 is described by de Moura and Ullrich [6].

The batch results have close precedents in deterministic and batched databases. Aria validates a batch of transactions against a snapshot and commits a transaction only if it has no write-after-write or read-after-write dependency on an earlier transaction in the batch; its serializability theorem gives serial execution in input order [15]. That rule is the pairwise rule of this paper, and for fast-forward and rebase members the batch result here is the same idea, machine-checked in a source-control model. Ding, Kot and Gehrke batch optimistic transactions and reorder them using an intra-batch

dependency graph [16], and they also discuss the write skew that per-member validation admits. What these systems do not have is a three-way merge: the extension to merged members, and the patch-compatibility condition it needs, has no counterpart there that we found. The merge counterexample is an instance of Thomas's write rule, under which an obsolete write is ignored and the result corresponds to a different serial order [5, Section 4.5].

On the merge side, Swierstra and Löh give a formal semantics of version control in which lines carry unique labels and independent patches merge, with a Coq model [17]. The line-level refinement here is the same abstraction. Sousa, Dillig and Lahiri verify three-way program merges against semantic conflict freedom, a property stated in terms of re-execution [18]. Mechanized proofs of isolation and transaction libraries exist for databases, for example VerIso for isolation guarantees [19] and vMVCC for a multi-version transaction library [20]. A search of these and related sources (30 sources, more than 40 queries) found no mechanized serializability proof for a Git-like source-control model with merges. Some bibliographic search services were unavailable during the search, so this is an absence of evidence, not a claim of priority.

13 Conclusion

The core of the transactional model of source control is machine-checked. In the model, a batch whose members are fast-forwards, rebases or conflict-free merges lands as the serial execution of its members in ticket order, provided no member reads or writes what an earlier member wrote and every merged member is patch-compatible with the head. Each hypothesis is necessary: write skew defeats the first, and a blind write that never read its merged path defeats the second. Patch compatibility is the exact condition for a merged landing to be a serial step, and for changesets it says that a change never writes a stale base value back over a path the head moved. Writing the model in Lean also corrected the merge rule, located the frame lemma's dependence on its two regions, and identified the premise that undeclared reads violate.

Three gaps remain between these theorems and the system. The merge is an idealization with stable line keys, not positional `diff3`. The implementation is linked to the model by reading code, not by proof. And whether the implementation's merged landings are patch-compatible is unknown; checking the condition at landing, or restricting changesets to paths whose content they change, would close that gap for changesets. Within the model, the foundation that the series' thesis needs is now proved by machine: transactions with declared read and write sets, landed in batches by one serialized authority, are serializable under conditions that can be checked at landing.

References

[1] A. Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. PhD thesis, Massachusetts Institute of Technology, March 1999. MIT-LCS-TR-786. `https://dspace.mit.edu/handle/1721.1/149899`.

[2] Nick G. and the source control team. Redesigning source control at Anthropic. Anthropic Engineering, 13 October 2026, updated 15 October 2026. `https://www.anthropic.com/engineering/redesigning-source-control`.

[3] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak, M. Świtakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia. Delta Lake: High-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411-3424, 2020. doi:10.14778/3415478.3415560.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1-10, 1995. doi:10.1145/223784.223785.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. `https://www.microsoft.com/en-us/research/people/philbe/book/`.

[6] L. de Moura and S. Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction, CADE 28*, Lecture Notes in Computer Science, pages 625-635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.

[7] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492-528, 2005. doi:10.1145/1071610.1071615.

[8] S. Khanna, K. Kunal, and B. C. Pierce. A formal investigation of diff3. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, pages 485-496. Springer, 2007. doi:10.1007/978-3-540-77050-3_40.

[9] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213-226, 1981. doi:10.1145/319566.319567.

[10] M. Long and The YonedaAI Collaboration. A transaction log for source control. Part I of *Source Code as a Transactional Abstraction*, YonedaAI Research Collective, author's manuscript, October 2026.

[11] M. Long and The YonedaAI Collaboration. Partitioned derived data. Part II of *Source Code as a Transactional Abstraction*, YonedaAI Research Collective, author's manuscript, October 2026.

[12] M. Long and The YonedaAI Collaboration. A cloud-portable architecture for serverless Git. Part III of *Source Code as a Transactional Abstraction*, YonedaAI Research Collective, author's manuscript, October 2026.

[13] M. Long and The YonedaAI Collaboration. Source code as a transactional abstraction: Foundations for transactional source control, with a serverless realization. YonedaAI Research Collective, author's manuscript, October 2026.

[14] M. Long. YonedaTx: Lean 4 formalization of the transactional model of source control. Directory `lean/` of the repository YonedaAI/yonedarepo-research, October 2026. `https://github.com/YonedaAI/yonedarepo-research/tree/main/lean`.

[15] Y. Lu, X. Yu, L. Cao, and S. Madden. Aria: A fast and practical deterministic OLTP database. *Proceedings of the VLDB Endowment*, 13:2047-2060, 2020. doi:10.14778/3407790.3407808.

[16] B. Ding, L. Kot, and J. Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment*, 12(2):169-182, 2018. doi:10.14778/3282495.3282502.

[17] W. Swierstra and A. Löh. The semantics of version control. In *Onward! 2014*, pages 43-54. ACM, 2014. doi:10.1145/2661136.2661137.

[18] M. Sousa, I. Dillig, and S. K. Lahiri. Verified three-way program merge. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1-29, 2018. doi:10.1145/3276535.

[19] S. Ghasemirad, S. Liu, C. Sprenger, L. Multazzu, and D. Basin. VerIso: Verifiable isolation guarantees for database transactions. *Proceedings of the VLDB Endowment*, 18(5):1362-1375, 2025. doi:10.14778/3718057.3718065.

[20] Y.-S. Chang, R. Jung, U. Sharma, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023. `https://www.usenix.org/conference/osdi23/presentation/chang`.

A Code evidence

Table 3 maps each Lean listing in this paper to the file and line of the declarations it quotes. Paths are relative to the root of the public repository [14]. Where a listing quotes several declarations, the line column gives the first line of each, in the order the listing shows them. Every row can be checked by opening the named file at the named lines.

Table 3: Lean listings and the declarations they quote.

Row	Claim	File	Lines	What the code shows
1	Paths, ancestry and overlap	`lean/YonedaTx/Paths.lean`	17, 20, 75, 111, 114, 117, 121	Listing 1: A path is a list of components. `anc` is component-wise prefix. `meets` is comparability under `anc`. `covered` and `region` decide membership in the subtrees of a set and in its overlap region.
2	Part I, Lemma 4.5	`lean/YonedaTx/Paths.lean`	81, 92, 95, 99	Listing 2: Overlap holds iff the two subtrees share a node. It is reflexive and symmetric, and it survives replacing an access by an ancestor.
3	States, agreement and confinement of changes	`lean/YonedaTx/State.lean`	15, 18, 23, 41	Listing 3: A state maps every path to optional content. `Agree` compares the overlap region of a set. `ChangesWithin` confines differences to covered paths.
4	A transaction carries R, W and a frame condition	`lean/YonedaTx/State.lean`	65	Listing 4: The structure has fields `R`, `W` and `f`, a `frame` proof that `f` changes only covered paths, and a `reads` proof that written values depend only on `R++W`.
5	Changesets are blind writes	`lean/YonedaTx/State.lean`	76, 80	Listing 5: `Txn.changeset` overlays Φ on the paths covered by W, whatever the state it runs on.
6	The log and its snapshots	`lean/YonedaTx/Log.lean`	19, 24, 29	Listing 6: An entry is a transaction and a base. `snap` folds the entries, landing each change computed on its (capped) base onto the previous snapshot with `landOn`.
7	Later write sets	`lean/YonedaTx/Log.lean`	102	Listing 7: `laterWriteslogbn` lists the write sets of the entries at list positions b to n-1, that is, versions b+1 to n.
8	Part I, Definition 4.15, at one granularity	`lean/YonedaTx/Validation.lean`	13, 24, 31	Listing 8: Four outcomes. Conflict when R meets a later write, FastForward when there is none, Merge when W meets one, Rebase otherwise. One list of write sets serves both tests.

Row	Claim	File	Lines	What the code shows
9	Logs with no Conflict landing, and clean logs	`lean/YonedaTx/Validation.lean`	123, 129	Listing 9: `NoConflict` requires each entry's base to precede it and its validation not to be Conflict. `Validated` requires FastForward or Rebase.
10	Three-way merge with the same-change clause	`lean/YonedaTx/Merge.lean`	32, 62, 65, 69, 221	Listing 10: `merge3Cell` returns theirs if ours is the base, ours if theirs is the base, ours if both agree, and a conflict otherwise. `landOn` installs the merge with the head as ours.
11	A conflict-free merge applies a diff	`lean/YonedaTx/Merge.lean`	173, 177	Listing 11: `applyDiffbch` takes c where c differs from b and h elsewhere. A conflict-free merge equals the diff of theirs applied to ours.
12	Lines with stable keys	`lean/YonedaTx/Merge.lean`	193, 196	Listing 12: Line ℓ of file f is the child location of f named ℓ. A file-granular conflict compares whole line maps.
13	The batch entry is an octopus merge	`lean/YonedaTx/Batch.lean`	17, 24, 29, 32	Listing 13: A member is a transaction and its base state. Each member's change is merged against the head, and `octopus` takes at each path the first merged change that differs from the head.
14	Serial forms and the pairwise rule	`lean/YonedaTx/Batch.lean`	35, 39, 43, 47, 51	Listing 14: `seqLand` lands members one at a time. `serialTxns` runs the transactions in order. `serialMerge` merges each change into the state so far. `BatchRule` is the rule of Part I, Proposition 7.4.
15	Patch compatibility	`lean/YonedaTx/Batch.lean`	152	Listing 15: On its write set, running the member on H equals its own diff against its base applied to H.
16	Acceptance of a batch	`lean/YonedaTx/Soundness.lean`	36, 46	Listing 16: A member is accepted when its base is at most the head and it validates as FastForward, Rebase, or a Merge with no conflict path. A batch also satisfies the rule pairwise in ticket order.
17	Serialization graph over declared accesses	`lean/YonedaTx/Graph.lean`	23, 30	Listing 17: Edges are write-write, write-read and read-write overlaps between entries placed at their base and landing versions. `SReach` is its transitive closure.
18	Part I, Lemma 6.3	`lean/YonedaTx/Log.lean`	154	Listing 18: If X meets no write set of the entries in (b, v], the snapshots at b and v agree on X.

Row	Claim	File	Lines	What the code shows
19	Part I, Theorem 6.4, parts 1 to 3	`lean/YonedaTx/Serializability.lean`	18, 52, 78	Listing 19: A landing that is not a Conflict read current values of R. A FastForward has base equal to the head and installs the change. A Rebase installs the change on W and the head elsewhere.
20	Part I, Theorem 6.4, serializability statement	`lean/YonedaTx/Serializability.lean`	43, 149	Listing 20: For a `Validated` log every snapshot equals serial execution in log order, and each entry's declared reads at its base equal its reads in that order.
21	Part I, Proposition 6.5, at file level	`lean/YonedaTx/Serializability.lean`	170	Listing 21: Outside W the landed state is the head. On W minus later writes it is the change. At every path without a conflict the merge is the landed value.
22	Part I, Theorem 6.7, validation part	`lean/YonedaTx/Validation.lean`	203, 222	Listing 22: Eq. 1 holds iff, in both landing orders, the second change is validated as Rebase against the first.
23	Part I, Theorem 6.7, state part	`lean/YonedaTx/Serializability.lean`	199	Listing 23: For independent changes on one base, both landing orders validate as Rebase and give the same state, which equals both serial orders.
24	Part I, Proposition 6.10, validation part	`lean/YonedaTx/Validation.lean`	169	Listing 24: Coarsening R, W and the later write sets keeps every Conflict a Conflict and every Merge a Merge or a Conflict.
25	Edges point forward and the graph is acyclic	`lean/YonedaTx/Graph.lean`	37, 59	Listing 25: In a log with no Conflict landing every edge goes from a lower to a higher position, so no node reaches itself.
26	Specification, uniqueness, commutativity, identities, idempotence	`lean/YonedaTx/Merge.lean`	73, 79, 96, 107, 127, 142, 146, 150, 160	Listing 26: The merge meets its specification without conflicts, is the unique state that does, is commutative including conflicts, has both identities, is idempotent, and has no conflict for disjoint write sets.
27	Edits to disjoint lines of one file merge cleanly by line	`lean/YonedaTx/Merge.lean`	204	Listing 27: If every line is unchanged on one side, no line is a conflict path, and each line's merge applies the diff of either side to the other.
28	Merged landing is a serial step iff patch-compatible	`lean/YonedaTx/Batch.lean`	157	Listing 28: For a conflict-free member, the merge with H equals running the transaction on H if and only if `PatchCompatible` holds.

Row	Claim	File	Lines	What the code shows
29	Patch compatibility of clean members and of changesets	`lean/YonedaTx/Batch.lean`	170, 183	Listing 29: Agreement of base and head on R and W gives patch compatibility. For a changeset it holds iff no written path left at its base content was moved by the head.
30	A blind write with an unread merged path is not serial in ticket order	`lean/YonedaTx/Soundness.lean`	258, 262, 264, 280	Listing 30: The changeset m declares x and y with no reads, is validated as Merge with no conflict path and accepted, yet the batch differs from running m on the head and m is not patch-compatible.
31	Headline: batch landing soundness	`lean/YonedaTx/Soundness.lean`	181	Listing 31: For a log with no Conflict and an accepted batch whose Merge members are patch-compatible, the batch equals serial execution in ticket order, equals the final snapshot of the log followed by the members, and the graph has no cycle.
32	Octopus equals sequential equals serial	`lean/YonedaTx/Batch.lean`	87, 120, 219	Listing 32: Disjoint writes make the octopus merge equal sequential landing. No conflict path makes that the serial merge. The rule and patch compatibility make that serial execution.
33	The members appended to the log	`lean/YonedaTx/Soundness.lean`	91, 111	Listing 33: After j members the snapshot is the sequential landing of the first j. The log followed by an accepted batch has no Conflict landing.
34	The merge form, and the clean case	`lean/YonedaTx/Soundness.lean`	144, 209	Listing 34: Without patch compatibility the batch equals `serialMerge`. With only FastForward and Rebase members it equals `serialTxns`.
35	The write-skew history of Part I, Theorem 7.3	`lean/YonedaTx/Batch.lean`	381, 384, 387, 391, 395	Listing 35: The initial state has x = y = 0. `c0` writes z. `ca` reads x and sets y to x + 1. `cb` reads y and sets x to y + 1.
36	Part I, Theorem 7.3: batch write skew	`lean/YonedaTx/Batch.lean`	412	Listing 36: Both members are validated as Rebase against the log, their write sets are disjoint, each reads what the other writes, the batch equals neither serial order, and landed one at a time the second would be a Conflict.

Row	Claim	File	Lines	What the code shows
37	Without the rule: accepted, not serial, cyclic	`lean/YonedaTx/Soundness.lean`	223	Listing 37: Both members are accepted, the rule fails, the batch equals neither serial order, and position 1 of the extended log reaches itself in the serialization graph.
38	Declared versus true reads	`lean/YonedaTx/UnderDeclared.lean`	26, 29, 34, 39	Listing 38: Validation uses declared sets `Di`. The write check requires only that no landing has a conflict path. `undeclared` is the true read set minus the declared one.
39	No lost update with under-declared reads	`lean/YonedaTx/UnderDeclared.lean`	48, 72	Listing 39: A landing changes p only if it writes p from a base that held the current value. Hence the value at i survives unless such a landing intervenes.
40	Under-declared counterexample	`lean/YonedaTx/UnderDeclared.lean`	128, 131, 146	Listing 40: With c_a's read of x undeclared, both landings validate and pass the write check, the final state is neither serial order, and with true reads c_a is a Conflict.
41	Under-declared reads that meet no later write are safe	`lean/YonedaTx/UnderDeclared.lean`	99	Listing 41: If the undeclared reads of each entry meet no later write set, every snapshot is serial and true reads agree with serial reads.
42	The condition cannot be weakened in general	`lean/YonedaTx/UnderDeclared.lean`	180	Listing 42: Whenever undeclared reads meet an intervening write set while the declared reads and a non-empty write set do not, some transactions with those sets are accepted and equal neither serial order.
43	Non-vacuity: a Merge with a file conflict, clean by line	`lean/YonedaTx/Witnesses.lean`	31, 45, 48, 51, 55, 58, 76	Listing 43: `m1` is validated as Merge, the file-granular merge of F conflicts, no path is a conflict, and the landing keeps both edits.
44	Non-vacuity of the headline	`lean/YonedaTx/Witnesses.lean`	130	Listing 44: A non-empty log, an accepted batch whose first member is validated as Merge and is patch-compatible, and the batch equal to serial execution with both edits and the increment present.
45	Non-vacuity of Theorem 6.4, Lemma 6.3 and the safe condition	`lean/YonedaTx/Witnesses.lean`	156, 170, 178, 192	Listing 45: A validated two-entry log whose second entry is a Rebase with base 0 and whose final snapshot is serial, snapshots 0 and 1 that differ at F/1 and agree on G, and a log validated with empty declared sets, whose second entry's undeclared reads are G, equal to serial execution.

Row	Claim	File	Lines	What the code shows
46	Part I, Theorem 5.4 and Corollary 5.5	`lean/YonedaTx/Graph.lean`	90, 150, 183, 211	Listing 47: Under the append discipline a ref's later values reach its earlier value by first parents. With generations, equal values at u and v mean no entry in between names the ref. And a landed commit stays an ancestor.
47	Part II, Lemma 13.2, and Part I, Proposition 8.3(5)	`lean/YonedaTx/Graph.lean`	272, 294, 313	Listing 48: The projected commit at j depends only on the first j entries, has the previous projection as first parent, and earlier projections lie on its first-parent chain.