MODULE *Config*

INSTANCE *Naturals*

INSTANCE *FiniteSets*

INSTANCE *Sequences*

INSTANCE *TLC*

An empty constant

CONSTANT *Nil*

Transaction type constants

CONSTANTS
TransactionChange,
TransactionRollback

Transaction status constants

CONSTANTS
TransactionPending,
TransactionValidating,
TransactionApplying,
TransactionComplete,
TransactionFailed

Configuration status constants

CONSTANTS
ConfigurationPending,
ConfigurationInitializing,
ConfigurationUpdating,
ConfigurationComplete,
ConfigurationFailed

The set of all nodes

CONSTANT *Node*

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

Example: *Target* $\triangleq$ [
*target*1 $\mapsto$ [
*path*1 $\mapsto$ {"*value*1", "*value*2"},
*path*2 $\mapsto$ {"*value*2", "*value*3"}],
*target*2 $\mapsto$ [
*path*2 $\mapsto$ {"*value*3", "*value*4"},
*path*3 $\mapsto$ {"*value*4", "*value*5"}]]

CONSTANT *Target*

ASSUME $Nil \in$ STRING

ASSUME $TransactionPending \in$ STRING
ASSUME $TransactionValidating \in$ STRING
ASSUME $TransactionApplying \in$ STRING
ASSUME $TransactionComplete \in$ STRING
ASSUME $TransactionFailed \in$ STRING

ASSUME $ConfigurationPending \in$ STRING
ASSUME $ConfigurationInitializing \in$ STRING
ASSUME $ConfigurationUpdating \in$ STRING
ASSUME $ConfigurationComplete \in$ STRING
ASSUME $ConfigurationFailed \in$ 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
$\land \forall\, p \in$ DOMAIN $Target[t] :$
$IsFiniteSet(Target[t][p])$

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 $TransactionType ::= type \in$
$\{TransactionChange,$
$TransactionRollback\}$

TYPE $TransactionStatus ::= status \in$
$\{TransactionPending,$
$TransactionValidating,$
$TransactionApplying,$
$TransactionComplete,$
$TransactionFailed\}$

TYPE Transaction $\triangleq$ [
$type ::= type \in TransactionType,$
$index ::= index \in Nat,$
revision ::= revision $\in Nat,$
$atomic ::= atomic \in$ BOOLEAN ,
$sync ::= sync \in$ BOOLEAN ,
$changes ::=$ [$target \in$ SUBSET (DOMAIN $Target$) $\mapsto$ [
$path \in$ SUBSET (DOMAIN $Target[target]$) $\mapsto$ [

```
        value ::= value ∈ STRING,
        delete ::= delete ∈ BOOLEAN ]]],
  rollback ::= index ∈ Nat,
  status ::= status ∈ TransactionStatus]

TYPE ConfigurationStatus ::= status ∈
  {ConfigurationPending,
   ConfigurationInitializing,
   ConfigurationUpdating,
   ConfigurationComplete,
   ConfigurationFailed}

TYPE Configuration ≜ [
  id       ::= id ∈ STRING,
  revision ::= revision ∈ Nat,
  target ::= target ∈ STRING,
  paths ::= [ path ∈ SUBSET (DOMAIN Target[target]) ↦ [
      value ::= value ∈ STRING,
      index ::= index ∈ Nat,
      deleted ::= delete ∈ BOOLEAN ]],
  txIndex ::= txIndex ∈ Nat,
  syncIndex ::= syncIndex ∈ Nat,
  term      ::= term ∈ Nat,
  status ::= status ∈ ConfigurationStatus]
```

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 configurations

VARIABLE *configuration*

A record of target states

VARIABLE *target*

A record of target masters

VARIABLE *master*

A history variable tracking past configuration changes

VARIABLE *history*

$vars \triangleq \langle transaction, configuration, master, target, history \rangle$

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

$SetMaster(n, t) \triangleq$

$\land\ master[t].master \neq n$
$\land\ master' = [master \text{ EXCEPT } ![t].term \quad = master[t].term + 1,$
$\qquad\qquad\qquad\qquad\qquad\qquad ![t].master = n]$
$\land$ UNCHANGED $\langle transaction,\ configuration,\ target,\ history\rangle$

$UnsetMaster(t) \triangleq$
$\land\ master[t].master \neq Nil$
$\land\ master' = [master \text{ EXCEPT } ![t].master = Nil]$
$\land$ UNCHANGED $\langle transaction,\ configuration,\ target,\ history\rangle$

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

$Value(s,\ t,\ p) \triangleq$
LET $value \triangleq$ CHOOSE $v \in s : v.target = t \land v.path = p$
IN
$[value \mapsto value.value,$
$\ delete \mapsto value.delete]$

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

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

$ValidValues(t,\ p) \triangleq$
UNION $\{\{[value \mapsto v,\ delete \mapsto$ FALSE$] : v \in Target[t][p]\},\ \{[value \mapsto Nil,\ delete \mapsto$ TRUE$]\}\}$

$ValidPaths(t) \triangleq$
UNION $\{\{v \mathbin{@@} [path \mapsto p] : v \in ValidValues(t,\ p)\} : p \in$ DOMAIN $Target[t]\}$

$ValidTargets \triangleq$
UNION $\{\{p \mathbin{@@} [target \mapsto t] : p \in ValidPaths(t)\} : t \in$ DOMAIN $Target\}$

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

The next available index in the transaction log.
This is computed as the max of the existing indexes in the log to
allow for changes to the log (*e.g.* log compaction) to be modeled.

$NextIndex \triangleq$
IF DOMAIN $transaction = \{\}$ THEN
1
ELSE
LET $i \triangleq$ CHOOSE $i \in$ DOMAIN $transaction$:
$\forall j \in$ DOMAIN $transaction : i \geq j$
IN $i + 1$

Add a set of changes 'c' to the transaction log
$Change(c) \triangleq$
$\land \quad transaction' = transaction \,@@\, (NextIndex :> [type \mapsto TransactionChange,$
$index \mapsto NextIndex,$
$atomic \mapsto$ FALSE,
$sync \mapsto$ FALSE,
$changes \mapsto c,$
$sources \mapsto \langle\rangle,$
$status \mapsto TransactionPending])$
$\land \quad$ UNCHANGED $\langle configuration, master, target, history\rangle$

Add a rollback of transaction 't' to the transaction log
$Rollback(t) \triangleq$
$\land\ transaction[t].type = TransactionChange$
$\land\ transaction' = transaction \,@@\, (NextIndex :> [type \mapsto TransactionRollback,$
$index \mapsto NextIndex,$
$atomic \mapsto$ FALSE,
$sync \mapsto$ FALSE,
$rollback \mapsto t,$
$status \mapsto TransactionPending])$
$\land$ UNCHANGED $\langle configuration, master, target, history\rangle$

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

Both types of transaction are reconciled in stages:
* Pending - waiting for prior transactions to complete
* Validating - validating the requested changes
* Applying - applying the changes to target configurations
* Complete - completed applying changes successfully
* Failed - failed applying changes

Reconcile a change transaction
$ReconcileChange(n,\ i) \triangleq$
If the transaction is Pending, begin validation if the prior transaction has already been applied. This simplifies concurrency control in the controller

```
and guarantees transactions are applied to the configurations in sequential order.
\/ /\ transaction[i].status = TransactionPending
   /\ \/ /\ i - 1 \in DOMAIN transaction
         /\ transaction[i - 1].status \in {TransactionComplete, TransactionFailed}
      \/ i - 1 \notin DOMAIN transaction
   /\ transaction' = [transaction EXCEPT ![i].status = TransactionValidating]
   /\ UNCHANGED <<configuration, history>>
If the transaction is in the Validating state, compute and validate the
Configuration for each target.
\/ /\ transaction[i].status = TransactionValidating
   If validation fails any target, mark the transaction Failed.
   If validation is successful, proceed to Applying.
   /\ \E valid \in BOOLEAN :
      IF valid THEN
         /\ transaction' = [transaction EXCEPT ![i].status = TransactionApplying]
      ELSE
         /\ transaction' = [transaction EXCEPT ![i].status = TransactionFailed]
   /\ UNCHANGED <<configuration, history>>
If the transaction is in the Applying state, update the Configuration for each
target and Complete the transaction.
\/ /\ transaction[i].status = TransactionApplying
   Update the target configurations, adding the transaction index to each updated path
   /\ configuration' = [
        t \in DOMAIN Target |->
          IF t \in DOMAIN transaction[i].changes THEN
            [configuration[t] EXCEPT
              !.paths = [p \in DOMAIN transaction[i].changes[t] |->
                           [index   |-> transaction[i].index,
                            value   |-> transaction[i].changes[t][p].value,
                            deleted |-> transaction[i].changes[t][p].delete]]
                        @@ configuration[t].paths,
              !.txIndex = transaction[i].index,
              !.status  = ConfigurationPending]
          ELSE
            configuration[t]]
   /\ history' = [r \in DOMAIN Target |-> Append(history[r], configuration'[r])]
   The transaction state is updated to include the source configuration modified.
   The source configuration is used to optimize rollbacks.
   Note that in a real-world implementation, the order of updates to the configuration
   and to add the source info to the transaction could have serious ramifications.
   If one is updated without the other, rollbacks may not be possible.
   /\ transaction' = [transaction EXCEPT
        ![i].status  = TransactionComplete,
        ![i].sources = [t \in DOMAIN transaction[i].changes |->
          LET updatePaths \triangleq {p \in DOMAIN transaction[i].changes[t] :
```

```
                                    ¬transaction[i].changes[t][p].delete}
            IN    [p ∈ updatePaths ∩ DOMAIN configuration[t].paths ↦ configuration[t].paths[p]]]]

 Reconcile a rollback transaction
ReconcileRollback(n, i) ≜
    If the transaction is Pending, begin validation if the prior transaction
    has already been applied. This simplifies concurrency control in the controller
    and guarantees transactions are applied to the configurations in sequential order.
  ∨ ∧ transaction[i].status = TransactionPending
    ∧ ∨ ∧ i − 1 ∈ DOMAIN transaction
          ∧ transaction[i − 1].status ∈ {TransactionComplete, TransactionFailed}
      ∨ i − 1 ∉ DOMAIN transaction
    ∧ transaction′ = [transaction EXCEPT ![i].status = TransactionValidating]
    ∧ UNCHANGED ⟨configuration, history⟩
    If the transaction is in the Validating state, validate the rollback.
    A transaction can only be rolled back if:
    1. The source transaction is in the log
    2. The source transaction was applied successfully (did not fail validation)
    3. The source transaction is the most recent change for each path is modified
  ∨ ∧ transaction[i].status = TransactionValidating
    ∧ ∨ ∧ transaction[transaction[i].rollback].status = TransactionComplete
        ∧ ∨ ∧ transaction[i].rollback ∈ DOMAIN transaction
              Determine whether the source transaction is the most recent change
              by comparing the configuration path indexes to the transaction index.
            ∧ LET canRollback ≜ ∀ t ∈ DOMAIN transaction[transaction[i].rollback].changes :
                                  ∀ p ∈ DOMAIN transaction[transaction[i].rollback].changes[t] :
                                    configuration[t].paths[p].index = transaction[i].rollback
              IN
                IF canRollback THEN
                    ∧ transaction′ = [transaction EXCEPT ![i].status = TransactionApplying]
                 ELSE
                    ∧ transaction′ = [transaction EXCEPT ![i].status = TransactionFailed]
          If the source transaction is not in the log, fail the rollback.
          ∨ ∧ transaction[i].rollback ∉ DOMAIN transaction
            ∧ transaction′ = [transaction EXCEPT ![i].status = TransactionFailed]
      If the source transaction failed, fail the rollback.
      ∨ ∧ transaction[transaction[i].rollback].status = TransactionFailed
        ∧ transaction′ = [transaction EXCEPT ![i].status = TransactionFailed]
    ∧ UNCHANGED ⟨configuration, history⟩
    If the transcation is in the Applying state, roll back the Configuration for
    each target and Complete the transaction.
  ∨ ∧ transaction[i].status = TransactionApplying
      Target configurations are rolled back by reverting to the source paths/values
      stored in the transaction when it was applied.
    ∧ configuration′ = [
```

```
        t ∈ DOMAIN Target ↦
          IF t ∈ DOMAIN transaction[transaction[i].rollback].changes THEN
             LET adds      ≜ {p ∈ DOMAIN transaction[transaction[i].rollback].changes[t] :
                                 ∧ p ∉ DOMAIN transaction[transaction[i].rollback].sources[t]
                                 ∧ ¬transaction[transaction[i].rollback].changes[t][p].delete}
                 updates   ≜ {p ∈ DOMAIN transaction[transaction[i].rollback].changes[t] :
                                 ∧ p ∈ DOMAIN transaction[transaction[i].rollback].sources[t]
                                 ∧ ¬transaction[transaction[i].rollback].changes[t][p].delete}
                 removes   ≜ {p ∈ DOMAIN transaction[transaction[i].rollback].changes[t] :
                                 ∧ p ∈ DOMAIN transaction[transaction[i].rollback].sources[t]
                                 ∧ transaction[transaction[i].rollback].changes[t][p].delete}
                 changes   ≜ adds ∪ updates ∪ removes
                 unchanges ≜ DOMAIN configuration[t].paths \ changes
             IN
                [configuration[t] EXCEPT
                   !.paths = [p ∈ unchanges ↦ configuration[t].paths[p]]
                                @@ [p ∈ updates ∪ removes ↦
                                       transaction[transaction[i].rollback].sources[t][p]],
                   !.txIndex = transaction[i].index,
                   !.status  = ConfigurationPending]
          ELSE
             configuration[t]]
      ∧ history' = [r ∈ DOMAIN Target ↦ Append(history[r], configuration'[r])]
      ∧ transaction' = [transaction EXCEPT ![i].status = TransactionComplete]

Reconcile a transaction in the transaction log
Transactions can be of one of two types: Change and Rollback.
The logic for processing different types of transactions differs.
ReconcileTransaction(n, i) ≜
   ∧ ∨ ∧ transaction[i].type = TransactionChange
       ∧ ReconcileChange(n, i)
     ∨ ∧ transaction[i].type = TransactionRollback
       ∧ ReconcileRollback(n, i)
   ∧ UNCHANGED ⟨master, target⟩
```

```
This section models the Configuration reconciler.
ReconcileConfiguration(n, c) ≜
   ∧ ∨ ∧ configuration[c].status = ConfigurationPending
       ∧ master[configuration[c].target].master ≠ Nil
          If the configuration is marked ConfigurationPending and mastership
          has changed (indicated by an increased mastership term), mark the
          configuration ConfigurationInitializing to force full re-synchronization.
       ∧ ∨ ∧ master[configuration[c].target].term > configuration[c].term
           ∧ configuration' = [configuration EXCEPT ![c].status = ConfigurationInitializing,
```

```
                                                    ![c].term  = master[configuration[c].target].term]
          ∧ history' = [history EXCEPT ![c] = Append(history[c], configuration'[c])]
        If the configuration is marked ConfigurationPending and the values have
        changed (determined by comparing the transaction index to the last sync
        index), mark the configuration ConfigurationUpdating to push the changes
        to the target.
      ∨ ∧ master[configuration[c].target].term = configuration[c].term
        ∧ configuration[c].syncIndex < configuration[c].txIndex
        ∧ configuration' = [configuration EXCEPT ![c].status = ConfigurationUpdating]
        ∧ history' = [history EXCEPT ![c] = Append(history[c], configuration'[c])]
    ∧ UNCHANGED ⟨target⟩
∨ ∧ configuration[c].status = ConfigurationInitializing
  ∧ master[configuration[c].target].master = n
    Merge the configuration paths with the target paths, removing paths
    that have been marked deleted
  ∧ LET deletePaths ≜ {p ∈ DOMAIN configuration[c].paths : configuration[c].paths[p].deleted}
        configPaths ≜ DOMAIN configuration[c].paths \ deletePaths
        targetPaths ≜ DOMAIN target[configuration[c].target] \ deletePaths
    IN
      ∧ target' = [target EXCEPT ![configuration[c].target] =
                    [p ∈ configPaths ↦ [value ↦ configuration[c].paths[p]]]
                      @@ [p ∈ targetPaths ↦ target[configuration[c].target][p]]]
        Set the configuration's status to Complete
  ∧ configuration' = [configuration EXCEPT ![c].status    = ConfigurationComplete,
                                           ![c].syncIndex = configuration[c].txIndex]
  ∧ history' = [history EXCEPT ![c] = Append(history[c], configuration'[c])]
If the configuration is marked ConfigurationUpdating, we only need to
push paths that have changed since the target was initialized or last
updated by the controller. The set of changes made since the last
synchronization are identified by comparing the index of each path-value
to the last synchronization index, syncIndex
∨ ∧ configuration[c].status = ConfigurationUpdating
  ∧ master[configuration[c].target].master = n
    Compute the set of updated and deleted paths by comparing
    their indexes to the target s last sync index.
  ∧ LET updatePaths ≜ {p ∈ DOMAIN configuration[c].paths :
                         configuration[c].paths[p].index > configuration[c].syncIndex}
        deletePaths ≜ {p ∈ updatePaths : configuration[c].paths[p].deleted}
        configPaths ≜ updatePaths \ deletePaths
        targetPaths ≜ DOMAIN target[configuration[c].target] \ deletePaths
    IN
      Update the target paths by adding/updating paths that have changed and
      removing paths that have been deleted since the last sync.
      ∧ target' = [target EXCEPT ![configuration[c].target] =
                    [p ∈ configPaths ↦ configuration[c].paths[p]]
```

$$@@\,[p \in targetPaths \mapsto target[configuration[c].target][p]]]$$

$\land\ configuration' = [configuration \text{ EXCEPT } ![c].status = ConfigurationComplete,$
$![c].syncIndex = configuration[c].txIndex]$

$\land\ history' = [history \text{ EXCEPT } ![c] = Append(history[c],\ configuration'[c])]$

If the configuration is not already *ConfigurationPending* and mastership has been lost revert it. This can occur when the connection to the target has been lost and the mastership is no longer valid.

TODO: We still need to model mastership changes

$\lor\ \land\ configuration[c].status \neq ConfigurationPending$
$\land\ master[configuration[c].target].master = Nil$
$\land\ configuration' = [configuration \text{ EXCEPT } ![c].status = ConfigurationPending]$
$\land\ history' = [history \text{ EXCEPT } ![c] = Append(history[c],\ configuration'[c])]$
$\land\ \text{UNCHANGED } \langle target \rangle$

$\land\ \text{UNCHANGED } \langle transaction,\ master \rangle$

Init and next state predicates

$$
\begin{aligned}
Init\ \triangleq\ & \\
& \land\ transaction = \langle\rangle \\
& \land\ configuration = [t \in \text{DOMAIN } Target \mapsto \\
& \qquad [target \mapsto t, \\
& \qquad\ paths \mapsto \\
& \qquad\quad [path \in \{\} \mapsto \\
& \qquad\qquad [path \mapsto path, \\
& \qquad\qquad\ value \mapsto Nil, \\
& \qquad\qquad\ index \mapsto 0, \\
& \qquad\qquad\ deleted \mapsto \text{FALSE}]], \\
& \qquad\ txIndex \mapsto 0, \\
& \qquad\ syncIndex \mapsto 0, \\
& \qquad\ term \mapsto 0, \\
& \qquad\ status \mapsto ConfigurationPending]] \\
& \land\ target = [t \in \text{DOMAIN } Target \mapsto \\
& \qquad [path \in \{\} \mapsto \\
& \qquad\quad [value \mapsto Nil]]] \\
& \land\ master = [t \in \text{DOMAIN } Target \mapsto [master \mapsto Nil,\ term \mapsto 0]] \\
& \land\ history = [t \in \text{DOMAIN } Target \mapsto \langle\rangle]
\end{aligned}
$$

$$
\begin{aligned}
Next\ \triangleq\ & \\
& \lor\ \exists\, c \in ValidChanges : \\
& \qquad Change(c) \\
& \lor\ \exists\, t \in \text{DOMAIN } transaction : \\
& \qquad Rollback(t) \\
& \lor\ \exists\, n \in Node : \\
& \qquad \exists\, t \in \text{DOMAIN } Target : \\
& \qquad\ SetMaster(n,\ t)
\end{aligned}
$$

$\lor \exists\, t \in \text{DOMAIN } \mathit{Target} :$
$\quad \mathit{UnsetMaster}(t)$
$\lor \exists\, n \in \mathit{Node} :$
$\quad \exists\, t \in \text{DOMAIN } \mathit{transaction} :$
$\quad\quad \mathit{ReconcileTransaction}(n,\ t)$
$\lor \exists\, n \in \mathit{Node} :$
$\quad \exists\, c \in \text{DOMAIN } \mathit{configuration} :$
$\quad\quad \mathit{ReconcileConfiguration}(n,\ c)$

$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}$

$\mathit{Inv} \triangleq$
$\land \forall\, a,\ b \in \text{DOMAIN } \mathit{transaction} :$
$\quad \mathit{transaction}[a].\mathit{index} > \mathit{transaction}[b].\mathit{index} \Rightarrow$
$\quad\quad (\mathit{transaction}[a].\mathit{status} \in \{\mathit{TransactionComplete},\ \mathit{TransactionFailed}\} \Rightarrow$
$\quad\quad\quad \mathit{transaction}[b].\mathit{status} \in \{\mathit{TransactionComplete},\ \mathit{TransactionFailed}\})$
$\land \forall\, t \in \text{DOMAIN } \mathit{Target} :$
$\quad \forall\, c \in \text{DOMAIN } \mathit{history}[t] :$
$\quad\quad \land\ \mathit{configuration}[t].\mathit{txIndex} \geq \mathit{history}[t][c].\mathit{txIndex}$
$\quad\quad \land\ \mathit{configuration}[t].\mathit{syncIndex} \geq \mathit{history}[t][c].\mathit{syncIndex}$

* Modification History
* Last modified *Tue Jan* 18 20:14:43 *PST* 2022 by *jordanhalterman*
* Created *Wed Sep* 22 13:22:32 *PDT* 2021 by *jordanhalterman*