NSO CDM Migration Guide

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

CONTENTS

CHAPTER 1

Introduction

In earlier releases of NSO, there could only be a single version of a YANG module loaded into the system. This had some limiting implications.

One limitation was that if two different devices were using two different revisions of a YANG module, NSO had to load one of them, usually the latest one. This often worked fine, but if the models were incompatible, the result was that NSO could not properly manage the incompatible parts of the model.

Another limitation was that if NSO implemented a model (for example "ietf-netconf-acm"), this model could not be used for a device. This scenario is the reason for the word *common* in Common Data Models (CDM).

The CDM solution solves these problems by using a technique called *YANG Schema Mount*, where all the data models from a device are mounted into the `/devices` tree in NSO. Each set of mounted data models is completely separated from the others (they are confined to a "mount jail"). This makes it possible to load different versions of the same YANG module for different devices.

In most cases, there are many devices running the same software version in the network managed by NSO, thus using the exact same set of YANG modules. With CDM, all YANG modules for a certain device (or family of devices) are contained in a *NED package* (or just *NED* for short). If the YANG modules on the device are updated in a backwards compatible way, the NED is also updated.

However, if the YANG modules on the device are updated in an incompatible way in a new version of the device's software, it might be necessary to create a new NED package for the new set of modules. Before CDM, this was not possible, since there would be two different packages that contained different versions of the same YANG module.

When a NED is being built, its YANG modules are compiled to be mounted into the NSO YANG model. This is done by *device compilation* of the device's YANG modules, and is performed via the `ncsc` tool provided by NSO.

CHAPTER 2

Internals

The YANG Module Namespace Identifier

Internally in NSO, a YANG module is identified by its *namespace*. Each such namespace must be unique. Before CDM, this namespace identifier was always the same as the XML namespace defined in the YANG module. With CDM, the namespace is constructed from a *mount-id* and the XML namespace. The resulting namespace is sometimes referred to as a *crunched namespace*.

The Mount Point

In order to implement CDM, NSO uses YANG Schema Mount, defined in RFC 8528. This document introduces a *mount point*, under which YANG models are mounted. NSO defines two such mount points, in `/devices/device/config` and `/devices/device/live-status`. Under these mount points, all the device's YANG modules are mounted.

This implies that traversing a path in the schema that crosses a mount-point signals that referencing a node under the mount point by using a module's name, prefix, or XML namespace may be ambiguous (since there may be multiple versions of the same module, with different definitions of the same node). In order to resolve this ambiguity, it is necessary to know the mount-id.

A NED package must define a `ned-id` that identifies the device-type for the NED. In NSO the ned-id is also the mount-id for the crunched namespaces.

Note This means that the ned-id must be unique for each NED and will serve the dual role of defining the device-type and mount-id.

So, when traversing a mount-point, NSO will internally lookup the ned-id for the specific device instance and resolve the ambiguities in module name, prefix or XML namespace. This way all user-code can and must use paths and XML namespaces just as before. There is no need for user-code to ever handle crunched namespaces.

CHAPTER 3

Backward Incompatibilities

North and south of NSO the all references and paths are described using XML namespaces, module names or prefixes (depending on protocol or API). The crunched namespaces are never be used. This to be able to keep service and other code intact at an NSO migration to a CDM release.

Efforts have been made to introduce CDM with as little backward incompatibilities as possible. However since the change to the NSO device schema tree is fundamental there are some changes to APIs and structures.

Schema Changes

Device Type

The device type in the tailf-ncs YANG module is a choice with four cases (netconf, cli, generic and snmp). For all these cases there is a ned-id leaf. With CDM the ned-id leaf has become mandatory, also for netconf, and the default value has been removed. In previous NSO releases the default was the identity 'ned:netconf'. This ned-id value still exists for backward compatible reasons but no NED build for a CDM release should use this value.

The ned-id is required to be globally unique over all other NED versions. In Chapter 6, *NED Internals* a scheme how to construct unique ned-ids are described.

When an existing NSO installation is migrated to CDM, NSO will automatically, as part of the migration process, attempt to fill in the proper ned id for each device. This also includes NETCONF devices. The prerequisite for this ned id replacement is that each NED package are replaced with it unique CDM compiled counterpart. In Chapter 4, *NSO CDM migration* is NSO migration procedure described in detail.

YANG import Statement

YANG modules that uses e.g. *leafrefs*, *when*, *must* and other xpath expressions need to have a *import* declaration for the modules that are referred to by the xpath expression.

Now if a device compiled (crunched) namespace is referred to in an xpath expression, this is by itself an possible ambiguity. Here the *import* statement will identify the used xml namespace. A number of crunched namespace could potentially then have the same XML namespace and be part of the xpath expression. The validation of these expressions are deferred at compile time and performed at load time instead.

Note When compiling a YANG module containing the above described situation there need to exist at least one crunched namespace for the XML namespace in the import statement. In makefiles or equivalent the crunched namespace should be referenced. These are normally found under `src/ncsc-out/module/yang` in the package directory of the compiled Ned.

Device Templates

Device templates have changed. The config container in the template list has moved under a new ned-id list keyed on ned-id YANG identities. The path `/devices/template[name]/config` have become `/devices/template[name]/ned-id[id]/config`.

The reason for this is that with CDM ambiguities between the same XML namespace for different ned-ids has been introduced. This new list makes it possible to handle such ambiguities in a template.

When an existing NSO installation is migrated to CDM, NSO will automatically, as part of the migration process, attempt to move existing templates in the configuration into this new form. However, templates stored as XML files will need to be changed manually.

Device group RPCs

Device group RPCs is the name of the functionality which makes it possible to invoke an RPC on all viable devices in a device group. The viable devices are the ones in the device group that have implemented the RPC.

The same problem with ambiguous XML namespaces occurs here as for device templates. Namely that the device RPC can have ambiguities in inputs and output for different ned-ids on the same XML namespace. Therefore the path in the NSO YANG model for the device group RPCs has moved from `/devices/device-group[name]/rpc` to `/devices/device-group[name]/ned-id[id]/rpc`.

As a result, the user will need to list the ned-ids for the RPC that should be invoked for the device-group. Note, that only the devices that have implemented the RPC can be invoked. In addition the ned-ids need also to match the device-type of the devices.

Notification Kickers

Notification kickers were earlier defined in a submodule to the tailf-ncs YANG module. This had some drawbacks where one major drawback was that it prevented the tailf-ncs module from being device compiled.

The notification kicker YANG module now has a new XML namespace and new prefix. Code that uses notification kickers might need to change.

API

In general the Java/C/Python APIs are kept intact. Still under the hood they have undergone major changes.

However certain API calls will fail if they internally try to traverse schema information past a mount-point e.g `/devices/device/config` without instance information so that an internal device-type/ned-id lookup can be performed.

Some API calls have therefore new variants which typically also passes a transaction and a path as arguments. The reason to pass also the transaction is to handle the possibility that new devices are created

in the same transaction that the internal lookup needs to retrieve. If no mount-points are traversed all API calls work as before.

Java API incompatibilities

In the Java API, schema node lookups was previously performed by the `MaapiSchemas.findCSNode(String nsName, String fmt, Object... arguments)` method. This still holds as long as the path, defined in fmt, does not traverse a `mount-point`. To traverse a `mount-point` a new method needs to be used, namely `MaapiSchemas.findCSNode(MountIdInterface mountGetter, String nsName, String fmt, Object... arguments)`. The latter method will hold for any node and is there for preferred. The following code snippet shows example of the usage:

```
// Schema traversal using the com.tailf.maapi.MountIdCb implementation
// of the interface MountIdInterface.

MaapiSchemas.CSNode node;
node = schemas.findCSNode(new MountIdCb(this.maapi, tid), Ncs.uri, path);

// If no current transaction id, tid, is available then -1 is allowed
node = schemas.findCSNode(new MountIdCb(this.maapi, -1), Ncs.uri, path);
```

In the Java API the ConfPath object is a primitive that is extensively used. The `ConfPath(String path, Object... arguments)` constructor is the most common and has the same problem of traversing a `mount-point`.

To change all usage of ConfPath with a new constructor would not be viable so instead the internal implementation of the `ConfPath(String path, Object... arguments)` constructor is changed so that a String path is deferred until it is used in a Maapi or Cdb context.

```
ConfPath p = new ConfPath("/ncs:devices/device{xyz}/config/top");

// At this point the parsing of this path is deferred since the
// moint-point /ncs:devices/device/config is traversed.
// This can be verified using the ConfPath.isParsingDeferred() method.
// In this case p.isParsingDeferred() will return true

// Any usage in Maapi or Cdb will work and resolve the parsing, for instance

boolean res = maapi.exists(tid, p);

// After this call p.isParsingDeferred() will return false.
```

However the caveat is that ConfPath can be used before parsing is resolved. Some methods are ok to use, but the following for methods will fail.

- `getCSNode()`
- `getKP()`
- `toString()`
- `toXPathString()`

If any of these methods are called before parsing is resolved they will throw a Conf.AmbiguousNamespaceException.

Keep in mind that this exception only occurs for ConfPath with deferred parsing because of mount-point traversal. And only before they are used in Maapi or Cdb. If this still is the case there are different ways to resolve it. First the problematic scenario:

```
ConfPath p = new ConfPath("/ncs:devices/device{xyz}/config/top");
```

```
p.toString() // this will throw an AmbiguousNamespaceException.
boolean res = maapi.exists(tid, p);
```

Resolving by moving:

```
ConfPath p = new ConfPath("/ncs:devices/device{xyz}/config/top");

boolean res = maapi.exists(tid, p);
p.toString() // now it is ok.
```

Resolving using new constructor:

```
// Use an constructor that makes the Maapi transaction available for
// parsing: ConfPath(maapi, th, path).
// There is also an corresponding constructor for Cdb: ConfPath(Cdb, th, path).

ConfPath p = new ConfPath(maapi, th, "/ncs:devices/device{xyz}/config/top");

p.toString() // now this works
boolean res = maapi.exists(tid, p);
```

Set a MountIdInterface instance to a deferred path:

```
ConfPath p = new ConfPath("/ncs:devices/device{xyz}/config/top");

if (p.isParsingDeferred()) {
    p.setMountIdGetter(new MountIdCb(maapi, tid));
}

p.toString() // now this works
boolean res = maapi.exists(tid, p);
```

Other Changes

NED compilations

The `ncsc` tool now have a argument --ncs-ned-id which is mandatory in conjunction with --ncs-compile-bundle and --ncs-compile-module.

Removed earlier deprecations

In NSO 5.1 there are some functionality that has been removed that was earlier deprecated.

In the java api the NavuCdbXXXSubscribers has been removed. How to handle this and do adaptations in the subscription code is covered in Chapter 10, *Appendix I*.

The old prime webui has been removed. See Chapter 11, *Appendix II*.

CHAPTER 4

NSO CDM migration

CDM is introduced in the NSO 5.1 release.

Note The supported target NSO releases for a NSO CDM release migration are from 4.x.y releases with x >= 5. For earlier NSO target releases the migration needs to be performed in two steps. First upgrade to a NSO release higher than 4.5 and verify that this works and then as a second step do the NSO migration to the NSO 5.x release.

The NSO CDM migration is more elaborate than NSO upgrades performed with earlier releases, hence the name migration instead of upgrade. This is because of the namespace changes of internally stored data. NSO CDM migration does, like earlier upgrades, support changes in YANG models and performs data upgrades accordingly for stored data in CDB. However since the CDM migration is more complex than usual, it is strongly recommended that the NED versions used for the migrated NSO deployment are kept exactly the same as the original target NSO deployment.

Note The probability to retrieve a CDM compiled NED release for a specific NED versions decreases with the age of the NED version. For old NED versions or NEDs that are not officially supported, e.g. developed by the customer, the makefiles or build scripts needs to be modified to incorporate the new --ncs-ned-id argument in the device compilation. See Chapter 7, *Converting old cli and generic NEDs to CDM* and Chapter 8, *Converting old netconf NEDs to CDM* on how this modification can be done.

Changing NED versions or adding NED versions (as new CDM functionality) can then be done as a separate step after the initial NSO CDM migration.

- **Step 1.** Create a backup of the NSO deployment.
- **Step 2.** Install the new CDM NSO release.
- **Step 3.** The CDB files need to be compacted this is done as a separate step by issuing the command.

```
ncs --cdb-compact ./ncs-cdb
```

Here the `./ncs-cdb` is the directory where the cdb files reside. When the command returns the cdb files are compacted. A recommendation is to store a copy of these in another location if the last step, the migration, has to be attempted again after failure.

- **Step 4.** Get hold of the NED packages compiled for the CDM release that have the same NED version as the ones used in the target deployment. Replace the old NED packages with the CDM compiled counterparts. Also compile any other package for the new NSO release.
- **Step 5.** Search for any initial XML files that are loaded by the system at NSO startup. These files are normally found under the `ncs-cdb` directory. Look for any XML files that contain device templates. NSO will not be able to start with these files if the device template structure is not changed. The resolution is to change this structure to the correct one. As an alternative, verify that these templates are already loaded and stored in CDB. In that case they can be moved/removed from the `ncs-cdb` directory and he NSO CDM migration code will handle the device template migration in CDB. After the migration the new device templates can then be saved to a file that will replace the original.
- **Step 6.** The code will need to be prepared for the changes necessary as described in Chapter 3, *Backward Incompatibilities*.
- **Step 7.** Start NSO to perform the migration.
- **Step 8.** Check the cdb-migration.log that there are no errors found during migration.

CHAPTER 5

Service Handling of Ambiguous Device Models

As stated above, efforts have been made to keep APIs etc backward compatible. A major concern has been to keep old service code functional without change. However this only works as long as there are no XML namespace ambiguities between different devices in the system.

In other words, after an initial NSO CDM migration with the same NEDs as before the migration, the service should work as before. Later, when new NED versions with diverging XML namespaces are introduced, adaptations might be needed in the services for these new NEDs. But not necessarily; it depends on where in the specific NED models that the ambiguities reside. Existing services might not refer to these parts of the model and in that case they do not need any adaptations.

Finding out if and where services need adaptations can be non-trivial. An important exception is template services which check and point out ambiguities at load time (NSO startup). In Java or Python code this is harder and basically falls back to code reviews and testing.

The changes in service code to handle ambiguities are straightforward but different for templates and code.

Template Services

In templates there are new processing instructions *if-ned-id* and *elif-ned-id*. When the template specifies a node in an XML namespace where an ambiguity exists, the *if-ned-id* process instruction is used to resolve that ambiguity.

The processing instruction *else* can be used in conjunction with *if-ned-id* and *elif-ned-id* to capture all other ned-ids.

For the nodes in the XML namespace where no ambiguities occur this process instruction is not necessary.

```
<config-template xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device foreach="{apache-device}">
      <name>{current()}</name>
      <config>
        <?if-ned-id apache-nc-1.0:apache-nc-1.0?>
          <vhosts xmlns="urn:apache">
            <vhost>
              <hostname>{/vhost}</hostname>
```

```
            <doc-root>/srv/www/{/vhost}</doc-root>
          </vhost>
        </vhosts>
      <?elif-ned-id apache-nc-1.1:apache-nc-1.1?>
        <public xmlns="urn:apache">
          <vhosts>
            <vhost>
              <hostname>{/vhost}</hostname>
              <aliases>{/vhost}.public</aliases>
              <doc-root>/srv/www/{/vhost}</doc-root>
            </vhost>
          </vhosts>
        </public>
      <?end?>
    </config>
  </device>
 </devices>
</config-template>
```

Java Services

In Java the service code must handle the ambiguities by code where the devices ned-id is tested before setting the nodes and values for the diverging paths.

The ServiceContext class has a new convenience method, *getNEDIdByDeviceName* which helps retrieving the ned-id from the device name string.

```
    @ServiceCallback(servicePoint="websiteservice",
                     callType=ServiceCBType.CREATE)
    public Properties create(ServiceContext context,
                             NavuNode service,
                             NavuNode root,
                             Properties opaque)
                             throws DpCallbackException {

...

                NavuLeaf elemName = elem.leaf(Ncs._name_);
                NavuContainer md = root.container(Ncs._devices_).
                    list(Ncs._device_).elem(elemName.toKey());

                String ipv4Str = baseIp + ((subnet<<3) + server);
                String ipv6Str = "::ff:ff:" + ipv4Str;
                String ipStr = ipv4Str;
                String nedIdStr =
                    context.getNEDIdByDeviceName(elemName.valueAsString());
                if ("webserver-nc-1.0:webserver-nc-1.0".equals(nedIdStr)) {
                    ipStr = ipv4Str;
                } else if ("webserver2-nc-1.0:webserver2-nc-1.0"
                           .equals(nedIdStr)) {
                    ipStr = ipv6Str;
                }

                md.container(Ncs._config_).
                    container(webserver.prefix, webserver._wsConfig_).
                    list(webserver._listener_).
                    sharedCreate(new String[] {ipStr, ""+8008});

                ms.list(lb._backend_).sharedCreate(
                    new String[]{baseIp + ((subnet<<3) + server++),
                                 ""+8008});
```

```
...

            return opaque;
        } catch (Exception e) {
            throw new DpCallbackException("Service create failed", e);
        }

    }
```

Python Services

In the Python API there is also a need to handle ambiguities by checking the ned-id before setting the diverging paths. Use get_ned_id() from ncs.application to resolve ned-ids

```
import ncs

def _get_device(service, name):
    dev_path = '/ncs:devices/ncs:device{%s}' % (name, )
    return ncs.maagic.cd(service, dev_path)

class ServiceCallbacks(Service):
    @Service.create
    def cb_create(self, tctx, root, service, proplist):
        self.log.info('Service create(service=', service._path, ')')

        for name in service.apache_device:
            self.create_apache_device(service, name)

        template = ncs.template.Template(service)
        self.log.info(
            'applying web-server-template for device {}'.format(name))
        template.apply('web-server-template')
        self.log.info(
            'applying load-balancer-template for device {}'.format(name))
        template.apply('load-balancer-template')

    def create_apache_device(self, service, name):
        dev = _get_device(service, name)
        if 'apache-nc-1.0:apache-nc-1.0' == ncs.application.get_ned_id(dev):
            self.create_apache1_device(dev)
        elif 'apache-nc-1.1:apache-nc-1.1' == ncs.application.get_ned_id(dev):
            self.create_apache2_device(dev)
        else:
            raise Exception('unknown ned-id {}'.format(get_ned_id(dev)))

    def create_apache1_device(self, dev):
        self.log.info(
            'creating config for apache1 device {}'.format(dev.name))
        dev.config.ap__listen_ports.listen_port.create(("*", 8080))
        dev.config.ap__clash = dev.name

    def create_apache2_device(self, dev):
        self.log.info(
            'creating config for apache2 device {}'.format(dev.name))
        dev.config.ap__system.listen_ports.listen_port.create(("*", 8080))
        dev.config.ap__clash = dev.name
```

CHAPTER 6

NED Internals

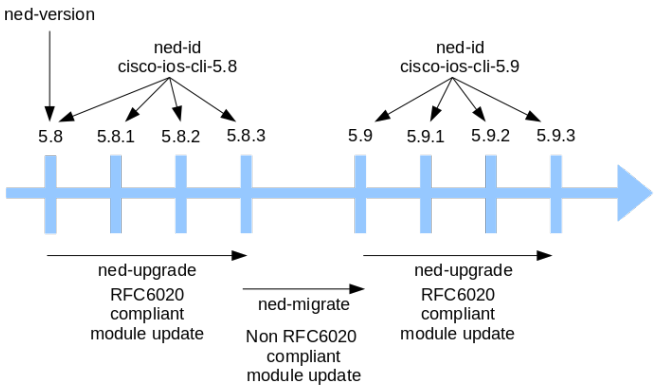
A NED has a version associated with it. A version consists of a sequence of numbers separated by dots ('.'). The first two number defines the major and minor version number, the third number defines the maintenance version number and any following numbers are patch release version numbers.

For instance, the 5.8.1 number indicates a maintenance release (1) on the minor release 5.8 and 5.8.1.1 indicates a patch release (1) on the maintenance release 5.8.1. Any incompatible YANG model change will require the major or minor version number to change, i.e. any 5.8.x version is to be backward compatible with the previous.

When a NED release is replaced with a later maintenance/patch release with the same major/minor version, NSO can do a simple data model upgrade to handle stored instance data in CDB. There is no risk that any data would be lost by this sort of upgrade.

On the other hand when a NED is replaced by a new major/minor release this becomes a NED migration. These are non trivial since the YANG model changes can result in loss of instance data if not handled correctly. With CDM the first and most important feature in NED migration is that both the old and new NED releases can be loaded at the same time and analysis of the YANG models and instance data can be performed before NED migration is attempted.

Figure 1. NED Version Scheme



• Backward Incompatibilities, page 16

Backward Incompatibilities

NED Settings

NED settings are YANG models augmented as config in NSO that controls behavior of the NED. These setting are augmented under `/devices/global-settings/ned-settings`, `/devices/profiles/ned-settings` and `/devices/device/ned-settings`. Traditionally these NED settings have been accompanied by a *when* expression specifying the ned-id for which the settings are legal. With the introduction of CDM such *when* expressions on specific ned-ids are not recommended since ned-id will change with NED releases.

Instead there is a need to introduce a 'family' identity that becomes base for all NED releases for a certain family. The *when* expressions can then use `derived-from` syntax to be legal for all NED releases in the family.

Schema Traversals in NED Java code

As stated above schema traversal works as before until a mount-point is reached in the path. At that point a lookup of the current mount-id (ned-id) is necessary to resolve any ambiguities in module name, prefix or XML namespace. Since the NED by definition works on devices under a NED any schema traversal in NED code falls under the latter case.

Pre CDM retrieving a CSNode from the Maapi Schema for a path was as simple as calling the `findCSNode(Namespace, Path)` function.

```
private  MaapiSchemas.CSNode getCSNode(String path) throws MaapiException {
  return schemas.findCSNode(Ncs.uri, path);
}
```

With CDM the original `findCSNode(Namespace, Path)` still exists for backward compatibility but in the NED code case all paths are under a mount-point and hence this function will return an error that a lookup cannot be performed. The reason for this is that a maapi call to the NSO service is necessary to retrieve the mount-id for the device. This is accomplished with a mount-id callback `MountIdCb(Maapi, Th)` which takes a Maapi instance and optionally a current transaction.

```
private  MaapiSchemas.CSNode getCSNode(String path) throws MaapiException {
  return schemas.findCSNode(new MountIdCb(this.mm, -1), Ncs.uri, path);
}
```

CHAPTER 7

Converting old cli and generic NEDs to CDM

This description expects the NED to be built with the structure as defined by the `ncs-make-package`. If this doesn't hold then the adaptations has to be done otherwise with an outcome that resembles the one described below.

- *package-meta-data.xml*

  In the `top` directory of the NED:

  ```
  mv package-meta-data.xml src/package-meta-data.xml.in
  ```

- *Makefile*

  In `src/Makefile` add the following make variable at the top after the other make variables, change XXXX and YYYY to name found in the module statement in the YANG file in `src/yang`.

  Check the existing YANG file for the existing ned-id, verify it is named according to the most common patterns and has the name:

  `tailf-ned-XXXX-YYYY-id.yang`

  this can be seen in the Makefile by searching for the string 'id'. If this is not the case the Make variable `JAVA_ID_FILE` has to be explicitly set.

  ```
  YANG_MODULE = tailf-ned-XXXX-YYYY
  NED_ID_ARG = $(shell [ -x ${NCS_DIR}/support/ned-ncs-ned-id-arg ] && \
               ${NCS_DIR}/support/ned-ncs-ned-id-arg package-meta-data.xml.in)
  ifeq ($(NED_ID_ARG),)
  NED_ID_FILE = $(YANG_MODULE)-id
  else
  NED_ID_FILE = tailf-ned-id-$(shell echo $(NED_ID_ARG) | cut -d: -f2)
  endif
  JAVA_ID_FILE = $(shell echo $(YANG_MODULE) | \
      perl -pe 's/-ned//;s/(^|-)./uc($$&)/ge;s/-//ge;s/(.)/lc($$&)/e')Id
  ```

  Add the following dependencies first among dependencies to the `all:` target:

  ```
  ../package-meta-data.xml ned-id-file
  ```

  to the `clean:` target add:

  ```
  rm -f ../package-meta-data.xml
  ```

  Add the following new targets:

  ```
  ../package-meta-data.xml: package-meta-data.xml.in
  ```

```
        rm -rf $@
        if [ -x ${NCS_DIR}/support/ned-make-package-meta-data ]; then \
           ${NCS_DIR}/support/ned-make-package-meta-data $<;          \
        else                                                          \
           cp $< $@;                                                  \
        fi
        chmod +w $@

ned-id-file:
        if [ -x ${NCS_DIR}/support/ned-make-package-meta-data ]; then   \
            echo -n "$(NED_ID_FILE) is built by: ";                     \
            echo "support/ned-make-package-meta-data";                  \
        else                                                            \
            $(NCSC) -c yang/$(NED_ID_FILE).yang                         \
            -o ../load-dir/$(NED_ID_FILE).fxs;                          \
        fi
        $(NCSC) $(JFLAGS)/$(JAVA_ID_FILE).java ../load-dir/$(NED_ID_FILE).fxs
```

The make variable `$(NED_ID_ARG)` needs to be added as an argument to all targets which are `ncsc` compiled. Look for everything which uses the option `'--ncs-device-type'`

And finally find and remove all the old dependencies and targets for the old ned-id file.

- *Remove identity from Java-code*

  If NED is implemented in Java remove the identity method:

```
-    /**
-     * Display NED identity
-     */
-    public String identity() {
-        return "asa-id:cisco-asa";
-    }
-
```

- *build.xml*

  Add the file `package-meta-data.xml` to the private jar file of the package by updating `src/java/build.xml`. Add the following, between .... snip ...., to the compile task:

```
<target name="package" depends="compile">
.... snip ....

    <jar update="true" destfile="${privatejar.dir}/${package}.jar"
         basedir="../.."
         includes="package-meta-data.xml"/>

.... snip ....
</target>
```

- *Verify changes*

  Verify `Makefile`

  Compile code for both versions < 5.0 and versions >= 5.0:

```
make -C src
make -C src
```

  twice without any errors then do:

```
make -C src clean
make -C src clean
```

twice without any errors.

Everything now should have been cleaned, if this is not the case update the `clean:` of the relevant `Makefile`

CHAPTER 8

Converting old netconf NEDs to CDM

This description expects the NED to be built with the structure as defined by the `ncs-make-package`. If this doesn't hold then the adaptations has to be done otherwise with an outcome that resembles the one described below.

For simple netconf NEDS use the `ncs-make-package` to generate a new package:

```
ncs-make-package --no-java --no-python --dest /tmp/new \
  --netconf-ned src/yang XXXX-YYYY
```

Use the generated package in `/tmp/new` as is or use it as a guide how a package is structured and the `Makefile` is written.

If the simple case is not applicable then for the general case follow the below steps:

- *package-meta-data.xml*
  In the top directory of the NED:

  ```
  mv package-meta-data.xml src/package-meta-data.xml.in
  ```

- *Makefile*
  In `src/Makefile` add the following make variable at the top after the other make variables:

  ```
  NED_ID_ARG = $(shell [ -x ${NCS_DIR}/support/ned-ncs-ned-id-arg ] && \
      ${NCS_DIR}/support/ned-ncs-ned-id-arg package-meta-data.xml.in)
  ```

  In `src/Makefile` add the following dependencies first among dependencies to the `all:` target:

  ```
  ../package-meta-data.xml ned-id-file
  ```

  and to the `clean:` target add:

  ```
  rm -f ../package-meta-data.xml
  ```

  Add the new targets:

  ```
  ../package-meta-data.xml: package-meta-data.xml.in
          rm -rf $@
          if [ -x ${NCS_DIR}/support/ned-make-package-meta-data ]; then \
             ${NCS_DIR}/support/ned-make-package-meta-data $<;          \
          else                                                          \
             cp $< $@;                                                  \
          fi
          chmod +w $@
  ```

```
ned-id-file:
        if [ -x ${NCS_DIR}/support/ned-make-package-meta-data ]; then   \
            echo -n "$(NED_ID_FILE) is built by: ";                     \
            echo "support/ned-make-package-meta-data";                  \
        fi
```

Add the make variable `$(NED_ID_ARG)` as an argument to all targets which are `ncsc` compiled. Look for everything which uses the option '--ncs-device-type'

- *Verify changes*

  Compile code for both versions < 5.0 and versions >= 5.0:

  ```
  make -C src
  make -C src
  ```

  twice without any errors, then do:

  ```
  make -C src clean
  make -C src clean
  ```

  twice without any errors.

  Everything now should have been cleaned, if this is not the case update the `clean:` of the relevant `Makefile`

CHAPTER 9

LSA and Common Data Models

LSA and Common Data Models

With the introduction of Common Data Models or CDM for short, there are new options for the designing a LSA deployment.

The CDM functionality implies that a device that have NED releases containing models which are incompatible with each other can still be loaded into NSO and coexist as separate packages. Without going into details here, the packages need different ned-ids and these ned-ids are used at device compilation to create internally unique namespaces.

So for a NED package not only the device-type but also the ned-id has become mandatory. For this reason also the LSA RFS service packages need to be device compiled with a specified ned-id. This ned-id is typically the predefined *lsa-netconf* and this is also what is maintained if `ncs-make-package` tool with directive `--lsa-netconf-ned` is used.

LSA and Common Data Models

CHAPTER 10

Appendix I

NavuContext(Cdb) and NavuContext(CdbSession) constructors deprecated

In NSO 4.2 the NAVU has been changed to fully support writable operational data transactions via MAAPI. For this reason the NavuContext(Cdb) and NavuContext(CdbSession) constructors has been deprecated. The NavuCdbXXXSubscribers was deprecated in NSO 4.2 and finally removed in NSO 5.1 and the strong recommendation is to change these to plain Cdb subscribers and if necessary in diff iterators make use of KeyPath2NavuNode.getNode() to be able to reuse old NavuCdbDiffIterator code.

The following is an example on how to make these changes. The prerequisite is an NavuCdbSubscriber that is used as an application component in a package.

We start with the following code which is part of the web-site-service example.

```
public class CdbUpdateHandler implements ApplicationComponent,
                                          NavuCdbConfigDiffIterate {
    private static Logger LOGGER = Logger.getLogger(CdbUpdateHandler.class);
    private static Logger PROPLOG = Logger.getLogger("PROPLOG");
    private NavuCdbSubscriber navuCfgSubscriber;

    private void configurePropLogger() throws IOException {
        SimpleLayout layout = new SimpleLayout();
        SimpleDateFormat fmt = new SimpleDateFormat("yyyy-MM-dd");
        String fileName = "logs/props.log";
        FileAppender appender = new FileAppender(layout, fileName);
        PROPLOG.addAppender(appender);
        PROPLOG.setAdditivity(false);
        PROPLOG.setLevel(Level.DEBUG);
    }

    @Override
    public void init() throws Exception {
        try {
            configurePropLogger();
            NcsMain ncsMain = NcsMain.getInstance();
            ConfPath p = new ConfPath("/ncs:services/properties/wsp:web-site");
            navuCfgSubscriber =
                NavuCdbSubscribers.configSubscriber(ncsMain.getNcsHost(),
                                                    ncsMain.getNcsPort());
```

```
            navuCfgSubscriber.register(this, p);
        } catch (Exception e) {
            throw new RuntimeException("Fail in init", e);
        }
    }


    @Override
    public void run() {
        navuCfgSubscriber.subscriberStart();
        LOGGER.info("subscribed: ready");
    }


    @Override
    public void finish() throws Exception {
        try {
            navuCfgSubscriber.subscriberStop();
            navuCfgSubscriber.awaitStopped();
        } catch (InterruptedException e) {
            LOGGER.error("Exception in shutdownSubscriber", e);
        } finally {
            navuCfgSubscriber.executor().shutdown();
        }
    }

    public void iterate(NavuCdbSubscriptionConfigContext ctx){
        try {
            NavuNode node = ctx.getNode();

            DiffIterateOperFlag op = ctx.getOperFlag();
            switch (op) {
            case MOP_CREATED:
                PROPLOG.info("CREATED : " + node.getKeyPath());
                break;
            case MOP_VALUE_SET:
                PROPLOG.info("SET : " + node.getKeyPath());
                PROPLOG.info("VALUE : " + ctx.getNewValue());
                break;
            case MOP_MODIFIED:
                PROPLOG.info("MODIFIED : " + node.getKeyPath());
                break;
            case MOP_DELETED:
                PROPLOG.info("DELETED : " + node.getKeyPath());
                break;
            }
        } catch (Exception e) {
            LOGGER.error("",e);
        }
        ctx.iterContinue();
    }
}
```

First change the use of the NavuCdbConfigDiffIterate interface to the plain CdbDiffIterate interface instead:

```
9c9,10
< public class CdbUpdateHandler implements ApplicationComponent, CdbDiffIterate {
---
> public class CdbUpdateHandler implements ApplicationComponent,
>                                          NavuCdbConfigDiffIterate {
```

Make use of a Cdb instance for the subscription and a Maapi instance for a NavuContext that will be used in the iterate method. We also define a requestStop flag to signal the finish of the subscription: We remove the reference to the NavuCdbSubscriber:

```
12,19c13
<
<     @Resource(type=ResourceType.CDB, scope=Scope.INSTANCE)
<     private Cdb cdb;
<     private CdbSubscription cdbsubscr;
<     @Resource(type=ResourceType.MAAPI, scope=Scope.INSTANCE)
<     private Maapi maapi;
<     private NavuContext ncontext;
<     private boolean requestStop = false;
---
>     private NavuCdbSubscriber navuCfgSubscriber;
```

In the application components init() method we start a user session for the Maapi and define the cdb subscription. The NavuCdbSubscribers registration is removed:

```
34,37d27
<             maapi.startUserSession("system", InetAddress.getByName(null),
<                                    "system", new String[]{},
<                                    MaapiUserSessionFlag.PROTO_TCP);
<             ncontext = new NavuContext(maapi);
39,42c29,34
<             cdbsubscr = cdb.newSubscription();
<             cdbsubscr.subscribe(1, 0, "/ncs:services/properties/wsp:web-site");
<             cdbsubscr.subscribeDone();
<             requestStop = false;
---
>             NcsMain ncsMain = NcsMain.getInstance();
>             ConfPath p = new ConfPath("/ncs:services/properties/wsp:web-site");
>             navuCfgSubscriber =
>                 NavuCdbSubscribers.configSubscriber(ncsMain.getNcsHost(),
>                                                     ncsMain.getNcsPort());
>             navuCfgSubscriber.register(this, p);
```

In the application components run() method we remove the NavuCdbSubscriber start and replace it with a read loop for the plain Cdb subscriber:

```
50a43
>         navuCfgSubscriber.subscriberStart();
52,69d44
<         try {
<             while (!requestStop) {
<                 try {
<                     int[] spoints = cdbsubscr.read();
<                     ncontext.startRunningTrans(Conf.MODE_READ);
<                     cdbsubscr.diffIterate(spoints[0], this);
<                     int th = ncontext.clearTrans();
<                     if (th > 0) {
<                         maapi.finishTrans(th);
<                     }
<                 } finally {
<                     cdbsubscr.sync(CdbSubscriptionSyncType.DONE_PRIORITY);
<                 }
<             }
<         } catch (Throwable e) {
<             LOGGER.error("Error in subscription:", e);
<         }
<         requestStop = false;
```

In the application components finish() method we set the requestStop flag, finish the maapi user session and unregister the resources. Here the ResourceManager.unregisterResources will close both the Cdb and Maapi sockets. We remove the earlier NavuCdbSubscriber stopping code:

```
75,78d49
<         requestStop = true;
<         try {
<             maapi.endUserSession();
<         } catch (Throwable ignore) {}
80,81c51,57
<             ResourceManager.unregisterResources(this);
<         } catch (Throwable ignore) {}
---
>             navuCfgSubscriber.subscriberStop();
>             navuCfgSubscriber.awaitStopped();
>         } catch (InterruptedException e) {
>             LOGGER.error("Exception in shutdownSubscriber", e);
>         } finally {
>             navuCfgSubscriber.executor().shutdown();
>         }
```

We change the iterate method to the CdbDiffIterate interface:

```
84,88c60
<     public DiffIterateResultFlag iterate(ConfObject[] kp,
<                                          DiffIterateOperFlag op,
<                                          ConfObject oldValue,
<                                          ConfObject newValue,
<                                          Object initstate) {
---
>     public void iterate(NavuCdbSubscriptionConfigContext ctx){
```

In the iterate() method we make use of the KeyPath2NavuNode.getNode() to obtain a NavuNode for the subscription entry:

```
90c62
<             NavuNode node = KeyPath2NavuNode.getNode(kp, ncontext);
---
>             NavuNode node = ctx.getNode();
```

In the iterate() method obtain the DiffIterateOperFlag and the newValue directly and we change we need to return with the DiffIterateResultFlag directly:

```
91a64
>             DiffIterateOperFlag op = ctx.getOperFlag();
98c71
<                 PROPLOG.info("VALUE : " + newValue);
---
>                 PROPLOG.info("VALUE : " + ctx.getNewValue());
110c83
<         return DiffIterateResultFlag.ITER_CONTINUE;
---
>         ctx.iterContinue();
```

The resulting code should look the as the following:

```
public class CdbUpdateHandler implements ApplicationComponent, CdbDiffIterate {
    private static Logger LOGGER = Logger.getLogger(CdbUpdateHandler.class);
    private static Logger PROPLOG = Logger.getLogger("PROPLOG");

    @Resource(type=ResourceType.CDB, scope=Scope.INSTANCE)
    private Cdb cdb;
    private CdbSubscription cdbsubscr;
```

```
@Resource(type=ResourceType.MAAPI, scope=Scope.INSTANCE)
private Maapi maapi;
private NavuContext ncontext;
private boolean requestStop = false;

private void configurePropLogger() throws IOException {
    SimpleLayout layout = new SimpleLayout();
    SimpleDateFormat fmt = new SimpleDateFormat("yyyy-MM-dd");
    String fileName = "logs/props.log";
    FileAppender appender = new FileAppender(layout, fileName);
    PROPLOG.addAppender(appender);
    PROPLOG.setAdditivity(false);
    PROPLOG.setLevel(Level.DEBUG);
}

@Override
public void init() throws Exception {
    try {
        maapi.startUserSession("system", InetAddress.getByName(null),
                               "system", new String[]{},
                               MaapiUserSessionFlag.PROTO_TCP);
        ncontext = new NavuContext(maapi);
        configurePropLogger();
        cdbsubscr = cdb.newSubscription();
        cdbsubscr.subscribe(1, 0, "/ncs:services/properties/wsp:web-site");
        cdbsubscr.subscribeDone();
        requestStop = false;
    } catch (Exception e) {
        throw new RuntimeException("Fail in init", e);
    }
}


@Override
public void run() {
    LOGGER.info("subscribed: ready");
    try {
        while (!requestStop) {
            try {
                int[] spoints = cdbsubscr.read();
                ncontext.startRunningTrans(Conf.MODE_READ);
                cdbsubscr.diffIterate(spoints[0], this);
                int th = ncontext.clearTrans();
                if (th > 0) {
                    maapi.finishTrans(th);
                }
            } finally {
                cdbsubscr.sync(CdbSubscriptionSyncType.DONE_PRIORITY);
            }
        }
    } catch (Throwable e) {
        LOGGER.error("Error in subscription:", e);
    }
    requestStop = false;
}


@Override
public void finish() throws Exception {
    requestStop = true;
    try {
        maapi.endUserSession();
```

```
            } catch (Throwable ignore) {}
            try {
                ResourceManager.unregisterResources(this);
            } catch (Throwable ignore) {}
        }

        public DiffIterateResultFlag iterate(ConfObject[] kp,
                                             DiffIterateOperFlag op,
                                             ConfObject oldValue,
                                             ConfObject newValue,
                                             Object initstate) {
            try {
                NavuNode node = KeyPath2NavuNode.getNode(kp, ncontext);

                switch (op) {
                case MOP_CREATED:
                    PROPLOG.info("CREATED : " + node.getKeyPath());
                    break;
                case MOP_VALUE_SET:
                    PROPLOG.info("SET : " + node.getKeyPath());
                    PROPLOG.info("VALUE : " + newValue);
                    break;
                case MOP_MODIFIED:
                    PROPLOG.info("MODIFIED : " + node.getKeyPath());
                    break;
                case MOP_DELETED:
                    PROPLOG.info("DELETED : " + node.getKeyPath());
                    break;
                }
            } catch (Exception e) {
                LOGGER.error("",e);
            }
            return DiffIterateResultFlag.ITER_CONTINUE;
        }
    }
```

CHAPTER 11

Appendix II

Old web-ui removed

The previous generation of the web ui (Prime) has been deprecated since NSO version 4.5. In NSO release 5.1 it will be removed completely. Any example or application that is integrated with Prime will no longer work with this release and needs to be modified to work as stand alone ui application. Once the application is decoupled from Prime, it can be used in conjunction with the other tools and be included as an item in the Application hub.

For more information on how to deploy your own application and use it together with the other tools, take a look at this simple example on the NSO-developer GitHub.