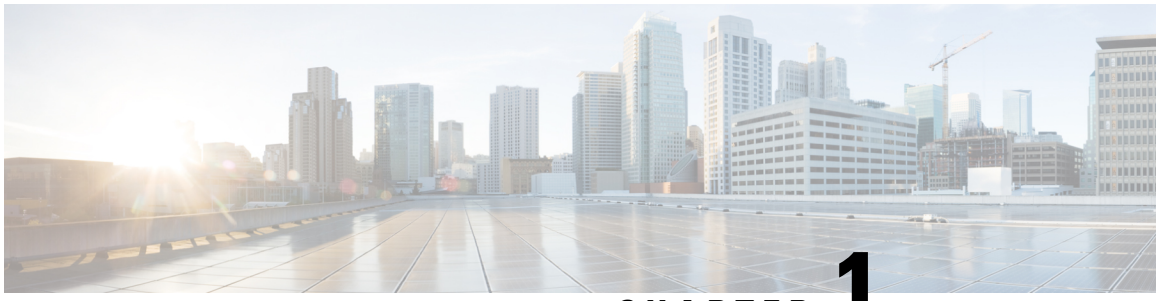
Cisco Crosswork Workflow Manager 1.0 Workflow Creator guide

First Published: 2023-06-01

Last Modified: 2023-07-26

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883



CHAPTER 1

Feature overview

This section contains the following topics:

Overview

Workflows help you automate business processes in a standardized manner to bridge the gap between expressing and modelling business logic.

Workflow definitions are written based on Serverless Workflow specification. For the Crosswork Workflow Manager version 1.0, only a subset of the specification is supported. This chapter describes all the supported features and gives practicable examples for each.

Wokflow definition features

A new workflow can be defined in either JSON or YAML formatting. The structure of the workflow definition is described in the specification.

The supported high-level components are as follows:

- id
- name
- description
- version

- start
- retries
- errors
- functions
- states
- metadata

Toplevel fields

Table 1: Toplevel fields

Parameter	Description
id	Unique identifier for the workflow.
name	Workflow name.
version	Workflow version based on Semantic Versioning.
specVersion	Version of Serverless Workflow specification release this definition adheres to. Current implementation is as per 0.9 specification.
description	Workflow description text.
start	State to be executed first.

Example in JSON:

```
{
"id": "MyWorkflow",
"version": "1.0.0",
"specVersion": "0.9",
"name": "My Workflow",
"description": "My Workflow Description",
"start": "SomeState",
"states": [],
"functions": [],
"retries":[]
}
```

Note If you prefer to use YAML instead of JSON, you can use a converter for the examples in this document.

Retry definitions

Retry definitions are policies that can be assigned to activities executing in a workflow to control how the workflow engine deals with faults and retries in the event of failure.

The following properties of retry definitions are supported:

Table 2: Retry definitions

Parameter	Definition
name	Definition name.
delay	Time delay between retry attempts in ISO 8601 format, for example "PT30S" for a 30 second delay.
maxAttempts	Maximum number of attempts. 0 for infinite. If you did not want any retries, maxAttempts should be set to 1.
maxDelay	Maximum amount of delay between retry attempts. Uses ISO 8601 format.
multiplier	Used to multiply delay value - if provided before each retry attempt. A float value. For example, if initial delay is 30 seconds, and multiplier is 1.5, the retries will increase by 50% each time.

Example:

```
"retries": [
     {
         "name": "Default",
         "delay": "PT1M",
         "maxAttempts": 5,
         "multiplier": 1.2
         "maxDelay": "PT3M"
     }
 ]
```

Error definitions

Error definitions describe errors that can occur during workflow execution. Whilst the serverless specification supports referencing an external file (JSON or YAML) that lists the errors, CWM will only handle errors defined in the Workflow definition.

The following properties of error definitions are supported:

Table 3: Error definitions

Parameter	Definition
name	Definition name.
code	Error code that could be returned - currently, this field is not used for error matching.
description	Should describe the error message. This description is used to match against error returned by activities.

Note Workflow Serverless specification doesn't have an option to specify error message which means that currently description is being used for matching against errors.

Example:

```
"errors": [
    {
        "name": "My Custom Error",
        "code": 0,
        "description": "Specific Error Message"
    }
    ]
```

Function definitions

Function definitions describe the function available for the workflow to execute and the name of the adapter and activity that should be invoked by the engine when that function is invoked. Whilst Serverless Workflow specification supports various types of functions, CWM will only support custom type functions that map to activities exposed via Adapters.

The following properties of function definitions are supported:

Table 4: Function definitions

Parameter	Definition
name	Name of function definition.
operation	Defines the adapter name and activity name that should be invoked by the engine. Format is <adapter name>.<activity name>, for example, the NSO Adapter has an activity called **RestconfGet** - an operation for this would be the name of activity as registered in worker, for example "RestconfGet". Please note this is case sensitive.
metadata	Allows modelling of information beyond the core definition of the Serverless Workflow specification. The "worker" key is used to define which Taskqueue the activities will be executed on. CWM 1.0 supports the concept of Workers that execute an Activity and are assigned Taskqueues that they listen to. To schedule an activity to run, the workflow engine places the activity on a Taskqueue. A worker process picks up the tasks to execute from Taskqueue and executes the activity.

Example:

```
"functions": [
        {
```

```
    "name": "NSO.RestconfGet",
    "operation": "restconf_Get"
    "metadata":  {
        "worker": "defaultWorker"
    }
},
{
    "name": "NSO.RestconfPut",
    "operation": "restconf_Put"
    "metadata":  {
        "worker": "defaultWorker"
    }
},
{
    "name": "NSO.RestconfPost",
    "operation": "restconf_Post"
    "metadata":  {
        "worker": "defaultWorker"
    }
},
{
    "name": "NSO.RestconfPatch",
    "operation": "restconf_Patch"
    "metadata":  {
        "worker": "defaultWorker"
    }
},
{
    "name": "NSO.RestconfDelete",
    "operation": "restconf_Delete"
    "metadata":  {
        "worker": "defaultWorker"
    }
},
{
    "name": "NSO.SyncFrom",
    "operation": "device_SyncFrom"
    "metadata":  {
        "worker": "defaultWorker"
    }
},
{
    "name": "REST.Post",
    "operation": "rest_Post"
    "metadata":  {
        "worker": "defaultWorker"
    }
} ]
```

States

States define the building blocks of workflow execution logic. Different types of states provide control flow logic to the Execution Engine and also allow you to define which activities to execute.

Common state properties

The following properties are common to all states:

Table 5: Common state properties

Parameter	Definition
name	State name.
type	Supported types are: "operation", "switch", "sleep", "inject", "foreach".
transition	Next transition of workflow - see below for further details. *Not applicable to SwitchState. For switch state, the transition option is defined on a per condition basis*.
end	If the workflow should end after this state - see below for further details. *Not applicable to SwitchState. For switch state, the end option is defined on a per condition basis.*
stateDataFilter	Filter data input and output for the state - not applicable to "sleep" state.
onErrors	Defines error handling for a given state, see below for further details. Can match based on Error Definition and control transition/end based on matched error including Compensation.
usedForCompensation	If `true`, this state is used to compensate another state. Default: `false`.
compensatedBy	Unique name of state which is responsible for compensation of this state. State identified here, is executed if "compensate" is set to true for transition/end property.

Note For any given state, you can only have one `transition` or `end` object. At least one must be present.

Compensation

Compensation provides you a way to define undoing the work done as part of a workflow. For each state, a compensation state can be defined. If, during execution, a condition is reached where compensation logic should be executed, a "compensate" flag can be set when defining a transition/end. The flag will result in executing states that are to be **usedForCompensation**. Refer to the Workflow Serverless specification for more information: Workflow compensation.

For CWM 1.0 implementation, each state marked for compensation is added to a queue. The compensation states are executed in terms of *Last in First Out*.

Transition

Serverless specification supports defining transition either as `string` or `object` with further properties. Crosswork Workflow Manager only supports the `object` format. Current CWM 1.0 implementation only supports the "nextState" property:

Table 6: Transition

Parameter	Definition
nextState	The name of state that workflow will transition to next.
compensate	If set to true, triggers workflow compensation before next transition is taken. Default: `false`.

End

Serverless specification supports defining end either as `string` or `object` with further properties. Crosswork Workflow Manager only supports the `object` format. Current CWM 1.0 implementation only supports the "nextState" property:

Table 7: End states

Parameter	Definition
terminate	Boolean value to define if this state should terminate the workflow.
compensate	If set to true, triggers workflow compensation before execution completes. Default: `false`.

stateDataFilter

State Data Filters allow you to define input and output data filters. Input Data filters allow you to select data that is required. Output Data filters are applied before transitioning to the next state, allowing you to filter data to be passed into the next state. More information on State Data Filters can be found here. Both the input and output filters are workflow expressions defined in jq. If no filters are specified, then all data is passed.

Table 8: stateDataFilter

Parameter	Definition
input	Input filter jq expression.
output	Output filter jq expression.

Example:

```
"states": [
        {
            "name": "step1",
            "type": "operation",
```

```
            "stateDataFilter" : {
                "input": "${ . }"
                "output": "${ . }"
            }
            "transition": {
                "nextState": "downloadImage"
            }
        },
        {
            "name": "step2",
            "type": "operation",
            "end": {
                "terminate": "true"
            }
        }
    ]
```

onErrors

onErrors property for a state defines errors that may occur during state execution and how they should be handled. More information on onErrors can be found here.

Table 9: onErrors

Parameter	Definition
errorRef or errorRefs	Define either a single errorDef or array of ErrorDefs to match for this state.
transition	Next transition of workflow if the error returned in state matches any of the error description in errorRef/errorRefs. Only `transition` or `end` can be defined.
end	The workflow should end if the error returned in state matches any of the error description in errorRef/errorRefs. Only `transition` or `end` can be defined.

Example:

```
"onErrors": [
        {
            "errorRef": "My Custom Error",
            "end" : {
                "terminate": true
                "compensate": true
            }
        }
    ]
```

Operation state overview

As per serverless workflow specification, operation states define sets of actions to be executed in sequence or parallel. Crosswork Workflow Manager only supports execution of actions in sequence.

An action can define invocation of 3 different types of services:

- Execution of function definition.
- Execution of another workflow definition as a child workflow (not supported in current implementation).
- Referencing events that may be "produced" or "consumed" (not supported in current implementation).

Note Only execution of function definition is supported in current implementation.

Action

Action definition specifies the function that should be executed for this state. The following properties are supported:

Parameter	Description
name	Action name.
functionRef	Object which defines the name of the function to be executed, and optionally arguments to pass into the activity the function points to. See below for further details.
retryRef	Name of retry definition defined globally. For example, `default`.
sleep	Object that optionally defines time to sleep either before or after action execution. See below for further details.
actionDataFilter	Filter to control what data should be passed to action, how to filter the results returned by action, and where to store the filtered results in the global state data. See below for further details.

functionRef

Parameter	Description
refName	Name of function referencing the function definition.

arguments	Arguments to be passed to the function - this can be a JSON object with complex structure. For Adapter activities, the structure has to be JSON as follows:
	`{`
	`"input": {`
	`...`
	`},`
	`"resource": {`
	`...`
	`}`
	`}`

actionDataFilter

Detailed information on actionDataFilter with examples can be found here.

Parameter	Description
fromStateData	Workflow expression in jq that filters data from state data to pass into function.
useResults	Boolean flag to control whether data returned from function execution should added/merged into state data output.
results	Workflow expression in jq that filters the data returned from function execution. Ignored if useResults is `false`. Default: `true`.
toStateData	Workflow expression defines state data where the results should be added/merged. If not specified, results merged at top level.

sleep

Parameter	Description
before	Amount of time to sleep before function is executed in ISO 8601 format e.g. "PT30S" - sleep for 30 seconds.
after	Amount of time to sleep after function is executed in ISO 8601 format e.g. "PT30S" - sleep for 30 seconds.

```
    {
        "id": "example",
        "version": "1.0",
        "specVersion": "0.9",
        "start": "step1",
        "functions": [
            {
                "name": "NSO.RestconfPost",
                "operation": "RestconfPost"
            }
        ],
        "retries": [
            {
                "name": "Default",
                "maxAttempts": 5,
                "delay": "PT30S",
                "multiplier": 1.1
            }
        ],
        "states": [
            {
                "name": "step1",
                "type": "operation",
                "sleep": {
                    "before": "PT1M"
                },
                "actions": [
                    {
                        "retryRef": "Default",
                        "name": "showVersion",
                        "functionRef": {
                            "refName": "NSO.RestconfPost",
                            "arguments": {
                                "input": {
                                 "path": "restconf/operations/devices/device=${ .deviceName
}/live-status/tailf-ned-cisco-ios-stats:exec/any",
                                    "data": "{\"input\": {\"args\": \"show version\"}}"
```

```
                                }
                            }
                        },
                        "actionDataFilter": {
                            "results": "${ if (.data) then .data |
fromjson.\"tailf-ned-cisco-ios-stats:output\".result else null end }",
                            "toStateData": "${ .showVersionPreCheck }"
                        }
                    }
                ],
                "end": {
                    "terminate": "true"
                }
            }
        ]
    }
```

Switch state overview

Switch states enable you to define decision points to route the workflow to a given path based on certain conditions. Workflow Serverless specification supports Data-based conditions and Event-based conditions. CWM only supports Data-based conditions.

dataConditions

The data condition property of Switch state is an array of conditions that are evaluated by the Execution engine. The Execution engine will select the first condition it matches and proceed along that path. If there are subsequent conditions that also match, they will be ignored.

Parameter	Description
name	Condition name.
condition	Workflow expression in jq that represents the condition. Must evaluate to `true`/`false`.
transition	Next transition of workflow if the condition matches.
end	The workflow should end if the condition matches.

Note You can provide only the `transition` object or the `end` object. At least one must be present.

defaultCondition

The default condition that is applied if none of the conditions match.

Parameter	Description
transition	Next transition of workflow if no conditions are matched.
end	The workflow should end if condition matches.

Note You can provide only the `transition` object or the `end` object. At least one must be present.

```
{
    "name": "ConditionName",
    "type": "switch",
    "dataConditions": [
        {
            "name": "IsTrue",
            "condition": "${ true  }",
            "transition": {
                "nextState": "TrueState"
            }
        },
        {
            "name": "IsFalse",
            "condition": "${ false }",
            "transition": {
                "nextState": "FalseState"
            }
        }
    ],
    "defaultCondition": {
        "end": {
            "terminate": true
        }
    }
}
```

Sleep state

Sleep state pauses workflow execution for a given duration.

Parameter	Description
duration	Duration the workflow should sleep for in ISO8601 format. For example, PT1M results in workflow sleeping for 1 minute.

```
{
    "name": "Sleep3Minutes",
    "type": "sleep",
    "duration": "PT3M",
    "transition": {
        "nextState": "NextState"
    }
}
```

Inject state

Inject state is used to inject static data into the State Data.

Parameter	Description
data	JSON object added to State Data.

```
{
    "id": "example",
    "version": "1.0",
    "specVersion": "0.9",
    "start": "HelloWorld",
    "states": [
        {
            "name": "HelloWorld",
            "type": "inject",
            "data": {
                "name": "Cisco",
                "message": "Hello World"
            },
            "stateDataFilter":{
                "output": "${ .message + \" from \" + .name + \"!\"  }"
            },
            "end": {
                "terminate": "true"
            }
        }
    ]
}
```

ForEach state

ForEach state allows you to define a set of actions to execute for each element in an array or list defined in State Data. For example, for Each device in device array, check the devices is in sync. Whilst the serverless workflow specification defines support for Parallel and Sequential execution of actions, current implementation only supports sequential execution of actions for each element in array.

Parameter	Description
inputCollection	Workflow expression in jq that points to an array in State Data.
iterationParam	Name of the parameter that can be referenced in action for each data element.
outputCollection	Workflow expression in jq that points to an array in State Data that the result will be appended to. If array doesn't exist, it will be created.

```
{
    "id": "example",
    "version": "1.0",
    "specVersion": "0.9",
    "start": "InjectData",
    "functions": [
        {
            "name": "HelloWorld",
            "operation": "HelloWorld"
        }
    ],
    "states": [
        {
            "name": "InjectData",
            "type": "inject",
            "data": {
                "people": [
                    {
                        "Firstname": "Peter",
```

```
                        "Surname": "Parker"
                    },
                    {
                        "Firstname": "Thor",
                        "Surname": "Odinson"
                    },
                    {
                        "Firstname": "Bruce",
                        "Surname": "Banner"
                    }
                ]
            },
            "transition":{
                "nextStat": "SayHelloToEveryone"
            }
        },
        {
            "name": "SayHelloToEveryone",
            "type": "foreach",
            "inputCollection": "${ .people }",
            "iterationParam": "person",
            "outputCollection": "${  .messages }",
            "actions": [
                {
                    "name": "SayHello",
                    "functionRef":{
                        "refName": "HelloWorld",
                        "arguments": {
                            "name": "${ .person.Firstname + \" \" + .person.Surname }"
                        }
                    }
                }
            ],
            "end": {
                "terminate": "true"
            }
        }
    ]
}
```

Parallel state

Parallel state allows you to define a collection of branches that are executed in parallel. Each branch in a state can define its own set of actions. Once the execution has completed, the parallel branches are joined into current path based on the **completionType** attribute.

The **completionType** attribute can define 2 values:

- allOf: All branches must complete execution before state can transition/end. This is the default value.
- atLeast: State can transition/end if the number of branches specified in atLeast has completed execution. If `completionType` attribute is "atLeast", numCompleted must also be set.

Parameter	Description
completionType	Define how to evaluate completion of state based on branch execution. "allOf" or "atLeast". Default: "allOf".

Parameter	Description
numCompleted	If `completionType` is "atLeast", this value must be specified. Defines the minimum number of branches that must be completed for the execution to proceed.

branches

List of branches that are to be executed in Parallel state. More information on branches can be found here.

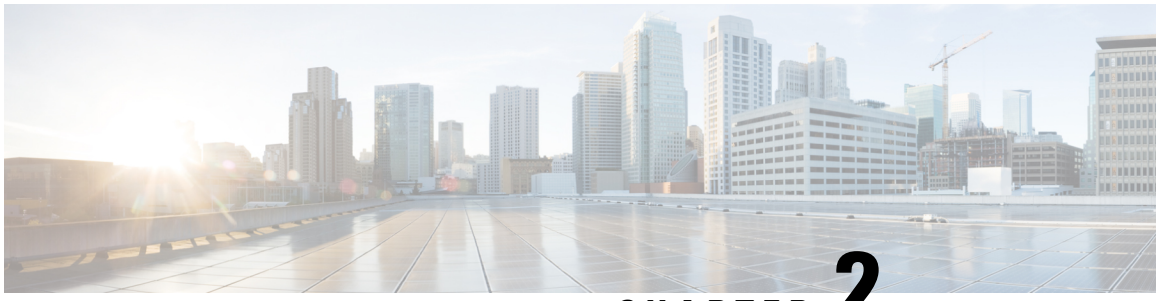
Parameter	Description
name	Name of branch.
actions	Actions to execute for this branch. A branch can support an array of actions. Definition for each action is the same as for Operation state type, see here.

State data

State data plays an important role during the lifecycle of the workflow. A state can filter data, inject data, and add data. Jq plays an important role in data filtering, creation and manipulation. For more information on how Data can be handled, see the Serverless Workflow specification.

When creating workflows, the following rules will apply when it comes to data management within CMW:

- Initial data passed into workflow execution is passed into State data as input.
- Data output from the last executed state is workflow output.
- If no State Input Filter is specified, all the data is passed into the state.
- If no State Output Filter is specified, all the data is passed into the next state.
- Workflow expressions in jq allow you to filter and manipulate data.
- Actions also allow for filtering data and also, if return data from action should be merged back into state data.
- Filters must return JSON object, if a jq workflow expression results in a string literal, this will result in an error.
- When working with jq, it is highly recommended to use https://jqplay.org/ to test the jq expressions. Alternatively, you can download jq locally and use it for testing.



CHAPTER 2

Create a workflow tutorial

This section contains the following topics:

Create a workflow – tutorial

This chapter shows you how you can structure a workflow based on a simple example that uses the `operation` and `switch` states to create a VPN service in Cisco NSO for some simulated devices. We go through the example workflow definition part by part to give you an idea how you can use different definition components in creating your original workflows.

If you need full information on how workflows can be defined, refer to the Serverless Workflow specification.

Example workflow overview

Workflows can be written either in JSON or YAML. For the example purpose, we'll pick the JSON formatting.

The purpose of the example workflow is to automatically create a VPN service for Cisco NSO devices.

First we point to the devices in the data input and then try to perform the NSO `check-sync` operation on them. Then, depending on the result:

- if not in sync, we push a device to perform a `sync-from`, and only then try to create a VPN for it;
- if in sync, we don't perform `sync-from` but directly create a VPN for the device.

If all the steps are executed successfully, CWM reports workflow execution completion and diplays the final data input. The results are visible in Cisco NSO too. If the engine encounters errors while performing a step, it uses the specified `retry` policy. In case errors persist beyond the retry limits, the execution engine ends the execution with a **Failed** status.

Go through the sections below to learn the details of how data input, functions, states, actions, and data filters are defined.

Provide data input

The workflow definition usually includes some input data at beginning of the JSON file. While the provided data is not part of the workflow, it is referred to within the workflow definition and can also be updated

between states, if such a data update is defined. For more details, see Workflow data input in the Serverless Workflow Specification.

In the quickstart example, we'll only need to provide two user-defined `deviceName` JSON object keys and values, which are the names of the test devices in the local NSO instance, and the `nsoResource` key, where we specify which CWM resource will be used in the workflow. The workflow data input in JSON should therefore look like this:

```
{
"device0Name": "ce0",
"device1Name": "ce1",
"nsoResource": "NSOLocal"
}
```

Define top-level parameters and functions

A workflow definition starts with the required workflow `id` key. Among other keys, `specVersion` is also required, defining the Serverless Workflow specification release version. The `start` key defines the name of the workflow starting state, but it is not required. In the `functions` key, you pass in Cisco NSO adapter activity name as function `name`, adapter activity ID as function `operation`, and provide the worker name under `metadata`: `worker` key:

```
{
    "id": "CreateL3VPN",
    "version": "1.0",
    "specVersion": "0.9",
    "name": "Create Layer3 VPN",
    "description": "Create an L3 VPN for MPLS devices",
    "start": "start",
    "functions": [
    {
        "name": "NSO.RestconfPost",
        "operation": "cisco.nso.v1.0.0.restconf.Post",
        "metadata": {
          "worker": "cisco.nso.v1.0.0"
        }
    }
  ],
}
```

Note Effectively, what you do under `functions` is you provide the workflow with the IDs of any activities as they are defined in the Cisco NSO adapter and presented in its `main.go` file. Also, under `metadata` you provide the name of the worker that will execute any actions that refer to the defined function.

Specify retry policy

With the `retries` key, you define the retry policies for state actions in the event that an action fails. Multiple retry policies can be specified under this key and they are reusable across multiple defined state actions.

```
"retries": [
    {
      "name": "Default",
      "maxAttempts": 4,
      "delay": "PT5S",
      "multiplier": 2
    },
```

```
    {
      "name": "Custom",
      "maxAttempts": 2,
      "delay": "PT30S",
      "multiplier": 1
    }
  ],
```

Note As you can see, the `Default` policy assumes that a failed action will be retried up to 4 times with an increasing delay between attempts: 5, 10, 20, 40 seconds between consecutive retries.

Define states

Workflow states are the building blocks of a workflow definition. In the present quickstart example, we will be using the `operation` and `switch` states, but others are possible. You can check them in detail in the Workflow states section of the Serverless specification.

operation state

```
"states": [
        {
        "name": "start",
        "type": "operation",
        "stateDataFilter": {
            "input": "${ . }"
        },
        "actions": [],
        "transition": {
            "nextState": "syncFromOrCreateVPN"
            }
        }
    ]
```

Inside the `operation` state, apart from state `name` and `type`, you define:

- `stateDataFilter` - point to the data input defined at beginning of example JSON file. In the `input` parameter, we state `${ . }`, which is a jq expression that means: use the whole of data input existing at this point of workflow execution.

Note For more information on how jq expressions are used in workflows, see the Workflow expressions chapter in the Serverless Workflow specification.

- `actions` - specify the function to be used by the action, and two basic `arguments`: `input` and `config`. Read more in the subsection below.
- `transition` or `end` - point to the next state to which the workflow should transition after executing the present one. If there are no more steps to be executed, use `end`.

switch state

```
  {
    "name": "syncFromOrCreateVPN",
```

```
        "type": "switch",
        "dataConditions": [
           {
             "name": "shouldSyncFrom",
            "condition": "${ if (.checkSyncResult0) then  .checkSyncResult0 != \"in-sync\"
 else null end }",
             "transition": {
               "nextState": "syncFrom"
             }
           },
           {
             "name": "shouldCreateVPN",
            "condition": "${ if (.checkSyncResult0) then  .checkSyncResult0 == \"in-sync\"
 else null end }",
            "transition": {
               "nextState": "createVPN"
             }
          }
         ]
      }
```

Inside the `switch` state, apart from state `name` and `type`, you define:

- `dataConditions` - define the conditions to be met by a device to be transitioned to a specified next state. You can view the `switch` state as a "gateway" for the workflow which directs the devices to appropriate states based on their status. Using the jq expression `${ if (.checkSyncResult0) then .checkSyncResult0 == \"in-sync\" else null end }` in the `condition` parameter, we create a boolean value that, if it evaluates to `true`, is used to transition the device directly to the `CreateVPN` state.

Specify actions

Let's analyse `actions` on the basis of the `checkSync` action of the `operation` state for device `ce0`.

```
{
"name": "checkSync",
"retryRef": "Default",
"functionRef": {
  "refName": "NSO.RestconfPost",
  "arguments": {
    "input": {
      "path": "restconf/operations/tailf-ncs:devices/device=${ .device0Name }/check-sync"
      },
    "config": {
      "resourceId": "${ .nsoResource }"
      }
    }
  },
"actionDataFilter": {
  "results": "${ if (.data) then .data | fromjson.\"tailf-ncs:output\".result else null end
 }",
  "toStateData": "${ .checkSyncResult0 }"
  }
}
```

Among the possible parameters, two are especially useful to consider:

- `functionRef` - refer to the function (aka an activity, from the NSO adapter perspective) to be used in action execution. Here, you need to pass in some `arguments`:
  - `input`:
    - `path` - point to a path for the adapter to send the request to.

- `data` - forward any data to be included in the request (not applicable for the `checkSync` action).

- `config`:
  - `resourceId` - provide ID of the resource you created for an external service. In the example workflow, the local host and the default port of the Cisco NSO instance is provided. The resource also points to the secret ID, which is used to provide authentication data for an external service: in this case, `username` and `password` to the Cisco NSO instance.

- `actionDataFilter` - define how to process the data passed on in the `checkSync` response from NSO:
  - `results` - use the jq expression `"${ if (.data) then .data | fromjson.\"tailf-ncs:output\".result else null end }"` to handle incoming NSO data. By using `fromjson` you translate the resultant `tailf-ncs:output` into JSON formatting, then with `.result` you cherrypick the `result` key value. In this case (if the device is in the in-sync state), the output of the expression would be `"in-sync"`.
  - `toStateData` - take the output of the expression defined in the `results` parameter above and save it as a key and value pair inside the workflow input data under any name that you pick: in this case, `.checkSyncResult0`.

Example workflow definition

The following example workflow definition is the end result of the workflow creation process presented in this chapter:

```
{
  "id": "CreateL3VPN-1.0",
  "name": "CreateL3VPN",
  "start": "start",
  "states": [
    {
      "name": "start",
      "type": "operation",
      "actions": [
        {
          "name": "checkSync",
          "retryRef": "Default",
          "functionRef": {
            "refName": "NSO.RestconfPost",
            "arguments": {
              "input": {
                "path": "restconf/operations/tailf-ncs:devices/device=${ .device0Name
}/check-sync"
              },
              "config": {
                "resourceId": "${ .nsoResource }"
              }
            }
          },
          "actionDataFilter": {
            "results": "${ if (.data) then .data | fromjson.\"tailf-ncs:output\".result
else null end }",
            "toStateData": "${ .checkSyncResult0 }"
          }
        },
        {
```

```
          "name": "checkSync",
          "retryRef": "Default",
          "functionRef": {
            "refName": "NSO.RestconfPost",
            "arguments": {
              "input": {
                "path": "restconf/operations/tailf-ncs:devices/device=${ .device1Name
}/check-sync"
              },
              "config": {
                "resourceId": "${ .nsoResource }"
              }
            }
          },
          "actionDataFilter": {
            "results": "${ if (.data) then .data | fromjson.\"tailf-ncs:output\".result
else null end }",
            "toStateData": "${ .checkSyncResult1 }"
          }
        }
      ],
      "transition": {
        "nextState": "syncFromOrCreateVPN"
      },
      "stateDataFilter": {
        "input": "${ . }"
      }
    },
    {
      "name": "syncFromOrCreateVPN",
      "type": "switch",
      "dataConditions": [
        {
          "name": "shouldSyncFrom",
          "condition": "${ if (.checkSyncResult0) then  .checkSyncResult0 != \"in-sync\"
else null end }",
          "transition": {
            "nextState": "syncFrom"
          }
        },
        {
          "name": "shouldCreateVPN",
          "condition": "${ if (.checkSyncResult0) then  .checkSyncResult0 == \"in-sync\"
else null end }",
          "transition": {
            "nextState": "createVPN"
          }
        },
        {
          "name": "shouldSyncFrom",
          "condition": "${ if (.checkSyncResult1) then  .checkSyncResult1 != \"in-sync\"
else null end }",
          "transition": {
            "nextState": "syncFrom"
          }
        },
        {
          "name": "shouldCreateVPN",
          "condition": "${ if (.checkSyncResult1) then  .checkSyncResult1 == \"in-sync\"
else null end }",
          "transition": {
            "nextState": "createVPN"
          }
        }
```

```
      ],
      "defaultCondition": {
        "end": {
          "terminate": true
        }
      }
    },
    {
      "name": "syncFrom",
      "type": "operation",
      "actions": [
        {
          "name": "syncFrom",
          "retryRef": "Default",
          "functionRef": {
            "refName": "NSO.RestconfPost",
            "arguments": {
              "input": {
                "path": "restconf/operations/tailf-ncs:devices/device=${ .device0Name
}/sync-from"
              },
              "config": {
                "resourceId": "${ .nsoResource }"
              }
            }
          },
          "actionDataFilter": {
            "results": "${ if (.data) then .data | fromjson.\"tailf-ncs:output\".result
else null end }",
            "toStateData": "${ .syncFromResult0 }"
          }
        },
        {
          "name": "syncFrom",
          "retryRef": "Default",
          "functionRef": {
            "refName": "NSO.RestconfPost",
            "arguments": {
              "input": {
                "path": "restconf/operations/tailf-ncs:devices/device=${ .device1Name
}/sync-from"
              },
              "config": {
                "resourceId": "${ .nsoResource }"
              }
            }
          },
          "actionDataFilter": {
            "results": "${ if (.data) then .data | fromjson.\"tailf-ncs:output\".result
else null end }",
            "toStateData": "${ .syncFromResult1 }"
          }
        }
      ],
      "transition": {
        "nextState": "createVPN"
      }
    },
    {
      "end": {
        "terminate": true
      },
      "name": "createVPN",
      "type": "operation",
```

```
        "actions": [
          {
            "name": "createVPN",
            "retryRef": "Custom",
            "functionRef": {
              "refName": "NSO.RestconfPost",
              "arguments": {
                "input": {
                  "data":
"{\"vpn\":{\"l3vpn\":[{\"name\":\"test-network\",\"route-distinguisher\":25,\"endpoint\":[{\"id\":\"office1\",\"ce-device\":\"ce0\",\"ce-interface\":\"GigabitEthernet0/5\",\"ip-network\":\"10.8.9.0/24\",\"bandwidth\":4500},{\"id\":\"office2\",\"ce-device\":\"ce1\",\"ce-interface\":\"GigabitEthernet0/5\",\"ip-network\":\"192.168.9.0/23\",\"bandwidth\":4500}]}]}}
 ",
                  "path": "restconf/data/l3vpn:vpn"
                },
                "config": {
                  "resourceId": "${ .nsoResource }"
                }
              }
            },
            "actionDataFilter": {
              "results": "${ if (.status) then .status else null end }",
              "toStateData": "${ .createServiceResult }"
            }
          }
        ]
      }
    ],
    "retries": [
      {
        "name": "Default",
        "delay": "PT30S",
        "multiplier": 2,
        "maxAttempts": 4
      },
      {
        "name": "Custom",
        "delay": "PT10S",
        "multiplier": 1,
        "maxAttempts": 2
      }
    ],
    "version": "1.0",
    "functions": [
      {
        "name": "NSO.RestconfPost",
        "metadata": {
          "worker": "cisco.nso.v1.0.0"
        },
        "operation": "cisco.nso.v1.0.0.restconf.Post"
      }
    ],
    "description": "",
    "specVersion": "0.9"
}
```

For a complete procedure on how to execute the example workflow in CWM and Cisco NSO to get tangible results, see the Quick Start guide