MODULE *Config*

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

Indicates that a configuration change is waiting to be applied to the network
CONSTANT *Pending*

Indicates that a configuration change has been applied to the network
CONSTANT *Complete*

Indicates that a configuration change failed
CONSTANT *Failed*

Indicates a change is a configuration
CONSTANT *Change*

Indicates a change is a rollback
CONSTANT *Rollback*

Indicates a device is connected
CONSTANT *Connected*

Indicates a device is disconnected
CONSTANT *Disconnected*

Indicates that an error occurred when applying a change
CONSTANT *Error*

The set of all nodes
CONSTANT *Node*

The set of all devices
CONSTANT *Device*

An empty constant
CONSTANT *Nil*

Per-node election state
VARIABLE *leader*

Per-node per-device election state
VARIABLE *master*

A sequence of network-wide configuration changes
Each change contains a record of 'changes' for each device
VARIABLE *networkChange*

A record of sequences of device configuration changes
Each sequence is a list of changes in the order in which they
are to be applied to the device

VARIABLE *deviceChange*

A record of device states - either *Available* or *Unavailable*

VARIABLE *deviceState*

A count of leader changes to serve as a state constraint

VARIABLE *electionCount*

A count of configuration changes to serve as a state constraint

VARIABLE *configCount*

A count of device connection changes to serve as a state constraint

VARIABLE *connectionCount*

Node variables

$nodeVars \triangleq \langle leader, master \rangle$

Configuration variables

$configVars \triangleq \langle networkChange, deviceChange \rangle$

Device variables

$deviceVars \triangleq \langle deviceState \rangle$

State constraint variables

$constraintVars \triangleq \langle electionCount, configCount, connectionCount \rangle$

$vars \triangleq \langle nodeVars, configVars, deviceVars, constraintVars \rangle$

This section models leader election for control loops and for devices. Leader election is modelled as a simple boolean indicating whether each node is the leader for the cluster and for each device. This model implies the ordering of leadership changes is irrelevant to the correctness of the spec.

Set the leader for node n to l

```
$SetNodeLeader(n, l) \triangleq$
    $\land leader' = [leader \text{ EXCEPT } ![n] = n = l]$
    $\land electionCount' = electionCount + 1$
    $\land \text{UNCHANGED } \langle master, configVars, deviceVars, configCount, connectionCount \rangle$
```

Set the master for device d on node n to l

```
$SetDeviceMaster(n, d, l) \triangleq$
    $\land master' = [master \text{ EXCEPT } ![n] = [master[n] \text{ EXCEPT } ![d] = n = l]]$
    $\land electionCount' = electionCount + 1$
    $\land \text{UNCHANGED } \langle leader, configVars, deviceVars, configCount, connectionCount \rangle$
```

This section models the northbound *API* for the configuration service.

Enqueue network configuration change c

```
$SubmitChange(c) \triangleq$
    $\land\ Cardinality(\textsc{domain}\ c) > 0$
    $\land\ networkChange' = Append(networkChange, [$
                                    $phase \mapsto Change,$
                                    $changes \mapsto c,$
                                    $value \mapsto Len(networkChange),$
                                    $state \mapsto Pending,$
                                    $attempt \mapsto 0])$
    $\land\ configCount' = configCount + 1$
    $\land\ \textsc{unchanged}\ \langle nodeVars, deviceChange, deviceVars, electionCount, connectionCount\rangle$

$RollbackChange(c) \triangleq$
    $\land\ networkChange[c].phase = Change$
    $\land\ networkChange[c].state\ = Complete$
    $\land\ networkChange' = [networkChange\ \textsc{except}\ ![c].phase = Rollback,\ ![c].state = Pending]$
    $\land\ configCount' = configCount + 1$
    $\land\ \textsc{unchanged}\ \langle nodeVars, deviceChange, deviceVars, electionCount, connectionCount\rangle$
```

This section models a configuration change scheduler. The role of the scheduler is to determine when network changes can be applied and enqueue the relevant changes for application by changing their state from *Pending* to *Applying*. The scheduler supports concurrent application of non-overlapping configuration changes (changes that do not impact intersecting sets of devices) by comparing *Pending* changes with Applying changes.

Return the set of all network changes prior to the given change

```
$PriorNetworkChanges(c) \triangleq$
    $\{n \in \textsc{domain}\ networkChange : n < c\}$
```

Return the set of all completed device changes for network change c

```
$NetworkCompletedChanges(c) \triangleq$
    $\{d \in \textsc{domain}\ networkChange[c].changes :$
        $\land\ c \in \textsc{domain}\ deviceChange[d]$
        $\land\ deviceChange[d][c].state = Complete\}$
```

Return a boolean indicating whether all device changes are complete for the given network change

```
$NetworkChangesComplete(c) \triangleq$
    $Cardinality(NetworkCompletedChanges(c)) = Cardinality(\textsc{domain}\ networkChange[c].changes)$
```

Return the set of all incomplete device changes prior to network change c

```
$PriorIncompleteDevices(c) \triangleq$
    $\textsc{union}\ \{\textsc{domain}\ networkChange[n].changes :$
                $n \in \{n \in PriorNetworkChanges(c) : \neg NetworkChangesComplete(n)\}\}$
```

Return the set of all devices configured by network change c

```
$NetworkChangeDevices(c) \triangleq \textsc{domain}\ networkChange[c].changes$
```

Return the set of all connected devices configured by network change c

$$\mathit{ConnectedDevices}(c) \triangleq \{d \in \textsc{domain}\ \mathit{networkChange}[c].\mathit{changes} : \mathit{deviceState}[d] = \mathit{Connected}\}$$

Return a boolean indicating whether network change c can be applied
A change can be applied if its devices do not intersect with past device
changes that have not been applied

```
$\mathit{CanApplyNetworkChange}(c) \triangleq$
    $\land\ \mathit{Cardinality}(\mathit{ConnectedDevices}(c) \cup \mathit{NetworkChangeDevices}(c)) = 0$
    $\land\ \mathit{Cardinality}(\mathit{NetworkChangeDevices}(c) \cap \mathit{PriorIncompleteDevices}(c)) = 0$
```

This section models the *NetworkChange* reconciler. The reconciler reconciles network changes when the change or one of its device changes is updated.

Return a boolean indicating whether a change exists for the given device
If the device is modified by the change, it must contain a device change
that's either *Complete* or with the same 'attempt' as the network change.

```
$\mathit{HasDeviceChange}(d,\ c) \triangleq$
    $\lor\ d \notin \textsc{domain}\ \mathit{networkChange}[c].\mathit{changes}$
    $\lor\ \land\ d \in \textsc{domain}\ \mathit{networkChange}[c].\mathit{changes}$
      $\land\ c \in \textsc{domain}\ \mathit{deviceChange}[d]$
      $\land\ \lor\ \mathit{deviceChange}[d][c].\mathit{attempt} = \mathit{networkChange}[c].\mathit{attempt}$
        $\lor\ \mathit{deviceChange}[d].\mathit{state} = \mathit{Complete}$
```

Return a boolean indicating whether device changes have been propagated
for the given network change

```
$\mathit{HasDeviceChanges}(c) \triangleq$
    $\mathit{Cardinality}(\{d \in \mathit{Device} : \mathit{HasDeviceChange}(d,\ c)\}) \neq 0$
```

Add or update the given device changes for the given network change.
If a device change already exists, update the 'attempt' field.

```
$\mathit{CreateDeviceChange}(d,\ c) \triangleq$
    IF $\mathit{Cardinality}(\textsc{domain}\ \mathit{deviceChange}[d]) = 0$ THEN
        $[x \in \{c\} \mapsto [$
                 $\mathit{phase}\ \ \ \mapsto \mathit{networkChange}[c].\mathit{phase},$
                 $\mathit{state}\ \ \ \ \mapsto \mathit{Pending},$
                 $\mathit{value}\ \ \ \ \mapsto \mathit{networkChange}[c].\mathit{value},$
                 $\mathit{attempt} \mapsto \mathit{networkChange}[c].\mathit{attempt}]]$
     ELSE
        IF $d \in \textsc{domain}\ \mathit{networkChange}[c].\mathit{changes}$ THEN
            IF $c \in \textsc{domain}\ \mathit{deviceChange}[d]$ THEN
                IF $\mathit{deviceChange}[d][c].\mathit{state} = \mathit{Complete}$ THEN
                    $\mathit{deviceChange}[d][c]$
                 ELSE
                    $[\mathit{deviceChange}[d]$ EXCEPT $![c].\mathit{attempt} = \mathit{networkChange}[c].\mathit{attempt},$
                                              $![c].\mathit{state} = \mathit{Pending}]$
             ELSE
            $[x \in \{c\} \mapsto [$
                $phase \mapsto networkChange[c].phase,$
                $state \mapsto Pending,$
                $value \mapsto networkChange[c].value,$
                $attempt \mapsto networkChange[c].attempt]] \; @@ \; deviceChange[d]$
        ELSE
            $deviceChange[d]$
```

Add or update device changes for the given network change

```
$CreateDeviceChanges(c) \triangleq$
    $deviceChange' = [d \in \text{DOMAIN } deviceChange \mapsto CreateDeviceChange(d, c)]$
```

Rollback device change c for device d

```
$RollbackDeviceChange(d, c) \triangleq$
    IF $c \in \text{DOMAIN } deviceChange[d]$ THEN
        $[deviceChange[d] \text{ EXCEPT } ![c].phase = Rollback, ![c].state = Pending]$
     ELSE
        $deviceChange[d]$
```

Roll back device changes

```
$RollbackDeviceChanges(c) \triangleq$
    $deviceChange' = [d \in \text{DOMAIN } deviceChange \mapsto RollbackDeviceChange(d, c)]$
```

Return a boolean indicating whether the given device change is *Failed*

```
$IsFailedDeviceChange(d, c) \triangleq$
    $\lor deviceChange[d][c].phase = Change$
    $\lor deviceChange[d][c].attempt = 0$
    $\lor \land deviceChange[d][c].attempt = networkChange[c].attempt$
      $\land deviceChange[d][c].state = Failed$

$IsDeviceChangeRollbackFailed(d, c) \triangleq$
    $\land c \in \text{DOMAIN } deviceChange[d]$
    $\land deviceChange[d][c].attempt = networkChange[c].attempt$
    $\land deviceChange[d][c].phase = Rollback$
    $\land deviceChange[d][c].state = Failed$

$IsDeviceChangeRollbackComplete(d, c) \triangleq$
    $\land c \in \text{DOMAIN } deviceChange[d]$
    $\land deviceChange[d][c].attempt = networkChange[c].attempt$
    $\land deviceChange[d][c].phase = Rollback$
    $\land deviceChange[d][c].state = Complete$
```

Return a boolean indicating whether the given device change is *Complete*

```
$IsCompleteDeviceChange(d, c) \triangleq$
    $\land deviceChange[d][c].phase = Change$
    $\land deviceChange[d][c].state = Complete$
```

```
Return a boolean indicating whether the given device change rollback is Complete
$IsCompleteDeviceRollback(d, c) \triangleq$
    $\land\ deviceChange[d][c].phase = Rollback$
    $\land\ deviceChange[d][c].state\ = Complete$

Return a boolean indicating whether any device change is Complete for the given network change
$HasCompleteDeviceChanges(c) \triangleq$
    $Cardinality(\{d \in \text{DOMAIN}\ deviceChange :$
        $\land\ c \in \text{DOMAIN}\ deviceChange[d]$
        $\land\ IsCompleteDeviceChange(d, c)\}) \neq 0$

Return a boolean indicating whether any device change is Failed for the given network change
$HasFailedDeviceChanges(c) \triangleq$
    $Cardinality(\{d \in \text{DOMAIN}\ deviceChange :$
        $\land\ c \in \text{DOMAIN}\ deviceChange[d]$
        $\land\ IsFailedDeviceChange(d, c)\}) \neq 0$

Return a boolean indicating whether all device changes have been rolled back for the given network change
$DeviceChangesRolledBack(c) \triangleq$
    $Cardinality(\{d \in \text{DOMAIN}\ deviceChange :$
        $\land\ c \in \text{DOMAIN}\ deviceChange[d]$
        $\land\ IsCompleteDeviceRollback(d, c)\}) =$
            $Cardinality(\text{DOMAIN}\ networkChange[c].changes)$

Return a boolean indicating whether all device changes are Complete for the given network change
$DeviceChangesComplete(c) \triangleq$
    $Cardinality(\{d \in \text{DOMAIN}\ deviceChange :$
        $\land\ c \in \text{DOMAIN}\ deviceChange[d]$
        $\land\ IsCompleteDeviceChange(d, c)\}) =$
          $Cardinality(\text{DOMAIN}\ networkChange[c].changes)$

Reconcile a network change state
$ReconcileNetworkChange(n, c) \triangleq$
    $\land\ leader[n]$
    $\land\ networkChange[c].state = Pending$
        Create device changes if necessary
    $\land\ \lor\ \land\ \neg HasDeviceChanges(c)$
            $\land\ CreateDeviceChanges(c)$
            $\land\ \text{UNCHANGED}\ \langle networkChange \rangle$
        $\lor\ \land\ HasDeviceChanges(c)$
                Reconcile a change
            $\land\ \lor\ \land\ networkChange[c].phase = Change$
                        If a device change failed rollback any applied changes
                    $\land\ \lor\ \land\ HasFailedDeviceChanges(c)$
                            $\land\ RollbackDeviceChanges(c)$
                            $\land\ \text{UNCHANGED}\ \langle networkChange \rangle$
                    If device changes have been rolled back, increment the attempt
                    ∨ ∧ DeviceChangesRolledBack(c)
                      ∧ networkChange' = [networkChange EXCEPT
                            ![c].attempt = networkChange[c].attempt + 1]
                    If device changes are complete, complete the network change
                    ∨ ∧ DeviceChangesComplete(c)
                      ∧ networkChange' = [networkChange EXCEPT
                            ![c].state = Complete]
                      ∧ UNCHANGED ⟨deviceChange⟩
                Reconcile a rollback
                ∨ ∧ networkChange[c].phase = Rollback
                  ∧ networkChange' = [networkChange EXCEPT
                        ![c].state    = Complete]
    ∧ UNCHANGED ⟨nodeVars, deviceVars, constraintVars⟩
```

This section models the *DeviceChange* reconciler.

```
ReconcileDeviceChange(n, d, c) ≜
    ∧ master[d][n]
    ∧ deviceChange[d][c].state = Pending
    ∧ deviceChange[d][c].attempt > 0
    ∧ ∨ ∧ deviceState[d] = Connected
        ∧ deviceChange' = [deviceChange EXCEPT
              ![d] = [deviceChange[d] EXCEPT ![c].state = Complete]]
      ∨ ∧ deviceState[d] = Disconnected
        ∧ deviceChange' = [deviceChange EXCEPT
              ![d] = [deviceChange[d] EXCEPT ![c].state = Failed]]
    ∧ UNCHANGED ⟨nodeVars, networkChange, deviceVars, constraintVars⟩
```

This section models device states. Devices begin in the *Unavailable* state and can only be configured while in the *Available* state.

```
 Set device d state to Connected
ConnectDevice(d) ≜
    ∧ deviceState' = [deviceState EXCEPT ![d] = Connected]
    ∧ connectionCount' = connectionCount + 1
    ∧ UNCHANGED ⟨nodeVars, configVars, electionCount, configCount⟩

 Set device d state to Disconnected
DisconnectDevice(d) ≜
    ∧ deviceState' = [deviceState EXCEPT ![d] = Disconnected]
    ∧ connectionCount' = connectionCount + 1
    ∧ UNCHANGED ⟨nodeVars, configVars, electionCount, configCount⟩
```

Init and next state predicates

```
$Init \triangleq$
    $\land\ leader\ = [n \in Node \mapsto \text{FALSE}]$
    $\land\ master = [n \in Node \mapsto [d \in Device \mapsto \text{FALSE}]]$
    $\land\ networkChange = \langle\rangle$
    $\land\ deviceChange = [d \in Device \mapsto [x \in \{\} \mapsto [phase \mapsto Change,\ state \mapsto Pending]]]$
    $\land\ deviceState = [d \in Device \mapsto Disconnected]$
    $\land\ electionCount = 0$
    $\land\ configCount = 0$
    $\land\ connectionCount = 0$

$Next \triangleq$
    $\lor\ \exists\, d \in \text{SUBSET}\ Device :$
        $SubmitChange([x \in d \mapsto 1])$
    $\lor\ \exists\, c \in \text{DOMAIN}\ networkChange :$
        $RollbackChange(c)$
    $\lor\ \exists\, n \in Node :$
        $\exists\, l \in Node :$
          $SetNodeLeader(n,\ l)$
    $\lor\ \exists\, n \in Node :$
        $\exists\, d \in Device :$
          $\exists\, l \in Node :$
            $SetDeviceMaster(n,\ d,\ l)$
    $\lor\ \exists\, n \in Node :$
        $\exists\, c \in \text{DOMAIN}\ networkChange :$
          $ReconcileNetworkChange(n,\ c)$
    $\lor\ \exists\, n \in Node :$
        $\exists\, d \in Device :$
          $\exists\, c \in \text{DOMAIN}\ deviceChange[d] :$
            $ReconcileNetworkChange(n,\ c)$
    $\lor\ \exists\, n \in Node :$
        $\exists\, d \in Device :$
          $\exists\, c \in \text{DOMAIN}\ deviceChange[d] :$
            $ReconcileDeviceChange(n,\ d,\ c)$
    $\lor\ \exists\, d \in Device :$
        $ConnectDevice(d)$
    $\lor\ \exists\, d \in Device :$
        $DisconnectDevice(d)$

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

\* Modification History
\* Last modified *Thu Dec* 12 17:37:01 *PST* 2019 by *jordanhalterman*
\* Created *Fri Sep* 27 13:14:24 *PDT* 2019 by *jordanhalterman*