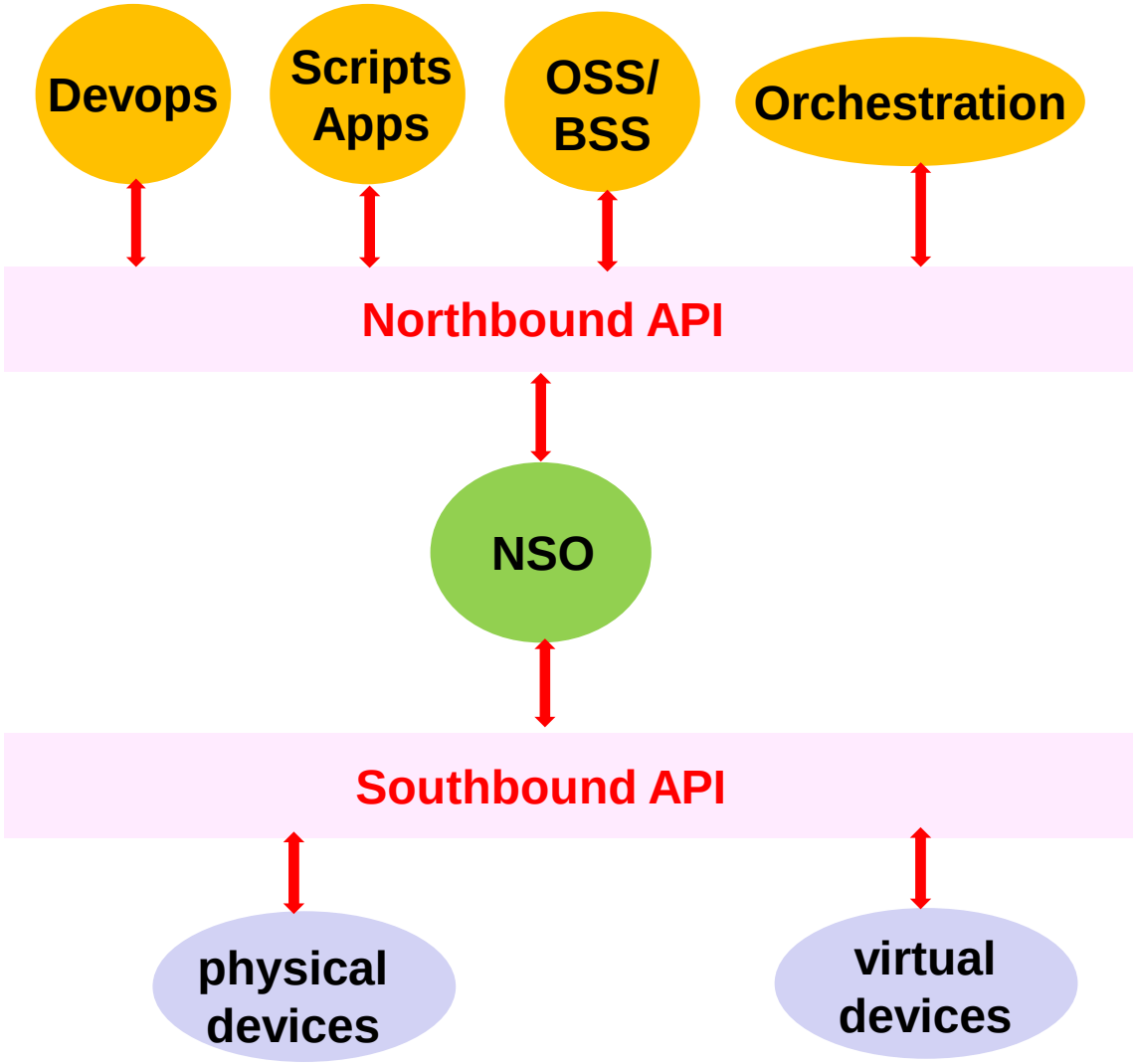
1 Cisco NSO

"Cisco **Network Services Orchestrator** (enabled by Tail-f)", since this is the changed name (due to the "Sell As A Name" service) of the product developed by a Swedish company acquired in 2014 by Cisco Systems. This is why you still get 'ncs' on their CLI. Automation was already on fire in those years, surfing the wave of SDN, VNF, and many other super fun acronyms. It is worldwide known and accepted that companies are slow in developing new services because (we)men (for the sake of inclusion) are lazy and prone to errors. By the way, I started doing 'automation' since I started working in the networking world in 2005. For the poor people working in the IT world, it's probably the only way to survive and maintain their mental brain sanity, for those who work as consultants it's even more necessary to do the job that 3.5 normal employees don't want to do.



Benefits of this product are … could you have guessed some of them ? They are all magical marketing keywords, belonging to that very small marketing dictionary that everybody is nowadays addicted to.

"

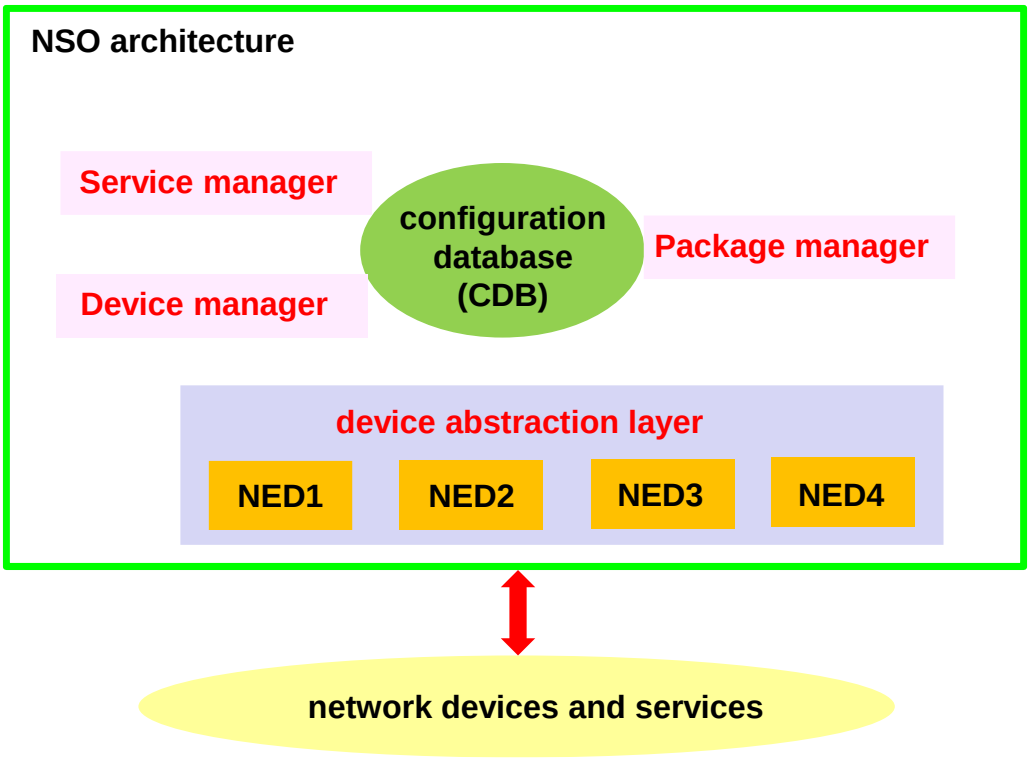
- **Accelerate revenue-generating services** with automated, self-service, on-demand provisioning that reduces activation times from months to minutes.
- **Increase agility and scale** with the capability to create, reconfigure, and repurpose services in real time with virtually unlimited horizontal scale.

- **Simplify your network operations** by automating the end-to-end service lifecycle and reducing manual configuration steps by up to 90 percent.
- **Promote open architectures** with multi-vendor capabilities to automate advanced device features, easily bundle multiple network services, and help assure them in real time.
- **Ensure trusted operations** with the industry's most robust feature set for ensuring trust across the network, other management systems and NSO clients, apps and users.

…
Cisco® Network Services Orchestrator (NSO) enabled by Tail-f® is in production in all of the top ten service providers and a number of large enterprises today. It provides end-to-end lifecycle service automation to design and deliver high-quality services faster and more easily. It lets you create and change services using standardized models **without the need for time-consuming custom coding** or service disruption. In addition, you can help ensure that your services are delivered in real time and meet even the most stringent service-level agreements (SLAs). The orchestrator **automates the full range of multivendor devices** across both physical and virtual environments and continually refines and repackages network services at the speed of your business."

The above has been taken from the following link:
https://www.cisco.com/c/en/us/solutions/service-provider/solutions-cloud-providers/network-services-orchestrator-solutions/at-a-glance.html#~true-agility-with-automation

Except for coffee, it seems to do everything you need, whoever you are, whatever your organization is. Except for giving you the next national lottery winning numbers (that's too much even for NSO).



NSO can interact through 'northbound API' with external systems (OSS, scripts, other orchestration systems). The most important pieces of this tool are:

- the 'service manager': services are represented through the YANG modeling language, and are probably the most important part of the job

- a local configuration database that is stored in MEMORY, even though there is also a persistent representation of it (of course there is one …)
- 'NED' is the abstraction layer through which the system interacts with the real or virtual devices

1.1 YANG

Looks like we're changing topic, but not really. This acronym is quite funny: though some of you (for instance Google) could know it as "'Young And Nasty Girl', the sad and less exciting truth is "Yet Another Next Generation (data modeling language)". Funny anyway: the real explanation, is not in the acronym but between parenthesis: YANG is a "**Data Modeling Language**". It has been 'standardized' by IETF with RFC 6020 and RFC 7950 (Yang version 1.1). You can define data structures, how they are organized, which variables types they are made of:

Type Name	Meaning
int8/16/32/64	Integer
uint8/16/32/64	Unsigned integer
decimal64	Non-integer
string	Unicode string
enumeration	Set of alternatives
boolean	True or false
bits	Boolean array
binary	Binary BLOB
leafref	Reference
identityref	Unique identity
empty	No value, void
	…and more

Common networking data types are defined in RFC6021 and can be imported, so that you don't need to reinvent the wheel:

```
import `ietf-yang-types` {
  prefix yang       ← reuse or extend the above imported data types with this prefix
}

leaf remote-ip {
  type yang:ipv4-address {
    pattern "10\\.0\\.0\\.[0-9]+";      ← add a check to the ip address syntax
  }
```

You can have:

- **group** statements (a list of other elements, containers, leafs, leaf-lists …)
- **container** statements (organizes leafs in structures)
- **list** statements (an array of elements)
- **leaf-list** statements (an array of elements with no children)
- **keys** (a dictionary, every key needs to be unique)
- **leaf references** (a link to another leaf)

Moreover there can be keywords for special purposes:

- unique (the value must be unique)
- range (specifies a range of possible values)
- error-message (error to be returned in case the variable doesn't match the range/pattern)
- pattern (regular expression to check for data content)
- starts-with (regular expression to check the beginning values of data)
- count (checks the number of occurrences of a specific variable)
- min-elements
- when (adds a conditional value)
- mandatory (specifies if the variable MUST be present or not)

A few examples about them hereafter:

```
EXAMPLE 1:
list l3mplsvpn-ce-config
{
  tailf:info "Used to Configure the CE Side of the MPLSL3VPN";
  key "vpn-name";
  unique vpn-id;            ← this value MUST be unique

  leaf vpn-id
  {
    tailf:info "Name of the VPN ID";
    type unint32            ← 32 bits unsigned integer
    {
      range 1..10;          ← an integer value identifies the vpn
      error-message "Invalid VPN ID"
    }
  }
}

EXAMPLE 2:
leaf ce-ip
{
  tailf:info "CE Interface IP Address";
! in this case there is a reference to the father leaf, it works like browsing
! through directories, '..' goes upwards one level, on that leaf there should
! be a leaf defined 'routing-protocol' whose value must be 'bgp'
  when "../routing-protocol='bgp'";
  mandatory false;
  type inet:ipv4-address
  {
    pattern "172\\.([1][6-9]|[2][0-9]|3[0-1])\\..*"
    {
      error-message
        "Invalid IP address. IP address should be in the 172.16.0.0/12 range.";
    }
  }
}

EXAMPLE 3: (look at how complex can expression be … )
augment "/ncs:services"
{
  list l3mplsvpn
  {
    tailf:info "Layer-3 MPLS VPN Service";
    key vpn-name;
    unique interface;

    leaf vpn-name
```

```
    {..}

    leaf device
    {..}

    leaf interface
    {
      tailf:info "Customer Facing Interface";
      type string;

# What the below expression expands to :
# "In NO other vpn configuration (vpn-name !=) the current device AND its
# current interface should be configured" which results in the expression to be
# zero.
    must "count(../../l3mplsvpn[vpn-name!=current()/../vpn-name]
[device=current()/../device][interface=current()]) = 0"

    # another way to do the same would be the following:
#// must "count(../../l3mplsvpn[vpn-name != current()/../vpn-name]" + "[device =
# current()/../device][interface=current()]) = 0"
# Notice the + in the above command, it is nothing but to ensure continuation of the entire
# path. It is not doing any arithmentic SUMMATION
      {
        error-message "Interface is already used for another link.";
      }
    }
  }
}
```

Without the aim of explaining 100% of what you can do with YANG, we wanted to pass the following message:

- YANG syntax is strict and many errors will be done, despite of tools/editors to help you writing them
- data representation can be complex as long as the object to be described is complex, and there is no way to automate this part of the job, except from reusing something standard written by someone else, and extending it. Managers, for your happiness, you will need a human being for this, or a high level and costly consultant
- parameters consistency checks can be VERY complex, especially when you need to reference different variables belonging to different leafs
- most checks are environment dependent, and are strictly related to the company where NSO needs to be deployed
- it's quite hard to write YANG data models checking that data is 100% consistent with your company's policies
- in case of syntax errors and mistakes, it's averagely speaking difficult to find out what is exactly wrong (it's not like debugging in Python)

Network topologies (like it happens on "Cisco WAE"), such as every possible information, could be represented using YANG. There's more than one way to do it, and they can be quite complex, long and difficult to be manually written and built. For example, you can find suggestions on other RFCs "A YANG Data Model for Routing Management" (RFC8022), which can save some time, but you will probably always need to change and adapt them for your specific purposes. The alternative, is that of **delegating an external script to perform all the complex checks** that maybe can't be done natively with YANG syntax and some of the complex stuff we've seen before.
As wrote before, in Cisco NSO YANG models are used to build SERVICES (for example 'create a l3 vpn'), we'll see how later on.

1.2 NSO netsim

NSO comes with a nice tool to emulate devices. They are not real virtual devices (like those used by VIRL, GNS, Boson) that need tons of memory, they are virtual devices belonging to a specific router's family. This topic gives us the chance of introducing **NED**s or Network Element Drivers: they are the abstraction layer that directly interacts with real devices and can support Netconf, CLI, snmp. Many 3rd party devices are currently supported, like Juniper, Checkpoint, Brocade, Amazon AWS. Some useful CLI commands to be used to create new devices:

```
ncs-netsim create-device cisco-ios mydevice
cd netsim
ncs-netsim start
ncs-netsim is-alive
```

- create netsim devices

```
cisco@NCS:~/nso-lab$ ncs-netsim create-network packages/cisco-ios 3 c
DEVICE c0 CREATED
DEVICE c1 CREATED
DEVICE c2 CREATED

cisco@NCS:~/nso-lab$ ncs-netsim add-to-network packages/dell-ftos 2 dell
DEVICE dell0 CREATED
DEVICE dell1 CREATED
cisco@NCS:~/nso-lab$
```

- start netsim devices

```
cisco@NCS:~/nso-lab$ ncs-netsim start
DEVICE c0 OK STARTED
DEVICE c1 OK STARTED
DEVICE c2 OK STARTED
DEVICE dell0 OK STARTED
DEVICE dell1 OK STARTED
cisco@NCS:~/nso-lab$
```

- login to a netsim device

```
cisco@NCS:~/nso-lab$ ncs-netsim cli-i c1
admin connected from 10.16.55.18 using ssh on NCS
c1>
c1> enable
c1#
c1# show running-config
no service pad
no ip domain-lookup
no ip http server
no ip http secure-server
ip routing
ip source-route
ip vrf my-forward
bgp next-hop Loopback 1
...
!
c1#
c1# exit
cisco@NCS:~/nso-lab$
```

The main reason to find these 'virtual devices' useful, is that you can retrieve configurations to later build the necessary templates to be used when configuring the services. If you have read my previous posts regarding Cisco ACI automation (part 1 and part 2) you know what I'm talking about.

```
admin@ncs> configure

admin@ncs% set devices device c0 config ios:snmp-server community BARFOO RW
admin@ncs% set devices device c0 config ios:snmp-server community FOOBAR RO

admin@ncs% commit dry-run outformat xml

result-xml {
       local-node {
              data <devices xmlns="http://tail-f.com/ns/ncs">
              <device>
               <name>c0</name>
                 <config>
                 <snmp-server xmlns="urn:ios">          ← entry config branch point
                   <community>
                     <name>BARFOO</name>
                     <RW/>
                   </community>
                   <community>
                     <name>FOOBAR</name>
                     <RO/>
                   </community>
                 </snmp-server>
                 </config>
              </device>
       </devices>
       }
}
```

The '**dry-run**' magic command is the one to tell to the system: "show me what you'll do, but don't do it". The highlighted part is the changed configurations in xml format: we do not use anymore cli-style commands. As we saw for ACI, there's not just json to accomplish such tasks, NSO uses xml which is more verbose and probably understandable due to the 'tags' explaining what the variable is used for. More complex configurations just follow the same principle. This is an **xml template example**, we'll see later on how building up a service is tied to templates, but the 'game' is very similar to what we've explained for ACI part 2:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0" servicepoint="snmpTemplate">
 <devices xmlns="http://tail-f.com/ns/ncs">
  <device>
   <name>{/device}</name>
    <config tags="merge">
    <!-- .....IOS..... -->          ← beginning of IOS configuration
    <snmp-server xmlns="urn:ios">
     <community>
      <name>{/comm-str}</name>      ← variable that will be passed to the template
      <{/comm-mode}/>               ← variable that will be passed to the template
     </community>
    </snmp-server>                  ← end of IOS configuration
    <!-- .....IOS-XR..... -->      ← beginning of IOS-XR configuration
    <snmp-server xmlns="http://tail-f.com/ned/cisco-ios-xr">
     <community>
      <name>{/comm-str}</name>      ← variable that will be passed to the template
      <{/comm-mode}/>               ← variable that will be passed to the template
     </community>
    </snmp-server>                  ← end of IOS-XR configuration
    </config>
```

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

This simple example starts showing the truth, **the concept of multivendor in real life**: there will be a moment in the whole design and provisioning process, when you will have to specify what you're dealing with. As you can see in the above template, you need to provide the template snippets for all the necessary devices: if you have also Juniper and Alcatel, you will need to have them both. Keep in mind that as showed above IOS is different from IOS-XR, which is different from NX-OS. A real 'multivendor' product (sold as such) should hide to the end user all the problems related to having a multivendor network: if I have to configure a bgp session, I should be able to configure all standard stuff without even knowing to which vendor or product-family the target belongs to. This would be a (too) big amount of work for the orchestration system, and is the reason for which **I still never saw a real multi-vendor product**. **Terraform** for the cloud area has exactly the same problem.
Even though the snmp community example doesn't seem to show real syntax differences between IOS and IOS-XR, a loopback address configuration shows better what I mean. Slight differences, but still differences.

```
<!-- IOS-XR -->
<interface xmlns="http://tail-f.com/ned/cisco-ios-xr">
  <Loopback>
    <id>{/loopback-intf}</id>
    <ipv4>
      <address>
        <ip>{/ip-address}</ip>
        <mask>255.255.255.255</mask>
      </address>
    </ipv4>
  </Loopback>
</interface>
<!-- IOS-XE -->
<interface xmlns="urn:ios">
  <Loopback>
    <name>{/loopback-intf}</name>
    <ip>
      <address>
        <primary>
          <address>{/ip-address}</address>
          <mask>255.255.255.255</mask>
        </primary>
      </address>
    </ip>
  </Loopback>
</interface>
```

Also beware that 'standard' CLI is no more an option with NSO, you will have to use xml templates. So if you have to **add a vlan to a trunk interface**, there is no way to do it using an xml template. You will have to retrieve the presently permitted vlans, add the new vlans to the list, and configure the whole new list of vlans. Quite tricky, and somewhat dangerous. Looks like other people have the same problem on the web:
https://stackoverflow.com/questions/68119910/allow-a-range-of-vlans-nso-cisco
… and the NED doesn't support the 'all' keyword too:
https://community.cisco.com/t5/nso-developer-hub-discussions/cisco-ios-ned-quot-switchport-mode-trunk-allowed-vlan-all-quot/td-p/3952561

At the following link:
https://github.com/NSO-developer/port-manager/blob/master/templates/port-manager-template.xml
… you can find something similar:

```
<!-- Trunk ports -->
 <GigabitEthernet-subinterface
  when="{/iosxr/GigabitEthernet and /port-type='trunk'}">
  <?foreach {/iosxr/GigabitEthernet}?>
    <GigabitEthernet>
      <id>{current()}.{/vlan-ids[1]}</id>
      <description>{/port-group-name}</description>
      <description when="{/description}">{/description}</description>
      <mode>l2transport</mode>
      <encapsulation>
        <dot1q>
          <vlan-id>{/vlan-ids}</vlan-id>
        </dot1q>
      </encapsulation>
      <rewrite>
        <ingress>
          <tag>
            <pop>1</pop>
            <mode>symmetric</mode>
          </tag>
        </ingress>
      </rewrite>
    </GigabitEthernet>
  <?end?>
 </GigabitEthernet-subinterface>
</interface>
```

… the problem seems to be that you can have a list of vlan ids, but you will anyway replace and overwrite the (possibly already) existing list of allowed vlans. Hopefully I'm wrong, unfortunately I don't have a test lab, nor had time to go deeper into some of these concepts.

End of the "How can such an easy configuration become an automation nightmare" chapter.

1.3 Service design

To design a new service, a CLI command will create the necessary directories and template files.

```
[root@nso LabDir45]# cd packages/
[root@nso packages]# ncs-make-package --service-skeleton python-and-template srvLoopEx

[root@nso packages]# cd srvLoopEx/
[root@nso srvLoopEx]# find .
.
./load-dir
./load-dir/srvLoopEx.fxs
./package-meta-data.xml
./python
./python/srvLoopEx                      ← service Python script directory
./python/srvLoopEx/__init__.py
./python/srvLoopEx/main.py
./README
./src
./src/java
./src/java/src
./src/Makefile
./src/yang
./src/yang/srvLoopEx.yang               ← service YANG representation
./templates
./templates/srvLoopEx-template.xml      ← configuration template, we've seen it already
(skip)
```

```
! template example created by NSO and slightly modified
module loopback {
  namespace "http://com/example/loopback";
  prefix loopback;

  import ietf-inet-types {
    prefix inet;
  }
  import tailf-ncs {
    prefix ncs;
  }
  import tailf-common {
    prefix tailf;
  }

  list loopback {
    key name;

    uses ncs:service-data;

    leaf device {
      mandatory true;
      type leafref {
               path "/ncs:devices/ncs:device/ncs:name";
      }
    }

    leaf loopback-intf {
      tailf:info "Loopback Interface Number";
      mandatory true;
      type uint16
    }

    leaf ip-address {
      tailf:info "Loopback /32 ip address";
      mandatory true;
      type inet:ipv4-address
    }

    leaf name {
      tailf:info "Service Instance Name";
            type string;
    }
  } # end of list loopback
}
```

The above simple example, represents the “link” between the following things:

- **services** are described through YANG and are made of leafs and leafs’ values. You can pass the required values using NSO CLI or **through NSO GUI**. NSO will show an html form in the GUI, one for each leaf, to be filled up to configure the service. When there is a ‘leaf reference’ to a network device, the GUI will show all the possible choices in a very useful drop-down menu, such as when there is a ‘leaf reference’ to the physical or logical interfaces belonging to the previously chosen device. What is (even) more complex and difficult to do (as we have previously explained), is filtering those choices based for example on the admin state of the interface, and/or being them without a description (to be sure they are not used)

- once the values are inserted (by a human being) in a simple html form, variables are passed to the xml configuration template we have seen before, where variables are referenced with "{/variable}" and are replaced with the present values (do you remember of know jinja2 templates ?)
- depending on the target device and its device type, the related xml configuration is pushed to it through the NED
- optionally, if existent a python or java script can be called (again, in the related directories we've previously shown) to perform many more complex things, checks and so on

This simple example should give you an idea of how services are built with Cisco NSO. Some other nice things that deserve to be mentioned:

- NSO keeps local configurations **synchronized** with those of the target. In case something is changed manually by a human being, you get an alarm on NSO dashboard. It's sufficient to '**sync-from**' devices to copy the devices' configurations locally on NSO, or 'sync-to' to copy NSO configurations toward the devices (this can be dangerous and disruptive when NSO has been installed for the first time and all local configuration are empty or not existent … be careful !!!). As usual, real life is always surprising for the possible outcomes: it seems that on IOS XR interfaces with no SFP appear in the 'preconfigure' mode, causing NSO to think he's out of sync (https://community.cisco.com/t5/nso-developer-hub-discussions/iosxr-interface-become-preconfigure-mode-cause-service-out-of/td-p/4460307)

- every configuration change can be checked before doing a real 'commit' and make them become effective: it's the 'dry-run' concept we have seen before. This is a concept that exists in netconf and also in IOS-XR, where you configure everything before doing a single unique 'commit' operation, rather than in IOS where every single command is immediately active (which can have sometimes unwanted side effects). This can be done through the GUI, the output being very intuitive (added lines are marked with '+', removed lines are marked with '-'). After all, xml configs are text files that can be compared to highlight differences.

- through Python scripts you can write pre-configuration checks, but also post-configuration checks. You can check for the output of some commands on target devices, do some pings, whatever you think it's useful and necessary to check that everything has gone has it had to (see paragraph 1.4).

1.3.1 Regarding YANG leaf references

At the following link you can find another github project published by Cisco employees:
https://github.com/NSO-developer/port-manager/blob/master/src/yang/port-manager.yang

A couple of things to notice:

- there are different YANG models depending on the device type (we report hereafter that for ios-xr, but you can find the others for ios and Juniper)
- look at how complex can leaf references and checks be … can you imagine how long you have to fight just because you forgot a "../", before you'll find the right syntax ? Those references work like normal surfing on a directory tree of folders and subfolders, since YANG data on its own can be seen as a tree.

To be honest, I still didn't understand 100% of how it works. More time spent on labs would be useful for sure.

```
container iosxr {
   when "derived-from(deref(../device)/../ncs:device-type" +
        "/ncs:cli/ncs:ned-id, 'iosxr-ned-id:cisco-iosxr-cli')" {
     tailf:dependency "../device";
   }

   choice xr-interface-type {
     mandatory true;

     leaf-list GigabitEthernet {
       tailf:info "Gigabit ethernet interface";
       type leafref {
         path "deref(../../device)/../ncs:config/cisco-ios-xr:interface" +
              "/cisco-ios-xr:GigabitEthernet/cisco-ios-xr:id";
       }
       // No more than 1 service can own each interface
            must "count(/ncs:services/port-manager" +
                 "[device=current()/../../device]" +
                 "/iosxr/GigabitEthernet[.=current()])<=1";
          }

          leaf-list TenGigE {
            tailf:info "TenGigE interface";
            type leafref {
              path "deref(../../device)/../ncs:config/cisco-ios-xr:interface" +
                   "/cisco-ios-xr:TenGigE/cisco-ios-xr:id";
            }
            must "count(/ncs:services/port-manager" +
                 "[device=current()/../../device]" +
                 "/iosxr/TenGigE[.=current()])<=1";
          }

        }
      }
```

1.4 Python service scripts

Follows later a basic python script that is created with the following command:

```
ncs-make-package --service-skeleton python-and-template srvLoopEx
```

Something similar is done with Java, for those who love that programming language. The script is automatically called by NSO once an operation is done, being it a 'dry-run' or a real 'commit'. This is reflected by the 'op' parameter that is passed by NSO and can be one of the following values:

```
OP =   1: 'CREATED',
       2: 'DELETED',
       3: 'MODIFIED',
       4: 'VALUE_SET',
       5: 'MOVED_AFTER',
       6: 'ATTR_SET'
```

In this way you know what kind of operation is being performed, and you can behave differently depending on that, writing the related (presently commented) functions. The 'lock' function can be used in case something needs to be done before configurations are 'virtually modified' (the dry-run concept we have previously mentioned), or really modified. Locking is necessary to avoid multiple transactions to be done at the same time, possibly making checks fail because someone overwrites another task's configurations, performed at the same time. This is that kind of problems you need to face in a multi-threaded operative

systems (with 'semaphore' mechanisms to avoid such problems). Variables filled in the html GUI, are of course also passed with their names and values to the python script.

```
import ncs
from ncs.application import Service

# ------------------------
# SERVICE CALLBACK EXAMPLE
# ------------------------
class ServiceCallbacks(Service):
    # The create() callback is invoked inside NCS FASTMAP and must always exist
    @Service.create
    def cb_create(self, tctx, root, service, proplist):
        self.log.info('Service create(service=', service._path, ')')
         # GUI variables are stored here
        vars = ncs.template.Variables()
        vars.add('this_host', '127.0.0.1')
        template = ncs.template.Template(service)
        # this is where the template xml file is used to perform configurations
        template.apply('srv-loop-template', vars)

    # The pre_modification() and post_modification() callbacks are optional,
    # and are invoked outside FASTMAP. pre_modification() is invoked before create,
    # update, or delete of the service, as indicated by the enum ncs_service_operation
    # 'op' parameter. Conversely post_modification() is invoked after create, update,
    # or delete of the service. These functions can be useful e.g. for allocations
    # that should be stored and existing also when the service instance is removed.
    #
    # @Service.pre_lock_create
    # def cb_pre_lock_create(self, tctx, root, service, proplist):
    #     self.log.info('Service plcreate(service=', service._path, ')')
    #
    # @Service.pre_modification
    # def cb_pre_modification(self, tctx, op, kp, root, proplist):
    #     self.log.info('Service premod(service=', kp, ')')
    #
    # @Service.post_modification
    # def cb_post_modification(self, tctx, op, kp, root, proplist):
    #     self.log.info('Service premod(service=', kp, ')')

# ---------------------------------------------
# COMPONENT THREAD THAT WILL BE STARTED BY NCS.
# ---------------------------------------------
class Main(ncs.application.Application):
       def setup(self):
       # The application class sets up logging for us. It is accessible
       # through 'self.log' and is a ncs.log.Log instance.
       self.log.info('Main RUNNING')
```

You can potentially use threading even in the Python script:

```
import threading
import time
import ncs
from ncs.application import Service

class BackgroundWorker (threading.Thread):
    def run(self):
        while True:
            print("Hello from background worker")
            time.sleep(1)

class Main(ncs.application.Application):
```

```
    def setup(self):
        self.log.info('Main RUNNING')
        self.BackgroundWorker = BackgroundWorker ()
        self.BackgroundWorker.start()

    def teardown(self):
        self.log.info('Main FINISHED')
        self.BackgroundWorker.stop()
```

On github under the main directory 'NSO-developer' you can find many different (and often complex) and instructive examples, just to mention a few:

https://github.com/NSO-developer/opa-example
https://github.com/gve-sw/NSO_to_EPNM_Optical_service

Things can get indefinitely complex as long as the task is. You can for example register your service with NSO to be wakened up or called in case specific events happen. More complex stuff is not so well documented and easy to be found in my opinion, but I would be glad to see that I'm wrong.

1.5 Input parameters consistency checks

All the complex consistency checks that should be made to avoid configuration errors or mistakes are difficult to be done through YANG keywords and commands. Of course you can check for the consistency of an ip address value (using 'pattern' regular expressions), but it's more difficult to check if a loopback interface has already been configured on another device, if a route-target or route-distinguisher has already been used somewhere else, if the chosen value for the vpn-id is really unique in the network, if the selected interfaces are really free or not. **Automation is always sold like 'less prone to errors'**, but this is only true if the above checks are made with a script in an automated way, which comes at a cost.
This is where things become even more tricky and complex. For example, as long as you perform checks related to configurations and network resources, you should somewhat freeze configurations to be sure that these checks are still valid when configurations are pushed to the target devices. Maybe an external database should be updated or searched for conflicts, using that custom service python script, assuming that this database is 100% updated and that nobody configures anything on a device, if all the resources have not yet been already taken and reserved on that database. Would you trust this approach ? if not, you should also check configurations locally on NSO. This should be quite fast since all configs are stored in memory (not just in a persistent database).

1.6 Release upgrade service

As you can see from the following posts inside Cisco community, with NSO you can even manage software upgrades:

https://community.cisco.com/t5/nso-developer-hub-videos/os-upgrade-using-nso/ba-p/3668033
https://community.cisco.com/t5/nso-developer-hub-videos/os-upgrade-version-2-using-nso/ba-p/4075202

As you can imagine, it's a very complex service that requires professional programmers to provide something good, it could be a product on its own. Of course once it's done, as an end user you don't need to know all the details: that's what happens with my car or my mobile phone. And of course, it's **not for free**. This is the list of features taken from the above links:

1. **Open architecture:** Generic upgrade platform to support user-configurable upgrade process for any device family.
2. **Batch upgrades**: Can upgrade multiple devices in parallel using individual upgrade threads per device.
3. **Multi path upgrade**: Ability to upgrade device(s) from any given source version to any given target version e.g from version A->B->C.
4. **Staging Images**: Can copy relevant images(s) to device(s) before upgrade maintenance window.
5. **Regional Image server selection**: Ability to select relevant images server (FTP, TFTP, SCP, SFTP etc) based on device region.
6. **Configuration Caveats Handling**: Can handle configuration issues which may arise during upgrade e.g config loss, pre/post upgrade config modifications etc.
7. **Pre-upgrade Validation-rules:** User can define a set of validation rules to perform health-checks on the device prior to upgrade.
8. **Plan of upgrade:** User can define the upgrade process with a sequence of commands to execute with success / failure criteria.
9. **Pre / Post Checks**: User can define the commands, which can be executed before & after the upgrade to identify any difference in device configs.
10. **Lazy Upgrade**: Runs the upgrade process on the device only when it is required. It skips the upgrade action if the device is already on target OS.
11. **Critical BUG Check**: validate if there is a critical bug(s) that is associated with the target image. BUG table is maintained within NSO CDB.
12. **Upgrade Report:** Generates side-by-side HTML color-coded report highlighting the difference between pre & post health-checks.

Point 10 is hard to be understood to be honest (or it's rather a genius marketing point), Point 4 more than a feature is a requirement or a prerequisite. Not to mention that pre-upgrade validation rules could be quite complex and customer dependent: for example in case of an ISP (Internet Service Provider) the points in the network where services are provided should be known to check for already present failures in the network and avoid service outages in case the upgrade is related to the unique node presently providing that service. In case of ring backbone topologies, you can't upgrade a node if there is already another failure or upgrade in the same ring. Traffic bandwidth on node's interfaces could also be compared before and after the activity, together with dynamic routing protocol neighbors, interfaces, cards, sent/received routes, stp status …
So on and so forth, if you want added value, there's a LOT of work to do, often putting together knowledge from different people (network, security, programmers). This **comes at a recurrent cost**, as all software.

1.7 Documentation

As often with Cisco, you can find a lot of stuff on the web inside their community, Devnet or even on **github**. Sometimes Cisco itself publishes little projects that help you understanding how it works, even **dCloud** is a valid resource. In any case, there is no real standard way to do anything, so you will need to find for someone who had your same doubts and/or problems, and has published something on the web. Not that easy. Sometimes there is really the sensation of feeling lost and hopeless, especially for more advanced stuff and things.

1.8 Conclusions

Now you can go back to the beginning and read again the first paragraph, with a more complete background about NSO.

The final question is … **is this REALLY automation ?** at a first glance, it looks like they have changed everything (YANG, netconf, python, xml …) so that it requires even more time to do easy configurations, rather than less time. Accessing the CLI of the target device to configure a loopback interface, is really different from inserting those values through a GUI, after a new specific service has been designed and developed (by another human beings) ? is this really faster, more secure and reliable than having a human being doing it ? do I really save so much time and money with one approach respect to the other ? can I fire every ops guy or half of them, assumed there are really too many (when the truth is, averagely speaking, just the opposite) ? Can I forget about having a multi vendor environment, if we have already shown that xml template MUST consider all device types and families anyway ?

The answer to all the above questions in my opinion **is NO**. Selling this as automation looks like telling Andersen's fairy tale: who's gonna see and yell that the king is naked ? Is it like the Windows-Linux battle, where Linux is not for everyone because you need to configure complex commands (which is anyway false since many years) ? Software companies like Google, Facebook or others for sure didn't wait for the wide spreading of the SDN magic word before actively doing REAL automation. NSO can be a first step, but you will need anyway a lot of line codes by professional programmers to get the job done, getting benefits from it. Supposed that **you are BIG ENOUGH** to manage so many concurrent configuration requests, that automation becomes a real advantage. And even in this case, for sure passing time filling up html forms for 100 different 'services' entries doesn't lower down the chance for copy/paste errors, nor makes things faster or easier in any way. An excel spreadsheet in this case would be a much better 'tool' than an html form based on a YANG modeled service.

Network automation is a MUST in the cloud or for (big) ISPs environments for standard services, for the enterprise market it's like waiting for Godot. For servers and data centers, automation is a must even for mid-sized enterprises, this is probably one of the reasons for which someone invented the cloud's "serverless" approach: in this case, you will not even care anymore about servers' maintenance, patching, software upgrades and so on.