Going Junos Native with NSO

Written 2019-11-22

by Jan Lindblad

NSO has had support for Junos devices pretty much since day one. The classical Junos NED is NETCONF based, but Juniper did not originally provide any YANG files, so the Tail-f team built a complex machinery to translate Juniper's XML Schema Description (XSD) files into YANGs for the Juniper-Junos NED.

Some time ago, Juniper started to provide native YANG files for Junos, and a few NSO users experimented with using them instead. Unfortunately, the Junos NETCONF hello message didn't match the Junos YANG files, so things did not work out.

In NSO 5.3, after a thorough discussion with Juniper engineering about how Junos devices behave, new device manager code in NSO works around the hello issue. Finally, the native Junos YANG files can be used directly in NSO.

This article describes how to make an existing NSO application work against a native Junos NETCONF NED. For application, I took the NSO mpls-vpn example from the NSO examples collection. Here are the high level steps:

1. Make the MPLS example run on a real Junos device with a classic NED
2. Build a Junos native NETCONF NED and configure NSO to use it
3. Update the service package to also work with the new NED
4. Reconcile service instances using new service

Much of this article may also be relevant if you're thinking about how to port a service package from using one NED to another.

Sounds interesting? Just follow along. Both hands-on and high level readers are welcome.

1. Make the MPLS example run on a real Junos device with a classic NED

In order to demonstrate how to port an existing application to use the native Junos NETCONF NED, I picked our favorite MPLS VPN example in Python from the NSO examples collection. You will find it under examples.ncs/getting-started/developing-with-ncs/17-mpls-vpn-python .

The first thing we need to do with the example is to switch out the simulated Junos device PE2 in the example for the real thing. Juniper Sweden has graciously enough provided me access to a vMX running Junos 18.4R1 and 19.1R1.6. To do the things described in this article, you need a Juniper device running Junos 18.4R1 or above and NSO 5.3 or above.

First, let's start up the example without any changes.

```
$ make all
$ make start
```

Then let's go into the NSO CLI, sync our devices and load up an initial service config for two MPLS VPNs, for our fictional customers IKEA and Spotify.

```
$ ncs_cli -Cu admin

admin connected from 127.0.0.1 using console on Host
admin@ncs# devices sync-from
```

All devices are netsim devices at this point. Let's turn off turn off autowizard and complete-on-space to make it possible to paste configs. If you leave the CLI, you need to do this again, if you intend to paste more configs.

```
admin@ncs# autowizard false
admin@ncs# complete-on-space false
```

Here is a service config with two MPLS VPNs we can use. There is nothing special with these service configurations, except the endpoints have been selected to pass through the PE node PE2, which is a Junos device.

```
config
vpn l3vpn ikea
 as-number 65101
 endpoint main-office
  ce-device    ce0
  ce-interface GigabitEthernet0/11
  ip-network   10.10.1.0/24
  bandwidth    12000000
 !
 endpoint branch-office1
  ce-device    ce1
  ce-interface GigabitEthernet0/11
  ip-network   10.7.7.0/24
  bandwidth    6000000
 !
 endpoint branch-office2
  ce-device    ce4
  ce-interface GigabitEthernet0/18
  ip-network   10.8.8.0/24
  bandwidth    300000
 !
 qos qos-policy GOLD
!
vpn l3vpn spotify
 as-number 65202
 endpoint main-office
  ce-device    ce2
  ce-interface GigabitEthernet0/8
  ip-network   10.0.1.0/24
  bandwidth    40000000
 !
 endpoint branch-office1
  ce-device    ce5
  ce-interface GigabitEthernet0/1
  ip-network   10.2.3.0/24
  bandwidth    10000000
 !
 endpoint branch-office2
  ce-device    ce3
  ce-interface GigabitEthernet0/4
  ip-network   10.4.5.0/24
  bandwidth    20000000
 !
 qos qos-policy GOLD
!
```

That the traffic will pass through PE2 can be verified by a simple command.

```
admin@ncs(config)# commit dry-run outformat native
```

Towards the end out this lengthy output, observe that some config changes are going to the PE2 device using the legacy (Cisco made) namespace http://xml.juniper.net/xnm/1.1/xnm .

```
device {
    name pe2
    data <rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
              message-id="1">
           <edit-config
             xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
             <target>
               <candidate/>
             </target>
             <test-option>test-then-set</test-option>
             <error-option>rollback-on-error</error-option>
             <with-inactive
               xmlns="http://tail-f.com/ns/netconf/inactive/1.0"/>
             <config>
               <configuration
                 xmlns="http://xml.juniper.net/xnm/1.1/xnm">
                 <interfaces>
                   <interface>
                     <name>xe-0/0/2</name>
                     <unit>
                       <name>102</name>
                       <description>Link to CE / ce5 -
                                    GigabitEthernet0/1</description>
                       <family>
                         <inet>
                           <address>
                             <name>192.168.1.22/30</name>
                           </address>
```

```
                            </inet>
                          </family>
                          <vlan-id>102</vlan-id>
                        </unit>
                      </interface>
                    </interfaces>
…
```

Looks good. Let’s commit this to our simulated network.

```
admin@ncs(config-l3vpn-spotify)# top
admin@ncs(config)# commit
```

Now that the service is in place and everything is working fine in our simulated network, let’s swap the PE2 device from netsim to the real thing, a Junos 18.4R1 or later device. First thing we need to do with the device is to ensure it has the proper NETCONF setup for what we want to do next: we want to configure the device using the legacy Junos NETCONF NED(!).

This may sound surprising, but is a very useful trick. This will give us the translation from the legacy YANG model made by Cisco to the native YANG on the device made by Juniper for free.

Let’s log in to the Junos device.

```
junos> configure
Entering configuration mode

[edit]
junos# edit system services netconf

[edit system services netconf]
junos# show
ssh;
```

This is what we want the NETCONF configuration to look like in order for the legacy YANG models to work. Specifically, “rfc-compliant” and “yang-compliant” modes *must not* (yet) be enabled for the legacy Junos NED to work.

Back in NSO, let’s change the PE2 config to point to the real device. Here are some reminders about all the commands you need to type.

```
devices authgroups group juniper default-map remote-name …

devices device pe2
authgroup juniper
address …
port 830
trace pretty
commit

ssh fetch-host-keys
connect
sync-from
```

The real Junos device is now under NSO management, but the services are obviously not yet in sync.

```
admin@ncs(config-device-pe2)# vpn l3vpn * check-sync
```

```
vpn l3vpn ikea check-sync
    in-sync false
vpn l3vpn spotify check-sync
    in-sync false
```

It should be easy to ensure our two service instances are in sync. Just redeploy them.

```
admin@ncs# vpn l3vpn * re-deploy
Aborted: RPC error towards pe2: invalid-value: for [edit class-of-
service interfaces]: mgd: 'per-unit-scheduler', 'hierarchical-
scheduler', or 'shared scheduler' for this interface is required for
bandwidth configuration on interface unit
```

Bummer. Some Googling reveals the service template for the Junos device we have in the L3VPN example service isn't perfectly adapted to the real world. While the template generates an acceptable configuration according to the YANG, the device implements further constraints not visible in the YANG. Apparently an additional leaf ("per-unit-scheduler") is needed for the configuration to be usable. Let's add it.

Inside the example, edit the packages/l3vpn/templates/l3vpn-pe.xml file to add the <per-unit-scheduler/> line, as shown below:

```
        <configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm"
                       tags="merge">
          <interfaces>
            <interface>
              <name>{$PE_INT_NAME}</name>
              <no-traps/>
              <vlan-tagging/>
              <per-unit-scheduler/> <!-- Added for real Junos -->
              <unit>
                <name>{$VLAN_ID}</name>
```

Once the file is edited, reload the packages to get the change into NSO.

```
admin@ncs# packages reload
```

Then let's check if we're in sync, re-deploy and then check sync again.

```
admin@ncs# vpn l3vpn * check-sync
vpn l3vpn ikea check-sync
    in-sync false
vpn l3vpn spotify check-sync
    in-sync false
admin@ncs# vpn l3vpn * re-deploy
System message at 2019-11-20 13:22:52...
Commit performed by admin via console using cli.
admin@ncs# vpn l3vpn * check-sync
vpn l3vpn ikea check-sync
    in-sync true
vpn l3vpn spotify check-sync
    in-sync true
```

That did it. The Junos machine is now managed by NSO, using the legacy Junos NED.

2. Build a Junos native NETCONF NED and configure NSO to use it

Now that the NSO application is working fine with the real Junos 18.4R1 or later device, it's time to start the actual port. The first step would be to build a NETCONF NED for the device. Starting with NSO 5.2, there is a netconf-ned-builder tool in NSO (no need for add-on packages like pioneer anymore). Let's use that.

For this to work, the Junos device needs to run in a mode that is as compatible with IETF NETCONF and YANG RFCs as possible. Log in to the Junos device again, and enable "rfc-compliant" and "yang-compliant" modes.

```
junos> configure
Entering configuration mode

[edit]
junos# edit system services netconf

[edit system services netconf]
junos# set rfc-compliant

[edit system services netconf]
junos# set yang-compliant

[edit system services netconf]
junos# show
ssh;
rfc-compliant;
yang-compliant;

[edit system services netconf]
junos# commit
commit complete
```

At this point, back in the NSO CLI, this change is not so great. We can no longer manage the device.

```
admin@ncs# config
admin@ncs(config)# devices device pe2 connect

result false
info Device pe2 does not advertise any known YANG modules
*** ALARM connection-failure: Device pe2 does not advertise any known
YANG modules
```

This is because the configured NED for the PE2 device is not compatible with "rfc-compliant" and "yang-compliant" Junos modes. The native Junos NED we are about to build will be, however, so this is a good move.

In order to build a NETCONF NED for the PE2 device, we need to change the ned-id that PE2 uses, from the legacy ned-id to "netconf". This is a generic id used with NETCONF devices that don't yet have a NED.

```
admin@ncs(config-device-pe2)# devices device pe2 device-type netconf
ned-id netconf
admin@ncs(config-device-pe2)# commit
```

```
Aborted: 'devices device pe2': Changing 'ned-id' not possible when
data exists. Run action 'migrate new-ned-id <ned-id>' to migrate the
data to the new NED identity.
```

I have to admit I did that on purpose just to show you. You can't simply change the ned-id of a device just like that. Since that would completely replace the YANG tree for the device with another, all sorts of things would get strange, like there would be no rollback from that operation. To prevent this situation from happening by mistake, changing the ned-id is only allowed when the device configuration is blank. We have to explicitly delete it – in NSO memory – in order for the ned-id change to be accepted.

```
admin@ncs(config-device-pe2)# revert
All configuration changes will be lost. Proceed? [yes, NO] yes

admin@ncs(config)# devices device pe2
admin@ncs(config-device-pe2)# no config
admin@ncs(config-device-pe2)# commit no-networking
Commit complete.
```

Note the "no-networking" flag to commit. We don't want to actually blank out the device configuration. Since we have changed the Junos NETCONF server mode to something that NSO can't use without a new NED, I suppose the deletion would not happen if we forgot the no-networking flag at this point. I'm not going to verify that calculation, however. I need my device, and I would not want to call Juniper Sweden and for a new one to be set up.

Once the device config is blank, the ned-id change succeeds:

```
admin@ncs(config)# devices device pe2
admin@ncs(config-device-pe2)# device-type netconf ned-id netconf
admin@ncs(config-device-pe2)# commit
admin@ncs(config)# exit
admin@ncs# exit
```

In order to get access to the netconf-ned-builder, we need to enable "devtools". If you leave the CLI, you need to do this again to access anything netconf-ned-builder related.

```
admin@ncs# devtools true
admin@ncs# config
admin@ncs(config)# netconf-ned-builder project juniper-junos-18.4R1
1.0 device pe2 vendor juniper local-user admin
admin@ncs(config-project-juniper-junos-18.4R1/1.0)# commit
admin@ncs(config-project-juniper-junos-18.4R1/1.0)# fetch-module-list
```

Here we created a new NETCONF NED project called "juniper-junos-18.4R1", in line with the NED naming recommendation (<vendor>-<family>-<version>). The version "1.0" refers to the NED version. Maybe we will generate new versions of the juniper-junos-18.4R1 NED, and any such versions may then be called "1.1" or "2.0" etc. Once the project has been committed, the first NED builder action is fetch-module-list, which makes NSO contact the device to discern which modules it may hold for download and build. The netconf-ned-builder uses several NETCONF methods to find out what modules are present.

By running an operational mode show command on the project, you can see a list of all the modules available.

```
do show netconf-ned-builder project juniper-junos-18.4R1 1.0
```

With most devices, it makes sense to not download and build a NED from all possible YANG modules. Many devices have, for example, both a native and an OpenConfig set of modules containing essentially the same data. It does not make much sense (and causes a lot of trouble) to include both of these views in the same NED.

In the Junos case at hand, there is an entry for a module called simply “junos”. In earlier attempts I have noticed it does not work well to download this module (and it’s not needed). I do want all the others, however, so this is how I wrote my module selection commands for the NED builder:

```
module junos-common-types 2019-01-01 select
module junos-conf* * select
module junos-rpc-* * select
```

To check on the download progress, exit to operational mode and run the following command every so often. On the Junos device, this takes only a minute or two. If it takes a long time and progress is not apparent, the download threads may have got stuck with an error situation they may incorrectly believe is fixable, so they keep retrying. The easiest way to cure this situation, should it occur, is currently to shut down and restart NSO.

```
# do show netconf-ned-builder project juniper-junos-18.4R1 1.0 module
status
NAME                                       REVISION    STATUS
----------------------------------------------------------------------
junos                                                  deselected,
                                                       download-error
junos-common-types                         2019-01-01  selected,downloaded
junos-conf-access                          2019-01-01  selected,downloaded
junos-conf-access-profile                  2019-01-01  selected,downloaded
…
```

Once the download is complete, start the build work with the “build-ned” command. If you are a bit too eager and try to start a NED build before the download is complete, you’ll just see an error message.

```
admin@ncs(config-project-juniper-junos-18.4R1/1.0)# build-ned
Error: Download is still in progress
```

The build of a NED as large as the Junos one will take a while. On my laptop it takes something like 30 minutes.

If you plan on making a series of similar NEDs, you can save your YANG module selection as a profile, and reuse this selection when working with other devices later.

```
save-selection profile juniper-junos-18.4R1-full
```

You will find this saved profile here:

```
show full-configuration netconf-ned-builder profile juniper-junos-
18.4R1-full
```

In this example, we chose to include as many YANG files as possible. A narrower selection could also have been chosen. If all you need from the device is covered by a smaller set of YANG modules, you can save a good deal of NSO YANG schema memory, instance config memory, and device sync speed by not including modules you don’t need.

Once the build is done, it's time to install it in your running NSO instance. The command to generate an installable package is called "export-ned". You could theoretically at least use this command to export the new NED right into the packages/ directory of the running NSO. In practice, this may not always work, however.

```
admin@ncs(config-project-juniper-junos-18.4R1/1.0)# export-ned to-
directory packages
Error: Missing permission for the file or one of its parents.
```

This has to do with the fact that I'm logged in to the NSO CLI as the "admin" user, which is not mapped to the host operating system, and therefore lacks filesystem write privileges in most locations.

That could certainly be fixed by properly configuring user information, but this article is not about aaa, so I will simply export the NED to /tmp, a directory with very liberal write permissions, then manually copy the package from there, as my ordinary operating system persona.

```
admin@ncs(config-project-juniper-junos-18.4R1/1.0)# export-ned to-
directory /tmp
tar-file /tmp/ncs-5.3.191112.84228-juniper-junos-18.4R1-nc-1.0.tar.gz
```

In a separate window, I then copy the file to the packages directory:

```
$ cp /tmp/ncs-5.3.191112.84228-juniper-junos-18.4R1-nc-1.0.tar.gz
packages/
```

Back in NSO, it's time to reload packages:

```
admin@ncs(config-project-juniper-junos-18.4R1/1.0)# top
admin@ncs(config)# exit
admin@ncs# packages reload
```

I tend to always want to check the package load status:

```
admin@ncs# show packages package oper-status

PACKAGE
                                  PROGRAM
                                  CODE     JAVA           PYTHON
NAME                         UP   ERROR    UNINITIALIZED  UNINITIALIZED
-----------------------------------------------------------------------
cisco-ios-cli-3.0            X    -        -              -
cisco-iosxr-cli-3.0          X    -        -              -
juniper-junos-18.4R1-nc-1.0  X    -        -              -
juniper-junos-nc-3.0         X    -        -              -
l3vpn                        X    -        -              -
```

Great, so the new Junos NETCONF NED is up and running! Let's switch our PE2 device to use the new ned-id, and see if we can connect to the device.

```
admin@ncs(config)# devices device pe2 device-type netconf ned-id
juniper-junos-18.4R1-nc-1.0
admin@ncs(config-device-pe2)# commit

admin@ncs(config-device-pe2)# connect
result true
```

Excellent. NSO will connect to the device automatically whenever needed, so “connect” is really quite unnecessary. I just wanted to see if we could connect or not. Let’s sync-from to see if things really work.

```
admin@ncs(config-device-pe2)# sync-from
```

That worked. Did we get any config?

```
admin@ncs(config-device-pe2)# show full config
devices device pe2
 config
  configuration version 20190319.203446_builder.r1013243
…
```

Yes. The NED seems to work.

3. Update the service package to also work with the new NED

Ok, so we have a Junos device running in “rfc-compliant” and “yang-compliant” mode. We have re-established contact with the device through a new native NETCONF NED. We have configured a couple of service instances, that we pushed with the traditional NED. Then we switched to the Junos native NED. So are we in sync then? Let’s check what NSO thinks about the state of affairs for the service instances.

```
admin@ncs(config)# vpn l3vpn * check-sync
vpn l3vpn ikea check-sync
    in-sync true
vpn l3vpn spotify check-sync
    in-sync true
```

Wow, cool, that was good news!? Or? A little unexpected, maybe. So we’re already in sync? Aren’t we supposed to port the application to use the new NED first? Let’s have a closer look; what does the ikea service instance look like on the device now?

```
admin@ncs(config)# vpn l3vpn ikea get-modifications
Error: No forward diff found for this service. Either
/services/global-settings/collect-forward-diff is false, or the
forward diff has become invalid. A re-deploy of the service will
correct the latter.
```

No forward diff at all? The service instance has never been created? What if we re-deployed the service, as suggested in the error message, but in dry-run mode?

```
admin@ncs# vpn l3vpn ikea re-deploy dry-run { outformat native }
native {
}
```

Hmm, empty change set. The pieces are slowly coming together. I see. That’s why NSO thinks the services are in sync. The NED change destroyed all device level references to the PE2 device, and because NSO thinks the services are generating an empty diff, absolutely any device configuration situation would be in sync. With no requirements, anything goes.

So why does NSO consider the output from a service re-deploy empty then? This comes down to how the service was defined. Like most modern NSO services, the MPLS VPN service uses a combination of (Python in this case) code and service templates. The service templates has template code for several different device types, or ned-id:s, if you will.

NSO will pick the right template content based on which namespace is declared for each snippet in the template file. Any snippet that has an XML namespace that matches a YANG namespace that is present in the NED for that device will be used. In this case, the template contains XML snippets for a couple of Cisco device types, and then the legacy Juniper NED with the single namespace http://xml.juniper.net/xnm/1.1/xnm . Nothing for the NED we just generated. Which is why the re-deploy came out empty.

To add XML template snippets for the new device namespace to the template(s) is the main piece of work for porting a service that uses templates to use a new YANG model.

So how do we come up with the additional XML template code for the new device type? Could be a lot of work, but with the trick we used here, it’s not too bad. Remember that we provisioned a service on the device using the legacy Junos NETCONF NED, then switched the

NED and did a sync-from. This means the service configuration is now present on the device and in the NSO database, and will be returned to us through the new YANG model if only we ask for it.

Let's have a look at the device interfaces, for example. I remember interface xe-0/0/2 was involved in delivering the service (have a look back at when we first provisioned the service, you'll see). If we display the device interface xe-0/0/2 configuration in the XML output format, we get the following:

```
admin@ncs# show running-config devices device pe2 config configuration
interfaces interface xe-0/0/2 | display xml
<config xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
  <device>
    <name>pe2</name>
      <config>
      <configuration xmlns="http://yang.juniper.net/junos/conf/root">
      <interfaces
        xmlns="http://yang.juniper.net/junos/conf/interfaces">
      <interface>
        <name>xe-0/0/2</name>
        <no-traps/>
        <per-unit-scheduler/>
        <vlan-tagging/>
        <unit>
          <name>101</name>
          <description>Link to CE / ce4 -
                       GigabitEthernet0/1</description>
          <vlan-id>101</vlan-id>
          <family>
            <inet>
              <address>
                <name>192.168.1.18/30</name>
              </address>
            </inet>
          </family>
        </unit>
        <unit>
          <name>102</name>
          <description>Link to CE / ce5 -
                       GigabitEthernet0/1</description>
          <vlan-id>102</vlan-id>
          <family>
            <inet>
              <address>
                <name>192.168.1.22/30</name>
              </address>
            </inet>
          </family>
        </unit>
      </interface>
      </interfaces>
…
```

Note the namespace attribute on the interfaces tag above,

```
<interfaces xmlns="http://yang.juniper.net/junos/conf/interfaces">
```

It's new. This is Junos' native YANG namespace for the interface configuration.

If we look in our packages/l3vpn/template, the l3vpn-pe.xml template refers to Junos functionality under the legacy namespace.

```
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm"
               tags="merge">
  <interfaces>
    <interface>
      <name>{$PE_INT_NAME}</name>
      <no-traps/>
      <vlan-tagging/>
      <per-unit-scheduler/> <!--Added for real Junos -->
```

As this namespace is different than what the device announces as supported, NSO understands this snippet isn't applicable for this device, so does not apply this template snippet. So what template is applied to the pe2 device when re-deploying/committing the service?

None. That's why the re-deploy comes out blank.

Time to get moving on producing those new XML template snippets. Open up packages/l3vpn/templates/l3vpn-pe.xml, and locate the XML tag with the old Junos namespace.

Enclosed by this tag (between <configuration> and </configuration>), you will find four top level tags: interfaces, routing-instances, policy-options and class-of-service. In the template there are plenty of references to variables, e.g. $PE_INT_NAME, in curly brackets { } that tell NSO that this is an expression needs to be evaluated.

```
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm"
               tags="merge">
  <interfaces>
    <interface>
      <name>{$PE_INT_NAME}</name>
      <no-traps/>
      <vlan-tagging/>
      <per-unit-scheduler/>
      <unit>
        <name>{$VLAN_ID}</name>
    …
  </interfaces>
  <routing-instances>
    …
  </routing-instances>
  <policy-options>
    …
  </policy-options>
  <class-of-service>
    …
  </class-of-service>
</configuration>
```

Our task is now to create a new template snippet that matches the new YANG, and uses the same variables. Basically, we need to match up the old template with the new config on the device. We need to look at the template and try to guess where the config would end up in the new model.

Since these the two YANG models we work with are very similar, that guessing is going to be pretty easy. Take /interfaces/interface in the old model, for instance. You will find it at (surprise!) /interfaces/interface in the new model as well.

When you found the corresponding location in the new model, show the running config for that part of the config and match against the template. The running config for the device interface xe-0/0/2 is displayed in XML format with the following command:

```
admin@ncs# show running-config devices device pe2 config configuration
interfaces interface xe-0/0/2 | display xml
```

Here they are, side by side, running config on the left, template on the right.

show running-config

```
<configuration xmlns="...">
 <interfaces xmlns="...">
  <interfaces>
   <interface>
    <name>xe-0/0/2</name>
    <no-traps/>
    <per-unit-scheduler/>
    <vlan-tagging/>
    <unit>
     <name>101</name>
```

original l3vpn-pe.xml

```
<configuration xmlns="...">
 <interfaces xmlns="...">
  <interfaces>
   <interface>
    <name>{$PE_INT_NAME}</name>
    <no-traps/>
    <per-unit-scheduler/>
    <vlan-tagging/>
    <unit>
     <name>{$VLAN_ID}</name>
```

The structure in the running-config (new YANG) and l3vpn-p3.xml (old YANG) is not identical, I had to swap the order of a few lines to get the above side-by-side view. I think it is fair to say that the previous XSD-based YANG model is pretty easy to match up with the new native YANG. Maybe there are areas where the differences are greater. Many other vendors have greater difference when porting between the various models for a particular device, and it will be more thought required when going from one vendor’s device to another. Regardless of whether it's easy or hard to match up with the new structure, this is the work that needs to be done.

Once things have been matched up like above, it's a small matter of writing down an additional template section in the same template file. Keep the old template definition intact if you want your service to work with devices running both kinds of YANG models. This is usually a good idea, since it may not be possible to convert all devices to use the new YANG interface at once. If you keep the old template for a while, it would also be easy to revert, in case you find some sort of trouble with the new device or its native YANG.

Before you proceed to make changes, take a backup copy of the original template:

```
$ cp l3vpn-pe.xml l3vpn-pe.xml.orig
```

The exact template lines to add for the snippet above would be:

```
        <configuration xmlns="http://yang.juniper.net/junos/conf/root"
                      tags="merge">
          <interfaces
            xmlns="http://yang.juniper.net/junos/conf/interfaces">
            <interface>
              <name>{$PE_INT_NAME}</name>
              <no-traps/>
              <per-unit-scheduler/>
              <vlan-tagging/>
              <unit>
                <name>{$VLAN_ID}</name>
```

Then add your new template lines to l3vpn-pe.xml. You can add it after the </configuration> end tag of the old Junos mapping. It needs to sit somewhere inside the <config> ... </config> tags in the template.

If you place this last in the template, the final lines of the updated l3vpn-pe.xml would read:

```
            ... tail of old junos template xml ...
          </class-of-service>
        </configuration>

        <configuration xmlns="http://yang.juniper.net/junos/conf/root"
                       tags="merge">
          <interfaces
            xmlns="http://yang.juniper.net/junos/conf/interfaces">
            ... your new junos template xml text ...
          </class-of-service>
        </configuration>

      </config>
    </device>
  </devices>
</config-template>
```

The full set of added template lines in l3vpn-pe.xml look like this:

```
        <configuration xmlns="http://yang.juniper.net/junos/conf/root"
                       tags="merge">
          <interfaces
            xmlns="http://yang.juniper.net/junos/conf/interfaces">
            <interface>
              <name>{$PE_INT_NAME}</name>
              <no-traps/>
              <per-unit-scheduler/>
              <vlan-tagging/>
              <unit>
                <name>{$VLAN_ID}</name>
                <description>Link to CE / {$CE} -
                             {$CE_INT_NAME}</description>
                <vlan-id>{$VLAN_ID}</vlan-id>
                <family>
                  <inet>
                    <address>
                      <name>{$LINK_PE_ADR}/{$LINK_PREFIX}</name>
                    </address>
                  </inet>
                </family>
              </unit>
            </interface>
          </interfaces>
          <routing-instances
         xmlns="http://yang.juniper.net/junos/conf/routing-instances">
            <instance>
              <name>{/name}</name>
```

```
    <instance-type>vrf</instance-type>
    <interface>
      <name>{$PE_INT_NAME}.{$VLAN_ID}</name>
    </interface>
    <route-distinguisher>
      <rd-type>{/as-number}:1</rd-type>
    </route-distinguisher>
    <vrf-import>{/name}-IMP</vrf-import>
    <vrf-export>{/name}-EXP</vrf-export>
    <vrf-table-label>
    </vrf-table-label>
    <protocols>
      <bgp>
        <group>
          <name>{/name}</name>
          <local-address>{$LINK_PE_ADR}</local-address>
          <peer-as>{/as-number}</peer-as>
          <local-as>
            <as-number>100</as-number>
          </local-as>
          <neighbor>
            <name>{$LINK_CE_ADR}</name>
          </neighbor>
        </group>
      </bgp>
    </protocols>
  </instance>
</routing-instances>
<policy-options
  xmlns="http://yang.juniper.net/junos/conf/policy-options">
  <policy-statement>
    <name>{/name}-EXP</name>
    <from>
      <protocol>bgp</protocol>
    </from>
    <then>
      <accept/>
    </then>
  </policy-statement>
  <policy-statement>
    <name>{/name}-IMP</name>
    <from>
      <protocol>bgp</protocol>
      <community>{/name}-comm-imp</community>
    </from>
    <then>
      <accept/>
    </then>
  </policy-statement>
  <community>
    <name>{/name}-comm-exp</name>
    <members>target:{/as-number}:1</members>
  </community>
  <community>
    <name>{/name}-comm-imp</name>
    <members>target:{/as-number}:1</members>
  </community>
</policy-options>
<class-of-service
xmlns="http://yang.juniper.net/junos/conf/class-of-service">
  <interfaces>
```

```
              <interface>
                <name>{$PE_INT_NAME}</name>
                <unit>
                  <name>{$VLAN_ID}</name>
                  <shaping-rate>
                    <rate>{$BW}</rate>
                  </shaping-rate>
                </unit>
              </interface>
            </interfaces>
          </class-of-service>
        </configuration>
```

Don't forget to add the xml attribute tags="merge" on the top tag (<configuration>) as shown above. Without it, you can be perfectly right in your mapping, but nothing would still come out.

You can use diff to check that there are no other changes to the template than the lines you added.

```
diff l3vpn-pe.xml l3vpn-pe.xml.orig
```

Back in NSO, you need to reload the package for the change to take effect. Also check that the package is up and running properly.

```
admin@ncs# packages reload
…
admin@ncs# show packages package oper-status

PACKAGE
                                 PROGRAM
                                 CODE     JAVA           PYTHON
NAME                          UP  ERROR    UNINITIALIZED  UNINITIALIZED
-------------------------------------------------------------------------
cisco-ios-cli-3.0             X   -        -              -
cisco-iosxr-cli-3.0           X   -        -              -
juniper-junos-18.4R1-nc-1.0   X   -        -              -
juniper-junos-nc-3.0          X   -        -              -
l3vpn                         X   -        -              -
```

With the l3vpn package up and running again, now with the new template in place, let’s have another go at check-sync:

```
admin@ncs# vpn l3vpn ikea check-sync
in-sync false
```

This is one of those times where check-sync for a service going from true to false is actually a step forward. Let’s see what a re-deploy dry-run can do with this would do.

```
admin@ncs# vpn l3vpn ikea re-deploy dry-run { outformat native }
native {
    device {
        name pe2
        data <rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
                  message-id="1">
                <edit-config
                  xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
                  <target>
                    <candidate/>
                  </target>
```

```
                        <test-option>test-then-set</test-option>
                        <config>
                          <configuration
                            xmlns="http://yang.juniper.net/junos/conf/root">
                            <class-of-service
                              xmlns="http://yang.juniper.net/junos/
                                     conf/class-of-service">
                              <interfaces>
                                <interface>
                                  <name>xe-0/0/2</name>
                                  <unit>
                                    <name>101</name>
```

Looks great, actually. Let’s do it for real.

```
admin@ncs# vpn l3vpn ikea re-deploy
admin@ncs#
System message at 2019-11-21 12:16:36...
Commit performed by admin via console using cli.
admin@ncs# vpn l3vpn ikea check-sync
in-sync true
```

We’re back in business! We are now able to deploy services on the Junos device using the new NETCONF interface.

4. Reconcile service instances using new service

So why is this article continuing? If the last sentence of the previous section is to be believed, we're already in business. We are already able to provision services, right? And we're in sync?

```
admin@ncs# vpn l3vpn ikea check-sync
in-sync true
```

Well, the last sentence of the previous section is true. It's just a bit deceiving. Look here:

```
admin@ncs# vpn l3vpn ikea get-modifications
cli {
    local-node {
        data  devices {
                   device ce0 {
                       config {
…
                   device pe2 {
                       config {
                           configuration {
                               class-of-service {
                                   interfaces {
                                       interface xe-0/0/2 {
                                           unit 101 {
                                               shaping-rate {
              -                                    rate 300k;
              +                                    rate 300000;
                                               }
```

What does actually say? Two important things, actually.

a) Apparently device canonical format for the rate is using suffixes. The device understands the "300000" that our template sends, but remembers that as "300k".

b) The rate was the only value set by the service, despite our efforts with the template. Nothing else was written by the ikea service instance.

If we are looking to avoid work, both of these insights are bad news.

Observation (a) is simply that the device expresses itself slightly differently than our service. Our service code is good enough so that the service gets deployed, but when checking later, the service will be out of sync because of this difference in dialect.

It may be possible to live with this difference, but for best results, we may want to adapt the service to use the same way to express these values as the device. In this case, that's pretty easy to do.

If we look at the YANG for the leaf shaping-rate/rate, we find that it's modeled as a string:

```
      container shaping-rate {
        description
          "Shaping rate";
        choice shaping_rate_choice {
          case case_1 {
            leaf rate {
              description
                "Shaping rate as an absolute rate";
              units "bits per second";
              type string;
```

In our service template, l3vpn-pe.xml, the XML looks like this. The actual rate value comes from a Python variable called BW.

```
        <configuration xmlns="http://yang.juniper.net/junos/conf/root"
                       tags="merge">
…
                  <shaping-rate>
                    <rate>{$BW}</rate>
                  </shaping-rate>
```

In the Python code, the variable binding like looks like this:

```
tv.add('BW', endpoint.bandwidth)
```

Since the BW variable is also used by other template snippets in the template file, it's not a good idea to change the BW value to use the Junos way with suffix. That would likely break the template for those other devices. Instead, let's add a new variable BW_SUFFIX that holds the BW value, but expressed with a suffix. Then have both variable bindings in the Python.

```
tv.add('BW_SUFFIX',
  ServiceCallbacks.int32_to_numeric_suffix_str(endpoint.bandwidth))
tv.add('BW', endpoint.bandwidth)
```

The int32_to_numeric_suffix_str() method translates any int32 value to a string with either no suffix, or a "k", "m" or "g" suffix.

```
    def int32_to_numeric_suffix_str(val):
        for suffix in ["", "k", "m", "g", ""]:
            suffix_val = int(val / 1000)
            if suffix_val * 1000 != val:
                return str(val) + suffix
            val = suffix_val
```

The template snippet for the native Junos YANG also needs to be updated to use the BW_SUFFIX value. Change the l3vpn-pe.xml, in the native Junos YANG, to the following:

```
        <configuration xmlns="http://yang.juniper.net/junos/conf/root"
tags="merge">
…
                  <shaping-rate>
                    <rate>{$BW_SUFFIX}</rate>
                  </shaping-rate>
```

After redeploying the package once again, and checking that it is up without error, we can see that the ikea service instance translates to a proper value for the device. The service instance is now in sync.

```
admin@ncs# vpn l3vpn ikea re-deploy
admin@ncs#
admin@ncs# vpn l3vpn ikea check-sync
in-sync true
```

As we have been deceived by in-sync true before, let's do a get-modifications to see what is actually being written to the device by this service instance.

```
admin@ncs# vpn l3vpn ikea get-modifications
```

While there is a lot of output, none of it is for the PE2 device. There is no diff for the rate leaf anymore, which means we fixed issue (a) above. That part is great, but an empty diff means (b) is still present more than ever. The updated rate leaf shows that the template is being applied, but nothing ‘sticks’.

It’s no mystery, really. This is due to the simple fact that the configuration the service wants to create for the service is already there, running on the device. That should come as no surprise, since we did push this service instance to the device using the legacy NED before we swapped NEDs.

NSO has no understanding (no memory) that the configuration now observed here is actually the same as NSO pushed earlier. Since the namespaces are all new, all the service meta data NSO had in the old device config is gone. The service meta data is NSO’s way of tracking which service created what on the device (or lower level services).

The good news is that “re-deploy reconcile” fixes this quickly. The re-deploy reconcile command comes in two variants. You can either discard-non-service-config or keep- it. Let’s try both here, in dry-run mode, to see what would happen.

```
admin@ncs# vpn l3vpn ikea re-deploy reconcile { discard-non-service-
config } dry-run
cli {
    local-node {
        data  devices {
                   device pe2 {
                       config {
                           configuration {
                               class-of-service {
                                   interfaces {
                                       interface xe-0/0/2 {
              -                            unit 102 {
              -                                shaping-rate {
              -                                    rate 10m;
              -                                }
              -                            }
                                       }
                                   }
                               }
                               interfaces {
                                   interface xe-0/0/2 {
              -                        unit 102 {
              -                            description "Link to CE /
ce5 - GigabitEthernet0/1";
              -                            vlan-id 102;
              -                            family {
…
```

That command would not add anything (we already knew that from the re-deploy that did nothing earlier), but it would remove a bunch of stuff. Sounds scary.

In a way, this should be scary. If we ran this command without the dry-run, the config lines marked with a minus would be removed, and something most likely would break. But why are they there and how did they get there in the first place?

The non-service-config referred to here are configurations created by other service instances, other systems or possibly by an operator. If it was created by other service instances, they would get back if we re-deployed all of them too.

If we try the keep-non-service-config alternative instead, it comes out much shorter. This is the whole thing.

```
admin@ncs# vpn l3vpn ikea re-deploy reconcile { keep-non-service-
config } dry-run
cli {
    local-node {
        data  devices {
                   device pe2 {
                       config {
                           configuration {
                               policy-options {
                                   policy-statement ikea-EXP {
                                       from {
                                       }
                                   }
                                   policy-statement ikea-IMP {
                                       from {
                                       }
                                   }
                                   community ikea-comm-exp {
                                   }
                                   community ikea-comm-imp {
                                   }
                               }
                               routing-instances {
                                   instance ikea {
                                   }
                               }
                           }
                       }
                   }
               }

    }
}
```

Still nothing added, as expected. And nothing deleted, just as might have been expected from a command with “keep-” in the name. But why are all these lines displayed then, even when nothing has been added or deleted? It’s not like it’s showing the entire configuration.

This is because it’s showing something that it’s hiding. The whole idea with the re-deploy reconcile command is to update NSO’s metadata about which NSO service instance owns which part of the (device) configuration. So the above shows all the objects that have been touched by the change, i.e. has updated service meta data. But the command isn’t printing the metadata itself.

So which should we go with, re-deploy reconcile with keep- or discard-non-service-config? Assuming there were no changes by other systems or operators, both approaches would end up the same if all service instances were re-deployed. Since each re-deploy would happen in a separate transaction, some actual device config deletions would happen before service is restored in the end, so that’s maybe not ideal. There is a possibility to “touch” a service to trigger a kind of re-deploy for many service instances in a single transaction, but touch doesn't have a reconcile flag.

Let's go with the keep-non-service-config. It's safe at the time of the re-deploy, in the sense that it won't delete things outside what the service touches. On the other hand, it risks leaving "garbage" configuration lingering even after the service instance is deleted.

```
admin@ncs# vpn l3vpn ikea re-deploy reconcile { keep-non-service-
config }
```

Having another look at the service modifications shows that now we have finally got the service metadata where we want it. The service content is now properly owned by the service instance (see the plus signs).

```
admin@ncs# vpn l3vpn ikea get-modifications
cli {
    local-node {
        data  devices {
                   device ce0 {
                       config {
                           ip {
…
                   device pe2 {
                       config {
                           configuration {
                               class-of-service {
                                   interfaces {
                                       interface xe-0/0/2 {
              +                            unit 101 {
              +                                shaping-rate {
              +                                    rate 300k;
              +                                }
              +                            }
                                       }
                                   }
                               }
                               interfaces {
                                   interface xe-0/0/2 {
              +                        no-traps;
…
```

Great. The ikea l3vpn service instance has been ported. Now all that remains is all the other service instances. Well, there's only one: spotify. In order to test the theory that the discard-non-service-config lines we were considering to remove came from the spotify service instance, let's try to reconcile this one with the discard-non-service-config flag. Since it's the last service instance, no other services should be clobbering any more.

The l3vpn services share interfaces, but not interface units.

```
admin@ncs# vpn l3vpn spotify re-deploy reconcile { discard-non-
service-config } dry-run
cli {
    local-node {
        data  devices {
                   device pe2 {
                       config {
                           configuration {
                               policy-options {
                                   policy-statement spotify-EXP {
                                       from {
                                       }
                                   }
                                   policy-statement spotify-IMP {
                                       from {
                                       }
                                   }
                                   community spotify-comm-exp {
                                   }
                                   community spotify-comm-imp {
                                   }
                               }
                               routing-instances {
                                   instance spotify {
                                   }
                               }
                           }
                       }
                   }
               }

    }
}
```

Yep. That would add nothing, remove nothing (only add service metadata, invisible here). Exactly as one would hope and expect.

```
admin@ncs# vpn l3vpn spotify re-deploy reconcile { discard-non-
service-config }
```

We can check-sync all l3vpn service instances.

```
admin@ncs# vpn l3vpn * check-sync
vpn l3vpn ikea check-sync
    in-sync true
vpn l3vpn spotify check-sync
    in-sync true
```

They look good. Admittedly, they have appeared good at several points earlier in this article, when the meaning of “true” has been a bit unintuitive. This time, however, it really means that the service is in sync in the classical meaning of that word.

Another test we can do is to re-deploy the services once more. Nothing would change.

```
admin@ncs# vpn l3vpn * re-deploy dry-run
vpn l3vpn ikea re-deploy
    cli {
    }
vpn l3vpn spotify re-deploy
    cli {
    }
```

We can also check the modifications again. Rather than copying all that text again, I will be lazy and simply count the lines of that output and cross check between service instances.

```
admin@ncs# vpn l3vpn ikea get-modifications | begin "device pe2"|count
Count: 84 lines
admin@ncs# vpn l3vpn spotify get-modifications | begin "device pe2" |
count
```

```
Count: 84 lines
```

Reasonable number of lines, and the same for both sounds reassuring. Finally, we can have a look at the actual service metadata that we worked so hard to get right.

```
admin@ncs# show running-config devices device pe2 config configuration
| display service-meta-data
devices device pe2
 config
  configuration version 20190319.203446_builder.r1013243
  configuration chassis fpc 0
   pic 0
    interface-type  xe
    number-of-ports 23
   !
  !
  ! Refcount: 2
  ! Backpointer: [ /l3vpn:vpn/l3vpn:l3vpn[l3vpn:name='ikea']
/l3vpn:vpn/l3vpn:l3vpn[l3vpn:name='spotify'] ]
  configuration class-of-service interfaces interface xe-0/0/2
   ! Refcount: 1
   ! Backpointer: [ /l3vpn:vpn/l3vpn:l3vpn[l3vpn:name='ikea'] ]
   unit 101
    ! Refcount: 1
    ! Refcount: 1 (/devices/device{pe2}/config/jc:configuration/jc-
class-of-service:class-of-service/interfaces/interface{xe-
0/0/2}/unit{101}/shaping-rate)
    ! Backpointer: [ /l3vpn:vpn/l3vpn:l3vpn[l3vpn:name='ikea'] ]
(/devices/device{pe2}/config/jc:configuration/jc-class-of-
service:class-of-service/interfaces/interface{xe-
0/0/2}/unit{101}/shaping-rate)
    shaping-rate rate 300k
   !
   ! Refcount: 1
   ! Backpointer: [ /l3vpn:vpn/l3vpn:l3vpn[l3vpn:name='spotify'] ]
   unit 102
…
```

It’s the refcount and backpointers that NSO uses to keep track of how many service instances and which ones that own a particular device configuration leaf.