MODULE *Config*

INSTANCE *Naturals*

INSTANCE *FiniteSets*

INSTANCE *Sequences*

INSTANCE *TLC*

An empty constant
CONSTANT *Nil*

Transaction type constants
CONSTANTS
Change,
Rollback

Transaction isolation constants
CONSTANTS
ReadCommitted,
Serializable

Phase constants
CONSTANTS
Initialize,
Validate,
Abort,
Commit,
Apply

$Phase \triangleq$
{*Initialize*,
Validate,
Abort,
Commit,
Apply}

Status constants
CONSTANTS
InProgress,
Complete,
Failed

$State \triangleq$
{*InProgress*,
Complete,

Failed}

State constants

CONSTANTS
Pending,
Validated,
Committed,
Applied,
Aborted

$Status \triangleq$
{*Pending*,
Validated,
Committed,
Applied,
Aborted}

CONSTANTS
Valid,
Invalid

CONSTANTS
Success,
Failure

The set of all nodes

CONSTANT *Node*

The number of transactions to use for model checking

CONSTANT *NumTransactions*

Target is the set of all targets and their possible paths and values.

Example:

```
Target ≜
  [target1 ↦
    [persistent ↦ FALSE, values ↦ [
      path1 ↦ {"value1", "value2"},
      path2 ↦ {"value2", "value3"}]],
  target2 ↦
    [persistent ↦ TRUE, values ↦ [
      path2 ↦ {"value3", "value4"},
      path3 ↦ {"value4", "value5"}]]]
```

CONSTANT *Target*

$Empty \triangleq [p \in \{\} \mapsto [value \mapsto Nil,\ delete \mapsto \text{FALSE}]]$

Configuration update/rollback requests are tracked and processed through two data types. Transactions represent the lifecycle of a single configuration change request and are stored in an append-only log. Configurations represent the desired configuration of a *gNMI* target based on the aggregate of relevant changes in the Transaction log.

```
TYPE Type ::= type ∈
  {Change,
   Rollback}

TYPE Phase ::= phase ∈
  {Initialize,
   Validate,
   Abort,
   Commit,
   Apply}

TYPE State ::= state ∈
  {InProgress,
   Complete,
   Failed}

TYPE Status ::= status ∈
  {Pending,
   Validated,
   Committed,
   Applied,
   Aborted}

TYPE Isolation ::= isolation ∈
  {ReadCommitted,
   Serializable}

TYPE Transaction ≜
  [type      ::= type ∈ Type,
   isolation ::= isolation ∈ Isolation
   change ::=
    [target ∈ SUBSET (DOMAIN Target) ↦
      [path ∈ SUBSET (DOMAIN Target[target].values) ↦
        [value ::= value ∈ STRING,
         delete ::= delete ∈ BOOLEAN ]]],
   rollback ::= index ∈ Nat,
   targets ::= targets ∈ SUBSET (DOMAIN Target)
   phase     ::= phase ∈ Phase,
   state     ::= state ∈ State,
   status ::= status ∈ Status]

TYPE Proposal ≜
  [type      ::= type ∈ Type,
   change    ::=
    [index ::= index ∈ Nat,
     values ::=
      [path ∈ SUBSET (DOMAIN Target[target].values) ↦
        [value ::= value ∈ STRING,
         delete ::= delete ∈ BOOLEAN ]]],
```

```
    rollback ::=
     [index ::= index ∈ Nat,
      values ::=
       [path ∈ SUBSET (DOMAIN Target[target].values) ↦
         [value ::= value ∈ STRING,
          delete ::= delete ∈ BOOLEAN ]]],
    dependency ::= [index ∈ Nat],
    phase      ::= phase ∈ Phase,
    state      ::= state ∈ State]
 TYPE Configuration ≜
   [config ::=
     [index ::= index ∈ Nat,
      term ::= term ∈ Nat,
      values ::=
       [path ∈ SUBSET (DOMAIN Target[target]) ↦
         [value ::= value ∈ STRING,
          index ::= index ∈ Nat,
          deleted ::= delete ∈ BOOLEAN ]]],
    proposal ::= [index ::= index ∈ Nat],
    commit ::= [index ::= index ∈ Nat],
    target ::=
     [index ::= index ∈ Nat,
      term ::= term ∈ Nat,
      values ::=
       [path ∈ SUBSET (DOMAIN Target[target]) ↦
         [value ::= value ∈ STRING,
          index ::= index ∈ Nat,
          deleted ::= delete ∈ BOOLEAN ]]],
    state ::= state ∈ State]
```

A transaction log. Transactions may either request a set of changes to a set of targets or rollback a prior change.

VARIABLE *transaction*

A record of per-target proposals

VARIABLE *proposal*

A record of per-target configurations

VARIABLE *configuration*

A record of target states

VARIABLE *target*

A record of target masterships

VARIABLE *mastership*

$vars \triangleq \langle transaction, proposal, configuration, mastership, target \rangle$

This section models *mastership* for the configuration service.

Mastership is used primarily to track the lifecycle of individual configuration targets and react to state changes on the southbound. Each target is assigned a master from the *Node* set, and masters can be unset when the target disconnects.

Set node n as the master for target t

$$
\begin{array}{l}
SetMaster(n,\ t)\ \triangleq \\
\quad \land\ mastership[t].master \neq n \\
\quad \land\ mastership' = [mastership\ \text{EXCEPT}\ ![t].term\ \ = mastership[t].term + 1, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ ![t].master = n] \\
\quad \land\ \text{UNCHANGED}\ \langle transaction,\ proposal,\ configuration,\ target\rangle
\end{array}
$$

$$
\begin{array}{l}
UnsetMaster(t)\ \triangleq \\
\quad \land\ mastership[t].master \neq Nil \\
\quad \land\ mastership' = [mastership\ \text{EXCEPT}\ ![t].master = Nil] \\
\quad \land\ \text{UNCHANGED}\ \langle transaction,\ proposal,\ configuration,\ target\rangle
\end{array}
$$

This section models configuration changes and rollbacks. Changes are appended to the transaction log and processed asynchronously.

$$
\begin{array}{l}
Value(s,\ t,\ p)\ \triangleq \\
\quad \text{LET}\ value\ \triangleq\ \text{CHOOSE}\ v \in s : v.target = t \land v.path = p \\
\quad \text{IN} \\
\qquad [value\ \mapsto value.value, \\
\qquad\ delete\ \mapsto value.delete]
\end{array}
$$

$$
\begin{array}{l}
Paths(s,\ t)\ \triangleq \\
\quad [p \in \{v.path : v \in \{v \in s : v.target = t\}\} \mapsto Value(s,\ t,\ p)]
\end{array}
$$

$$
\begin{array}{l}
Changes(s)\ \triangleq \\
\quad [t \in \{v.target : v \in s\} \mapsto Paths(s,\ t)]
\end{array}
$$

$$
\begin{array}{l}
ValidValues(t,\ p)\ \triangleq \\
\quad \text{UNION}\ \{\{[value \mapsto v,\ delete \mapsto \text{FALSE}] : v \in Target[t].values[p]\},\ \{[value \mapsto Nil,\ delete \mapsto \text{TRUE}]\}\}
\end{array}
$$

$$
\begin{array}{l}
ValidPaths(t)\ \triangleq \\
\quad \text{UNION}\ \{\{v\ @@\ [path \mapsto p] : v \in ValidValues(t,\ p)\} : p \in \text{DOMAIN}\ Target[t].values\}
\end{array}
$$

$$
\begin{array}{l}
ValidTargets\ \triangleq \\
\quad \text{UNION}\ \{\{p\ @@\ [target \mapsto t] : p \in ValidPaths(t)\} : t \in \text{DOMAIN}\ Target\}
\end{array}
$$

The set of all valid sets of changes to all targets and their paths.

The set of possible changes is computed from the *Target* model value.

$$
\begin{array}{l}
ValidChanges\ \triangleq \\
\quad \text{LET}\ changeSets\ \triangleq\ \{s \in \text{SUBSET}\ ValidTargets : \\
\qquad\qquad\qquad\qquad \forall\, t \in \text{DOMAIN}\ Target\ \ : \\
\qquad\qquad\qquad\qquad\quad \forall\, p \in \text{DOMAIN}\ Target[t].values : \\
\qquad\qquad\qquad\qquad\qquad Cardinality(\{v \in s : v.target = t \land v.path = p\}) \leq 1\}
\end{array}
$$

IN

$\{Changes(s) : s \in changeSets\}$

Add a set of changes 'c' to the transaction log

$RequestChange(i,\ c) \triangleq$

$\land\ i = Len(transaction) + 1$

$\land\ \exists\, isolation \in \{ReadCommitted,\ Serializable\} :$

$\land\ transaction' = transaction\ @@\ (i :> [type \mapsto Change,$
$isolation \mapsto isolation,$
$change \mapsto c,$
$targets \mapsto \{\},$
$phase \mapsto Initialize,$
$state \mapsto InProgress,$
$status \mapsto Pending])$

$\land$ UNCHANGED $\langle proposal,\ configuration,\ mastership,\ target\rangle$

Add a rollback of transaction 't' to the transaction log

$RequestRollback(i,\ j) \triangleq$

$\land\ i = Len(transaction) + 1$

$\land\ \exists\, isolation \in \{ReadCommitted,\ Serializable\} :$

$\land\ transaction' = transaction\ @@\ (i :> [type \mapsto Rollback,$
$isolation \mapsto isolation,$
$rollback \mapsto j,$
$targets \mapsto \{\},$
$phase \mapsto Initialize,$
$state \mapsto InProgress,$
$status \mapsto Pending])$

$\land$ UNCHANGED $\langle proposal,\ configuration,\ mastership,\ target\rangle$

This section models the Transaction log reconciler.

Transactions come in two flavors : − *Change* transactions contain a set of changes to be applied to a set of *targets* − *Rollback* transactions reference a prior change transaction to be reverted to the previous state

Transacations proceed through a series of phases:
* *Initialize* − create and link Proposals
* *Validate* − validate changes and rollbacks
* *Commit* − *commit* changes to Configurations
* *Apply* − *commit* changes to Targets

Reconcile a transaction

$ReconcileTransaction(n,\ i) \triangleq$

Initialize is the only transaction phase that's globally serialized. While in the *Initializing* phase, the reconciler checks whether the prior transaction has been *Initialized* before creating Proposals in the *Initialize* phase. Once all of the transaction's proposals have

```
    been Initialized, the transaction will be marked Initialized. If any
    proposal is Failed, the transaction will be marked Failed as well.
∧ ∨ ∧ transaction[i].phase = Initialize
  ∧ ∨ ∧ transaction[i].state = InProgress
        All prior transaction must be initialized before proceeding
        to initialize this transaction.
        ∧ ¬∃ j ∈ DOMAIN transaction :
              ∧ j < i
              ∧ transaction[j].phase = Initialize
              ∧ transaction[j].state  = InProgress
          If the transaction's targets are not yet set, create proposals
          and add targets to the transaction state.
        ∧ ∨ ∧ transaction[i].targets = {}
              If the transaction is a change, the targets are taken
              from the change values.
            ∧ ∨ ∧ transaction[i].type = Change
                ∧ transaction' = [transaction EXCEPT ![i].targets = DOMAIN transaction[i].change]
                ∧ proposal' = [t ∈ DOMAIN proposal ↦
                      IF t ∈ DOMAIN transaction[i].change THEN
                         proposal[t] @@ (i :> [type      ↦ Change,
                                               change    ↦
                                                 [index  ↦ i,
                                                  values ↦ transaction[i].change[t]],
                                               rollback  ↦
                                                 [index  ↦ 0,
                                                  values ↦ Empty],
                                               dependency ↦ [index ↦ 0],
                                               phase     ↦ Initialize,
                                               state     ↦ InProgress])
                      ELSE
                         proposal[t]]
              If the transaction is a rollback, the targets affected are
              the targets of the change transaction being rolled back.
              ∨ ∧ transaction[i].type = Rollback
                  If the rollback index is a valid Change transaction,
                  initialize proposals for all of the Change targets.
                ∧ ∨ ∧ transaction[i].rollback ∈ DOMAIN transaction
                    ∧ transaction[transaction[i].rollback].type = Change
                    ∧ transaction' = [transaction EXCEPT ![i].targets =
                                          DOMAIN transaction[transaction[i].rollback].change]
                    ∧ proposal' = [t ∈ DOMAIN proposal ↦
                          IF t ∈ DOMAIN transaction[transaction[i].rollback].change THEN
                             proposal[t] @@ (i :> [type      ↦ Rollback,
                                                   change    ↦
                                                     [index  ↦ 0,
```

$$
\begin{array}{l}
\quad values \mapsto Empty], \\
rollback \quad \mapsto \\
\quad [index \ \mapsto transaction[i].rollback, \\
\quad values \mapsto Empty], \\
dependency \mapsto [index \mapsto 0], \\
phase \qquad \mapsto Initialize, \\
state \qquad \mapsto InProgress])
\end{array}
$$

ELSE

$proposal[t]]$

If the rollback index is not a valid *Change* transaction
fail the *Rollback* transaction.

$$
\begin{array}{l}
\lor \ \land \ \lor \ \land transaction[i].rollback \in \text{DOMAIN } transaction \\
\qquad\qquad \land transaction[transaction[i].rollback].type = Rollback \\
\qquad\quad \lor transaction[i].rollback \notin \text{DOMAIN } transaction \\
\qquad \land transaction' = [transaction \text{ EXCEPT } ![i].state = Failed] \\
\qquad \land \text{UNCHANGED } \langle proposal \rangle
\end{array}
$$

If the transaction's proposals have been initialized, check proposals
for completion or failures.

$$
\begin{array}{l}
\lor \ \land transaction[i].targets \neq \{\} \\
\qquad \text{If all proposals have been } \textit{Complete}\text{, mark the transaction } \textit{Complete}. \\
\quad \land \ \lor \ \land \forall \, t \in transaction[i].targets : \\
\qquad\qquad \land proposal[t][i].phase = Initialize \\
\qquad\qquad \land proposal[t][i].state \ = Complete \\
\qquad \land transaction' = [transaction \text{ EXCEPT } ![i].state = Complete] \\
\qquad \land \text{UNCHANGED } \langle proposal \rangle \\
\qquad \text{If any proposal has been } \textit{Failed}\text{, mark the transaction } \textit{Failed}. \\
\quad \lor \ \land \exists \, t \in transaction[i].targets : \\
\qquad\qquad \land proposal[t][i].phase \ = Initialize \\
\qquad\qquad \land proposal[t][i].state = Failed \\
\qquad \land transaction' = [transaction \text{ EXCEPT } ![i].state = Failed] \\
\qquad \land \text{UNCHANGED } \langle proposal \rangle
\end{array}
$$

Once the transaction has been *Initialized*, proceed to the *Validate* phase.
If any of the transaction's proposals depend on a *Serializable* transaction,
verify the dependency has been *Validated* to preserve serializability before
moving the transaction to the *Validate* phase.

$$
\begin{array}{l}
\lor \ \land transaction[i].state = Complete \\
\quad \land \forall \, t \in transaction[i].targets : \\
\qquad \land proposal[t][i].dependency.index \in \text{DOMAIN } transaction \\
\qquad \land transaction[proposal[t][i].dependency.index].isolation = Serializable \\
\qquad \Rightarrow transaction[proposal[t][i].dependency.index].status \in \{ Validated, Committed, Applied, A \\
\quad \land transaction' = [transaction \text{ EXCEPT } ![i].phase = Validate, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad ![i].state \ = InProgress] \\
\quad \land \text{UNCHANGED } \langle proposal \rangle
\end{array}
$$

If the transaction failed initialization, proceed to the *Abort* phase
to ensure indexes are still updated for the target configurations.

$\lor \land transaction[i].state = Failed$
$\land transaction' = [transaction \text{ EXCEPT } ![i].phase = Abort,$
$\qquad ![i].state = InProgress]$
$\land \text{UNCHANGED } \langle proposal \rangle$
$\lor \land transaction[i].phase = Validate$
$\land \lor \land transaction[i].state = InProgress$
Move the transaction's proposals to the *Validating* state
$\land \lor \land \exists\, t \in transaction[i].targets :$
$\land proposal[t][i].phase \neq Validate$
$\land proposal' = [proposal \text{ EXCEPT } ![t] =$
$[proposal[t] \text{ EXCEPT } ![i].phase = Validate,$
$\qquad ![i].state = InProgress]]$
$\land \text{UNCHANGED } \langle transaction \rangle$
If all proposals have been *Complete*, mark the transaction *Complete*.
$\lor \land \forall\, t \in transaction[i].targets :$
$\land proposal[t][i].phase = Validate$
$\land proposal[t][i].state = Complete$
$\land transaction' = [transaction \text{ EXCEPT } ![i].state = Complete,$
$\qquad ![i].status = Validated]$
$\land \text{UNCHANGED } \langle proposal \rangle$
If any proposal has been *Failed*, mark the transaction *Failed*.
$\lor \land \exists\, t \in transaction[i].targets :$
$\land proposal[t][i].phase = Validate$
$\land proposal[t][i].state = Failed$
$\land transaction' = [transaction \text{ EXCEPT } ![i].state = Failed]$
$\land \text{UNCHANGED } \langle proposal \rangle$
Once the transaction has been *Validated*, proceed to the *Commit* phase.
If any of the transaction's proposals depend on a *Serializable* transaction,
verify the dependency has been *Committed* to preserve serializability before
moving the transaction to the *Commit* phase.
$\lor \land transaction[i].state = Complete$
$\land \forall\, t \in transaction[i].targets :$
$\land proposal[t][i].dependency.index \in \text{DOMAIN } transaction$
$\land transaction[proposal[t][i].dependency.index].isolation = Serializable$
$\Rightarrow transaction[proposal[t][i].dependency.index].status \in \{Committed, Applied, Aborted\}$
$\land transaction' = [transaction \text{ EXCEPT } ![i].phase = Commit,$
$\qquad ![i].state = InProgress]$
$\land \text{UNCHANGED } \langle proposal \rangle$
If the transaction failed validation, proceed to the *Abort* phase
to ensure indexes are still updated for the target configurations.
$\lor \land transaction[i].state = Failed$
$\land transaction' = [transaction \text{ EXCEPT } ![i].phase = Abort,$
$\qquad ![i].state = InProgress]$
$\land \text{UNCHANGED } \langle proposal \rangle$
$\lor \land transaction[i].phase = Commit$

$\land \lor \land transaction[i].state = InProgress$

Move the transaction's proposals to the *Committing* state

$\land \lor \land \exists\, t \in transaction[i].targets :$
$\land proposal[t][i].phase \neq Commit$
$\land proposal' = [proposal \text{ EXCEPT } ![t] =$
$[proposal[t] \text{ EXCEPT } ![i].phase = Commit,$
$![i].state = InProgress]]$
$\land \text{UNCHANGED } \langle transaction \rangle$

If all proposals have been *Complete*, mark the transaction *Complete*.

$\lor \land \forall\, t \in transaction[i].targets :$
$\land proposal[t][i].phase = Commit$
$\land proposal[t][i].state = Complete$
$\land transaction' = [transaction \text{ EXCEPT } ![i].state = Complete,$
$![i].status = Committed]$
$\land \text{UNCHANGED } \langle proposal \rangle$

Once the transaction has been *Committed*, proceed to the *Apply* phase.
If any of the transaction's proposals depend on a *Serializable* transaction,
verify the dependency has been *Applied* to preserve serializability before
moving the transaction to the *Apply* phase.

$\lor \land transaction[i].state = Complete$
$\land \forall\, t \in transaction[i].targets :$
$\land proposal[t][i].dependency.index \in \text{DOMAIN } transaction$
$\land transaction[proposal[t][i].dependency.index].isolation = Serializable$
$\Rightarrow transaction[proposal[t][i].dependency.index].status \in \{Applied, Aborted\}$
$\land transaction' = [transaction \text{ EXCEPT } ![i].phase = Apply,$
$![i].state = InProgress]$
$\land \text{UNCHANGED } \langle proposal \rangle$

$\lor \land transaction[i].phase = Apply$
$\land transaction[i].state = InProgress$

Move the transaction's proposals to the *Applying* state

$\land \lor \land \exists\, t \in transaction[i].targets :$
$\land proposal[t][i].phase \neq Apply$
$\land proposal' = [proposal \text{ EXCEPT } ![t] =$
$[proposal[t] \text{ EXCEPT } ![i].phase = Apply,$
$![i].state = InProgress]]$
$\land \text{UNCHANGED } \langle transaction \rangle$

If all proposals have been *Complete*, mark the transaction *Complete*.

$\lor \land \forall\, t \in transaction[i].targets :$
$\land proposal[t][i].phase = Apply$
$\land proposal[t][i].state = Complete$
$\land transaction' = [transaction \text{ EXCEPT } ![i].state = Complete,$
$![i].status = Applied]$
$\land \text{UNCHANGED } \langle proposal \rangle$

If any proposal has been *Failed*, mark the transaction *Failed*.

$\lor \land \exists\, t \in transaction[i].targets :$

```
                    ∧ proposal[t][i].phase  = Apply
                    ∧ proposal[t][i].state = Failed
                ∧ transaction' = [transaction EXCEPT ![i].state = Failed]
                ∧ UNCHANGED ⟨proposal⟩
        The Aborting state is used to clean up transactions that have failed during
        the Initializing or Validating phases.
        ∨ ∧ transaction[i].phase = Abort
          ∧ transaction[i].state  = InProgress
              Move the transaction's proposals to the Aborting state
          ∧ ∨ ∧ ∃ t ∈ transaction[i].targets :
                    ∧ proposal[t][i].phase ≠ Abort
                    ∧ proposal' = [proposal EXCEPT ![t] =
                                       [proposal[t] EXCEPT ![i].phase = Abort,
                                                           ![i].state  = InProgress]]
                ∧ UNCHANGED ⟨transaction⟩
              If all proposals have been Complete, mark the transaction Complete.
            ∨ ∧ ∀ t ∈ transaction[i].targets :
                    ∧ proposal[t][i].phase  = Abort
                    ∧ proposal[t][i].state = Complete
                ∧ transaction' = [transaction EXCEPT ![i].state  = Complete,
                                                     ![i].status = Aborted]
                ∧ UNCHANGED ⟨proposal⟩
    ∧ UNCHANGED ⟨configuration, mastership, target⟩

  Reconcile a proposal
ReconcileProposal(n, t, i) ≜
    ∧ ∨ ∧ proposal[t][i].phase = Initialize
        ∧ proposal[t][i].state  = InProgress
        ∧ proposal' = [proposal EXCEPT ![t] =
              [proposal[t] EXCEPT ![i].state  = Complete,
                                  ![i].dependency.index = configuration[t].proposal.index]]
        ∧ configuration' = [configuration EXCEPT ![t].proposal.index = i]
        ∧ UNCHANGED ⟨target⟩
      While in the Validate phase, validate the proposed changes.
      If validation is successful, the proposal also records the changes
      required to roll back the proposal and the index to which to roll back.
      ∨ ∧ proposal[t][i].phase = Validate
        ∧ proposal[t][i].state  = InProgress
        ∧ configuration[t].commit.index = proposal[t][i].dependency.index
            For Change proposals validate the set of requested changes.
        ∧ ∨ ∧ proposal[t][i].type = Change
              ∧ LET rollbackIndex  ≜ configuration[t].config.index
                    rollbackValues ≜ [p ∈ DOMAIN proposal[t][i].change.values ↦
                                        IF p ∈ DOMAIN configuration[t].config.values THEN
                                            configuration[t].config.values[p]
```

```
                ELSE
                  [value  ↦ Nil,
                   delete ↦ TRUE]]
    Model validation successes and failures with Valid and Invalid results.
IN  ∃ r ∈ {Valid, Invalid} :
      If the Change is Valid, record the changes required to roll
      back the proposal and the index to which the rollback changes
      will roll back the configuration.
    ∨ ∧ r = Valid
      ∧ proposal' = [proposal EXCEPT ![t] =
                       [proposal[t] EXCEPT ![i].rollback.index  = rollbackIndex,
                                           ![i].rollback.values = rollbackValues,
                                           ![i].state           = Complete]]
    ∨ ∧ r = Invalid
      ∧ proposal' = [proposal EXCEPT ![t] =
                       [proposal[t] EXCEPT ![i].state = Failed]]
For Rollback proposals, validate the rollback changes which are
proposal being rolled back.
∨ ∧ proposal[t][i].type = Rollback
    Rollbacks can only be performed on Change type proposals.
  ∧ ∨ ∧ proposal[t][proposal[t][i].rollback.index].type = Change
      Only roll back the change if it's the lastest change made
      to the configuration based on the configuration index.
    ∧ ∨ ∧ configuration[t].config.index = proposal[t][i].rollback.index
      ∧ LET changeIndex    ≜ proposal[t][proposal[t][i].rollback.index].rollback.index
            changeValues   ≜ proposal[t][proposal[t][i].rollback.index].rollback.values
            rollbackValues ≜ proposal[t][proposal[t][i].rollback.index].change.values
        IN  ∃ r ∈ {Valid, Invalid} :
              If the Rollback is Valid, record the changes required to
              roll back the target proposal and the index to which the
              configuration is being rolled back.
            ∨ ∧ r = Valid
              ∧ proposal' = [proposal EXCEPT ![t] =
                               [proposal[t] EXCEPT ![i].change.index    = changeIndex,
                                                   ![i].change.values   = changeValues,
                                                   ![i].rollback.values = rollbackValues,
                                                   ![i].state           = Complete]]
            ∨ ∧ r = Invalid
              ∧ proposal' = [proposal EXCEPT ![t] =
                               [proposal[t] EXCEPT ![i].state = Failed]]
      If the Rollback target is not the most recent change to the configuration,
      fail validation for the proposal.
    ∨ ∧ configuration[t].config.index ≠ proposal[t][i].rollback.index
      ∧ proposal' = [proposal EXCEPT ![t] = [proposal[t] EXCEPT ![i].state = Failed]]
If a Rollback proposal is attempting to roll back another Rollback,
```

```
                fail validation for the proposal.
              ∨ ∧ proposal[t][proposal[t][i].rollback.index].type = Rollback
                ∧ proposal' = [proposal EXCEPT ![t] =
                                [proposal[t] EXCEPT ![i].state  = Failed]]
        ∧ UNCHANGED ⟨configuration, target⟩
    While in the Commit state, commit the proposed changes to the configuration.
    ∨ ∧ proposal[t][i].phase = Commit
      ∧ proposal[t][i].state  = InProgress
        Only commit the proposal if the prior proposal has already been committed.
      ∧ configuration[t].commit.index = proposal[t][i].dependency.index
      ∧ configuration' = [configuration EXCEPT ![t].config.values = proposal[t][i].change.values,
                                               ![t].config.index  = proposal[t][i].change.index,
                                               ![t].commit.index  = i]
      ∧ proposal' = [proposal EXCEPT ![t] = [proposal[t] EXCEPT ![i].state = Complete]]
      ∧ UNCHANGED ⟨target⟩
    While in the Apply phase, apply the proposed changes to the target.
    ∨ ∧ proposal[t][i].phase = Apply
      ∧ proposal[t][i].state  = InProgress
      ∧ configuration[t].target.index = proposal[t][i].dependency.index
      ∧ configuration[t].target.term  = mastership[t].term
      ∧ mastership[t].master = n
        Model successful and failed target update requests.
      ∧ ∃ r ∈ {Success, Failure} :
          ∨ ∧ r = Success
            ∧ target' = [target EXCEPT ![t] = proposal[t][i].change.values @@ target[t]]
            ∧ configuration' = [configuration EXCEPT
                                  ![t].target.index  = i,
                                  ![t].target.values = proposal[t][i].change.values
                                    @@ configuration[t].target.values]
            ∧ proposal' = [proposal EXCEPT ![t] = [proposal[t] EXCEPT ![i].state = Complete]]
          If the proposal could not be applied, update the configuration's applied index
          and mark the proposal Failed.
          ∨ ∧ r = Failure
            ∧ configuration' = [configuration EXCEPT ![t].target.index = i]
            ∧ proposal' = [proposal EXCEPT ![t] = [proposal[t] EXCEPT ![i].state = Failed]]
            ∧ UNCHANGED ⟨target⟩
    ∨ ∧ proposal[t][i].phase = Abort
      ∧ proposal[t][i].state  = InProgress
          The commit.index will always be greater than or equal to the target.index.
          If only the commit.index matches the proposal's dependency.index, update
          the commit.index to enable commits of later proposals, but do not
          mark the Abort phase Complete until the target.index has been incremented.
      ∧ ∨ ∧ configuration[t].commit.index = proposal[t][i].dependency.index
          ∧ configuration' = [configuration EXCEPT ![t].commit.index = i]
          ∧ UNCHANGED ⟨proposal⟩
```

```
                If the configuration's target.index matches the proposal's dependency.index,
                update the target.index and mark the proposal Complete for the Abort phase.
              ∨ ∧ configuration[t].commit.index ≥ i
                ∧ configuration[t].target.index  = proposal[t][i].dependency.index
                ∧ configuration' = [configuration EXCEPT ![t].target.index = i]
                ∧ proposal' = [proposal EXCEPT ![t] = [proposal[t] EXCEPT ![i].state = Complete]]
                If both the configuration's commit.index and target.index match the
                proposal's dependency.index, update the commit.index and target.index
                and mark the proposal Complete for the Abort phase.
              ∨ ∧ configuration[t].commit.index = proposal[t][i].dependency.index
                ∧ configuration[t].target.index  = proposal[t][i].dependency.index
                ∧ configuration' = [configuration EXCEPT ![t].commit.index = i,
                                                         ![t].target.index  = i]
                ∧ proposal' = [proposal EXCEPT ![t] = [proposal[t] EXCEPT ![i].state = Complete]]
        ∧ UNCHANGED ⟨target⟩
    ∧ UNCHANGED ⟨transaction, mastership⟩

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

This section models the Configuration reconciler.
ReconcileConfiguration(n, t) ≜
    ∧ ∨ ∧ Target[t].persistent
        ∧ configuration[t].state ≠ Complete
        ∧ configuration' = [configuration EXCEPT ![t].state = Complete]
        ∧ UNCHANGED ⟨target⟩
      ∨ ∧ ¬Target[t].persistent
        ∧ ∨ mastership[t].term > configuration[t].config.term
          ∨ ∧ mastership[t].term = configuration[t].config.term
            ∧ mastership[t].master = Nil
        ∧ configuration' = [configuration EXCEPT ![t].config.term = mastership[t].term,
                                                 ![t].state       = InProgress]
        ∧ UNCHANGED ⟨target⟩
      ∨ ∧ configuration[t].state = InProgress
        ∧ mastership[t].term = configuration[t].config.term
        ∧ mastership[t].master = n
        ∧ target' = [target EXCEPT ![t] = configuration[t].target.values]
        ∧ configuration' = [configuration EXCEPT ![t].target.term = mastership[t].term,
                                                 ![t].state       = Complete]
    ∧ UNCHANGED ⟨proposal, transaction, mastership⟩

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

Formal specification, constraints, and theorems.
Init ≜
    ∧ transaction = [i ∈ {} ↦
                        [type   ↦ Change,
```

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\quad phase \mapsto Initialize, \\
\qquad\qquad\qquad\qquad\quad state \;\mapsto InProgress, \\
\qquad\qquad\qquad\qquad\quad status \mapsto Pending]] \\
\quad \land\, proposal = [t \in \text{DOMAIN } Target \mapsto \\
\qquad\qquad\qquad\quad [i \in \{\} \;\mapsto \\
\qquad\qquad\qquad\qquad [phase \mapsto Initialize, \\
\qquad\qquad\qquad\qquad\; state \;\mapsto InProgress]]] \\
\quad \land\, configuration = [t \in \text{DOMAIN } Target \mapsto \\
\qquad\qquad\qquad\qquad\quad [state \;\mapsto InProgress, \\
\qquad\qquad\qquad\qquad\quad config \mapsto \\
\qquad\qquad\qquad\qquad\qquad [index \;\mapsto 0, \\
\qquad\qquad\qquad\qquad\qquad\; term \;\mapsto 0, \\
\qquad\qquad\qquad\qquad\qquad\; values \mapsto \\
\qquad\qquad\qquad\qquad\qquad\quad [path \in \{\} \mapsto \\
\qquad\qquad\qquad\qquad\qquad\qquad [path \;\mapsto path, \\
\qquad\qquad\qquad\qquad\qquad\qquad\; value \;\mapsto Nil, \\
\qquad\qquad\qquad\qquad\qquad\qquad\; index \;\mapsto 0, \\
\qquad\qquad\qquad\qquad\qquad\qquad\; deleted \mapsto \text{FALSE}]]], \\
\qquad\qquad\qquad\qquad\quad proposal \;\mapsto [index \mapsto 0], \\
\qquad\qquad\qquad\qquad\quad commit \;\mapsto [index \mapsto 0], \\
\qquad\qquad\qquad\qquad\quad target \;\mapsto \\
\qquad\qquad\qquad\qquad\qquad [index \;\mapsto 0, \\
\qquad\qquad\qquad\qquad\qquad\; term \;\mapsto 0, \\
\qquad\qquad\qquad\qquad\qquad\; values \mapsto \\
\qquad\qquad\qquad\qquad\qquad\quad [path \in \{\} \mapsto \\
\qquad\qquad\qquad\qquad\qquad\qquad [path \;\mapsto path, \\
\qquad\qquad\qquad\qquad\qquad\qquad\; value \;\mapsto Nil, \\
\qquad\qquad\qquad\qquad\qquad\qquad\; index \;\mapsto 0, \\
\qquad\qquad\qquad\qquad\qquad\qquad\; deleted \mapsto \text{FALSE}]]]]] \\
\quad \land\, target = [t \in \text{DOMAIN } Target \mapsto \\
\qquad\qquad\qquad [path \in \{\} \mapsto \\
\qquad\qquad\qquad\quad [value \mapsto Nil]]] \\
\quad \land\, mastership = [t \in \text{DOMAIN } Target \mapsto [master \mapsto Nil,\ term \mapsto 0]]
\end{array}
$$

$$
\begin{array}{l}
Next \;\triangleq \\
\quad \lor \exists\, i \in 1 \,..\, NumTransactions : \\
\qquad \exists\, c \in ValidChanges : \\
\qquad\quad RequestChange(i,\ c) \\
\quad \lor \exists\, i \in 1 \,..\, NumTransactions : \\
\qquad \exists\, j \in \text{DOMAIN } transaction : \\
\qquad\quad RequestRollback(i,\ j) \\
\quad \lor \exists\, n \in Node : \\
\qquad \exists\, t \in \text{DOMAIN } Target : \\
\qquad\quad SetMaster(n,\ t) \\
\quad \lor \exists\, t \in \text{DOMAIN } Target :
\end{array}
$$

```
    UnsetMaster(t)
  ∨ ∃ n ∈ Node :
      ∃ t ∈ DOMAIN transaction :
        ReconcileTransaction(n, t)
  ∨ ∃ n ∈ Node :
      ∃ t ∈ DOMAIN proposal :
        ∃ i ∈ DOMAIN proposal[t] :
          ReconcileProposal(n, t, i)
  ∨ ∃ n ∈ Node :
      ∃ c ∈ DOMAIN configuration :
        ReconcileConfiguration(n, c)

Spec ≜ Init ∧ □[Next]_vars ∧ WF_vars(Next)

Order ≜
  ∀ t ∈ DOMAIN proposal :
    ∀ i ∈ DOMAIN proposal[t] :
      ∧ ∧ proposal[t][i].phase = Commit
        ∧ proposal[t][i].state  = InProgress
        ⇒ ¬∃ j ∈ DOMAIN proposal[t] :
              ∧ j > i
              ∧ proposal[t][j].phase = Commit
              ∧ proposal[t][j].state  = Complete
      ∧ ∧ proposal[t][i].phase = Apply
        ∧ proposal[t][i].state  = InProgress
        ⇒ ¬∃ j ∈ DOMAIN proposal[t] :
              ∧ j > i
              ∧ proposal[t][j].phase = Apply
              ∧ proposal[t][j].state  = Complete

Consistency ≜
  ∀ t ∈ DOMAIN target :
    LET
        Compute the transaction indexes that have been applied to the target
      targetIndexes ≜ {i ∈ DOMAIN transaction :
                          ∧ i ∈ DOMAIN proposal[t]
                          ∧ proposal[t][i].phase = Apply
                          ∧ proposal[t][i].state  = Complete
                          ∧ t ∈ transaction[i].targets
                          ∧ ¬∃ j ∈ DOMAIN transaction :
                                ∧ j > i
                                ∧ transaction[j].type = Rollback
                                ∧ transaction[j].rollback = i
                                ∧ transaction[j].phase = Apply
                                ∧ transaction[j].state  = Complete}
        Compute the set of paths in the target that have been updated by transactions
```

$appliedPaths \triangleq \text{UNION } \{\text{DOMAIN } proposal[t][i].change.values : i \in targetIndexes\}$

Compute the highest index applied to the target for each path

$pathIndexes \triangleq [p \in appliedPaths \mapsto \text{CHOOSE } i \in targetIndexes :$
$\forall j \in targetIndexes :$
$\land i \geq j$
$\land p \in \text{DOMAIN } proposal[t][i].change.values]$

Compute the expected target configuration based on the last indexes applied to the target for each path.

$expectedConfig \triangleq [p \in \text{DOMAIN } pathIndexes \mapsto proposal[t][pathIndexes[p]].change.values[p]]$

IN

$target[t] = expectedConfig$

$Isolation \triangleq$
$\forall i \in \text{DOMAIN } transaction :$
$\land \land transaction[i].phase = Commit$
$\land transaction[i].state = InProgress$
$\land transaction[i].isolation = Serializable$
$\Rightarrow \neg\exists j \in \text{DOMAIN } transaction :$
$\land j > i$
$\land transaction[j].targets \cap transaction[i].targets \neq \{\}$
$\land transaction[j].phase = Commit$
$\land \land transaction[i].phase = Apply$
$\land transaction[i].state = InProgress$
$\land transaction[i].isolation = Serializable$
$\Rightarrow \neg\exists j \in \text{DOMAIN } transaction :$
$\land j > i$
$\land transaction[j].targets \cap transaction[i].targets \neq \{\}$
$\land transaction[j].phase = Apply$

$Safety \triangleq \Box(Order \land Consistency \land Isolation)$

THEOREM $Spec \Rightarrow Safety$

$Terminated(i) \triangleq$
$\land i \in \text{DOMAIN } transaction$
$\land transaction[i].phase \in \{Apply, Abort\}$
$\land transaction[i].state = Complete$

$Termination \triangleq$
$\forall i \in 1 \,..\, NumTransactions : Terminated(i)$

$Liveness \triangleq \Diamond Termination$

THEOREM $Spec \Rightarrow Liveness$

Type assumptions.

ASSUME $Nil \in$ STRING

ASSUME $\forall\, phase \in Phase : phase \in$ STRING

ASSUME $\forall\, state \in State : state \in$ STRING

ASSUME $\forall\, status \in Status : status \in$ STRING

ASSUME $\land\, IsFiniteSet(Node)$
$\land\, \forall\, n \in Node :$
$\land\, n \notin$ DOMAIN $Target$
$\land\, n \in$ STRING

ASSUME $\land\, \forall\, t \in$ DOMAIN $Target :$
$\land\, t \notin Node$
$\land\, t \in$ STRING
$\land\, Target[t].persistent \in$ BOOLEAN
$\land\, \forall\, p \in$ DOMAIN $Target[t].values :$
$IsFiniteSet(Target[t].values[p])$

* Modification History
* Last modified *Thu Feb* 10 15:59:15 *PST* 2022 by *jordanhalterman*
* Created *Wed Sep* 22 13:22:32 *PDT* 2021 by *jordanhalterman*