This module specifies the logic for the tranasction controller. This controller is responsible for doing the vast majority of the processing in µ*ONOS Config*.

The transaction controller processes transactions as they're appended to the transaction *log* by the northbound server. Each transaction may go through up to two phases: *Change* and *Rollback*. All transactions are initially processed through the *Change* phase after being appended to the *log*. Once they've been committed to the internal configuration, transactions can be rolled back by request via the northbound server. Additionally, there are two stages through which the transaction must progress within each phase. In the *Commit* stage, the controller will validate the change and commit it to the internal configuration, making it immediately queryable via the northbound *API*. Once a change has been committed, it will be sent to the target during the *Apply* stage.

The transaction controller provides a few important guarantees for processing transactions : − Changes and rollbacks will always be committed to the configuration before
they're applied to the *target*
− Changes will always be committed and applied in the order in which they were appended to the transaction *log*
− *Rollbacks* will always be committed and applied in reverse chronological order
- Transactions will always be applied in the order in which they were *committed*
− If a transaction passes validation but is rejected by the target, the rejected transaction and any later changes that are pending must be rolled back before any subsequent change can be applied (due to the semantics of guarantee $\neq 2$)

MODULE *Transaction*

INSTANCE *Naturals*

INSTANCE *FiniteSets*

INSTANCE *Sequences*

INSTANCE *TLC*

An empty constant
CONSTANT *Nil*

Transaction phase constants
CONSTANTS
Change,
Rollback

Transaction stage constants
CONSTANTS
Commit,
Apply

Status constants representing the state associated with a transaction state (commit/apply)
CONSTANTS
Pending,
InProgress,

Complete,
Aborted,
Canceled,
Failed

$Status \triangleq \{Pending, InProgress, Complete, Aborted, Canceled, Failed\}$

$Done \triangleq \{Complete, Aborted, Canceled, Failed\}$

The set of all nodes
CONSTANT *Node*

The set of possible paths and values
CONSTANT *Path*, *Value*

$Empty \triangleq [p \in \{\} \mapsto Nil]$

The state variables that are managed by other controllers.
VARIABLES
configuration,
mastership,
conns,
target

A transaction log. Changes are appended to the log and can be rolled back by modifying the state of a transaction.
VARIABLE *transactions*

A history of transaction change/rollback commit/apply events used for model checking.
VARIABLE *history*

Transactions are managed externally by two operations: append and rollback. Transactions are appended in the *Change* phase. Once a transaction has been committed successfully, it can be rolled back by simply moving it to the Rollback phase.

Add a change for revision 'i' to the transaction log

$$
\begin{array}{l}
AppendChange(i) \triangleq \\
\quad \land\ Len(transactions) = i - 1 \\
\quad \land\ \exists\, p \in Path,\ v \in Value : \\
\qquad \text{LET } transaction \triangleq [\\
\qquad\qquad index \quad \mapsto Len(transactions) + 1, \\
\qquad\qquad phase \quad \mapsto Change, \\
\qquad\qquad change \mapsto [\\
\qquad\qquad\quad index \quad \mapsto Len(transactions) + 1, \\
\qquad\qquad\quad ordinal \mapsto 0, \\
\qquad\qquad\quad values \quad \mapsto (p :> v),
\end{array}
$$

$commit \mapsto Pending,$
$apply \mapsto Pending],$
$rollback \mapsto [$
$index \mapsto 0,$
$ordinal \mapsto 0,$
$values \mapsto Empty,$
$commit \mapsto Nil,$
$apply \mapsto Nil]]$
IN $\land transactions' = Append(transactions, transaction)$
$\land$ UNCHANGED $\langle configuration, mastership, conns, target, history\rangle$

Add a rollback of revision 'i' to the transaction log
$RollbackChange(i) \triangleq$
$\land i \in$ DOMAIN $transactions$
$\land transactions[i].phase = Change$
$\land transactions[i].change.commit = Complete$
$\land transactions' = [transactions$ EXCEPT $![i].phase = Rollback,$
$![i].rollback.commit = Pending,$
$![i].rollback.apply = Pending]$
$\land$ UNCHANGED $\langle configuration, mastership, conns, target, history\rangle$

This is the heart of the transaction controller logic. The controller is responsible for four general actions :
– Validate/commit a *change*
– *Apply* a change to the *target*
– Roll back a committed *change*
– Roll back a change on the target

Commit transaction 'i' change on node 'n'
$CommitChange(n, i) \triangleq$
If the change commit is *Pending*, check if the prior transaction (Change or *Rollback*) has been committed before moving the change to *InProgress*.
$\lor \land transactions[i].change.commit = Pending$
$\land configuration.committed.change = i - 1$
$\land \lor \land configuration.committed.target \neq i$
$\land configuration.committed.index = configuration.committed.target$
$\land configuration.committed.index \in$ DOMAIN $transactions \Rightarrow$
$\lor \land configuration.committed.target = configuration.committed.index$
$\land transactions[configuration.committed.index].change.commit \in Done$
$\lor \land configuration.committed.target < configuration.committed.index$
$\land transactions[configuration.committed.index].rollback.commit \in Done$
Updates to the configuration store and transaction store are separate atomic operations, and therefore the model must account for the potentail for failures to occur between the two. This models the scenario in which the configuration and transaction are both updated, and the scneario

```
                of a failure occuring after the configuration has been updated.
               ∧ configuration' = [configuration EXCEPT !.committed.target = i]
               ∧ history' = Append(history, [
                                    phase  ↦ Change,
                                    event  ↦ Commit,
                                    index  ↦ i,
                                    status ↦ InProgress])
                We store the rollback index and values in the transaction when moving
                from commit Pending to InProgress to ensure the current state of the
                relevant paths/values in the configuration are persisted for use during
                rollbacks. Storing the prior values for rollbacks is a performance
                optimization. It allows the implementation to do rollbacks without having
                to search the transaction log.
               ∧ ∨ LET rollbackIndex ≜ configuration.committed.revision
                       rollbackValues ≜ [
                          p ∈ DOMAIN transactions[i].change.values ↦
                            IF p ∈ DOMAIN configuration.committed.values THEN
                               configuration.committed.values[p]
                             ELSE Nil]
                   IN
                      transactions' = [transactions EXCEPT ![i].change.commit   = InProgress,
                                                           ![i].rollback.index  = rollbackIndex,
                                                           ![i].rollback.values = rollbackValues]
                 ∨ UNCHANGED ⟨transactions⟩
                If the configuration was updated but the commit is still Pending, move
                it to InProgress.
               ∨ ∧ configuration.committed.target = i
                 ∧ LET rollbackIndex ≜ configuration.committed.revision
                       rollbackValues ≜ [
                          p ∈ DOMAIN transactions[i].change.values ↦
                            IF p ∈ DOMAIN configuration.committed.values THEN
                               configuration.committed.values[p]
                             ELSE Nil]
                   IN
                      transactions' = [transactions EXCEPT ![i].change.commit   = InProgress,
                                                           ![i].rollback.index  = rollbackIndex,
                                                           ![i].rollback.values = rollbackValues]
                 ∧ UNCHANGED ⟨configuration, history⟩
            If the change commit is InProgress, validate the change and commit it to the
            configuration if valid. If the change is invalid, the change commit is marked Failed.
            ∨ ∧ transactions[i].change.commit = InProgress
              ∧ ∨ ∧ configuration.committed.change ≠ i
                    In the implementation, the change is validated prior to committing
                    it to the configuration. This section models both valid (Complete)
                    and invalid (Failed) changes.
```

Again, we model partial commits due to failures during the commit process.

$\land \lor \land$ LET $ordinal \triangleq configuration.committed.ordinal + 1$
$values \triangleq transactions[i].change.values \mathbin{@@} configuration.committed.values$
IN
$\land configuration' = [configuration$ EXCEPT $!.committed.index = i,$
$!.committed.change = i,$
$!.committed.revision = i,$
$!.committed.ordinal = ordinal,$
$!.committed.values = values]$
$\land history' = Append(history, [$
$phase \mapsto Change,$
$event \mapsto Commit,$
$index \mapsto i,$
$status \mapsto Complete])$
To ensure the change will be applied in the order in which it was committed, store the sequential ordinal to be used to sequence changes.
$\land \lor transactions' = [transactions$ EXCEPT $![i].change.commit = Complete,$
$![i].change.ordinal = ordinal]$
$\lor$ UNCHANGED $\langle transactions \rangle$
In the event of a validation failure, the transaction status is updated *before* the configuration indexes. This is safe because the configuration values are not updated with invalid change values. Updating the transaction status first ensures important error information is retained in the event of a failure while processing the change.
$\lor \land transactions' = [transactions$ EXCEPT $![i].change.commit = Failed,$
$![i].change.apply = Canceled]$
$\land history' = Append(history, [$
$phase \mapsto Change,$
$event \mapsto Commit,$
$index \mapsto i,$
$status \mapsto Failed])$
$\land \lor configuration' = [configuration$ EXCEPT $!.committed.index = i,$
$!.committed.change = i]$
$\lor$ UNCHANGED $\langle configuration \rangle$
If the configuration was updated but the commit is still *InProgress*, move it to *Complete*.
$\lor \land configuration.committed.change = i$
$\land transactions' = [transactions$ EXCEPT $![i].change.commit = Complete,$
$![i].change.ordinal = configuration.committed.ordinal]$
$\land$ UNCHANGED $\langle configuration, history \rangle$
If the change commit was marked *Failed* but a failure occurred before the configuration could be updated, update the configuration indexes to unblock subsequent transactions.
$\lor \land transactions[i].change.commit = Failed$
$\land configuration.committed.change < i$

```
    ∧ configuration' = [configuration EXCEPT !.committed.index  = i,
                                              !.committed.change = i]
    ∧ UNCHANGED ⟨transactions, history⟩
```

Apply transaction 'i' change on node 'n'

```
ApplyChange(n, i) ≜
  ∨ ∧ transactions[i].change.apply = Pending
        \* To ensure changes are applied in the same order in which they were committed,
        \* we use the change ordinal to serialize application of changes/rollbacks to
        \* the target.
    ∧ ∨ ∧ configuration.applied.ordinal = transactions[i].change.ordinal − 1
        ∧ configuration.applied.target ≠ i
        ∧ configuration.applied.index ∈ DOMAIN transactions ⇒
            ∨ ∧ configuration.applied.target = configuration.applied.index
              ∧ transactions[configuration.applied.index].change.apply ∈ Done
            ∨ ∧ configuration.applied.target < configuration.applied.index
              ∧ transactions[configuration.applied.index].rollback.apply ∈ Done
        \* Just checking if the previous change/rollback has been committed
        \* is not enough to guarantee sequential ordering. In the event of
        \* a target failure, gaps in the log need to be accounted for.
        \* We use the applied revision to determine whether the prior
        \* change/rollback (the rollback index) was successful.
        ∧ ∨ ∧ configuration.applied.revision = transactions[i].rollback.index
            ∧ configuration' = [configuration EXCEPT !.applied.target = i]
            ∧ history' = Append(history, [
                            phase  ↦ Change,
                            event  ↦ Apply,
                            index  ↦ i,
                            status ↦ InProgress])
            ∧ ∨ transactions' = [transactions EXCEPT ![i].change.apply = InProgress]
              ∨ UNCHANGED ⟨transactions⟩
          \* In the event that a prior change apply Failed, all subsequent changes
          \* must be Aborted and ultimately rolled back. This forces administrator
          \* intervention in the unlikely event of a mismatch between the config
          \* and target configuration models, in which case the change may pass
          \* validation but fail being applied to the target.
          ∨ ∧ configuration.applied.revision < transactions[i].rollback.index
            ∧ transactions' = [transactions EXCEPT ![i].change.apply = Aborted]
            ∧ history' = Append(history, [
                            phase  ↦ Change,
                            event  ↦ Apply,
                            index  ↦ i,
                            status ↦ Aborted])
            ∧ ∨ LET ordinal ≜ transactions[i].change.ordinal
                IN  configuration' = [configuration EXCEPT !.applied.target  = i,
```

$!.applied.index = i,$
$!.applied.ordinal = ordinal]$

$\lor$ UNCHANGED $\langle configuration \rangle$

If the configuration was updated but the apply is still *Pending*, move it to *InProgress*.

$\lor \land configuration.applied.target = i$
$\land transactions' = [transactions$ EXCEPT $![i].change.apply = InProgress]$
$\land$ UNCHANGED $\langle configuration, history \rangle$

$\land$ UNCHANGED $\langle target \rangle$

$\lor \land transactions[i].change.apply = InProgress$

If the change has not yet been applied, attempt to apply it to the target. This logic models the possibility of failures when applying changes to the *target* – despite them having already been validated - providing administrators an avenue to safely reconcile errors caused by mismatches between the system configuration models or between a model and the real-world implementation of it.

$\land \lor \land configuration.applied.ordinal \neq transactions[i].change.ordinal$

In the event of a node or target restart or another failure resulting in a *mastership* change, the configuration must be re-pushed to the target before the transaction controller can resume processing transactions to be applied to the target.

If the configuration is still being synchronized with the target, wait for the synchronization to be complete. The configuration state must be *Complete* in the current master's term before the change can be applied.

$\land configuration.state = Complete$
$\land configuration.term = mastership.term$
$\land conns[n].id = mastership.conn$
$\land conns[n].connected$
$\land target.running$

The change is successfully applied to the target.

$\land \lor \land$ LET $ordinal \triangleq transactions[i].change.ordinal$
$values \triangleq transactions[i].change.values$ @@ $configuration.applied.values$
IN
$\land target' = [target$ EXCEPT $!.values = transactions[i].change.values$ @@ $target.values]$
$\land configuration' = [configuration$ EXCEPT $!.applied.index = i,$
$!.applied.ordinal = ordinal,$
$!.applied.revision = i,$
$!.applied.values = values]$
$\land history' = Append(history, [$
$phase \mapsto Change,$
$event \mapsto Apply,$
$index \mapsto i,$
$status \mapsto Complete])$
$\land \lor transactions' = [transactions$ EXCEPT $![i].change.apply = Complete]$
$\lor$ UNCHANGED $\langle transactions \rangle$

The target rejects the change with an unexpected error.

```
                    ∨ ∧ transactions' = [transactions EXCEPT ![i].change.apply = Failed]
                      ∧ history' = Append(history, [
                                       phase  ↦ Change,
                                       event  ↦ Apply,
                                       index  ↦ i,
                                       status ↦ Failed])
                      ∧ ∨ LET ordinal ≜ transactions[i].change.ordinal
                            IN  configuration' = [configuration EXCEPT !.applied.index   = i,
                                                                       !.applied.ordinal = ordinal]
                        ∨ UNCHANGED ⟨configuration⟩
                      ∧ UNCHANGED ⟨target⟩
                  If the change has been applied, update the transaction status.
                ∨ ∧ configuration.applied.ordinal = transactions[i].change.ordinal
                  ∧ transactions' = [transactions EXCEPT ![i].change.apply = Complete]
                  ∧ UNCHANGED ⟨configuration, target, history⟩
              If the apply has been marked Aborted or Failed, increment the applied ordinal
              to unblock subsequent changes/rollbacks.
            ∨ ∧ transactions[i].change.apply ∈ {Aborted, Failed}
              ∧ configuration.applied.ordinal < transactions[i].change.ordinal
              ∧ configuration' = [configuration EXCEPT !.applied.target  = i,
                                                       !.applied.index   = i,
                                                       !.applied.ordinal = transactions[i].change.ordinal]
              ∧ UNCHANGED ⟨transactions, target, history⟩

          Reconcile transaction 'i' change on node 'n'
        ReconcileChange(n, i) ≜
          ∧ transactions[i].phase = Change
          ∧ ∨ ∧ CommitChange(n, i)
              ∧ UNCHANGED ⟨target⟩
            ∨ ∧ transactions[i].change.commit = Complete
              ∧ ApplyChange(n, i)

          Commit transaction 'i' rollback on node 'n'
        CommitRollback(n, i) ≜
          ∨ ∧ transactions[i].rollback.commit = Pending
            ∧ configuration.committed.revision = i
            ∧ ∨ ∧ configuration.committed.target = i
                ∧ configuration.committed.index = configuration.committed.target
                ∧ ∨ ∧ configuration.committed.index = i
                    ∧ transactions[configuration.committed.index].change.commit = Complete
                  ∨ ∧ configuration.committed.index > i
                    ∧ transactions[configuration.committed.index].rollback.commit = Complete
                ∧ configuration' = [configuration EXCEPT !.committed.target = transactions[i].rollback.index]
                ∧ history' = Append(history, [
                                 phase  ↦ Rollback,
```

```
                                    event  ↦ Commit,
                                    index  ↦ i,
                                    status ↦ InProgress])
                  ∧ ∨ transactions' = [transactions EXCEPT ![i].rollback.commit = InProgress]
                    ∨ UNCHANGED ⟨transactions⟩
              ∨ ∧ configuration.committed.target = transactions[i].rollback.index
                ∧ transactions' = [transactions EXCEPT ![i].rollback.commit = InProgress]
                ∧ UNCHANGED ⟨configuration, history⟩
        ∨ ∧ transactions[i].rollback.commit = InProgress
          We do not model validation here under the assumption the prior state
          to which we are rolling back was already validated.
          ∧ ∨ ∧ configuration.committed.revision = i
              When completing the commit, the configuration status must be updated
              before the rollback commit can be marked Complete to avoid multiple
              transactions being processed concurrently.
              ∧ LET ordinal  ≜ configuration.committed.ordinal + 1
                    revision ≜ transactions[i].rollback.index
                    values   ≜ transactions[i].rollback.values @@ configuration.committed.values
                IN
                  ∧ configuration' = [configuration EXCEPT !.committed.index    = i,
                                                           !.committed.ordinal  = ordinal,
                                                           !.committed.revision = revision,
                                                           !.committed.values   = values]
                  ∧ history' = Append(history, [
                                    phase  ↦ Rollback,
                                    event  ↦ Commit,
                                    index  ↦ i,
                                    status ↦ Complete])
                  Model partial commits due to failures during processing.
                  ∧ ∨ transactions' = [transactions EXCEPT ![i].rollback.commit  = Complete,
                                                           ![i].rollback.ordinal = ordinal]
                    ∨ UNCHANGED ⟨transactions⟩
              In the event of a partial commit, Complete the rollback commit.
            ∨ ∧ configuration.committed.revision = transactions[i].rollback.index
              ∧ transactions' = [transactions EXCEPT ![i].rollback.commit  = Complete,
                                                     ![i].rollback.ordinal = configuration.committed.ordinal]
              ∧ UNCHANGED ⟨configuration, history⟩

Apply transaction 'i' rollback on node 'n'
ApplyRollback(n, i) ≜
  ∨ ∧ transactions[i].rollback.apply = Pending
    Before the rollback can be applied, any changes that are Pending or InProgress
    must first be canceled.
    If the change is Pending apply, mark the apply stage Aborted.
    ∧ ∨ ∧ transactions[i].change.apply = Pending
```

```
    The change cannot be aborted until the previous scheduled change/rollback is complete.
  ∧ configuration.applied.ordinal = transactions[i].change.ordinal − 1
  ∧ configuration.applied.target ≠ i
  ∧ configuration.applied.index ∈ DOMAIN transactions ⇒
        ∨ ∧ configuration.applied.target = configuration.applied.index
          ∧ transactions[configuration.applied.index].change.apply ∈ Done
        ∨ ∧ configuration.applied.target < configuration.applied.index
          ∧ transactions[configuration.applied.index].rollback.apply ∈ Done
  ∧ transactions′ = [transactions EXCEPT ![i].change.apply = Aborted]
  ∧ history′ = Append(history, [
                    phase  ↦ Change,
                    event  ↦ Apply,
                    index  ↦ i,
                    status ↦ Aborted])
  ∧ ∨ configuration′ = [configuration EXCEPT !.applied.target  = i,
                                             !.applied.index  = i,
                                             !.applied.ordinal = transactions[i].change.ordinal]
    ∨ UNCHANGED ⟨configuration⟩
    If the change apply is InProgress, mark the apply stage Failed rather than
    aborting it. This is necessary to indicate that the change may or may not have
    been applied to the target. Since the apply was already in progress, a prior
    step may have already attempted to push the change to the target, and a failure
    during that attempt could have left the system in an inconsistent state. Once
    the change apply is InProgress, it must be rolled back completely through the
    apply phase even if it was never marked Complete.
∨ ∧ transactions[i].change.apply = InProgress
  ∧ configuration.applied.ordinal ≠ transactions[i].change.ordinal
  ∧ transactions′ = [transactions EXCEPT ![i].change.apply = Failed]
  ∧ history′ = Append(history, [
                    phase  ↦ Change,
                    event  ↦ Apply,
                    index  ↦ i,
                    status ↦ Failed])
  ∧ ∨ configuration′ = [configuration EXCEPT !.applied.index  = i,
                                             !.applied.ordinal = transactions[i].change.ordinal]
    ∨ UNCHANGED ⟨configuration⟩
    If the transaction was Aborted or Failed but the configuration status hasn't
    been updated, update the configuration status to unblock subsequent changes/rollbacks.
    This can happen in the event a failure occurs while applying the change.
∨ ∧ transactions[i].change.apply ∈ {Aborted, Failed}
  ∧ configuration.applied.ordinal < transactions[i].change.ordinal
  ∧ configuration′ = [configuration EXCEPT !.applied.target  = i,
                                           !.applied.index  = i,
                                           !.applied.ordinal = transactions[i].change.ordinal]
  ∧ UNCHANGED ⟨transactions, history⟩
```

```
        Finally, the transaction's change phase really is done (based on the applied ordinal),
        apply the rollback in commit order (by rolback ordinal).
      ∨ ∧ transactions[i].change.apply ∈ Done
        ∧ configuration.applied.ordinal = transactions[i].rollback.ordinal − 1
            Model partial failures that can occur while transitioning the transaction.
        ∧ ∨ ∧ configuration.applied.target ≠ transactions[i].rollback.index
            ∧ ∨ ∧ configuration.applied.index = i
                ∧ transactions[configuration.applied.index].change.apply ∈ Done
              ∨ ∧ configuration.applied.index > i
                ∧ transactions[configuration.applied.index].rollback.apply ∈ Done
            ∧ configuration′ = [configuration EXCEPT !.applied.target = transactions[i].rollback.index]
            ∧ history′ = Append(history, [
                                  phase  ↦ Rollback,
                                  event  ↦ Apply,
                                  index  ↦ i,
                                  status ↦ InProgress])
            ∧ ∨ transactions′ = [transactions EXCEPT ![i].rollback.apply = InProgress]
              ∨ UNCHANGED ⟨transactions⟩
            A failure left the system in an inconsistent state, resume the transaction
            by moving it to InProgress.
          ∨ ∧ configuration.applied.target = transactions[i].rollback.index
            ∧ transactions′ = [transactions EXCEPT ![i].rollback.apply = InProgress]
            ∧ UNCHANGED ⟨configuration, history⟩
    ∧ UNCHANGED ⟨target⟩
  ∨ ∧ transactions[i].rollback.apply = InProgress
      If this transaction has not yet been applied, attempt to apply it.
    ∧ ∨ ∧ configuration.applied.ordinal ≠ transactions[i].rollback.ordinal
        In the event of a node or target restart or another failure resulting in a
        mastership change, the configuration must be re-pushed to the target before
        the transaction controller can resume processing transactions to be applied
        to the target.
        If the configuration is still being synchronized with the target, wait for
        the synchronization to be complete. The configuration state must be Complete
        in the current master's term before the change can be applied.
      ∧ configuration.state = Complete
      ∧ configuration.term = mastership.term
      ∧ conns[n].id = mastership.conn
      ∧ conns[n].connected
      ∧ target.running
      ∧ LET ordinal  ≜ transactions[i].rollback.ordinal
            revision ≜ transactions[i].rollback.index
            values   ≜ transactions[i].rollback.values @@ configuration.applied.values
        IN
          ∧ target′ = [target EXCEPT !.values = transactions[i].rollback.values @@ target.values]
          ∧ configuration′ = [configuration EXCEPT !.applied.index  = i,
```

```
                                                    !.applied.ordinal  = ordinal,
                                                    !.applied.revision = revision,
                                                    !.applied.values   = values]
                    ∧ history' = Append(history, [
                                    phase  ↦ Rollback,
                                    event  ↦ Apply,
                                    index  ↦ i,
                                    status ↦ Complete])
                    ∧ ∨ transactions' = [transactions EXCEPT ![i].rollback.apply = Complete]
                      ∨ UNCHANGED ⟨transactions⟩
                  If the change has been applied, update the transaction status.
                ∨ ∧ configuration.applied.ordinal = transactions[i].rollback.ordinal
                  ∧ configuration.applied.revision = transactions[i].rollback.index
                  ∧ transactions' = [transactions EXCEPT ![i].rollback.apply = Complete]
                  ∧ UNCHANGED ⟨configuration, target, history⟩

Reconcile transaction 'i' rollback on node 'n'
ReconcileRollback(n, i) ≜
    ∧ transactions[i].phase = Rollback
    ∧ ∨ ∧ CommitRollback(n, i)
        ∧ UNCHANGED ⟨target⟩
      ∨ ∧ transactions[i].rollback.commit = Complete
        ∧ ApplyRollback(n, i)

Reconcile transaction 'i' on node 'n'
ReconcileTransaction(n, i) ≜
    ∧ i ∈ DOMAIN transactions
    ∧ mastership.master = n
    ∧ ∨ ReconcileChange(n, i)
      ∨ ReconcileRollback(n, i)
    ∧ UNCHANGED ⟨mastership, conns⟩

─────────────────────────────────────────────────────────────────────────

TypeOK ≜
  ∀ i ∈ DOMAIN transactions :
    ∧ transactions[i].index ∈ Nat
    ∧ transactions[i].phase ∈ {Change, Rollback}
    ∧ transactions[i].change.commit ∈ Status
    ∧ transactions[i].change.apply ∈ Status
    ∧ ∀ p ∈ DOMAIN transactions[i].change.values :
        transactions[i].change.values[p] ≠ Nil ⇒
          transactions[i].change.values[p] ∈ STRING
    ∧ transactions[i].rollback.commit ≠ Nil ⇒
        transactions[i].rollback.commit ∈ Status
    ∧ transactions[i].rollback.apply ≠ Nil ⇒
```

```
        transactions[i].rollback.apply ∈ Status
    ∧ ∀ p ∈ DOMAIN transactions[i].rollback.values :
        transactions[i].rollback.values[p] ≠ Nil ⇒
          transactions[i].rollback.values[p] ∈ STRING

LOCAL State ≜ [
  transactions  ↦ transactions,
  configuration ↦ configuration,
  mastership    ↦ mastership,
  conns         ↦ conns,
  target        ↦ target]

LOCAL Transitions ≜
  LET
      indexes  ≜ {i ∈ DOMAIN transactions' :
                    i ∈ DOMAIN transactions ⇒ transactions'[i] ≠ transactions[i]}
  IN  [transactions ↦ [i ∈ indexes ↦ transactions'[i]]] @@
        (IF configuration' ≠ configuration THEN [configuration ↦ configuration'] ELSE Empty) @@
        (IF target' ≠ target THEN [target ↦ target'] ELSE Empty) @@
        (IF Len(history') > Len(history) THEN [event ↦ history'[Len(history')]] ELSE Empty)

Test ≜ INSTANCE Test WITH
  File ← "Transaction.test.log"
```