20-12-2023

Milestone M7.4
Example Workflow Orchestrator with IMS and IPAM Integration

Contractual Date:	30-12-2023
Actual Date:	20-12-2023
Grant Agreement No.:	101100680
Work Package:	WP7
Task Item:	Task 3
Nature of Milestone:	Supporting Document, Specification]
Dissemination Level:	PU (Public)
Lead Partner:	SURF
Document ID:	GN5-1-23-8a1761
Authors:	Hans Trompert (SURF)

Abstract

A set of best common practices is explained using an orchestrator implementation based on the open source Workflow Orchestrator. A basic set of NREN products and workflow are modelled for a Node, Core Link, Customer port and L2VPN service, together with an IMS and IPAM integration using NetBox.

Table of Contents

Table of Figures

Executive Summary

More and more NRENs are making use of the open-source Workflow Orchestrator (WFO) for the automation and orchestration of their operational network procedures and flows of information. Each time an NREN creates an orchestrator based on the WFO framework, custom integration code needs to be written that is business specific. To support NRENs in helping each other write the needed code, facilitate collaboration towards the further development of the framework, and achieve a set of standardised products and workflows, a set of best common practices (BCP) are set forth in this document.

To start with, a standard folder layout is described to organise the custom integration code base. This helps in quickly finding similar code in different implementations. To help illustrate the BCPs, an example orchestrator has been implemented for a virtual NREN. The defined products model a network node, a core link between nodes, a customer port, and a customer L2VPN service between those ports. For all products, the complete set of create, modify, terminate, and validate workflows are implemented. Products and product blocks are described in Domain Models that are designed to help the developer manage complex data models and interact with the objects in a user-friendly way. The how and why for each step are described in detail.

Finally, the use of services is introduced. A service comprises collections of helper functions that deliver a service to other parts of the code base. The common programming pattern of function overloading is used for the implementation of a service, which can be as simple as the generation of a description based on the product block domain model, or a complete interface to query, create, update or delete objects in an OSS or BSS. In the example orchestrator, a service is implemented to interface with NetBox that is being used as IMS and IPAM. The mappings between the product blocks and the objects in NetBox are described, and the interface is fully implemented for the supported products.

The example orchestrator is fully functional and showcases how the WFO can be integrated with NetBox.

1 Introduction

To capture the Best Common Practices for implementing a network orchestrator using the Workflow Orchestrator (WFO) software framework, an example orchestrator is implemented that can be found at https://github.com/workfloworchestrator/example-orchestrator. This document can be used as a reading guide for the code base that has been written, and will highlight many best practices and the reasoning behind them. A basic understanding of the inner workings of the Workflow Orchestrator is assumed up to a level as discussed in GN5-1 Milestone M7.3 *Common NREN Network Service Product Models* [GN5-1_M7.3] and explained in the workshops that can be found on the Workflow Orchestrator website [WFO]. Basic knowledge in designing and operating computer networks and an accompanying product portfolio and procedures is also assumed.

The products and workflows implemented in the example orchestrator are based on a simple fictional NREN that has the following characteristics:

- The network consists of Provider and Provider Edge network nodes
- The network nodes are connected to each other through core links
- On top of this substrate a set of services like Internet Access, L3VPN and L2VPN are offered
- The Operations Support Systems (OSS) used are:
  - An IP Administration Management (IPAM) tool
  - A network Inventory Management System (IMS)
  - A Network Resource Manager (NRM) to provision the network
- There is no Business Support System (BSS) yet

This NREN decided on a phased introduction of automation in their organisation, only automating some of the procedures and flows of information while leaving others unautomated for the time being, as follows:

- Automated administration and provisioning of:
  - Network nodes including loopback IP addresses
  - Core links in between network nodes including point-to-point IP addresses
  - Customer ports
  - Customer L2VPN's
- Not automated administration and provisioning of:
  - Role, make and model of the network nodes
  - Sites where network nodes are installed
  - Customer services like Internet Access, L3VPN, …
  - Internet peering

[NetBox] is used as IMS and IPAM, and serves as the source of truth for the complete IP address administration and physical and logical network infrastructure. It has a REST-based API that makes it easy to integrate with the Workflow Orchestrator.

2 Example Orchestrator

To automate the administration and provisioning of the nodes, core links, customer ports and L2VPN's of the virtual NREN, an orchestrator is implemented making use of the WFO framework.

2.1 Folder Layout

Creating an orchestrator based on the WFO framework needs custom integration code that is business specific. This code can be organised as described below. A standard folder layout not only makes it easier to navigate different orchestrator implementations, but also helps with keeping the code organised while the number of products and associated workflows increases. The following layout is recommended and is, for example, also being used in the WFO workshops and by many of the WFO users.

```
├── migrations
│   └── versions
│       └── schema
├── products
│   ├── product_blocks
│   ├── product_types
│   └── services
│       └── <service>
├── services
│   └── <service>
├── templates
├── translations
├── utils
└── workflows
    ├── <product>
    └── tasks
```

Figure 1: Folder layout

2.1.1 migrations/versions/schema

This is the default location used by Alembic to store migration files. Alembic is a lightweight database migration tool that is part of [SQLAlchemy], and uses multiple HEADs to allow both the orchestrator-core package and the implementation using this package to maintain its own list of migrations. Usually there is at least a migration file for each new product plus associated workflows that are added to the implementation.

2.1.2 products/product_types

Each product has its own file, named after the product, that describes the product domain model in all its lifecycle stages. For example, the filename for a L2VPN product would be `l2vpn.py`.

2.1.3 products/product_blocks

Products can use one or more product blocks, and product blocks can be shared by different products. Every product block that is defined, has a file with the same name as the product block, to store the domain models in all its lifecycle stages. For example, the core port product block used by the core link product has a file called `core_port.py` in this folder.

2.1.4 products/services

This is a collection of helper functions that deliver a service to product-related code, such as the generation of descriptions, or payload for OSS/BSS API's, for different product and product blocks. For example, the folder `products/services/netbox/` contains the NetBox API payload service.

2.1.5 services

Similar to the product services but with code base wide helper functions. For example, the folder `services/netbox/` contains the service that interfaces with the NetBox API.

2.1.6 templates

A list of product configuration templates, with a template per product. Based on a template, currently an experimental feature, the WFO can generate skeleton code for the product and product block domain models, all four types of workflows including input forms, registration of the product and workflows, and the corresponding database migration.

2.1.7 translations

The translations for the WFO GUI for input form fields, subscriptions, subscription instances, and workflows.

2.1.8 utils

A collection of helper functions that are not directly related to the code base but are, for example, used to setup a deployment environment or generate documentation.

2.1.9 workflows

Every product has a folder here, named after the product. Each folder contains the collection of workflows for that product. Every workflow has its own file, and the filename is prefixed with the type of workflow. For example, the folder `workflows/port/` contains the workflows for the Port product, and the file `workflows/port/create_port.py` contains the Port subscription create workflow.

2.1.10 shared

Throughout the code base, shared folders are used that contain helper functions for that module and below. As good coding practice, it is best to define the helper functions as locally as possible.

2.2 Main Application

The main application is as simple as shown below, and can be deployed by an ASGI server like [Uvicorn].

```
from orchestrator import OrchestratorCore
from orchestrator.settings import AppSettings

import products
import workflows

app = OrchestratorCore(base_settings=AppSettings())
```

Figure 2: Main application

All other code is referenced by importing the products and workflows modules.

2.3 Implemented Products

In the `product.product_types` module the following products are defined:

- Node
- CoreLink
- Port
- L2vpn

And in the `product.product_blocks` module the following product blocks are defined:

- NodeBlock
- CoreLinkBlock
- CorePortBlock
- PortBlock
- SAPBlock
- VirtualCircuitBlock

Usually, the top-level product block of a product is named after the product, but this is not true for the top-level product block of the L2VPN product. The more generic name `VirtualCIrcuitBlock` allows the reuse of this product block by other services such as Internet Access and L3VPN.

The Service Access Point (SAP) product block `SAPBlock` is used to encapsulate transport-specific service endpoint information, in our case Ethernet 802.1Q is used and the SAP holds the VLAN used on the indicated port.

When this example orchestrator is deployed, it can create a growing graph of product blocks as is shown in Figure 3 below.

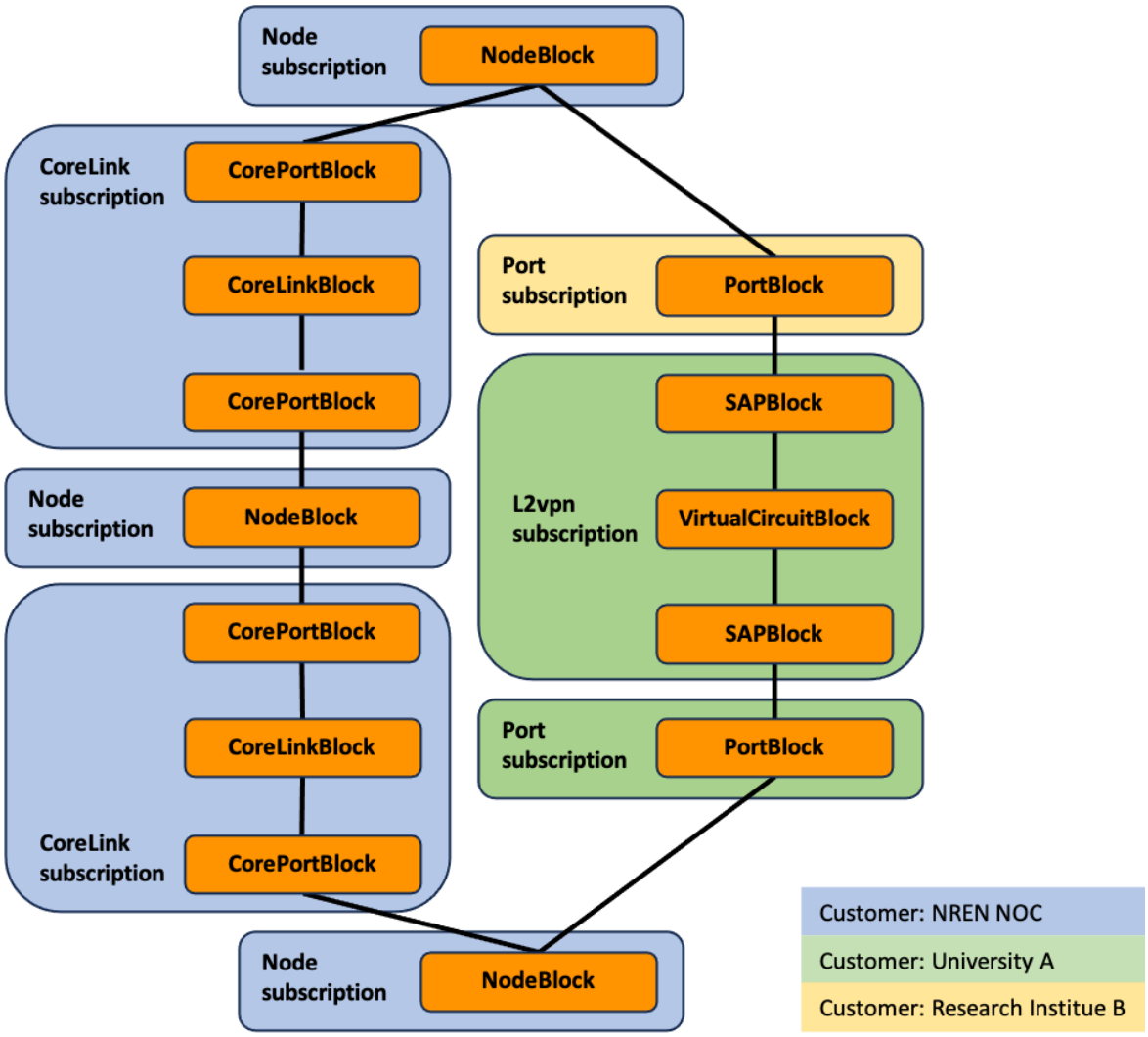

Figure 3: Product block graph

2.4 How to Use

Human workflows for the delivery of products to customers are often comprehensive. To limit the scope of this example orchestrator but still demonstrate the BCP when automating procedures, only inventory management and provisioning are modelled. The implemented products and workflows are designed with particular procedures in mind. For example, it is assumed that the following are administered in IMS outside of the orchestrator:

- Sites
- Device roles
- Device Manufactures
- Device Types
- IPv4 and IPv6 prefix for node loopback addresses
- IPv4 and IPv6 prefix for core link addressing

The *Bootstrap Netbox* task takes care of initialising NetBox with a default set of this information. For convenience, a *Wipe Netbox* task is added as well that will remove all objects from NetBox again, including those created by the different workflows. Task can be found in the orchestrator GUI under **`Tasks->New Task`**.

2.4.1 Node

The Node create workflow will read all configured, sites and device roles, and manufactures types, and allow the user to choose appropriate values using dropdowns. The only things that needs to be entered by hand are a unique name for the node and an optional description. This is enough to create a node subscription and administer the node in the IMS.

Network interfaces are installed in the nodes by field engineers. The Node product has an "Update node interfaces" workflow that will discover all interfaces on a physical node and will add or remove interfaces from the IMS as needed. For this implementation, this workflow will always return a preconfigured list of 10 and 100 Gbit/s network interfaces. In a real world implementation, this could have been fetched from the network device with SNMP, NETCONF, gNMI, or similar protocol. Only basic information on the interfaces is added, which make them available to be used by the *create* workflows of the core link and customer port products.

There are variants of the node product that allow the creation of nodes for different manufactures, and only the matching device types will be shown in the dropdown.

2.4.2 CoreLink

To build a core link, at least two node subscriptions should already exist. This is to satisfy the constraint that the A and B side of the core link need to be different. On each node, there should be at least one port available that matches the requested core link speed.

2.4.3 Port

To create a customer port, at least one node should exist with at least one free interface of the requested port speed. The type of port can be untagged, tagged, or a link member, but note that currently only one network service product is implemented and that product only supports tagged ports.

2.4.4 L2VPN

To create a L2VPN services for a customer, at least two customer ports should exist, and every port can only be used once in the same L2VPN. This product is only supported on tagged interfaces, and VLAN retagging is not supported.

3 Products

Products are described in Domain Models that are designed to help the developer manage complex subscription models and interact with the objects in a user-friendly way. Domain models use [Pydantic] with some additional functionality to dynamically cast variables from the database, where they are stored as a string, to their correct type in Python at runtime. Pydantic uses Python type hints to validate that the correct type is assigned. The use of typing, when used together with type checkers, already helps to make the code more robust, and now everything assigned to the model is also checked at runtime which greatly improves reliability.

The definition of a product is divided into describing the product type and the product blocks. The product type describes the fixed inputs and the top-level product blocks. The fixed inputs are used to differentiate between variants of the same product, for example the speed of a network port. There is always at least one top level product block that contains the resource types to administer the customer facing input. Besides resource types, the product blocks usually also contain links to other product blocks. If a fixed input needs a custom type, then it is defined here together with fixed input definition.

3.1 Product Types

The product types in the code are upper camel cased, like all other type definitions. Per default, the product type is declared for the inactive, provisioning and active lifecycle states, and the product type name is suffixed with the state if the lifecycle is not active. Usually, the lifecycle state starts with *inactive*, and then transitions through provisioning to *active*, and finally to *terminated*. During its life, the subscription, an instantiation of a product for a particular customer, can transition from *active* to *provisioning* and back again many times, before it ends up terminated. The terminated state does not have its own type definition.

```
class PortInactive(SubscriptionModel, is_base=True):
    speed: PortSpeed
    port: PortBlockInactive


class PortProvisioning(PortInactive,
lifecycle=[SubscriptionLifecycle.PROVISIONING]):
    speed: PortSpeed
    port: PortBlockProvisioning


class Port(PortProvisioning, lifecycle=[SubscriptionLifecycle.ACTIVE]):
    speed: PortSpeed
    port: PortBlock
```

Figure 4: Product domain model

As can be seen in the above example, the inactive product type definition is subclassed from SubscriptionModel, and the following definitions are subclassed from the previous one. This product has one fixed input called speed and one port product block (see below about naming). Note that the port product block matches the lifecycle of the product, for example, the PortInactive product has a PortBlockInactive product block, but it is totally fine to use product blocks from different lifecycle states if that suits your use case.

Because a port is only available in a limited number of speeds, a separate type is declared with the allowed values, see below.

```
from enum import IntEnum

class PortSpeed(IntEnum):
    _1000 = 1000
    _10000 = 10000
    _40000 = 40000
    _100000 = 100000
    _400000 = 400000
```

Figure 5: Type definition

This type is not only used to ensure that the speed fixed input can only take these values but also in user input forms to limit the choices, and in the database migration to register the speed variant of this product.

Products need to be registered in two places. All product variants have to be added to the **SUBSCRIPTION_MODEL_REGISTRY**, in **products/__init__.py**, as shown below.

```
from orchestrator.domain import SUBSCRIPTION_MODEL_REGISTRY
from products.product_types.core_link import CoreLink

SUBSCRIPTION_MODEL_REGISTRY.update(
    {
        "core link 10G": CoreLink,
        "core link 100G": CoreLink,
    }
)
```

Figure 6: Subscription model registry

All variants also have to be entered into the database using a migration. The migration uses the *create helper* function from **orchestrator.migrations.helpers** that takes the following dictionary as an argument, as shown below. Note that the name of the product and the product type need to match with the subscription model registry.

```
from orchestrator.migrations.helpers import create

new_products = {
    "products": {
        "core link 10G": {
            "product_id": uuid4(),
            "product_type": "CoreLink",
            "description": "Core link",
            "tag": "CORE_LINK",
            "status": "active",
            "product_blocks": [
                "CoreLink",
                "CorePort",
            ],
            "fixed_inputs": {
                "speed": CoreLinkSpeed._10000.value,
            },
        },

def upgrade() -> None:
    conn = op.get_bind()
    create(conn, new_products)
```

Figure 7: Product migration

3.2 Product Blocks

Like product types, the product blocks are declared for the inactive, provisioning and active lifecycle states. The names of product blocks are suffixed with the word Block, to clearly distinguish them from the product types, and again suffixed by the state if the lifecycle is not active.

Every time a subscription is transitioned from one lifecycle to another, an automatic check is performed to ensure that resource types that are not optional are in fact present on that instantiation of the product block. This safeguards for incomplete administration for that lifecycle state. The resource types on an inactive product block are usually all optional to allow the creation of an empty product block instance. All resource types that are used to hold the user input for the subscription is stored using resource types that are no longer optional in the provisioning lifecycle state. All resource types used to store information that is generated while provisioning the subscription is stored using resource types that are optional while provisioning but are no longer optional for the active lifecycle state. Resource types that are still optional in the active state are used to store non-mandatory information.

```
class NodeBlockInactive(ProductBlockModel, product_block_name="Node"):
    type_id: int | None = None
    node_name: str | None = None
    ims_id: int | None = None
    nrm_id: int | None = None
    node_description: str | None = None


class NodeBlockProvisioning(
    NodeBlockInactive, lifecycle=[SubscriptionLifecycle.PROVISIONING]
):
    type_id: int
    node_name: str
    ims_id: int | None = None
    nrm_id: int | None = None
    node_description: str | None = None


class NodeBlock(NodeBlockProvisioning,
lifecycle=[SubscriptionLifecycle.ACTIVE]):
    type_id: int
    node_name: str
    ims_id: int
    nrm_id: int
```

Figure 8: Product block domain model

In the simplified node product block shown above, the type and the name of the node are supplied by the user and stored on the NodeBlockInactive. Then, the subscription transitions to Provisioning and a check is performed to ensure that both pieces of information are present on the product block. During the provisioning phase the node is administered in IMS and the handle to that information is stored on the NodeBlockProvsioning. Next, the node is provisioned in the NRM and the handle is also stored. If both of these two actions were successful, the subscription is transitioned to *Active* and it is checked that the type and node name, and the IMS and NRM ID, are present on the product block. The description of the node remains optional, even in the active state. These checks ensure that information that is necessary for a particular state is present so that the actions that are performed in that state do not fail.

Sometimes there are resource types that depend on information stored on other product blocks, even on linked product blocks that do not belong to the same subscription. This kind of types need to be calculated at run time so that they include the most recent information. Consider the following example of a (stripped down version) of a port and node product block, and a title for the port block that is generated dynamically.

A *@serializable_property* has been added that will dynamically render the title of the port product block. Even after a *modify* workflow has been run to change the node name on the node subscription, the title of the port block will always be up to date. The title can be referenced as any other resource type using *subscription.port.title*. This is not a random example, the title of a product block is used by the orchestrator GUI while displaying detailed subscription information.

```python
class NodeBlock(NodeBlockProvisioning, lifecycle=[SubscriptionLifecycle.ACTIVE]):
    node_name: str


class PortBlock(PortBlockProvisioning, lifecycle=[SubscriptionLifecycle.ACTIVE]):
    port_name: str
    node: NodeBlock

    @serializable_property
    def title(self) -> str:
        return f"{self.port_name} on {self.node.node_name}"


class Port(PortProvisioning, lifecycle=[SubscriptionLifecycle.ACTIVE]):
    port: PortBlock
```

Figure 9: Serialisable property on domain model

4 Workflows

Four types of workflows are defined, including three that are lifecycle related to create, modify and terminate subscriptions, and fourth to validate subscriptions against the OSS and BSS. The decorators `@create_workflow`, `@modify_workflow`, `@terminate_workflow`, and `@validate_workflow` are used to define the different types of workflow, and the `@step` decorator is used to define workflow steps that can be used in any type of workflow.

Information between workflow steps is passed using `State`, which is nothing more than a collection of key/value pairs, represented in Python by a `Dict`, with string keys and arbitrary values. Between steps the `State` is serialised to JSON and stored in the database. The step decorator is used to turn a function into a workflow step, all arguments to the step function will automatically be initialised with the value from the matching key in the `State`. In turn the step function will return a `Dict` of new and/or modified key/value pairs that will be merged into the `State` to be consumed by the next step. The serialisation and deserialisation between JSON and the indicated Python types is done automatically, which is why it is important to correctly type the step function parameters.

The input form is where a user can enter the details for a subscription on a certain product at the start of the workflow, or can enter additional information during the workflow. The input forms are dynamically generated in the backend and use Pydantic to define the type of the input fields. This also allows for the definition of input validations. Input forms are (optionally) used by all types of workflows to gather and validate user input. It is possible to have more than one input form, with the ability to navigate back and forth between the forms, until the last input form is submitted, and the first (or next) step of the workflow is started. This allows for on-the-fly generation of input forms, where the content of the next form(s) depends on the input of the previous form(s). For example, when creating a core link between two nodes, a first input form could ask to choose two nodes from a list of active nodes, and the second form will present two lists with ports on these two nodes to choose from.

While developing a new product, the workflows can be written in any order. Those who use a test-driven development style will probably start with the *validate* workflow but in general the *create* workflow will usually be used to start with as it is helpful to discuss the product model (the information involved) and the workflows (the procedures involved) with stakeholders to clarify requirements. Once the minimal viable *create* workflow is implemented, the *validate* workflow can be written to ensure that all information is administered correctly in all touched OSS and BSS and is not changed again by hand because human workflows were not yet correctly adapted. Then, after the *terminate* workflow is written, the complete lifecycle of the product can be tested. Even when the *modify* workflow is not implemented, a change to a subscription can be carried out by terminating the subscription and creating it again with the modified input. Finally, the *modify* workflow is implemented to allow changes to a subscription with minimal or no impact to the customer.

4.1 Create Workflow

A *create* workflow needs an initial input form generator and defines the steps to create a subscription on a product. The **@create_workflow** decorator adds some additional steps to the workflow that are always part of a *create* workflow. The steps of a *create* workflow in general follow the same pattern, as described below using the *create node* workflow as an example.

```
@create_workflow("Create node", initial_input_form=initial_input_form_generator)
def create_node() -> StepList:
    return (
        begin
        >> construct_node_model
        >> store_process_subscription(Target.CREATE)
        >> create_node_in_ims
        >> reserve_loopback_addresses
        >> provision_node_in_nrm
    )
```

Figure 10: Create workflow

1. Collect input from user (**initial_input_form**)
2. Instantiate subscription (**construct_node_model**):
   a. Create inactive subscription model
   b. Assign user input to subscription
   c. Transition to subscription to provisioning
3. Register *create* process for this subscription (**store_process_subscription**)
4. Interact with OSS and/or BSS, in this example:
   a. Administer subscription in IMS (**create_node_in ims**)
   b. Reserve IP addresses in IPAM (**reserve_loopback_addresses**)
   c. Provision subscription in the network (**provision_node_in_nrm**)
5. Transition subscription to active and 'in sync' (**@create_workflow**)

As long as every step remains as idempotent as possible, the work can be divided over fewer or more steps as desired.

The input form is created by subclassing the FormPage and adding the input fields together with the type and indication if they are optional or not. Additional form settings can be changed via the Config class, such as for example the title of the form page.

```
class CreateNodeForm(FormPage):
    class Config:
        title = product_name

    role_id: node_role_selector(node_type)
    node_name: str
    node_description: str | None
```

Figure 11: Create workflow input form

By default, Pydantic validates the input against the specified type and will signal missing input fields. But custom validations can also be added, like a check on the validity of the entered VLAN ID as shown below.

```
@validator("vlan", allow_reuse=True)
def valid_vlan(cls, v: int):
    if v < 2 or v > 4094:
        raise AssertionError("VLAN ID must be between 2 and 4094 (inclusive)")
    return v
```

Figure 12: Input validator

The node role is defined as type Choice and will be rendered as a dropdown that is filled with a mapping between the role IDs and names as defined in NetBox.

```
def node_role_selector() -> Choice:
    roles = {str(role.id): role.name for role in netbox.get_device_roles()}
    return Choice("RolesEnum", zip(roles.keys(), roles.items()))
```

Figure 13: Choice definition

When more than one item needs to be selected, a choice_list() can be used to specify the constraints, for example to select multiple ports for a L2VPN:

```
choice = Choice("PortsEnum", zip(ports.keys(), ports.items()))
return choice_list(choice, min_items=2, max_items=8, unique_items=True)
```

Figure 15: Choice list definition

Finally, a summary form is shown with the user supplied values. When a value appears to be incorrect, the user can go back to the previous form to correct the mistake, otherwise, when the form is submitted, the workflow is kicked off.

```
summary_fields = ["role_id", "node_name", "node_description"]
yield from create_summary_form(user_input_dict, product_name, summary_fields)
```

Figure 14: Create workflow summary form

4.2 Modify Workflow

A *modify* workflow also follows a general pattern, as described below. The **@modify_workflow** decorator adds

```
@modify_workflow("Modify node", initial_input_form=initial_input_form_generator)
def modify_node() -> StepList:
    return (
        begin
        >> set_status(SubscriptionLifecycle.PROVISIONING)
        >> update_subscription
        >> update_node_in_ims
        >> update_node_in_nrm
        >> set_status(SubscriptionLifecycle.ACTIVE)
    )
```

Figure 16: Modify workflow

some additional steps to the workflow that are always needed.

1. Collect input from user (**initial_input_form**)
2. Necessary subscription administration (**@modify_workflow**):
   a. Register modify process for this subscription
   b. Set subscription 'out of sync' to prevent the start of other processes
3. Transition subscription to Provisioning (**set_status**)
4. Update subscription with the user input
5. Interact with OSS and/or BSS, in this example
   a. Update subscription in IMS (**update_node_in_ims**)
   b. Update subscription in NRM (**update_node_in_nrm**)
6. Transition subscription to active (**set_status**)
7. Set subscription 'in sync' (**@modify_workflow**)

Like the *create* workflow, the *modify* workflow also uses an initial input form but in this case only to collect the values from the user that need to be changed. Usually, only a subset of the values may be changed. To assist the user, additional values can be shown in the input form using read-only fields. In the example below, the name of the node is shown but cannot be changed, the node status can be changed and the dropdown is set to the current node status, and the node description is still optional.

```
class ModifyNodeForm(FormPage):
    node_name: str = ReadOnlyField(node.node_name)
    node_status: node_status_selector() = node.node_status
    node_description: str | None = node.node_description
```

Figure 18: Modify workflow summary form

After a summary form has been shown that lists the current and the new values, the *modify* workflow is started.

```
summary_fields = ["node_status", "node_name", "node_description"]
yield from modify_summary_form(user_input_dict, subscription.node, summary_fields)
```

4.3 Terminate Workflow

At the end of the subscription lifecycle, the *terminate* workflow updates all OSS and BSS accordingly, and the **@terminate_workflow** decorator takes care of most of the necessary subscription administration.

```
@terminate_workflow("Terminate node", initial_input_form=initial_input_form_generator)
def terminate_node() -> StepList:
    return (
        begin
        >> load_initial_state
        >> delete_node_from_ims
        >> deprovision_node_in_nrm
```

Figure 19: Terminate workflow

1. Show subscription details and ask user to confirm termination (**initial_input_form**)
2. Necessary subscription administration (**@terminate_workflow**):
   a. Register terminate process for this subscription
   b. Set subscription 'out of sync' to prevent the start of other processes
3. Get subscription and add information for following steps to the State (**load_initial_state**)
4. Interact with OSS and/or BSS, in this example
   a. Delete node in IMS (**delete_node_in ims**)
   b. Deprovision node in NRM (**deprovision_node_in_nrm**)
5. Necessary subscription administration (**@terminate_workflow**)
   a. Transition subscription to terminated
   b. Set subscription 'in sync'

The initial input form for the terminate workflow is very simple, it only has to show the details of the subscription:

```
class TerminateForm(FormPage):
    subscription_id: DisplaySubscription = subscription_id
```

Figure 20: Terminate workflow input form

4.4 Validate Workflows

And finally, the *validate* workflow, used to check if the information in all OSS and BSS is still the same with the information in the subscription. One way to do this is to reconstruct the payload sent to the external system using information queried from that system, and compare this with the payload that would have been sent by generating a payload based on the current state of the subscription. The **@validate_workflow** decorator takes care of necessary subscription administration. There is no initial input form for this type of workflow.

```
@validate_workflow("Validate l2vpn")
def validate_l2vpn() -> StepList:
    return (
        begin
        >> validate_l2vpn_in_ims
        >> validate_l2vpn_terminations_in_ims
        >> validate_vlans_on_ports_in_ims
    )
```

Figure 21: Validate workflow

1. Necessary subscription administration (**@validate_workflow**):
   a. Register validate process for this subscription
   b. Set subscription 'out of sync', even when subscription is already out of sync
2. One or more steps to validate the subscription against all OSS and BSS:
   a. Validate subscription against IMS:
      i. **validate_l2vpn_in_ims**
      ii. **validate_l2vpn_terminations_in_ims**
      iii. **validate_vlans_on_ports_in_ims**
3. Set subscription 'in sync' again (**@validate_workflow**)

When one of the validation steps fails, the subscription will stay 'out of sync', prohibiting other workflows from being started for this subscription. The failed validation step can be retried as many times as needed until it succeeds, which finally will set the subscription 'in sync' and allow other workflows to be started again. This safeguards workflows from being started for subscription with mismatching information in OSS and BSS which would make them likely to fail.

It is better to limit the number of validations done in each step. This will make it easier to see any discrepancies found at a glance and will make a retry of any failed steps much faster. A commonly used strategy is to use separate steps for each OSS and BSS, and separate steps per external system for each payload that was sent. This can be done by comparing a payload created for a product block in the orchestrator with a payload that is generated by querying the external system.

As well as performing validations per subscription, is also possible to validate other requirements. For example, to make sure that there are no L2VPNs administered in IMS that do not have a matching subscription in the orchestrator, a task (a workflow with **Target.SYSTEM**) can be written that will retrieve a list of all L2VPNs from IMS and compare it against a list of all L2VPN subscriptions from the orchestrator.

5 Services

Services are collections of helper functions that deliver a service to other parts of the code base. The common programming pattern of function overloading is used for the implementation of the service. Function overloading allows the use of multiple functions with the same name that will execute the right function based on the type of the parameters. Python does not allow function overloading, but similar functionality can be achieved through the use of the single dispatch feature that is part of the standard Python library.

First, an interface is defined and decorated with **`@singledispatch`**. Then different nameless functions can be registered that implement that interface but for different parameters. Note that only the first parameter will be taken into account to decide which one of the functions needs to be executed.

A helper function called **`single_dispatch_base()`** is used to keep track of all registered functions and the type of their first argument. This allows for more informative error messages when the single dispatch function is called with an unsupported parameter.

5.1 Subscription Descriptions

An example of a service is the generation of descriptions for subscriptions or product block instances, where the description is generated based on the type of subscription or product block instance. In this way, there is one place where every description is being generated, and changes to the way a description is generated will automatically appear in all places where that description is being used.

The *single dispatch* description enables a first argument of type product model, product block model, or subscription model, and will call the matching function.

```
@singledispatch
def description(
    model: Union[ProductModel, ProductBlockModel, SubscriptionModel]
) -> str:
    return single_dispatch_base(description, model)
```

Figure 22: Single dispatch description

Implementations of the description function can then be registered, such as the generation of a description for a Node product, starting from the provisioning lifecycle state, that will show the name of the node followed by the status of the node in parenthesis.

```
@description.register
def _(product: NodeProvisioning) -> str:
    return f"node {product.node.node_name} ({product.node.node_status})"
```

Figure 23: Single dispatch description register

5.2 NetBox

The NetBox service is an interplay between several single dispatch functions, one to generate the payload for a specific product block, and two others that create or modify an object in NetBox based on the type of payload. The [Pynetbox] Python API client library is used to interface with NetBox.

5.2.1 Payload

The **build_payload()** single dispatch enables a first argument of type product block model, and a subscription model parameter that is used when related information is needed from other parts of the subscription. The specified return type is the base class that is used for all Netbox payload definitions.

```
@singledispatch
def build_payload(
        model: ProductBlockModel, subscription: SubscriptionModel, **kwargs: Any
) -> netbox.NetboxPayload:
    return single_dispatch_base(build_payload, model)
```

Figure 24: Single dispatch Netbox payload

When the payload is generated from a product block, the correct mapping is made between the types used in the orchestrator and the types used in the OSS or BSS, for example, the Port product block maps on the Interface type in NetBox, as can be seen below.

```
@build_payload.register
def _(
    model: PortBlockProvisioning, subscription: SubscriptionModel
) -> netbox.InterfacePayload:
    return build_port_payload(model, subscription)

def build_port_payload(
    model: PortBlockProvisioning, subscription: SubscriptionModel
) -> netbox.InterfacePayload:
    return netbox.InterfacePayload(
        device=model.node.ims_id,
        name=model.port_name,
        type=model.port_type,
        tagged_vlans=model.vlan_ims_ids,
        mode="tagged" if model.port_mode == PortMode.TAGGED else "",
        description=model.port_description,
        enabled=model.enabled,
        speed=subscription.speed * 1000,
    )
```

Figure 25: Single dispatch Netbox payload register

The values from the product block are copied to the appropriate place in the Interface payload. The interface payload field names match those that are expected by NetBox. The speed of the interface is taken from the fixed input speed with the same name on the subscription, and multiplication by 1000 is used to convert between Mbit/s and Kbit/s.

5.2.2 Create

To create an object in NetBox based on the type of Netbox payload, the single dispatch **`create()`** is used:

```
@singledispatch
def create(payload: NetboxPayload, **kwargs: Any) -> int:
    return single_dispatch_base(create, payload)
```

Figure 26: Single dispatch Netbox create

When registering the payload type, a keyword argument is used to inject the matching endpoint on the NetBox API that is used to create the desired object. In the example below, it can be seen that *interface payload* is to be used against the **`api.dcim.interfaces`** endpoint.

```
@create.register
def _(payload: InterfacePayload, **kwargs: Any) -> int:
    return _create_object(payload, endpoint=api.dcim.interfaces)
```

Figure 27: Single dispatch Netbox create register

Finally, the payload is used to generate a dictionary as expected by the NetBox API endpoint. Note that the names of the fields of the Netbox payload must match the names of the fields that are expected by the NetBox API.

```
def _create_object(payload: NetboxPayload, endpoint: Endpoint) -> int:
    object = endpoint.create(payload.dict())
    return object.id
```

Figure 28: Netbox service create object

The ID of the object that is created in NetBox is returned so that it can be registered in the subscription for later reference, e.g. when the object needs to be modified or deleted.

5.2.3 Update

The single dispatch **`update()`** is defined in a similar way, the only difference being that an additional argument is used to specify the ID of the object that needs to be updated in NetBox.

```
@update.register
def _(payload: InterfacePayload, id: int, **kwargs: Any) -> bool:
    return _update_object(payload, id, endpoint=api.dcim.interfaces)
```

Figure 29: Single dispatch Netbox update

The ID is used to fetch the object from the NetBox API, update the object with the dictionary created from the supplied payload, and send the update to NetBox.

```
def _update_object(payload: NetboxPayload, id: int, endpoint: Endpoint) -> bool:
    object = endpoint.get(id)
    object.update(payload.dict())
    return object.save()
```

Figure 30: Netbox service update object

5.2.4 Get

The NetBox service also defines other helpers, for example, to get an single object, or a list of objects, of a specific type from NetBox.

```
def get_interfaces(**kwargs) -> List:
    return api.dcim.interfaces.filter(**kwargs)


def get_interface(**kwargs):
    return api.dcim.interfaces.get(**kwargs)
```

Figure 31: Netbox service get object(s)

Both types of helpers accept keyword arguments that can be used to specify the object(s) that are wanted. For example **`get_inteface(id=3)`** will fetch the single interface object with ID equal to 3 from NetBox. And **`get_interfaces(speed=1000000)`** will get a list of all interface objects from NetBox that have a speed of 1Gbit/s.

5.2.5 Delete

Another set of helpers is defined to delete objects from NetBox. For example, to delete an Interface object from NetBox, see below:

```
def delete_interface(**kwargs) -> None:
    delete_from_netbox(api.dcim.interfaces, **kwargs)
```

Figure 32: Netbox service delete

The keyword arguments allow for different ways of selecting the object to be deleted, as long as the supplied arguments result in a single object.

```
def delete_from_netbox(endpoint, **kwargs) -> None:
    object = endpoint.get(**kwargs):
    object.delete()
```

Figure 33: Netbox service delete object

5.2.6 Product Block to NetBox Object Mapping

The modelling used in the orchestrator does not necessarily have to match the modelling in your OSS or BSS exactly. In many cases, different names are used, or a one-to-many or many-to-one relation needs to be created. To facilitate future transition to a different external system, any needed mappings, or translations between the models, are isolated as much as possible in the workflow step(s) that deal with those external systems.

The diagram below shows the product blocks and relations as used in a core link between two nodes, and how they map to the objects as administered in NetBox. The product blocks are shown in orange and the NetBox objects in green.

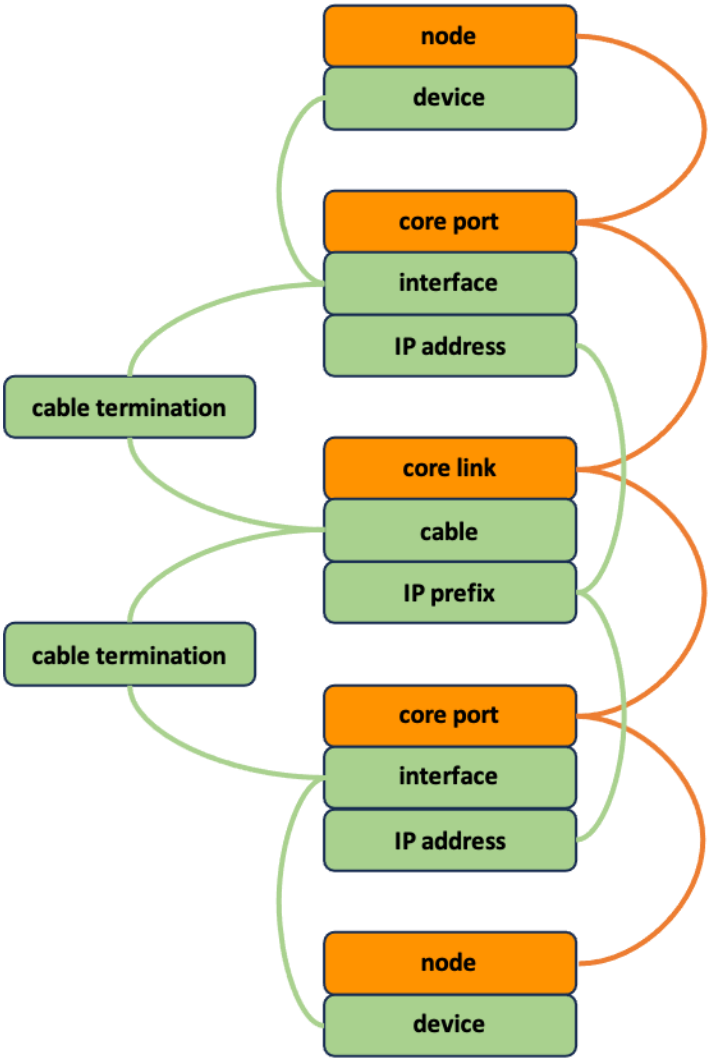

Figure 34: Node and core link type mapping

Finally, the next diagram below shows the mapping and relation between product blocks and NetBox objects for a L2VPN on customer ports between two nodes.

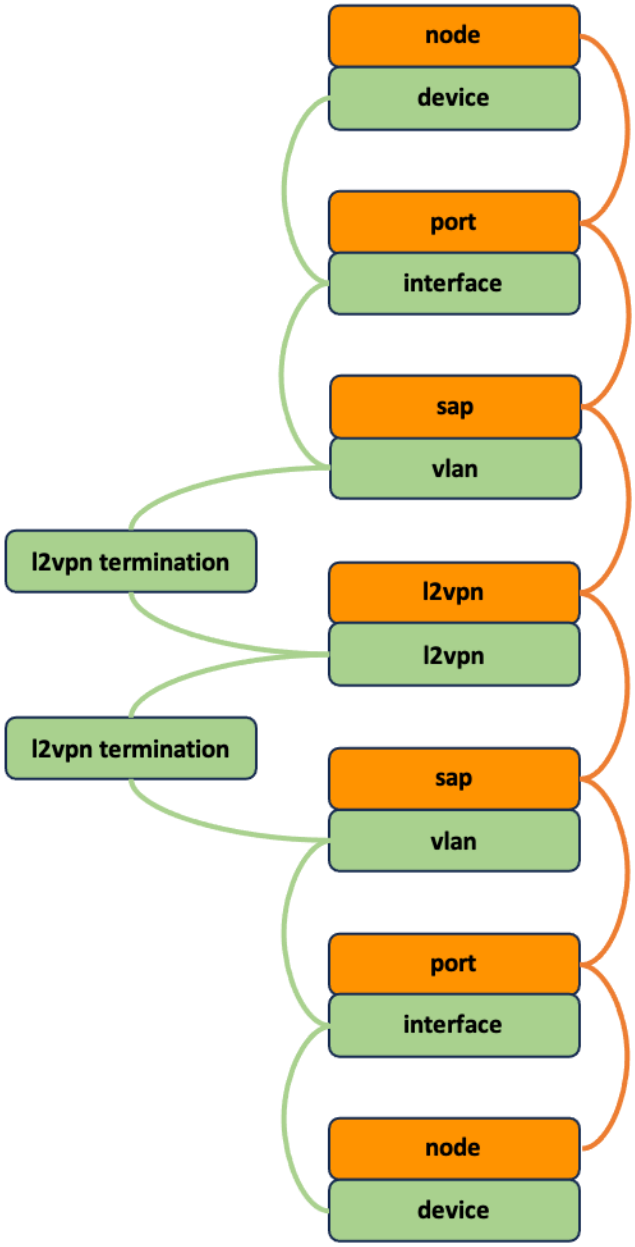

Figure 35: Node, port and L2VPN type mapping

Glossary

API	Application Programming Interface
ASGI	Asynchronous Server Gateway Interface
BCP	Best Common Practice
BSS	Business Support System
gNMI	gRPC Network Management Interface
gRPC	generic Remote Procedure Call
GUI	Graphical User Interface
IMS	Inventory Management System
IPAM	IP Address Management
L2VPN	Layer 2 Virtual Private Network
L3VPN	Layer 3 Virtual Private Network
NETCONF	NETwork CONFiguration protocal
NREN	National Research and Education Network
OSS	Operation Support System
REST	REpresentational state transfer
SAP	Service Access Point
SNMP	Simple Network Management Protocol
WFO	WorkFlow Orchestrator

References

[GN5-1_M7.3]	*Common NREN Network Service Product Models* https://resources.geant.org/wp-content/uploads/2023/06/M7.3_Common-NREN-Network-Service-Product-Models.pdf
[WFO]	https://workfloworchestrator.org/orchestrator-core/
[NetBox]	A tool for data centre infrastructure and IP address management https://netbox.dev
[SQAlchemy]	The Python SQL Toolkit and Object Relational Mapper https://www.sqlalchemy.org
[Uvicorn]	ASGI server https://www.uvicorn.org
[Pydantic]	Pydantic is a data validation library for Python https://pydantic.dev/
[Pynetbox]	Pynetbox Python API https://github.com/netbox-community/pynetbox