NYAT: Converting CLI Test Vectors to NETCONF

tail-f a Cisco company

Table of Contents

NYAT: Converting CLI Test Vectors to NETCONF

1. Overview

Device management in one form or the other has been around as long as devices themselves. Historically, various paths have been taken to address the problem of configuring and tracking the device. NETCONF frequently and most usually takes role of a newcomer that replaces existing legacy APIs like CLI. How can we take advantage of existing management API test vectors and re-use them with new NETCONF interfaces.

2. Utilizing Legacy Tests

In this application note, we will describe a method for translating legacy network element CLI tests to NETCONF and use the translated tests to verify the NETCONF interface of the device. This methodology is very useful when developing test vectors for use with NETCONF & YANG Automation Testing (NYAT).

For our needs, we define a test vector as an exact and stable configuration state of a device, declared in any supported management API format.

Devices being developed are usually backed with a bulk of test vectors (test cases/scenarios) previously created to verify the functionality of both the device itself and of the management API.

Enabling a new management API typically requires effort for the backing test coverage as well. Already invested effort can ideally be leveraged by converting the legacy CLI tests into the new API format as well.

While implementing the conversion tool in the form of a "format conversion" (or data format grammar translation) is possible, doing so may require considerable effort and finetuning.

If the developed device provides both legacy CLI, and new NETCONF northbound management interfaces, the most straightforward approach would be to utilize the device implementation itself.

Multiple paths can be taken, depending on the architecture in place, tools available, etc.

3. Architecture of The Process

3.1 Overview

For our app-note needs, we will utilize Cisco Network Services Orchestrator (NSO) as the main management framework.

Cisco NSO is a data model (YANG) driven platform for automating network orchestration. It supports multi-vendor networks through a rich variety of Network Element Drivers (NEDs). NSO supports the process of validating, implementing, and abstracting your network configuration and network services, providing support for the entire transformation into Intent Based Networking (IBN).

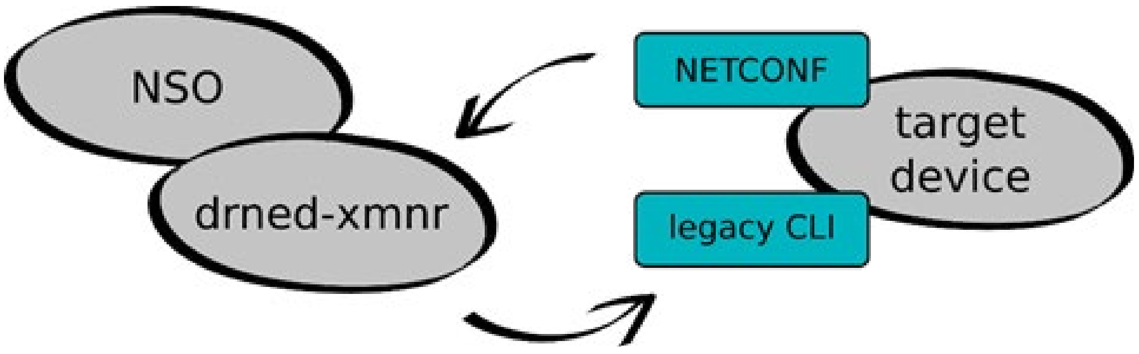
For more info on Cisco NSO, a brief solution overview is available here or a bit more technical overview is available on Cisco DevNet here.

In addition to Cisco NSO, we will augment our toolset with DrNED Examiner (drned-xmnr) as a testing related tool. drned-xmnr helps you with testing your NED (Network Element Driver) or device. Using the tool's actions, you can:

- Set up the environment (or at least get help with that)
- Save/import device configurations (or states) to be used later in tests
- Invoke DrNED tests that verify if the device transitions between those states smoothly
- Make DrNED generate reports on how well the state transitions cover the device model
- Log tools allow reviewing the communication between NSO and the device, debugging situations when the device configuration violates the device's YANG contract, and when testing transactionality properties

The core concept of our conversion can be easily summarized into a few key points:

- Target device has both legacy CLI and NETCONF API available
- Test vectors/configurations in the legacy device's CLI format are available
- We use drned-xmnr action "import-convert-cli-files", that internally:
    - Pushes the config into target device via its legacy CLI, utilizing the device driver (see further down)
    - •Synchronizes the NSO device instance from the target device using NETCONF, creating the configuration state in NSO/CDB format for further processing



This imposes some requirements on the target device to successfully follow our scenario - a device needs to have:

- A filesystem to store received files/data
- A way to apply stored files/data as a running configuration on the device
- A way to save the running config to the filesystem
- CLI and NETCONF interfaces enabled

3.2 DrNED Examiner import-convert-cli-files Action

The entire process, as executed by the DrNED Examiner action import-convert-cli-files, covers a few more points to make the user experience smoother:

- Save the original running config of the device in case something goes wrong
- Copy the input CLI config / test vector to the device filesystem
- Apply the copied data to the running configuration of the device
- Look for potential errors and report if something goes wrong
- Synchronize the device state to NSO using NETCONF <get_config> operation
- Save the configuration in XML (NETCONF) format and as a state in drned-xmnr
- Restore the original config on the device

The whole iteration can be automatically repeated for a batch of input test vectors, as defined by the action input parameters. Batch processing via the file-path-pattern param can lead to potential conflicts when several states already exist or the process is interrupted by some error, with only partial success of file set import.
More parameters are available to address the existing state conflicts. The overwrite parameter allows you to overwrite all the existing states in the batch if encountered. The skip-existing parameter can ignore conflicting filenames/states and keep them unchanged.
The process spans across several layers - NSO, drned-xmnr action callback, device driver, and target device. It helps to clearly identify where error is happening. The import action internally parses the errors triggered in the layers below the drned-xmnr callback and tries to enhance the error messages appropriately.

4. Getting It Done

4.1 Overview

Please, note that following guide is not intended as s 100% reproducible and effortless command list to be copy and pasted to achieve a working scenario. The entire pipeline includes several components and APIs that may need to be verified/prepared.

In case of missing puzzle pieces, please, refer to either of:

- Various NSO docs - starting/user/administrator/etc. guides
- NETCONF and YANG Automation Testing User Guide (NYAT)

The following sub-chapters describe conceptually important steps that should be covered but may need to be fine tuned or replaced by others depending on the end-user/test scenario.

All the sub-sections of this chapter, up until "Binding Device's Legacy Driver", are described in the NYAT User Guide and are best followed with the extensive details and explanations which are covered there. Here, we will just show excerpts and potentially shortened details.

Several example commands contain "variables" like ${USER}, etc., that are to be replaced with end-user specific values.

4.2 Defining Device in NSO

Before taking any NSO related steps, we can verify the target device's NETCONF API by using the tool netconf-console delivered with NSO to send a trivial NETCONF "ping" to the target device:

```
# Bash command:
$ netconf-console --user ${USER} --password ${PWD} --host
${IP} \
         --port ${PORT} --hello

<?xml version="1.0" encoding="UTF-8"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</
capability>
    <capability>urn:ietf:params:netconf:base:1.1</
capability>
    ...
```

Initial steps in the NSO CLI are the same as for many other use-cases.

Define an authentication group for a device to execute NETCONF operations on:

```
admin@ncs(config)# devices authgroups group nyat default-
map \
         remote-name admin remote-password admin
admin@ncs(config-group-nyat)# commit
Commit complete.
```

The authgroup setting is used by default for NETCONF access to a device, as well as CLI legacy API access in the driver, unless the driver retrieves the authentication information from another source (NSO database, local file, etc.).

Create a device with target IP/port and fetch the comm. keys:

```
admin@ncs(config-group-nyat)# top
admin@ncs(config)# devices device myrouter device-type
netconf \
         ned-id netconf
admin@ncs(config-device-myrouter)# address ${IP} port
${PORT} \
         authgroup nyat
admin@ncs(config-device-myrouter)# state admin-state
unlocked
admin@ncs(config-device-myrouter)# trace raw
admin@ncs(config-device-myrouter)# commit
Commit complete.
admin@ncs(config-device-myrouter)# ssh fetch-host-keys
result ...
fingerprint {
    algorithm ssh-rsa
    value ...
}
```

Enable some extra debugging/tracing items:

```
admin@ncs(config)# drned-xmnr log-detail cli all
admin@ncs(config)# python-vm logging level level-debug
admin@ncs(config)# commit
Commit complete.
```

4.3 Build Device NETCONF NED

Enable developer tools in the NSO CLI and fetch supported modules from device:

```
admin@ncs# devtools true
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# netconf-ned-builder project myrouter 1.0
\
         device myrouter local-user admin
Value for 'vendor' (<string>): test
admin@ncs(config-project-myrouter/1.0)# commit
Commit complete.
admin@ncs(config-project-myrouter/1.0)# fetch-module-list
admin@ncs(config-project-myrouter/1.0)#
```

We select the required modules:

```
admin@ncs# netconf-ned-builder project myrouter 1.0
module * *
// optionally deselect unneeded modules according to your
use-case...
```

And finally check the NED settings & start its build:

```
admin@ncs# show netconf-ned-builder project myrouter 1.0
module status
NAME              REVISION    STATUS
-----------------------------------------------------
ATM-MIB           1998-10-19  selected,downloaded
ATM-TC-MIB        1998-10-19  selected,downloaded
IANAifType-MIB    2006-03-31  selected,downloaded
IF-MIB            2000-06-14  selected,downloaded
SNMPv2-TC                     selected,downloaded
ietf-inet-types   2013-07-15  selected,downloaded
ietf-yang-smiv2   2012-06-22  selected,downloaded
...
admin@ncs# netconf-ned-builder project myrouter 1.0
build-ned
admin@ncs#
```

At this point, we should be ready to communicate with the target device in NSO. We assign the newly built NED to the NSO device and initialize drned-xmnr data for device:

```
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# devices device myrouter
```

```
admin@ncs(config-device-myrouter)# device-type netconf \
         ned-id myrouter-nc-1.0
admin@ncs(config-device-myrouter)# drned-xmnr setup setup-
xmnr \
         overwrite true
success XMNR set up for device myrouter
admin@ncs(config-device-myrouter)#
```

4.4 Build Device's Legacy Driver

Several of drned-xmnr's actions utilize the target device's legacy CLI to achieve its goals. Inter-connection of NSO APIs and legacy CLI is handled by a so-called "device driver". The driver consists of functions to copy files to and from the device and a simple state machine for the commands needed to apply CLI configuration to the target device. The most frequent and probable tool for this is a SSH connection.

The driver is an instance of the "Devcli" Python object with the prescribed interface defined by drned-xmnr. DrNED Examiner uses this object internally together with the pexpect tool to execute specific actions in the device's CLI. See the following pseudo-code:

```
class Devcfg(object):

    def init _ params(self, **args):
        pass    # parameters passed from the DrNED
Examiner via **args
                # can be stored in the object dictionary
and used to
                # connect to and control the device CLI
                # (params like device IP, CLI port, user/
password, ...)

    # some mandatory items to cover common
functionality...
    def get _ address(self):
        pass    # ip address of the target CLI device
    def get _ port(self):
        pass    # CLI port of the target device
    def get _ username(self):
        pass    # username for device CLI access
    def get _ password(self):
        pass    # password for device CLI access
    def get _ prompt(self):
        pass    # default CLI prompt format of the device

    # and primary driver implementation
    def get _ state _ machine(self):
        # state machine defining the expected CLI
prompts,
        # functions to invoke on state entry,
        # and sub-sequent next state transitions, in
format like:
        #
        #   state-id: [
        #       ("expected-cli-prompt",
        #        callback-function-to-run,
        #        "next-state-id"
        #       ),
        #       ...
        #   ],
        pass
```

For an example of how to implement and use the device driver, see the netsim_l.py (inside the drned-xmnr sub-directory ./test/lux/cfgs/device/) file, which is part of drned-xmnr's test suite. The pattern can be followed and implemented for theoretically any device with SSH access or other.

Having the device driver available, we can bind the driver file to the NSO device to handle legacy CLI operations:

```
admin@ncs(config-device-myrouter)# drned-xmnr driver
${PATH _ TO _ DRIVER}.py
admin@ncs(config-device-myrouter)# commit
Commit complete.
admin@ncs(config-device-myrouter)#
```

4.5 Execute Test Vector Conversion

At this point, all the pre-requisites for the conversions should be in place and we can proceed with attempts to convert input test vectors. Let us assume we have the test vector file "if-test-001.txt":

```
hostname ciscoios-1085-1
interface GigabitEthernet2
ip address 2.2.2.2 255.255.255.0
ip ospf network point-to-point
negotiation auto
exit
interface GigabitEthernet2.1
encapsulation dot1Q 11 second-dot1q 12
ip address 21.3.1.21 255.255.255.0
exit
```

This is the native CLI command format for a Cisco IOS XE router, and an example of how a small test vector may look like. We can apply the test vector to the target NSO device:

```
admin@ncs(config-device-myrouter)# drned-xmnr \
          state import-convert-cli-files \
          file-path-pattern ${PATH _ TO _ TEST _ VECTOR}/test-
vector-1.txt \
          overwrite true
importing state test-vector-1
success Imported states: test-vector-1
admin@ncs(config-device-myrouter)#
System message at 2021-09-10 12:51:40...
Commit performed by admin via ssh using cli.
admin@ncs(config-device-myrouter)#
```

4.6 Verify the Results

We can verify the new state has been created/imported:

```
admin@ncs(config-device-myrouter)# drned-xmnr state list-
states
success Saved device states: ['test-vector-1']
admin@ncs(config-device-myrouter)#
```

Further, we can display the imported configuration state (test vector):

```
admin@ncs(config-device-myrouter)# drned-xmnr state \
         view-state state-name test-vector-1
success <config xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>myrouter</name>
      <config>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-
XE-native">
          <version>17.5</version>
          <boot-start-marker/>
          <boot-end-marker/>
          <memory>
...
```

While the input CLI command only contains a few lines specific to IP configuration of the OSPF protocol, the retrieved configuration state is for the entire NETCONF API supported configuration state. This allows for stable and explicit state definition and further repeated usage of states and their testing.

5. Conclusion

We have shown import and conversion of a single test vector. The action also allows wildcards and can batch process many existing files at once.

We can copy the output for other needs with external tools. At the same time, a "state" created can be used by other drned-xmnr actions – enable on device, use with other created states to try transitions between stated, etc. This can be particularly useful for testing the programmability of the device.

6. For More Information

For more information about NETCONF & YANG Automation Testing (NYAT): https://info.tail-f.com/netconf_yang_automation_testing

NEPs can sign-up to receive free support for NYAT: https://info.tail-f.com/netconf-yang-automation-testing-program

For more information about NSO, visit: https://www.cisco.com/c/en/us/products/cloud-systems-management/network-services-orchestrator/index.html

tail-f a Cisco company

www.tail-f.com
info@tail-f.com

Corporate Headquarters

Sveavagen 25
111 34 Stockholm
Sweden
+46 8 21 37 40

©2021 Tail-f Systems All rights reserved.