Make Model Driven Network Automation Pythonic

Term Project

Department of Computer Science

OST – University of Applied Sciences
Campus Rapperswil-Jona

Semester: Autumn 2022

Version: 1.0
Date: 2022-12-23 15:27:10Z
Git Version: 5b29ffc

Project Team: Dejan Jovicic
Dominic Walther
Project Advisor: Urs Baumann

School of Computer Science
OST Eastern Switzerland University of Applied Sciences

Acknowledgements

We would like to take this opportunity to thank all those who have supported us in this study.

We would especially like to thank our supervisor Urs Baumann. At the weekly meetings he always took enough time to answer our questions, and through his experience and expertise he was able to provide us with input that helped us to understand and solve problems. In particular, we would like to thank him for always being available quickly and without complications throughout the entire work, even outside the agreed meetings.

Abstract

YANG is a data modelling language used to define data structures transmitted over either the NETCONF or RESTCONF protocol. Such models can be used to perform so-called *Model Driven Network Automation.*

The goal of this project is to create a proof-of-concept to show how YANG models could be translated to Python data-structures based on pydantic. These data-structures can in turn be initialized with configuration values, serialized into a RESTCONF payload and sent to a network device, applying the configuration. If successful, this would facilitate configuring network devices through Python code, without requiring the user to have prior knowledge of YANG.

We started by analysing the Python ecosystem surrounding YANG, including projects such as Pyang, PyangBind, yangson and Pyang-Pydantic, as well as pydantic and datamodel-code-generator. Our analysis revealed that most of the projects in the ecosystem have either been abandoned for years or are nowhere near robust or reliable enough to be used in a productive environment.

After some consideration, we settled on Pyang as our YANG parser and used the Pyang-Pydantic plugin as a starting point for our project. We then proceeded to gradually add features designed to make the generated Python models as intuitive and convenient to work with as possible.

This resulted in the creation of Pydantify, a tool for translating YANG models into executable Python code which can in turn generate valid RESTCONF payloads. On top of validating the concept as feasible, we successfully managed to implement a large section of the YANG specification, enabling some real-world models to be converted without issue.

This paper documents the development of Pydantify and outlining the challenges we faced along the way, providing a stepping stone for future projects in this domain.

Lay Summary

Initial Situation

The common goal in the IT industry is to automate every task as much as possible, and in the context of networking, one of the key components is YANG. YANG is used to describe data structures containing the configuration of a network device in a process called *Model Driven Network Automation*. Such data structures can be transmitted to network devices, typically in either XML or JSON payloads, to modify the device's configuration remotely. Writing such payloads by hand and without errors is tedious, so tools are employed to simplify the process. There are however very few Python-based tools for this purpose.

Approach

Our goal was to create a proof-of-concept to demonstrate that YANG models could be translated to Python data structures which in turn can generate RESTCONF payloads when initialized with configuration values by the user. These payloads can then be sent to a network device where the configuration changes get applied. Such a tool would help network engineers with little to no YANG experience create configurations in a Python environment and with the aid of type-hints provided by their IDE of choice.

Results

By creating Pydantify, we were able to demonstrate the feasibility of the concept and how *Model Driven Network Automation* based on YANG models could be improved in a Pythonic way.

Further Work

As Pydantify is only a proof-of-concept, the functionality does not cover all aspects of YANG. Further improvements to the capability and usability of Pydantify can certainly be made, however many of its limitations lie with its dependencies, requiring work on external projects to fully resolve.

Table of Contents

Listings

List of Figures

List of Tables

Glossary

container

contains a group of related nodes. 11, 25

Flit

is a simple way to put Python packages and modules on PyPi. 35

Gantt

is a type of bar chart that illustrates a project schedule. 38

intramodel

is someting that is within a model. 26

Jinja2

is a full-featured template engine for Python. 31

JSONSchema

is a declarative language that allows you to annotate and validate JSON documents. 12, 28

kanban board

is a scheduling system for lean manufacturing. 38

leaf

represents a single value in YANG. 4

leaf-list

contains a sequence of leaf nodes. 5

library

is a collection of prewritten code that user can use to optimize tasks. 9, 24, 35, 45

netconf

The Network Configuration Protocol, defines data retrieval, upload, manipulation and deletion of configuration data on network devices. 2, 3, 6

payload

is the data to be transmitted in networks. iii, 8, 29

PDM

is a Python package and dependency manager. 35

Poetry

is a dependency and packaging manager. 35

Pyang

is a YANG validator, transformator and code generator, written in Python. 8–10, 13, 19, 24, 28, 35

Pyang-Pydantic

is a Pyang plugin that generates a pydantic class out of YANG modules, supporting only basic container types and has no input validation nor default values. 10, 43

PyangBind

is a Pyang plugin, which converts a YAML schema to valid Python code that can be edited, instantiated with data and sent to a switch. ii, 10

pydantic

is a Python library for data validation. 2, 11, 18, 25, 26, 32, 35, 45

Pytest

makes it easy to write small, readable tests, and can scale to support complex functional testing for applications and libraries. 32

restconf

is a HTTP based protocol for configurig YANG data using datastore concepts from NETCONF. 2, 6, 20, 24, 26, 28, 35

SNMP

Simple Network Management Protocol used to collect, organize and modify information from network devices. 3

typedef

is a statement used to define types derived from a base type. 20, 26

usability test

is a test used to evaluate a product by testing it on users. 22, 46

yangson

is a Python 3 library offering programmers tools for working with configuration and other data modelled with YANG. ii, 9

Acronyms

CI/CD

Continous Integration / Continous Deployment. 22, 46

CLI

Command-Line Interface. 9, 24, 31

CRUD

Create, Read, Update and Delete. 6

DMCG

Datamodel Code Generator. 12, 25, 28, 31, 32, 35

IDE

Integrated Development Environment. iii, 14, 19, 27

IETF

Internet Engineering Task Force. 2, 3

INS

Institute for Networked Solutions. 22, 46

JSON

JavaScript Object Notation. 2, 20, 24

RPC

Remote Procedure Call. 6

XML

Extensible Markup Language. 2, 5

XPath

XML Path Language. 7

yang

Yet Another Next Generation. 2, 3, 18, 24, 25, 28, 35, 39

Part I

Technical Report

Chapter 1

Vision

1.1 Problem Statement

The introduction of the NETCONF and RESTCONF protocols led to the creation of a data modeling language by the IETF to define a standard for data models, which would be used by both protocols to get, update, and push configurations from or to network devices. This led to the publication of YANG in 2010. The data modeling language itself has similar structures to XML or JSON and contains nested statements and references to other files, making it difficult to understand and use. The configuration data needs to be in XML or JSON format, depending on the protocol being used. A single YANG file can contain over a thousand lines of code, and the associated data is usually just as vast, often making it impractical to work with the entire model at once. Therefore, the typical workflow of a network engineer is to narrow down the chunk of data they work on by using filters in their NETCONF or RESTCONF request. After that, they can modify their chosen section of configuration data before sending it back to the network device using the same filters as before. However, adding to or modifying the structure of the data usually requires consulting the YANG model itself, which is significantly more tedious.

1.2 Our Solution

Our project aims to demonstrate how a YANG model can be translated into a pydantic[1] model as a proof-of-concept. This could be useful for network engineers who want to create RESTCONF payloads from scratch using their Python skills, limited YANG knowledge, and with the added help provided by type hints. The result would be generated models in Python code that can be used to create JSON structures, which can then be used to configure a Cisco router or switch through by means of RESTCONF.

In summary, the primary focus of this thesis is to show that it is possible to make model-driven network automation more Pythonic, which can be further developed in future theses or by the public YANG community.

[1]pydantic is a popular Python library for data validation and type hinting

Chapter 2

Technical Framework

2.1 Overview

YANG (short for "Yet Another Next Generation") is a data modeling language that was designed to improve upon the limitations of SNMP[1] in configuration management. While SNMP is commonly used as a network management system to detect errors on network devices, its disadvantages sparked a need for the creation of a better protocol. In 2006, the IETF published NETCONF, a standardized protocol for automating network configurations. NETCONF allows for the retrieval, upload, manipulation, and deletion of configuration data. However, NETCONF only defines the process of transmitting and modifying data, not the structure of the data itself. This lead to the to the development of YANG. YANG makes it easier to understand data models and is widely used in the networking industry.

YANG is hierarchical, provides high extensibility and can distinguish between configurations and status. In other words, YANG complements NETCONF so that it is possible to define configuration and state data, notifications and Remote Procedure Calls using NETCONF-based operations.

A YANG module defines a single data model, however, it can reference definitions from other modules by using the *`import`* and *`include`* statements. YANG, as defined in RFC6020, has four primary node types[2] as shown in Table 2.1.

Type	**Description**	**Comparable to**
`leaf`	represents a single value	a variable
`leaf-list`	contains a sequence of leaf nodes	an array
`container`	contains a group of related nodes	a class
`list`	contains a sequence of nodes, each uniquely identified by one or more key attributes	a database table

Table 2.1: YANG node types

[1]Simple Network Management Protocol

```
grouping passive-interface-grouping {
  container passive-interface {
    description
      "Suppress routing updates on an interface";
    choice passive-interface-choice {
      leaf default {
        description
          "Suppress routing updates on all interfaces";
        type empty;
      }
      leaf-list interface {
        type string;
      }
    }
  }
  container disable {
    when '../passive-interface/default';
    list passive-interface {
      key "interface";
      leaf interface {
        type string;
      }
    }
  }
}
```

Listing 2.1: YANG Model Interface Common Example

2.1.1 typedef

By using the *typedef* statement, one can define types derived from a base type. This base type can either be a built-in type, such as *string* or *uint8*, or an already derived type. This allows for a basic form of inheritance between types, where the root type is always a built-in type.

For example, a type called *percent* can be defined, which is derived from the base type *uint8* and has its value restricted to between 0 and 100, as seen in Listing 2.2.

```
typedef percent {
    type uint8 {
        range "0 .. 100";
    }
}
```

Listing 2.2: typedef percent

Such a type can then be referenced by other statements, such as leafs. In Listing 2.3, the *completion* leaf can be thought of as a variable of type *percent*[2].

```
leaf completion {
    type percent;
}
```

Listing 2.3: typedef percent usage example

2.1.2 leafref

The *leafref* type is a little bit more complicated to understand than the other types.

The *leafref* type is used to reference other *leaf* instances in the tree via its "path" sub-statement. The "path" statement takes a string as an argument and must refer to an existing leaf or a leaf-list node, dangling references are not allowed.[2]

```
list interface {
  key "name";
  leaf name {
    type string;
  }
  leaf admin-status {
    type admin-status;
  }
  list address {
    key "ip";
    leaf ip {
      type yang:ip-address;
    }
  }
}

leaf mgmt-interface {
  type leafref {
    path "../interface/name";
  }
}
```

Listing 2.4: Simple leafref YANG example

In Listing 2.4, we can see that the *leafref* refers to the path *"../interface/name"*, meaning that the leaf it is referring to can be found by leaving the scope of the current node, navigating into the "interface" list and finally locating the node called "name". A corresponding XML is shown in Listing 2.5.

```
<interface>
    <name>eth0</name>
</interface>
<interface>
    <name>lo</name>
</interface>

<mgmt-interface>eth0</mgmt-interface>
```

Listing 2.5: Simple leafref XML example

2.1.3 RESTCONF

In section 2.1 Overview, we took a short excursion into the history of the creation of NETCONF and YANG, but there is one more component to it - the RESTCONF protocol. RESTCONF is a protocol based on HTTP that provides an interface to access data defined in YANG, while using the data store concepts of NETCONF.

The goal of both YANG, NETCONF and RESTCONF is to facilitate the automation of network configurations. While NETCONF is based on RPCs[2], something many companies would need to re-train their IT-Engineers for in order to use, RESTCONF uses HTTP-based RESTful APIs, which are much more ubiquitous in the industry.

RESTCONF tends to be easier to work with for simple CRUD operations when compared to NETCONF, and its ability to work with both XML and JSON data makes it a bit more flexible to integrate with other software, such as in our case.

All in all, RESTCONF is an easy way for applications to access configuration and state data, data-model-specific RPC operations, and event notifications.

The operations provided by RESTCONF are defined in RFC 8040 and can be summarized thusly[1]:

- **OPTIONS**
  - Is sent to discover which methods (options) are supported by the opposing side for a specified resource
- **GET**
  - Is sent to retrieve data and metadata of a specified resource
- **HEAD**
  - Is sent to retrieve just the header fields from a specified resource
- **POST**
  - Is sent by the client to create a data resource or invoke a Remote Procedure Call
- **PUT**
  - Is sent to create or replace the data of a specified resource
- **PATCH**
  - Is used to provide an extensible framework for resource patching mechanisms and can be used to create or modify a child resource within the specified resource
- **DELETE**
  - Is used to delete the specified resource

[2]Remote Procedure Call

2.1.4 XPath

XML Path Language, or XPath for short, was designed to support the query of XML structures. The simplest form an XPath statement can take would be a complete and internal path, such as `interfaces/ip`, which selects a child node called `ip` located in a node called `interfaces`. There are many, much more complex queries that can be performed through XPath, an introduction to which can be found on the W3Schools website.

In YANG, XPath is used in for referencing other nodes or to specify restrictions on them. Here are a few examples of where XPath finds application in YANG:

Must Statements are used to constraint nodes, in Listing 2.6 we want to make sure that the value of `count` is exactly 10.

```
container interface {
    must "count = 10";
    leaf count {
        type uint8
    }
}
```

Listing 2.6: XPath must statement

If/When Statements are used to make instances of YANG conditional. The `when` statements can change at run-time, whereas `if` statements are set on boot-time. Listing 2.7 describes a `leaf` called "name", which is only present in the data if the value of `percent` is below 50.

```
container test {
    leaf percent {
        type uint8
    }
    leaf name {
        when "../percent < 50";
        type string
    }
}
```

Listing 2.7: XPath when statement

Path Statements, as described in subsection 2.1.2 leafref, are the most common application of XPath within YANG. They are essential for the functioning of leafrefs as they specify which YANG node is being referenced. They can also be used to add additional restrictions on the properties the node to be referenced must have, allowing them to be used as a form of *foreign key constraint.*

XPath in Pydantify

All in all, XPath is a powerful feature, but making use of it is not a trivial affair. For instance the `path` statements can only be checked as the model is being instantiated with data, meaning that the validity of the input can only be tested after Pydantify has already completed its task of generating a pydantic output model.

In addition, queries like `if` and `when` can alter which components are included in the model dynamically based on input values. To fully reproduce this behaviour in the output model would require the output model to be self-modifying, adding a layer of meta-programming which would easily exceed the scope of our project.

It would however be possible to validate the input after the instantiation of the output model by having Pyang check the generated RESTCONF payload against the YANG model. This would allows the the configuration data to be validated fully, without having to implement the validation ourselves. The only downside being that the YANG files have to be preserved alongside the output model in order to provide this functionality.

2.2 Python Ecosystem Overview

2.2.1 yangson

Yangson is a Python 3 library that offers programmers tools for working with configuration and other data modelled with the YANG data modelling language.[8] The documentation places great emphasis on the ability to modify YANG models directly from within Python which, while being a valuable feature, does not directly align with the goals of Pydantify (although it would likely offer what is required). It is still being actively maintained, however it does not appear to be as widely used as other projects in this chapter.

After sampling the project's repository to get a sense of the code quality we rather quickly decided to not use this project as the base for Pydantify. While it is based on Python3.6 and makes good use of type annotations and docstrings, the quality of the code itself does not inspire a lot of confidence. The code is littered throughout with string operations and inconsistent uses of exceptions and return values.

A commented-out unit test with the note "Commented out because it fails."[7] ultimately settled the matter.

2.2.2 Pyang

Pyang describes itself as "a YANG validator, transformator and code generator, written in Python". It is capable of translating YANG modules between various formats including YANG, YIN, DSDL, jsTree, among others. The library is extensible through plugins, most notably PyangBind and Pyang-Pydantic.

Being widely used in the industry while having released version 1.0 over 12 years ago *should* indicate that it is a fairly reliable tool. However, being that old comes at a cost: it maintains backwards compatibility all the way to Python 2.7. It does so by using only the subset of Python instructions valid in both Python 2.7 and 3.6, occasionally using branching code paths where necessary. It therefore comes with no type hints, f-strings, list comprehensions, match-case statements or any number of other features that have improved the legibility of Python code since the Python 2.7 release.

Usage

Pyang is primarily designed to be used as a CLI tool, but it does offer a few additional options: There are other options however:

- **Using Pyang as an intermediate translation step**
  - **pro:** this would expand our tool options. For example YIN (which is effectively XML) can be interpreted by off-the-shelve Python libraries, making it easier for us to parse.
  - **pro:** we would not need to concern ourselves with any of the Pyang internals.
  - **con:** on its own, this would require any YANG to pydantic conversion to be done in multiple steps (Eg. translating YANG to YIN using Pyang, then generating a pydantic model from said YIN output).
- **Using Pyang as a library**

    - **pro:** models could be used directly upon loading, without translating them to another format first. This could eliminate the need for an additional dependency in the chain.
    - **con:** as this is not the primary intended use-case, there is no intended interface for us to use, leading to increased coupling as we would need to access the library's internals directly.
- **Extending Pyang with a plugin of our own**
    - **pro:** it's the intended interface by which to add functionality to Pyang, giving us greater guarantees that the components we would rely on will stay the same.
    - **con:** implemented this way, our project would effectively be a pure translator, running only when the user requires a model to be converted from one format to another. This would limit our ability to add functionality designed to aid the user after the initial conversion. For example, if the user intends to modify the pydantic schema we would likely not be able to offer any editing tools (Eg. pruning a branch of the model-tree would have to be done manually by the user). Any mistakes by the user would only be noticeable when attempting to convert the pydantic model back to a YANG module, requiring a trial-and-error approach.
    - **additional complication:** neither of us has prior experience writing extensions for third-party software, leading to more uncertainty in our estimates for potential risks and challenges.

2.2.3 PyangBind

PyangBind is a Pyang plugin which, at least in theory, should be exactly what we need. It purports to convert a YAML schema to valid Python code that can then be edited, instantiated with data and sent to a switch. There are however several factors that do not inspire confidence in the project.

- As of September 2022, PyangBind has not received any new commits on any publicly available branch since August 2018 despite 82 unresolved issues, leading us to believe that the project has been abandoned by the author.
- The aforementioned unresolved issues generally revolve around serialization and deserialization issues, making it a questionable option for critical work.
- The quality of the code leaves a lot to be desired. The issues range from "un-Pythonic code" to severe structural problems.

For these reasons and for the time investment necessary to address them, we have opted not to pursue this plugin any further and consider it an option of last resort.

2.2.4 Pyang-Pydantic

As mentioned in section 9.3 Realized Opportunities, Pyang-Pydantic saw its first public commit in the first week of this project and while it has not nearly as many features as PyangBind, what it does have is much more concise and clean in terms of its implementation.

Currently, it functions as a Pyang plugin that simply receives a set of interpreted YANG modules, recursively iterates through their tree-structure and generates a pydantic class for each YANG type it encounters, then prints it to the output. It only directly supports the basic container types and it adds neither input validation nor default values.

So while it does not satisfy the full scope of our project it would be a great starting-off point for us, as we could start expanding upon its functionality without delays caused by having to refactor anything.

2.2.5 Pydantic

Pydantic is a data validation library for Python with some very appealing features:

- It can do runtime type-checking of arguments when instantiating classes or assigning to one of its member fields
- It allows even complex class structures with inheritance and compositions to be instantiated with a JSON-like dictionary
- It can serialize the content of a class to JSON, provided it consists only of python-native types or if custom serialization functions are provided for non-native types
- It can serialize and de-serialize a class structure, allowing it to be stored or sent as JSON
- It can do automatic type-casting and conversion between native types, allowing it to inter-operate with applications written in untyped languages, such as JavaScript
- Despite relying heavily on generic and dynamically generated classes, it provides a lot of type-hinting information to the IDE, making it easy to work with

These features are provided through a series of classes any developer should be familiar with when using pydantic for their project. The rest of this subsection aims to provide an overview of said classes and their purpose.

BaseModel

The `BaseModel` class lies at the heart of every pydantic project. It is the base class each class needs to inherit from, if it wants to make use of the aforementioned pydantic features and become a *Pydantic Model*.

When a class inherits from `BaseModel`, it fundamentally changes how the class works. For instance, member fields are no longer declared in the `__init__()` method, instead needing to be declared directly in the class body like conventional static members. Type annotation also plays a crucial role, as pydantic tries to convert any input given during construction to the annotated type - if no type is provided or the provided type is unknown to pydantic (meaning it does not inherit from `BaseModel`, nor provide its own validator), an exception is raised by default. Additionally, such a class can automatically be instantiated from a dictionary of its fields, even without declaring an `__init__()` method explicitly.

The integration with Python's dictionary type does not end there however. Any class inheriting from `BaseModel` can be serialized to a schema-dictionary via the `.schema()` method. Any properties pertaining to the data of the class are preserved in said schema, such as field types, names, defaults and value constraints in a way that is compatible with **JSON**

Schema Core and **OpenAPI**. Other class attributes such as methods, however, are not included, which is relevant when used in combination with the Datamodel Code Generator (DMCG).

Config

Defining a `Config` class within a model adds the option of modifying model-wide settings. Some of the most widely used settings include[4]:

- `allow_mutation`: whether `__setattr__()` is allowed
- `arbitrary_types_allowed`: whether to allow arbitrary user types for fields (validation simply consists of checking if the type matches when enabled)
- `extra`: whether to ignore, allow, or forbid extra attributes during initialization
- `underscore_attrs_are_private`: whether to treat any underscore fields as private, or leave them as is
- `validate_assignment`: whether to perform validation on assignment to attributes

Type Annotation and Validation

Input validation in pydantic is primarily declared via type-annotation. Pydantic's validation supports numerous types and therefore will be summarized aggressively here.[3]

- Most native Python types are supported, including but not limited to: `bool`, `int`, `str`, `bytes`, `list`, `dict` and `tuple`.
- Various types of `enum` are supported.
- Various types found in the **ipaddress** library are supported for IP validation.
- Most types provided by the **typing** library are supported, including `Optional`, `Union`, `Sequence`, `Type`, `Callable`, `Pattern` and `Annotated`.
- Pydantic offers several additional *constrained types*, such as `stricturl`, `PositiveFloat` , `conint` and `constr`. Most of these are built upon other types, with additional customizable restrictions.
- If the supported types are not sufficient, custom ones can be added by creating classes that provide their own validators via a `__get_validators__()` method, even if they do not inherit from `BaseModel`.

2.2.6 datamodel-code-generator

Datamodel-code-generator (or DMCG for short) is a project which aims to translate data models written in either the ApenAPI 3 or JSONSchema format into Python class structures based on pydantic. While it is primarily designed as a CLI tool, it can easily be used as a library and integrated into other projects (though the lack of documentation surrounding this use-case requires some reverse-engineering).

Some notable features include:

[3] A complete list of supported types can be found at https://pydantic-docs.helpmanual.io/usage/types/

- automatic generation of import statements based on the types and methods present in the schema being translated
- support for annotating classes via Python docstrings for ease of use within IDEs
- ability to re-use and reference other classes within the schema (meaning two classes containing a field of the same type do not lead to said type appearing twice in the output model)
- ability to rename classes and fields in order to not cause syntax errors, while still allowing initialization by the original name through pydantic's `alias` attribute

As of September 2022, datamodel-code-generator only supports pydantic V1, however the developer has been made aware of the impending V2 update in issue #803.

Additionally, the DMCG supports annotating classes with docstrings based on the description field of a class via the `use_schema_description` flag, however, there currently is no such option for member fields of classes. This leads to field descriptions only being visible in the field annotation, rendering them invisible to any IDE. The lack of such a feature has already been raised in issue #857 [3] and its implementation would greatly improve the usability of Pydantify.[4]

2.3 Pyang In-Depth

In this chapter, we will cover the Pyang project in more detail, specifically the parts relevant to our project. It is by no means a comprehensive review, but it should serve as a crash-course for anyone improving upon or maintaining our project.

2.3.1 The plugin system

Pyang can be extended at runtime through a fairly typical plugin interface. The `pyang` console-command supports a `--plugindir="<path>"` flag that, if present, prompts Pyang to look for additional plugins situated at the given path and import them through the **importlib** library.

For Pyang to recognise a Python script as a valid plugin, it needs to contain a `pyang_plugin_init()` function. This function will be called by Pyang once it is ready to initialize plugins and must in turn call `register_plugin()` with an instance of a class which inherits from `PyangPlugin` as its argument. Said class must:

- Call the super-class' constructor with its own name as the argument.
- Implement `add_output_format(self, fmts)` to associate the plugin with a given output file format.
- Implement `emit(self, ctx, modules, fd)`, which gets called after Pyang has parsed a YANG model if the user requests the output to be in the associated format.

The Listing 2.8 Barebones Pyang plugin shows a Pyang plugin with no additional functionality.

[4]This feature has since been implemented as part of the project. See subsection 5.1.1 Field annotation via docstrings.

```python
from pyang.plugin import PyangPlugin, register_plugin
from pyang.statements import ModSubmodStatement
from pyang.context import Context
from typing import List, Dict


def pyang_plugin_init():
  register_plugin(MyPlugin())


class MyPlugin(PyangPlugin):
  def __init__(self):
    # Pass on the name of the plugin
    super().__init__(name="my-plugin-name")

  def add_output_format(self, fmts: Dict[str, PyangPlugin]):
    # Register self as the plugin in charge of "my-format" inputs
    fmts["my-format"] = self

  def emit(self, ctx: Context, modules: List[ModSubmodStatement], fd):
    # Main functionality goes here.
    # Once converted, write the output to the "fd" file-descriptor.
    pass
```

Listing 2.8: Barebones Pyang plugin

This approach works quite well, but it does come with a slight disadvantage, namely that the plugins folder can not contain any non-plugin files in order to avoid Pyang logging it as an error. This has slight implications on the project's structure, as it requires a separate folder just for the plugin's entry-point.

2.3.2 Pyang classes

Statements

Statements represent most of the common YANG keywords such as *`module`*, *`list`*, *`leaf`*, *`container`*, *`type`*, etc. These are effectively the nodes in the YANG tree structure and are all derived from a common *`Statement`* class, often with very few additions.

`Statement` instances make heavy use of Python *`__slots__`*, which is effectively a whitelist of field names that are allowed within the instance. This means that, unlike conventional classes, arbitrary fields cannot be added to an instance at runtime. Additionally, the way these *`__slots__`* are used in Pyang leads to many of the fields being declared but not initialized, causing exceptions to be raised when accessed, even by an IDE. This, combined with YANG's heavy use of optional substatements can lead to situations in which the majority of a *`Statement`*'s fields are undefined. Direct access to fields therefore needs careful consideration.

The *`Statement`* class also offers most of the functionality required for tree traversal, including *`search()`* and *`search_one()`* to locate the statement's children and substatements by either their *`keyword`* or *`arg`* values along with *`main_module()`*, used to find the root module of the tree. To simplify navigation even further, each *`Statement`* contains a reference to its *`parent`*. For debugging and logging purposes, each *`Statement`* also contains a *`pos`* field, referencing the file and line number from which it was parsed - a welcome addition when working on cross-referential models split up across several files.

TypeSpecs

`TypeSpec` and its derived classes hold information about the underlying type of a node. They contain the base type (for example an integer, float or string) and the restrictions the value must adhere to to be considered valid. These restrictions typically consist of value ranges, length restrictions, regular expressions (called "patterns") or pre-defined values (called "enums").

These classes also contain a `validate()` function which, as the name implies, validates whether a given value matches the restrictions imposed on the type. Unfortunately for Pydantify, these functions are tightly coupled to the Pyang project and cannot be easily repurposed for input validation in the output model.

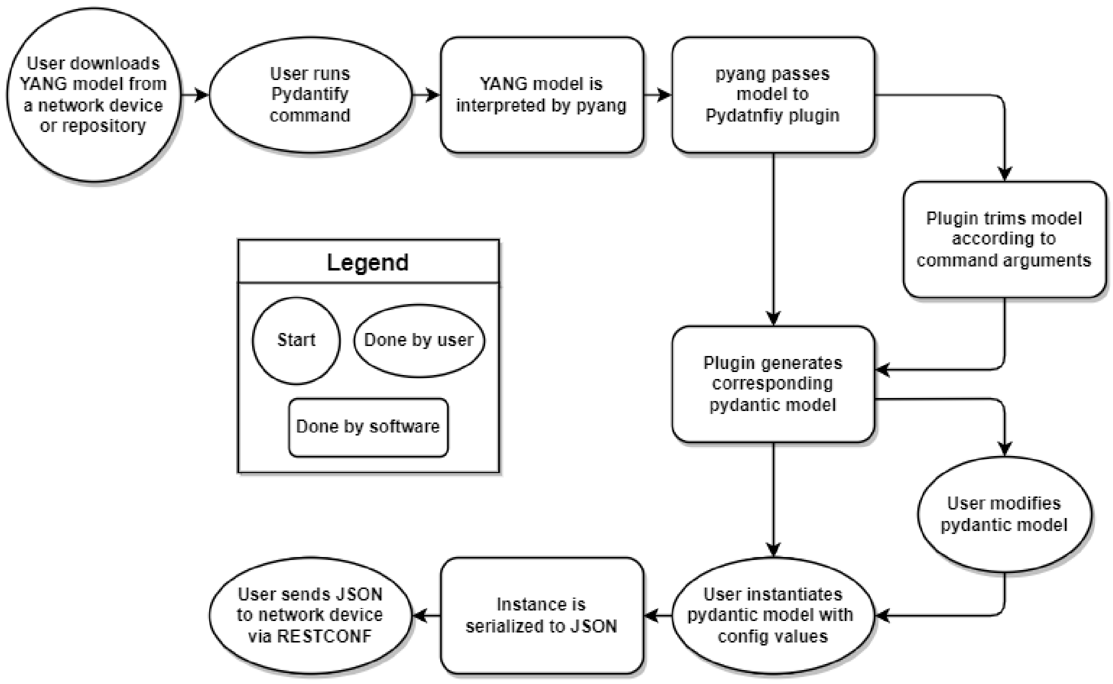
Part II

Product Documentation

Chapter 3

Requirements

3.1 Storyboard



User downloads YANG model from a network device or repository
User runs Pydantify command
YANG model is interpreted by pyang
pyang passes model to Pydatnfiy plugin
Plugin trims model according to command arguments
Plugin generates corresponding pydantic model
User modifies pydantic model
User instantiates pydantic model with config values
Instance is serialized to JSON
User sends JSON to network device via RESTCONF
Legend
Start
Done by user
Done by software

Figure 3.1: Expected workflow of the end-user

One of the potential issues that stands out in Figure 3.1 is the large number of user interactions. This emphasises the need for intuitive and well documented interactions between the user and Pydantify, especially at the main point of contact, namely inside the output model.

3.2 Functional Requirements

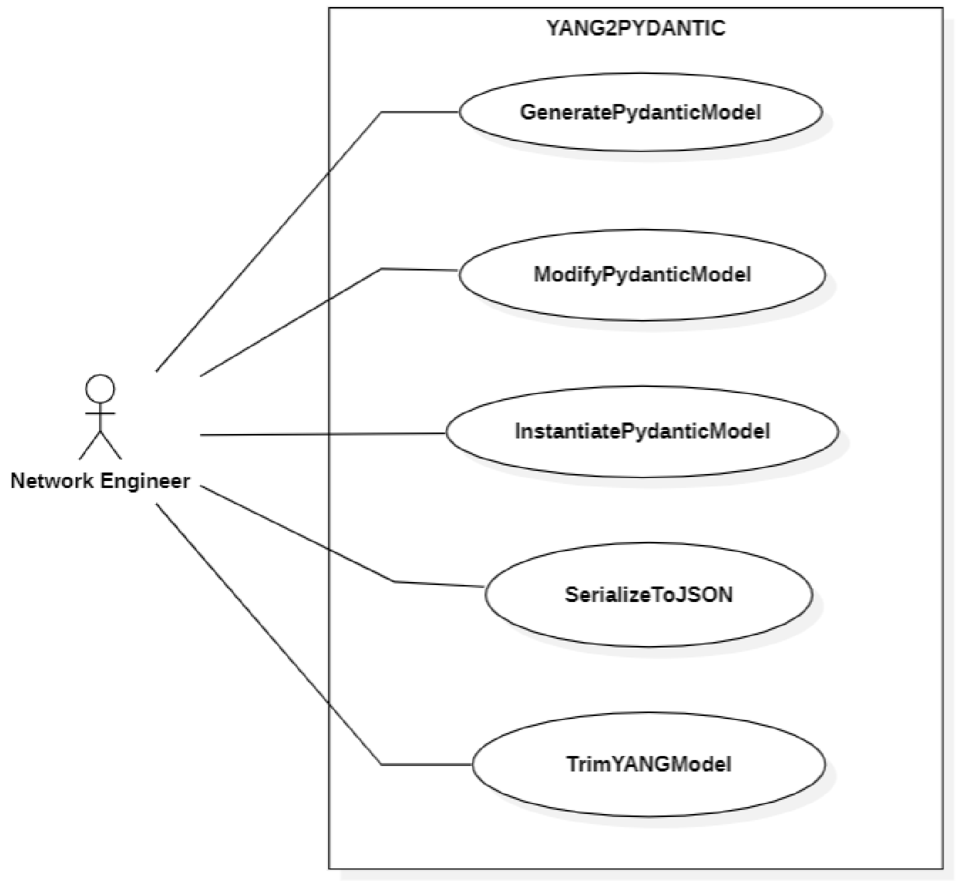


Figure 3.2: Use Cases

3.2.1 Actors

The sole actor in our system is the network engineer, who will fetch a YANG module, then generate and modify a corresponding pydantic model. The pydantic model can finally be instantiated with configuration values and sent to a network device.

3.2.2 Validation

The validation of the functional requirements was done in terms of the System Tests.

3.2.3 Actions

GeneratePydanticModel

Actor	Network Engineer
Success Scenario	The network engineer wants to generate a pydantic model from a YANG model. He can do so by running a command-line command. The command prompts a custom Pyang plugin to generate a pydantic model representing the YANG model.
Validation Result	An OpenConfig YANG model was downloaded and subsequently passed to Pydantify through the command *`pydantify openconfig-interfaces.yang`*. This resulted in the successful generation of a valid output model. Generating pydantic models from YANG 1.1 models is currently not fully supported by Pydantify due to issues within the Pyang dependency.

Table 3.1: Functional Requirement - GeneratePydanticModel

ModifyPydanticModel

Actor	Network Engineer
Success Scenario	The network engineer wants to modify the generated pydantic model. He can do so manually in an IDE of his choice.
Validation Result	After the generation of the pydantic model, it was possible to modify the generated model within Visual Studio Code. We specifically tested modifying docstrings, patterns and integer ranges.

Table 3.2: Functional Requirement - ModifyPydanticModel

InstantiatePydanticModel

Actor	Network Engineer
Success Scenario	The network engineer wants to instantiate the pydantic model with configuration values. He can do so by fetching the network device's configuration in JSON form through a RESTCONF request. He can then pass the configuration to the constructor of the pydantic model to create an instance.
Validation Result	Instantiating a pydantic model with configuration values was possible to do by inserting the values by hand. Instantiating a pydantic model by fetching the network device's configuration in JSON form through a RESTCONF request was not possible, as our solution exclusively uses qualified names (e.g. *"openconfig-interfaces:interfaces"*), whereas RESTCONF replies only use qualified names under certain conditions, which causes a name mismatch. (See subsection 4.1.2 Necessary information to generate each pydantic class) **This requirement has therefore only been partly met.**

Table 3.3: Functional Requirement - InstantiatePydanticModel

SerializeToJSON

Actor	Network Engineer
Success Scenario	The network engineer wants to serialize the pydantic model to valid JSON, so that it can be sent to a network device via RESTCONF. He can do so by calling *.json()* on an instance of the pydantic model.
Validation Result	We can call *.json(alias=True)* on an instance of the Model class in the pydantic model. This serializes its contents to a valid JSON payload which can be used to configure a network device over RESTCONF.

Table 3.4: Functional Requirement - SerializeToJSON

TrimYANGModel

Actor	Network Engineer
Success Scenario	The network engineer wants to work only with one branch of the YANG model tree. He can do so by passing a command-line argument containing a YANG-model-internal path to the node of the tree he wants included. The plugin then filters the model, ignoring the nodes leading up to the selected node unless they are relevant to the branch (eg. typedefs).
Validation Result	By using the command argument *-t/-trim* (e.g. *"pydantify -t openconfig-interfaces/interfaces/interface/config openconfig-interfaces.yang"*), it was possible to only include the *config* node and its children in the output model.

Table 3.5: Functional Requirement - TrimYANGModel

3.3 Non-Functional Requirements

The process of validating the Non-Functional Requirements will be done by Dejan Jovicic, following called "The Validator". Screenshots of the validations can be seen in section B.1 NFR Validation Screenshots.

Portability

- ***The project SHOULD be able to run on both Linux and Windows.***
  - **Validation process:**

    Try to run Pydantify on both Operating Systems.
  - **Validation results:**

    Pydantify was installed and ran on both Operating Systems without any issues.
- ***The project SHOULD be installable via a single command, provided the appropriate Python version is pre-installed.***
  - **Validation process:**

    The Validator will try to install Pydantify with the command `pip install 'path'`
  - **Validation results:**

    It was possible to install Pydantify with `pip install .` if the project was downloaded prior to your computer. Another possibility is to download and install it directly from gitlab with the command `pip install git+https://gitlab.ost.ch/pydantic-sa/pydantify`

Performance

- ***After the YANG model is parsed, generating a pydantic model from it MUST take no longer than 5 minutes.***
  - **Validation process:**

    A YANG model with a length of 150 lines (approximately the average YANG file to be used) will be used by the Validator to generate pydantic models to check if it will take longer than 5 minutes.
  - **Validation results:**

    A YANG model with a length of approximately 150 lines of node statements was used and the time needed to generate the pydantic model out of the YANG model was 0.121 seconds.
- ***After the YANG model is parsed, generating a pydantic model from it MUST scale sub-exponentially with the size of the inputted model.***
  - **Validation process:**

    The Validator will use three different YANG models with the size of "Base", "5xBase" and "10xBase", meaning that the "Base" will be our standard model size, which will then be multiplied by 5 and 10.

- **Validation results:**

  The result with "Base" was 0.121 seconds as mentioned before in Performance. The results for "Base", "5xBase" and "10xBase" were as follows:

  - Base: 0.121 seconds
  - 5xBase: 0.470 seconds
  - 10xBase: 0.904 seconds

    We can read from the results that the performance of Pydantify is scaling sub-linear and therefore fulfills the requirement to scale sub-exponentially.

Maintainability

- ***The plugin SHOULD be easily adaptable to pydantic 2.0.***
  - **Validation process:**

    No validation possible, as pydantic v2.0 has not been released yet at this point. Planned to be released in 2023.

Usability

- ***A network engineer SHOULD be able to realize in less than 5 minutes whether this plugin meets his needs.***
  - **Validation process:**

    At the end of the project, the Validator will do a usability test with selected network engineers from the Institute for Networked Solutions (INS), the results will determine if this NFR is passed or not.

  - **Validation results:**

    The results of the Usability Test can be seen in subsection 10.2.6 Usability Test Result.

Testability

- ***The project SHOULD contain tests covering all supported YANG node types.***
  - **Validation process:**

    The test coverage should be higher than 90%.

  - **Validation results:**

    On Gitlab we can see in the Continous Integration / Continous Deployment (CI/CD) of the project the stage "run_pytests" is run, which runs a code coverage report from which we can read the code coverage percentage, which is currently at 94.95% .

Reliability

- ***The plugin MUST raise an exception when a YANG statement is encountered it does not recognise.***
    - **Validation process:**

      The Validator will try to generate a pydantic model out of a YANG model which has an unsupported YANG statement, an exception is expected.
    - **Validation results:**

      Tried to generate a pydantic model out of a YANG model with an unsupported statement, an exception was raised.
- ***The generated pydantic model SHOULD be complete if no exception was raised.***
    - **Validation process:**

      The Validator will try to generate a pydantic model out of a YANG model and afterwards try to push the generated JSON file to a switch via RESTCONF.
    - **Validation results:**

      The generation of a pydantic model out of a YANG model and then sending it to a switch via RESTCONF worked without any issues.

Chapter 4

Architecture

4.1 Architecture considerations

The architecture of our project was arrived at by attempting to address several questions regarding the generation process.

- How can Pyang (primarily a CLI tool) be integrated into the project?
- Which pieces of information need to be gathered to generate an output class?
- What constraints does the RESTCONF JSON structure impose on the structure of the output model?
- Given that a YANG statement can reference a definition located somewhere else in the model, how can we resolve said definition without parsing it multiple times?
- To what extent can input validation be done by the output model?

The rest of this section aims to answer the questions posed above and list the constraints those answers pose for the architecture of our project.

4.1.1 Pyang integration

As explained in subsection 2.2.2 Pyang, Pyang is designed to be a stand-alone tool and is not primarily intended to be used as a library. Additionally, being a legacy codebase compatible with both python 2.7 and 3, a large part of it is implemented in a non-standard way. A key area of concern regarding the integration into Pydantify is the installation process. Unlike more modern Python projects where installation is done via a ".toml" file in conjunction with a package manager, it is instead installed via a "setup.py" script. The issue is, that as part of the process, the Python based start-up script is moved from the project's "bin" directory to a scripts folder, with the unfortunate side-effect of rendering the start-up file non-importable by other projects.

4.1.2 Necessary information to generate each pydantic class

When the pydantic model is to be generated, the following information needs to be available for each and every class in the output model:

- Class name
- Base class to derive from
- The class' *doc-string*, if present
- The `Config` to use
- The Class' member fields

Class Name

Can be generated by taking the `arg` field of the original model and appending to it the type it originates from. Eg. a node of type `container` called `"interface"` results in a class called `InterfaceContainer`. In case the combination of `arg` value and type is not unique in the model, it needs to be made unique in some other way, otherwise the DMCG cannot distinguish these and will fall back to only ever referencing the first occurrence.

Base Class

Depending on the YANG type being represented, the following options present themselves:

- The YANG type being represented is a container
  - The base class can be pydantic's own `BaseModel`.
- The YANG type being represented is a list
  - A class representing a member of the list is created based on `BaseModel`. The list itself gets created by annotating the field holding it with `List[<member-type>]`, without the need for a class of its own.
- The YANG type being represented is a primitive
  - The base class can be derived from pydantic's own template-types, such as `constr`, `conint` and `condate`.

Class Doc-string

A class' doc-string can be generated from the node's `description` field, if present.

Config

As explained in section 2.2.5 Config, the `Config` class provides settings relevant to the creation process of pydantic models and is not included in the output model. For our purposes, the same `Config` class can be used for all classes as the settings are independent of the input model.

Class Members

Class members are added to the model via a dictionary which takes the form of `Dict[<name>, <type-info>]`, where the name corresponds to the YANG statement's sanitized `arg` field. `type-info`, on the other hand, has the following signature:

```
Annotated[<type>, Field(<default>, alias=<alias>)]
```

This requires additional information to be gathered before model creation:

- `<type>`: a reference to the class of the field. Any input validation is performed by said class.
- `<default>`: the value provided by the YANG `default` statement.
- `<alias>`: the name to be used for the field during serialization to JSON

The `alias` is required due to the name of the field needing to be sanitized in the output model for Python to recognise it as a valid identifier, whereas RESTCONF expects the original, unmodified name.

4.1.3 Resolving type definitions and references

YANG models can contain intramodel references both to make use of custom type definitions and to create references between different nodes via `leafref`. The statements being referenced can be located virtually anywhere in the model, meaning that common tree traversals such as preorder and postorder may encounter nodes with references to previously not encountered statements. Additionally, such references may contain other references as long as no circular references arise.

In order to avoid duplicate classes in the output model, it is important that each statement in the input model gets interpreted at most once (even if referenced multiple times), resulting in a single object representing it. This in turn requires the ability to keep track of already encountered statements and referencing the generated output instance directly if encountered again, without parsing it again.

4.1.4 Input validation in the output model

As explained in subsection 2.1.1 typedef, `typedef` statements allow only a handful of basic types, namely strings and various kinds of numbers. During instantiation of the output model, pydantic automatically coerces a value to the type dictated by the field it's being assigned to, raising an exception if the conversion cannot be done. Additionally, pydantic V2 plans to add a ”strict” mode, disabling the automatic conversion in favour of dictating the required type to be given by the user.[5] Basic type validation is therefore covered automatically by pydantic, provided the expected type is annotated correctly in the output model.

Restrictions on the value of a node however are not as trivial. YANG allows for nested typedefs, where each layer imposes additional restrictions on the underlying type. The valid input range for a given type therefore consists of all the values that fit the cumulative sum of all its restrictions. Pydantic's `con`-types (such as `conint` and `condecimal`) provide most of the building-blocks needed to implement YANG's full spectrum of restrictions, but it is much more basic in its implementation. For example, a YANG integer can be restricted to be within the ranges `"0..1 | 5..10 | 20..max"`, all in a single statement, whereas pydantic's `conint` can only be restricted to be between two values at most. Similarly, string-based types can have multiple regex checks imposed on them, whereas Pydantic's `constr` only allows a single one. In addition, YANG uses the xmlschema regex spec[6], which is not entirely covered by Python's regex implementation.

4.1.5 Implications

Pyang integration: Due to the legacy status of the codebase and the inability to import the startup file, only two options remain: running Pyang as a console-command from inside Pydantify or copying the startup script into the project as-is, effectively adding a wrapper around Pyang's main function.

The first option comes with a major downside that makes it unattractive, namely debuggability. Pydantify would need to execute Pyang via a console-command, which in turn imports the project as a plugin and executes it. Any exceptions raised within Pyang would result in an early exit with errors; the IDE would only be able to report that the console-command could not be executed successfully, with no further insights.

Copying the startup script into the project is also undesirable as it requires any future changes to the script in Pyang to be ported to its now detached sibling. In terms of maintainability and code cleanliness however, it is the lesser of two evils.

Necessary information to generate each pydantic class: Since nodes are primarily defined by the statements they contain, these sub-statements have to be visited before the current node can be fully initialized. This necessitates a post-order traversal of the input model.

The possibility of identically named nodes in the input model requires a mechanism to make the class names in the output model uniquely identifiable. To facilitate this, class names can be enumerated sequentially if they are not otherwise unique.

Resolving type definitions and references: When a reference is encountered to a statement that has not been visited yet, the post-order traversal has to be temporarily abandoned to resolve the statement being referenced. If the reference instead points to an already visited statement, the resulting output class should be re-used, which necessitates keeping track of all visited statements and their corresponding output classes.

Input validation in the output model: Validating or coercing the type of an input value during initialization can be done rather easily by finding the root type and mapping it to a type known to pydantic.

Layered regular expressions can be implemented by turning each pattern into a lookahead-check, concatenating them and capturing everything if all checks pass.[10] Unfortunately, the issue regarding YANG using a different regex spec remains and is outlined in more detail in chapter 7 Further Work.

When it comes to validating the value range of numbers however, currently only basic sanity checks are possible without implementing custom validators, as "or" logic would be needed to properly validate disjointed value ranges.

4.2 Component overview

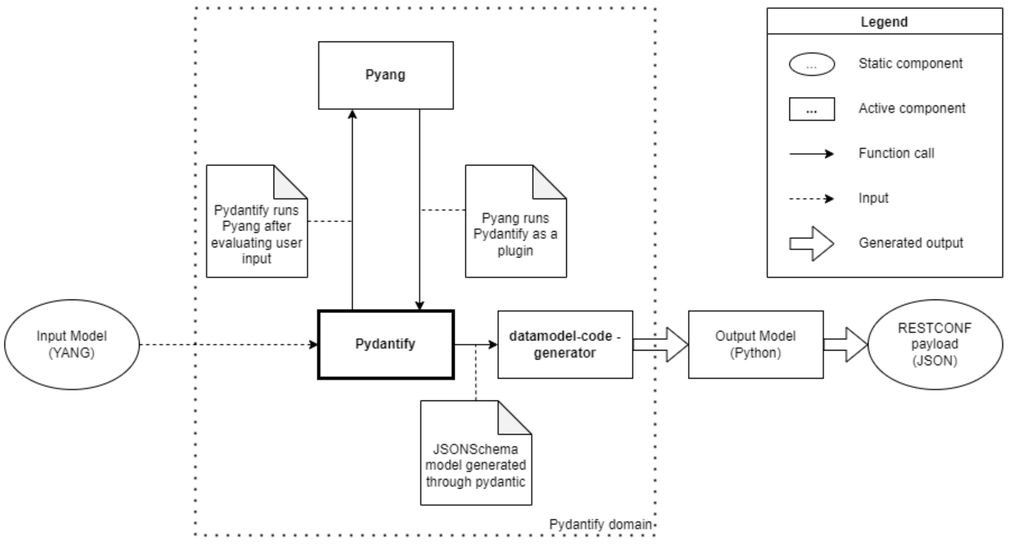


Figure 4.1: Component overview

As seen in the picture above, there are four core components to the project. These are, from left to right: the input model, the "Pydantify domain", the output model, and the RESTCONF payload.

4.2.1 Input Model

The input model, consisting of a main YANG file and its dependencies (if any), is passed to Pydantify through file paths.

4.2.2 Pydantify Domain

Pydantify itself consists of three components: the Pydantify plugin, Pyang and DMCG.

- The Pydantify plugin first parses the user's input, then runs Pyang. It later translates what it receives back from Pyang into a JSONSchema model and passes it on to the DMCG.
- Pyang is in charge of parsing the input model into a data structure. This structure is passed on to Pydantify when done.
- DMCG transforms the JSONSchema model into the Python code which constitutes the output model.

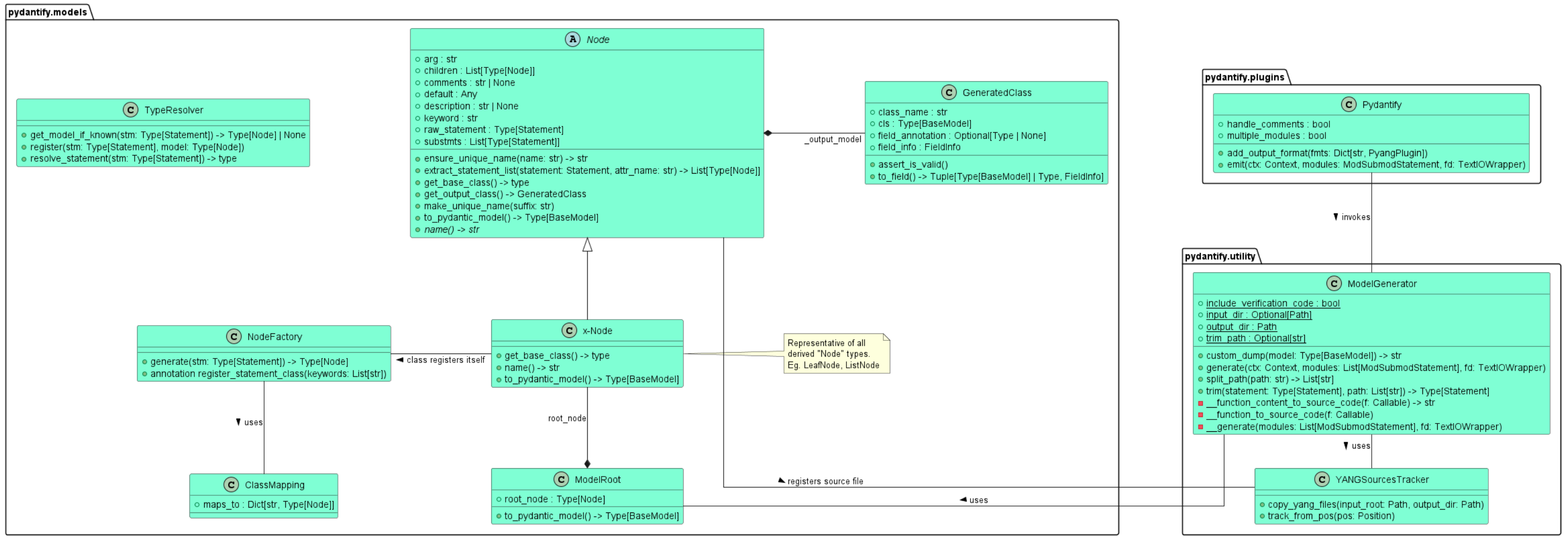
4.2.3 Output Model

The output model consists of an executable Python file. Said file contains a class structure which can be initialized with values by the user and then converted to a RESTCONF payload via a built-in function. Utility functions and tips on how to use the model can be included at the end of the file at the user's request.

4.2.4 RESTCONF Payload

The RESTCONF payload is the final product in the chain. It contains the configuration data set by the user in a format compatible with the input model. This payload can be copied from the terminal when the model is run or sent directly to a network device via the **requests** library, a code snippet for which is included in the output model to make the process as easy as possible.

4.3 Class Diagram



pydantify.models
Node
arg : str
children : List[Type[Node]]
comments : str | None
default : Any
description : str | None
keyword : str
raw_statement : Type[Statement]
substmts : List[Type[Statement]]
ensure_unique_name(name: str) -> str
extract_statement_list(statement: Statement, attr_name: str) -> List[Type[Node]]
get_base_class() -> type
get_output_class() -> GeneratedClass
make_unique_name(suffix: str)
to_pydantic_model() -> Type[BaseModel]
name() -> str
TypeResolver
get_model_if_known(stm: Type[Statement]) -> Type[Node] | None
register(stm: Type[Statement], model: Type[Node])
resolve_statement(stm: Type[Statement]) -> type
GeneratedClass
class_name : str
cls : Type[BaseModel]
field_annotation : Optional[Type | None]
field_info : FieldInfo
assert_is_valid()
to_field() -> Tuple[Type[BaseModel] | Type, FieldInfo]
_output_model
NodeFactory
generate(stm: Type[Statement]) -> Type[Node]
annotation register_statement_class(keywords: List[str])
class registers itself
x-Node
get_base_class() -> type
name() -> str
to_pydantic_model() -> Type[BaseModel]
Representative of all derived "Node" types. Eg. LeafNode, ListNode
uses
root_node
ClassMapping
maps_to : Dict[str, Type[Node]]
ModelRoot
root_node : Type[Node]
to_pydantic_model() -> Type[BaseModel]
registers source file
uses
pydantify.plugins
Pydantify
handle_comments : bool
multiple_modules : bool
add_output_format(fmts: Dict[str, PyangPlugin])
emit(ctx: Context, modules: ModSubmodStatement, fd: TextIOWrapper)
invokes
pydantify.utility
ModelGenerator
include_verification_code : bool
input_dir : Optional[Path]
output_dir : Path
trim_path : Optional[str]
custom_dump(model: Type[BaseModel]) -> str
generate(ctx: Context, modules: List[ModSubmodStatement], fd: TextIOWrapper)
split_path(path: str) -> List[str]
trim(statement: Type[Statement], path: List[str]) -> Type[Statement]
__function_content_to_source_code(f: Callable) -> str
__function_to_source_code(f: Callable)
__generate(modules: List[ModSubmodStatement], fd: TextIOWrapper)
uses
YANGSourcesTracker
copy_yang_files(input_root: Path, output_dir: Path)
track_from_pos(pos: Position)

Figure 4.2: Resulting class diagram

Note for future developers: Because class diagrams tend to go out of date rather quickly, the Pydantify project contains a convenience-command in order to quickly generate a new one directly from the source-code. By running `pdm run class_diagram` in the root folder of the source repository, a new diagram can be generated automatically. While it is not as well formatted as in Figure 4.2, it does come with the guarantee of being up-to-date.

Chapter 5

Work done on external projects

As part of the project, some contributions were made to the projects Pydantify depends on. This chapter aims to document these contributions.

5.1 Contributions to Datamodel Code Generator

5.1.1 Field annotation via docstrings

As outlined in subsection 2.2.6 datamodel-code-generator, the DMCG did not support annotating class fields via docstrings. We therefore opted to implement this feature ourselves.

We tried to align our addition as close as possible to the existing code, drawing heavy inspiration from the `use_schema_description` feature. Our implementation consists of a series of distinct steps:

- Parsing the `--use-field-description` CLI flag
- Relaying the state of the flag to the relevant section of code
- Generating a docstring through a property when requested and when the flag is set
- Removing the description from the field annotation when the flag is set
- Printing the docstring to the output file through Jinja2 when a docstring is present

In addition to the above, a unit test was added to not negatively impact the existing code coverage score of the project. This contribution resulted in merge-request#918[13], which was merged into the master branch without alterations on the 17th of November, officially releasing as part of the 0.14.0 update.

We also contributed a more general fix for DMCG's docstrings, as the indentation of docstrings which span more than a single line was not done correctly, leading to issues with how they got rendered inside the IDE. Our solution to this issue was to pass the string to the `indent()` function within Jinja2 before they get printed to the output file. This resulted in in merge-request#938[12], which was merged without changes into the main branch on the 22nd of December.

5.1.2 Pydantic deprecation warnings fix

While implementing automated tests via Pytest, we noticed a large number of warnings stemming from the outdated use of `copy_on_model_validation`. After some investigation, we found that the issue was within the DMCG itself. An update to pydantic had changed how this flag worked, turning it from a simple boolean toggle to a string-selection consisting of three options. After changing the default value set by the DMCG from `False` to `"None"`, Pytest warnings regarding Pydantify were reduced from 130 to none, while warnings regarding the DMCG were reduced from 6145 to 4. This change resulted in merge-request#927[11] in which the project owner added a case distinction based on the installed pydantic version before merging it into main on the 22nd of November.

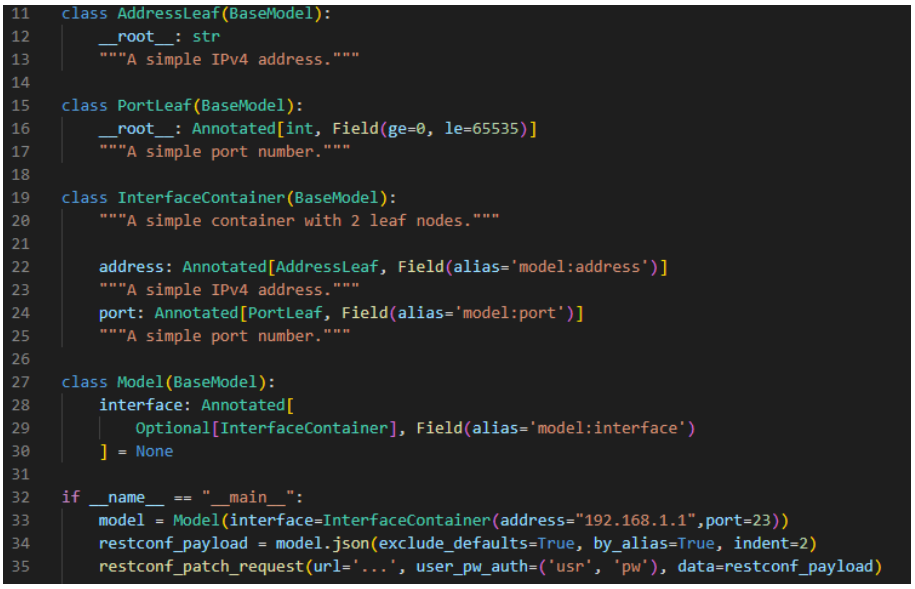
Chapter 6

Results

In this thesis, we demonstrated the feasibility of generating valid pydantic data structures from YANG models in a way that enables them to serialized to valid RESTCONF payloads, all in a way that is as intuitive and user-friendly as possible. Along the way we conducted research on a wide variety of technologies and projects, and documented our findings.

All non-functional requirements have been satisfied and while not all functional requirements have been met, the remaining ones likely require work on external projects to implement cleanly.

While there is still a lot of work to be done to turn Pydantify it into a production-ready tool, the concept is sound and shows potential. We look forward to seeing its continued development.



```python
class AddressLeaf(BaseModel):
    __root__: str
    """A simple IPv4 address."""

class PortLeaf(BaseModel):
    __root__: Annotated[int, Field(ge=0, le=65535)]
    """A simple port number."""

class InterfaceContainer(BaseModel):
    """A simple container with 2 leaf nodes."""

    address: Annotated[AddressLeaf, Field(alias='model:address')]
    """A simple IPv4 address."""
    port: Annotated[PortLeaf, Field(alias='model:port')]
    """A simple port number."""

class Model(BaseModel):
    interface: Annotated[
        Optional[InterfaceContainer], Field(alias='model:interface')
    ] = None

if __name__ == "__main__":
    model = Model(interface=InterfaceContainer(address="192.168.1.1",port=23))
    restconf_payload = model.json(exclude_defaults=True, by_alias=True, indent=2)
    restconf_patch_request(url='...', user_pw_auth=('usr', 'pw'), data=restconf_payload)
```

Figure 6.1: Closing Example of a Generated Output Model

Chapter 7

Further Work

7.1 Work to be done within Pydantify

7.1.1 Tests

While expanding the feature-set of Pydantify, we repeatedly encountered the need to update the pydantic models we compare against in the integration tests. This was primarily due to changes in formatting or code structure propagating through all the tests. For this reason, we recommend keeping the number of integration tests to a minimum. Additionally, improving the legibility of the error messages raised when comparing output models could drastically simplify the debugging process by indicating the precise location where the output model deviates from the expected result.

7.1.2 Instantiation of the output model

One of the pain points surrounding the use of Pydantify are the error messages raised when the output model is being instantiated wrongly by the user. These may improve by upgrading to Python 3.11 or by changes in the pydantic project. Should these improvements not materialize however, using the **friendly** Python library should be considered, as it drastically simplifies locating issues in case of nested constructors and complex single-line expressions, at the cost of an additional dependency in the output model.

7.2 Proposed work in the ecosystem

7.2.1 XML-Schema Regular Expressions

We discovered relatively late in the project that YANG uses regular expressions as described in the xmlschema, which is not equivalent to Python's implementation of regex.[6] While they are mostly similar, at least the `"\p{...}"` token (called a "category escape") appears to be a nonstandard regex feature which leads to an exception being raised when parsed by Python's **re**-library. This exception would be raised within pydantic while initializing the output model.

We are of the opinion that despite the context the exception is raised in, covering the xmlschema implementation of regex is not the responsibility of pydantic. Instead, we would like to propose a project that converts xmlschema regular expressions into Python-native regular expressions. Such a library could be utilized by Pydantify as a further processing step when generating an output model.

7.2.2 Pyang

In our opinion, the Pyang project suffers from its ongoing compatibility with older Python versions. The addition of type annotations in Python 3.8 could drastically improve the legibility of the code, simplifying both the maintainability of the project as well as the development of plugins.

Additionally, dropping support for Python versions below 3.7 would enable the use of **setuptools**, which would resolve most of the issues relating to differences of installation between Windows and Linux, such as issue #310[9]. It would also alleviate the frustrations described in subsection 4.1.1 Pyang integration, as the startup script would not need to be moved programmatically in a custom installation process.

7.2.3 Multi-alias support in pydantify and DMCG

Currently, Pydantify does not fully support output model initialization directly from RESTCONF payloads due to the lack of support for multiple aliases on individual fields of the output model. This may change with the release of pydantic V2 but as of the time of writing, circumventing this limitation would require compromises in either the maintainability of the source code or the versatility of the output model.

For this reason, we opted to forego such an attempt in the scope of this thesis and would instead like to propose the implementation of multiple aliases in Pydantic and by extension DMCG.

7.2.4 Poetry and other package managers

Package managers generally install the latest version of a dependency unless the project sets a specific restriction. This is a great approach for keeping projects up to date but a breaking change in a dependency requires maintainers to either drop support for the older version or add an alternate branch in the code depending on the version of the dependency installed. This became apparent to us while working on the Pydantic deprecation warnings fix contribution to DMCG, where we noticed a need for the ability to install the oldest supported version of all dependencies of a Python project in order to validate the advertised range of compatibility. Currently neither Poetry, PDM nor Flit support such a feature, leaving maintainers with the option of either manually locking dependencies to a particular version during testing or trying to manually keep track of changes and hoping to spot compatibility issues in contributions made by other other developers during pull requests.

Part III

Project Documentation

Chapter 8

Project Plan

8.1 Organization

In typical projects, the costs, timeline and scope of a project have to be considered and balanced out, this is typically referred to as the ”project management triangle”. For our purposes however, the final deadline is non-negotiable and the costs are non-existent. This means our scope is limited to whatever can be achieved within the timeframe, making artifacts like Gantt charts and milestone-trend analysis rather pointless for planning purposes. We instead opted to subdivide our allotted time into phases, akin to RUP, and use a simple kanban board to keep track of outstanding tasks and assign the associated responsibilites. This keeps the organizational overhead to a minimum, leaving more time for productive work.

8.2 Project Time Plan

8.2.1 Phases

Inception

The inception phase starts on 19.09.2022 and will end on 07.10.2022. In this phase, we are making ourselves familiar with YANG and the general network automation environment. Reaching a consensus on the project scope with the stakeholder is another goal of this phase.

Elaboration

The elaboration phase starts on 07.10.2022 and lasts till 04.11.2022. We check what tools and projects we can implement in our project. In this phase, we define what our risks, our technical requirements and NFRs are. We set up our programming environment, so that we are ready for the construction phase.

Construction

The construction phase starts on 04.11.2022 and ends on 12.12.2022, in this time span the programming will take place.

Transition

This phase will take place in the last week before the final hand-in of the project from 12.12.2022 to 23.12.2022 17:00. In this phase we will be finalizing the documentation and preparing it for the final hand-in.

8.2.2 Milestones

M1: End of Inception - 07.10.2022

The goal of this milestone is to understand the assignment and get a grasp of the YANG and network automation environment. A project time plan is created.

Products:

- Documentation of our research
- Project time line

M2: End of Elaboration - 04.11.2022

At the end of the elaboration the decision should have been made, with which tools and projects we will work in the future, what our risks are and the definition of the requirements.

Products:

- Risk analysis
- FRs and NFRs
- Use cases
- Barebones proof of concept

M3: End of Construction - 12.12.2022

The project should be finished by then, only small adjustments and the documentation are left to be finished.

Products:

- Test protocol
- Usability protocol
- Working proof of concept
- Abstract (due 19.12.2022)

M4: End of Transition - 23.12.2022 17:00

Everything should be finished and the project should be handed-in with the complete documentation.

Products:

- Finished project documentation

- Finished proof of concept

M5: Presentation - TBD

The presentation for the term project is prepared and the team is ready to presentate it. The poster according to the guidelines is created.

Products:

- Presentation

8.3 Roles

8.3.1 Advisor and Stakeholder

Urs Baumann is responsible for the evaluation of the project and is simultaneously the stakeholder. He is available for any help of the project if the team is facing any sort of issues.

8.3.2 Developers

Dejan Jovicic and Dominic Walther will be the developers and responsible for the success of the project.

8.4 Meetings

Every friday a meeting will be held with the advisor and stakeholder Urs Baumann and the two developers. The notes to which can be found in **?? ??**.

8.5 Planning Tools

- clockify.me (timetracker)
- Gitlab (issue tracker, file storage & versioning)
- Microsoft Teams (communication channel & meeting)
- Visual Studio Code (used to write the documentation in LaTeX)

Chapter 9

Risk Management

9.1 Risks

This chapter deals with the risks that might occur during the project. The properties of the risks are:

- ID: Identifier of the risk.
- Description: Short text explaining the risk.
- Probability: The probability of the risk measured in percentage. 100% meaning that it will occur at least once during the project.
- Maximum Time Loss: The maximum time loss we will have because of this risk, measured in hours
- Mitigation: Description of the precautions we can take to mitigate the risk.
- Behaviour: Describes the behaviour of what is done if the risk happens.
- Severity: The severity shows the extent of the damage the risk will do to the goals and objectives to the project. It is measured qualitatively with ”low”, ”mid” and ”high” severity.

ID	Description	Probability	Max. time loss	Mitigation	Behaviour	Severity
R1	A team member gets sick	low	17h	No mitigation possible	Communication between team members, prioritize tasks which need to be done	low
R2	Irreparable corruption of the git server	low	34h	Weekly off-site backups on the devices of the team members	Restore data with your backup	mid
R3	Irreparable corruption of the clockify data	low	0.5h	Weekly off-site backups on the devices of the team members	Restore data with your backup	low
R4	Unrealistic timeline	low	17h	Detailed project plan	Reduction of the project scope	mid
R5	Specifications cannot be implemented as intended	mid	51h	Weekly meetings with the advisor	Getting help from the advisory & experts	mid
R6	Familiarization with Pyang, pydantic or other plugin takes longer than expected	low	34h	Watch tutorials to gain knowledge of technology	Reduction of the project scope	mid
R7	Hardware of a team member is kaput	low	34h	No mitigation possible	Buy new hardware and install all tools again - redo destroyed work	mid

Table 9.1: Risk analysis

9.2 Opportunities

ID	Description	Probability	Impact
O1	Pydantic V2.0 is scheduled to be released at the end of October 2022, latest at the end of the year 2022	high	high
O2	An existing GitHub repository gives us an idea on how to solve a problem	mid	high

Table 9.2: Opportunity analysis

9.3 Realized Opportunities

ID	Description	Impact	Date	OID
RO1	The ”Pyang-Pydantic” repository offered us a great starting-off point for our project	mid	22.09.22	O2

Table 9.3: Realized opportunity analysis

Chapter 10

Quality Measures

10.1 Code Guidelines

The principles described in this section should always be adhered to to ensure a consistent code-style. Change proposals are always welcome.

Editor: The editor to be used is VSCode. Files associated with said editor can be committed to the repository provided the contained settings are not user-dependent. The recommended plugins should be installed.

Linter: The linter specified in the settings file should be installed and enabled at all times with the provided settings. Exceptions to the linting rules should be avoided. Linting rules may be changed if an agreement is found.

Classes: Class names are written in *CamelCase*. Member fields are annotated with their expected type. Should annotation not be possible the regular way (e.g. due to circular imports), annotations are done via so-called "forward references".

Function Names: Class names are written in *snake_case*.

Variable Names: Class names are written in *snake_case*. Too explicit is better than too short. Abbreviations are only allowed if they are immediately obvious in their context.

Comments: Comments should primarily explain why something was done, not how it works. Comments explaining how something works are to be treated as a code-smell.

Comprehensions: Comprehensions should only be used for trivially understandable operations. Use explicit for-loops otherwise.

Exception handling: Exceptions are to be treated as errors and should only be raised if the program cannot proceed safely. Exceptions should not be caught unless absolutely necessary. Exceptions may be caught and re-thrown to add additional context to the exception.

10.2 Testing

10.2.1 Code Coverage

Our aim is to reach a code coverage of at least 90%, with every supported YANG statement appearing in at least one test. This gives us enough confidence that the tests cover all supported YANG node types as requested in section 3.3 Non-Functional Requirements.

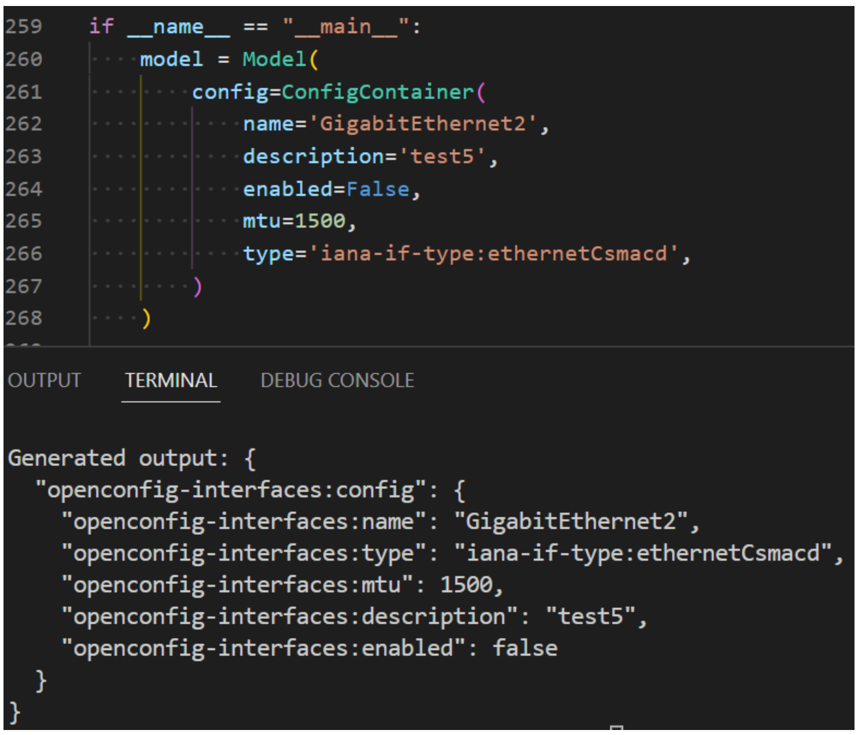
10.2.2 Integration Tests

The integration tests are realised with the **pytest** library. It verifies the correct generation of pydantic models by converting YANG models to pydantic models. The generated Python files are then compared against ones we validated by hand.

This makes detecting regressions easy, at the expense of complicating the process of integrating new features, as changes tend to propagate throughout most if not all tests, quickly necessitating multiple tests to be updated.

10.2.3 System Tests

The system testing was done with a switch from the cisco devnet sandbox[1]. We did so by generating an output model from the "interfaces/interface/config" branch of the "openconfig-interfaces.yang" model. This model was then instantiated with values and serialized to JSON as seen in Figure 10.1.

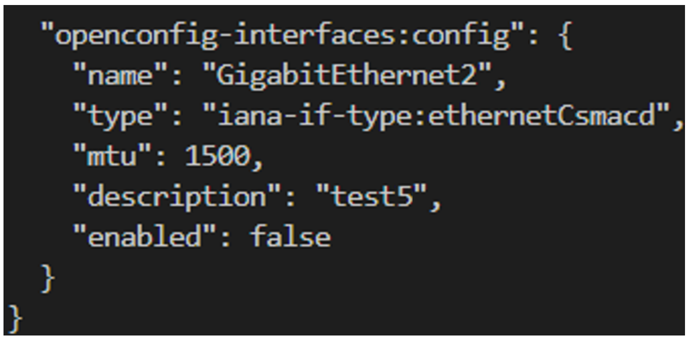

```python
if __name__ == "__main__":
    model = Model(
        config=ConfigContainer(
            name='GigabitEthernet2',
            description='test5',
            enabled=False,
            mtu=1500,
            type='iana-if-type:ethernetCsmacd',
        )
    )
```

```
Generated output: {
  "openconfig-interfaces:config": {
    "openconfig-interfaces:name": "GigabitEthernet2",
    "openconfig-interfaces:type": "iana-if-type:ethernetCsmacd",
    "openconfig-interfaces:mtu": 1500,
    "openconfig-interfaces:description": "test5",
    "openconfig-interfaces:enabled": false
  }
}
```

Figure 10.1: Model serialization during system test

[1]The devnet sandbox is free for everyone to use, to learn how to configure a product, development or API testing

After sending the JSON payload seen in Figure 10.1 to the sandbox switch via a RESTCONF-PATCH request, the resulting configuration received back from the switch was as seen in Figure 10.2. With this test, the concept was successfully proven to work.



```
  "openconfig-interfaces:config": {
    "name": "GigabitEthernet2",
    "type": "iana-if-type:ethernetCsmacd",
    "mtu": 1500,
    "description": "test5",
    "enabled": false
  }
}
```

Figure 10.2: Sandbox configuration after RESTCONF-PATCH request

10.2.4 Usability Tests

The usability testing will be done by the network engineers of the INS, as they are the ideal target audience for Pydantify. They will receive a zip-file with the project files and a word file with tasks and fields to give us feedback. As they are network engineers, they are supposed to solve the tasks with Pydantify without any external help, other than the readme and the CLI help command.

The usability test is planned to be done in Week 13, giving us enough time to fix minor imperfections and improving the provided hints as necessary. The Usability Test Protocol and the filled out word files can be reviewed in section B.2 Usability Test Protocols Results

Goal

The goal of the usability test is that all participants will rate the readme file, the help command and the IDE hints with a minimum score of 4 on a scale from 1 to 5. Also the participants should have now problems to complete the usability tests. If that is the case, we consider the non-functional requirement Usability as passed.

10.2.5 CI/CD Pipeline

To avoid "it works on my machine" scenarios, we added a CI/CD Pipeline for the Pydantify source code, as well as the documentation.

The documentation CI pipeline consists of two stages: "build_doc", which generates a PDF out of the latest LaTeX-files and "build_doc_diff", which builds a PDF in which the changes that were made since the last meeting are highlighted. This is especially helpful during meetings, as it makes demonstrating progress trivial.

The Pydantify source code CI/CD contains the "run_pytest" stage, which runs pytest and generates a code coverage report. If all tests pass, a second "deploy" stage takes said coverage report and hosts it on "GitLab Pages", allowing users to interactively navigate the source code with added coverage information.

10.2.6 Usability Test Result

As explained in subsection 10.2.4 Usability Tests, our usability testers were network engineers of the INS. In total there were 2 participants. The filled out usability test protocols can be found in section B.2 Usability Test Protocols Results.

Results

Both participants had no trouble in finishing our usability test protocol, as both gave us feedback that everything worked well. The feedback for the subtask **2-Hello World** was about the same in both cases. In one case it was mentioned that it should be possible to change the output name of the generated file if someone needs multiple pydantic models. The other feedback was that it would be beneficial if Pydantify printed the path to the output folder.

When asked how comprehensible the help in the readme, help command and IDE hints was, both participants answered with at least a 4.

Conclusion

While the sample size is admittedly small, we agree with the testers that it would be beneficial for the users to be able to change the output name and to print out the path of the output folder. Hence, we are planning to implement this after the submission of the term project.

The NFR Usability is passed, as we have received a minimum score of 4 in every aspect and the participants had no issues in completing the usability tests.

Part IV

Appendix

Appendix A

Assignment

A.1 Supervisor and Expert

This student project will be developed for the Institute for Network and Security at OST internally. It will be supervised by Urs Baumann (urs.baumann@ost.ch), OST.

A.2 Students

This project is conducted in the context of the module "Studienarbeit" in the department "Informatik" by:

- Dominic Walther
- Dejan Jovicic

A.3 Introduction

Since the introduction of NETCONF (rfc6241) and later RESTCONF (rfc8040) the data modeling language YANG (rfc6020) got very popular. YANG is the modeling language used to describe the data structure received through NETCONF or RESTCONF. The data itself is usually in JSON or XML. Network automation using this kind of model is also called **Model Driven Automation**. Model Driven Streaming Telemetry also takes advantage of YANG models. A collection of YANG models can be found in the following repository YangModels/yang

There has been a growing ecosystem around working with YANG models and some projects are already not maintained anymore. This is an incomplete list:

pyang https://github.com/mbj4668/pyang

libyang https://github.com/CESNET/libyang

pyangbind https://github.com/robshakir/pyangbind

ydk-gen https://github.com/CiscoDevNet/ydk-gen

yangson https://github.com/CZ-NIC/yangson

yang2swagger https://github.com/bartoszm/yang2swagger

Pydantic is a popular Python library for data validation using type hints.

A.4 Goals of the Project

The goal of this project is to show how YANG modules can be translated to `pydantic` models and so make model-driven network automation pythonic. After analysing the ecosystem and the technologies a proof-of-concept should be created to show how model-driven network automation can be done with pydantic. The generated models should be able to be used for sending and receiving RESTCONF configuration from a Cisco router or switch.

A.5 Documentation

This project must be documented according to the guidelines of the "Informatik" department. This includes all analysis, design, implementation, project management, etc. sections. All documentation is expected to be written in English. The project plan also contains the documentation tasks. All results must be complete in the final upload to the archive server. There is no need to print out the documentation

A.6 Important Dates

Date	Event
19.09.2022	Start of the student project
19.12.2022	Hand-in of the abstract using the online tool abstract.rj.ost.ch
23.12.2022 17:00	Final hand-in of the report using the online tool avt.i.ost.ch
TBD	Presentation

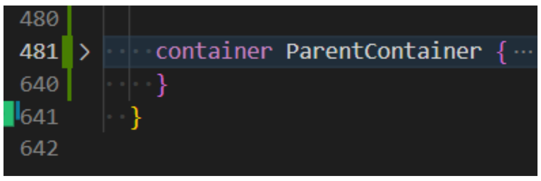
A.7 Evaluation

Criterion	Weight
Organization and implementation	20%
Formal quality of the report	20 %
Analysis, design and evaluation	20 %
Technical implementation	40 %

Appendix B

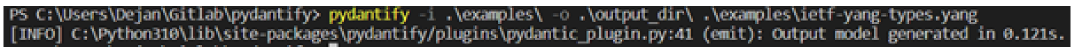
Screenshots

B.1 NFR Validation Screenshots



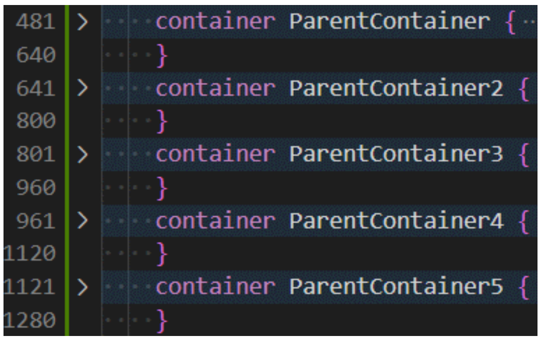
```
    container ParentContainer {⋯
    }
}
```

Figure B.1: NFR Performance Base YANG



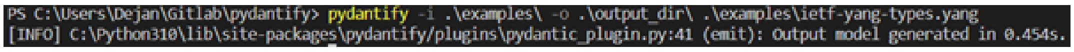
```
PS C:\Users\Dejan\Gitlab\pydantify> pydantify -i .\examples\ -o .\output_dir\ .\examples\ietf-yang-types.yang
[INFO] C:\Python310\lib\site-packages\pydantify/plugins\pydantic_plugin.py:41 (emit): Output model generated in 0.121s.
```

Figure B.2: NFR Performance Base YANG result



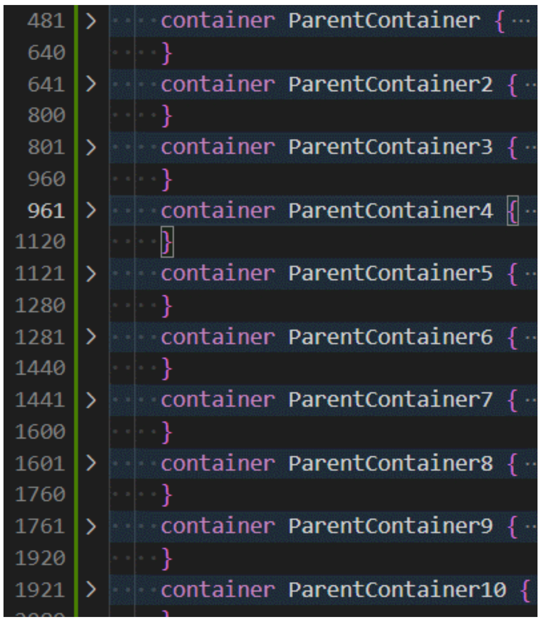
```
    container ParentContainer {⋯
    }
    container ParentContainer2 {
    }
    container ParentContainer3 {
    }
    container ParentContainer4 {
    }
    container ParentContainer5 {
    }
```

Figure B.3: NFR Performance 5xBase YANG



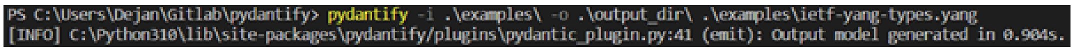
```
PS C:\Users\Dejan\Gitlab\pydantify> pydantify -i .\examples\ -o .\output_dir\ .\examples\ietf-yang-types.yang
[INFO] C:\Python310\lib\site-packages\pydantify/plugins\pydantic_plugin.py:41 (emit): Output model generated in 0.454s.
```

Figure B.4: NFR Performance 5xBase YANG result



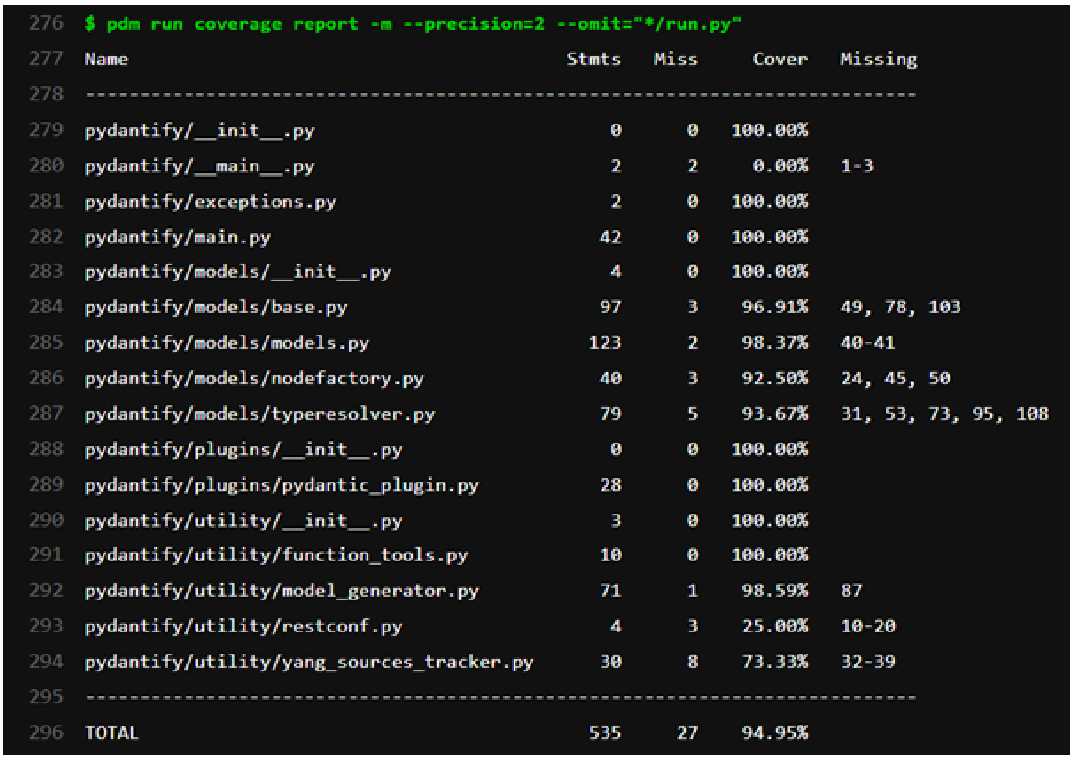
```
    container ParentContainer {···
    }
    container ParentContainer2 {···
    }
    container ParentContainer3 {···
    }
    container ParentContainer4 {···
    }
    container ParentContainer5 {···
    }
    container ParentContainer6 {···
    }
    container ParentContainer7 {···
    }
    container ParentContainer8 {···
    }
    container ParentContainer9 {···
    }
    container ParentContainer10 {
```

Figure B.5: NFR Performance 10xBase YANG



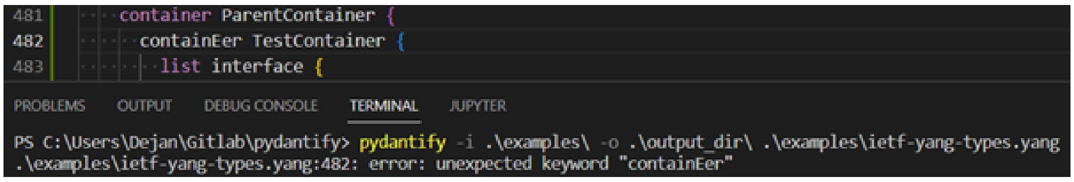
```
PS C:\Users\Dejan\Gitlab\pydantify> pydantify -i .\examples\ -o .\output_dir\ .\examples\ietf-yang-types.yang
[INFO] C:\Python310\lib\site-packages\pydantify/plugins\pydantic_plugin.py:41 (emit): Output model generated in 0.904s.
```

Figure B.6: NFR Performance 10xBase YANG result



```
$ pdm run coverage report -m --precision=2 --omit="*/run.py"
Name                                          Stmts   Miss    Cover   Missing
-----------------------------------------------------------------------------
pydantify/__init__.py                             0      0  100.00%
pydantify/__main__.py                             2      2    0.00%   1-3
pydantify/exceptions.py                           2      0  100.00%
pydantify/main.py                                42      0  100.00%
pydantify/models/__init__.py                      4      0  100.00%
pydantify/models/base.py                         97      3   96.91%   49, 78, 103
pydantify/models/models.py                      123      2   98.37%   40-41
pydantify/models/nodefactory.py                  40      3   92.50%   24, 45, 50
pydantify/models/typeresolver.py                 79      5   93.67%   31, 53, 73, 95, 108
pydantify/plugins/__init__.py                     0      0  100.00%
pydantify/plugins/pydantic_plugin.py             28      0  100.00%
pydantify/utility/__init__.py                     3      0  100.00%
pydantify/utility/function_tools.py              10      0  100.00%
pydantify/utility/model_generator.py             71      1   98.59%   87
pydantify/utility/restconf.py                     4      3   25.00%   10-20
pydantify/utility/yang_sources_tracker.py        30      8   73.33%   32-39
-----------------------------------------------------------------------------
TOTAL                                           535     27   94.95%
```

Figure B.7: NFR Testability Code Coverage

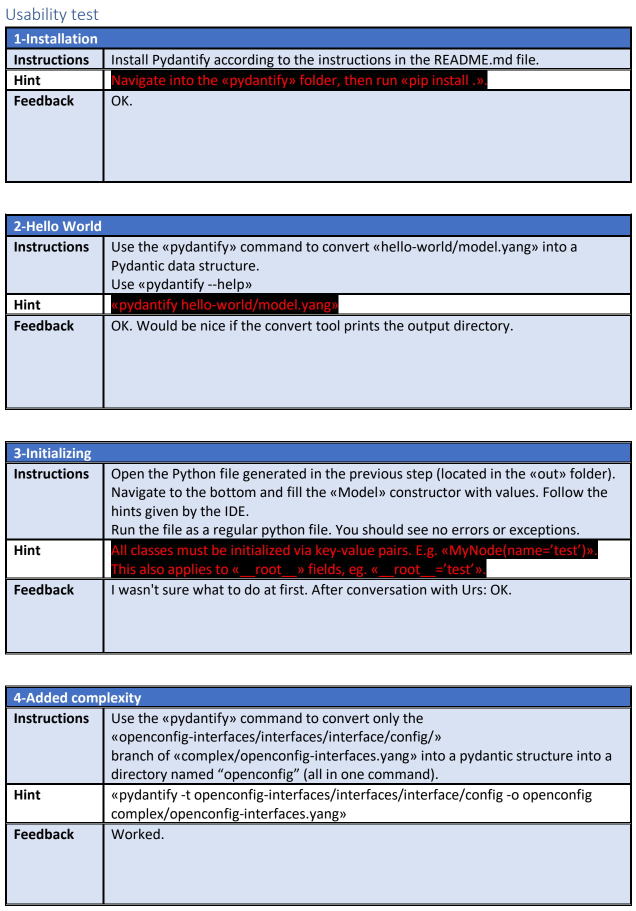


```
    container ParentContainer {
      containEer TestContainer {
        list interface {
```

```
PS C:\Users\Dejan\Gitlab\pydantify> pydantify -i .\examples\ -o .\output_dir\ .\examples\ietf-yang-types.yang
.\examples\ietf-yang-types.yang:482: error: unexpected keyword "containEer"
```

Figure B.8: NFR Reliability Unsupported Yang Statement

B.2 Usability Test Protocols Results



Usability test

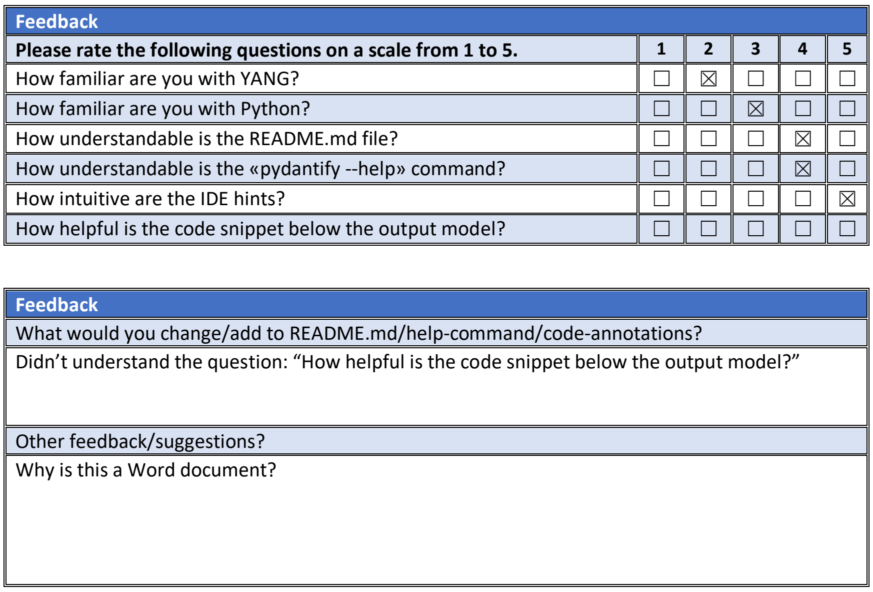
1-Installation	
Instructions	Install Pydantify according to the instructions in the README.md file.
Hint	Navigate into the «pydantify» folder, then run «pip install .».
Feedback	OK.

2-Hello World	
Instructions	Use the «pydantify» command to convert «hello-world/model.yang» into a Pydantic data structure. Use «pydantify --help»
Hint	«pydantify hello-world/model.yang»
Feedback	OK. Would be nice if the convert tool prints the output directory.

3-Initializing	
Instructions	Open the Python file generated in the previous step (located in the «out» folder). Navigate to the bottom and fill the «Model» constructor with values. Follow the hints given by the IDE. Run the file as a regular python file. You should see no errors or exceptions.
Hint	All classes must be initialized via key-value pairs. E.g. «MyNode(name='test')». This also applies to «__root__» fields, eg. «__root__='test'».
Feedback	I wasn't sure what to do at first. After conversation with Urs: OK.

4-Added complexity	
Instructions	Use the «pydantify» command to convert only the «openconfig-interfaces/interfaces/interface/config/» branch of «complex/openconfig-interfaces.yang» into a pydantic structure into a directory named "openconfig" (all in one command).
Hint	«pydantify -t openconfig-interfaces/interfaces/interface/config -o openconfig complex/openconfig-interfaces.yang»
Feedback	Worked.

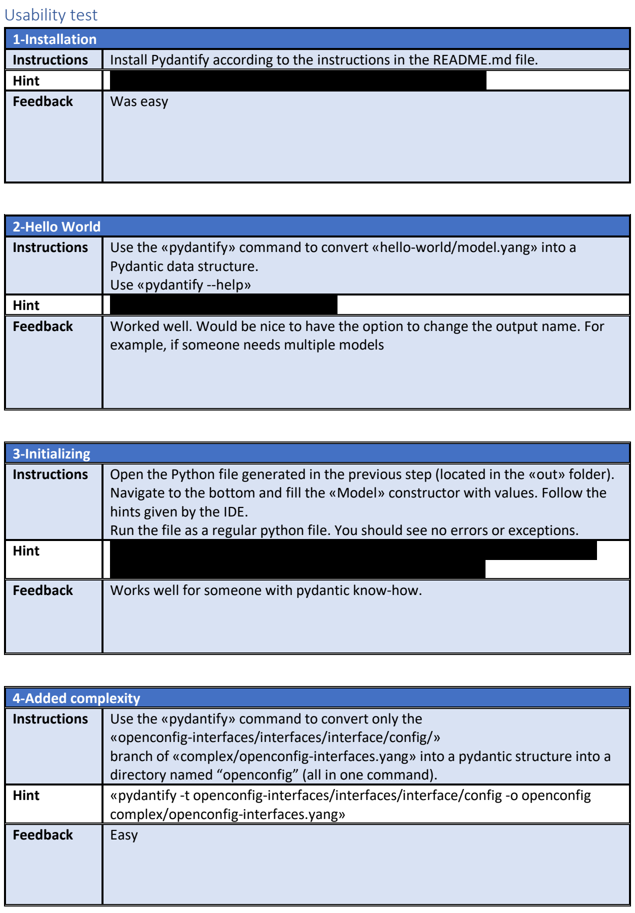
Figure B.9: Usability Test Form 1



Feedback					
Please rate the following questions on a scale from 1 to 5.	**1**	**2**	**3**	**4**	**5**
How familiar are you with YANG?	☐	☒	☐	☐	☐
How familiar are you with Python?	☐	☐	☒	☐	☐
How understandable is the README.md file?	☐	☐	☐	☒	☐
How understandable is the «pydantify --help» command?	☐	☐	☐	☒	☐
How intuitive are the IDE hints?	☐	☐	☐	☐	☒
How helpful is the code snippet below the output model?	☐	☐	☐	☐	☐

Feedback
What would you change/add to README.md/help-command/code-annotations?
Didn’t understand the question: “How helpful is the code snippet below the output model?”
Other feedback/suggestions?
Why is this a Word document?

Figure B.10: Usability Test Feedback 1



Usability test

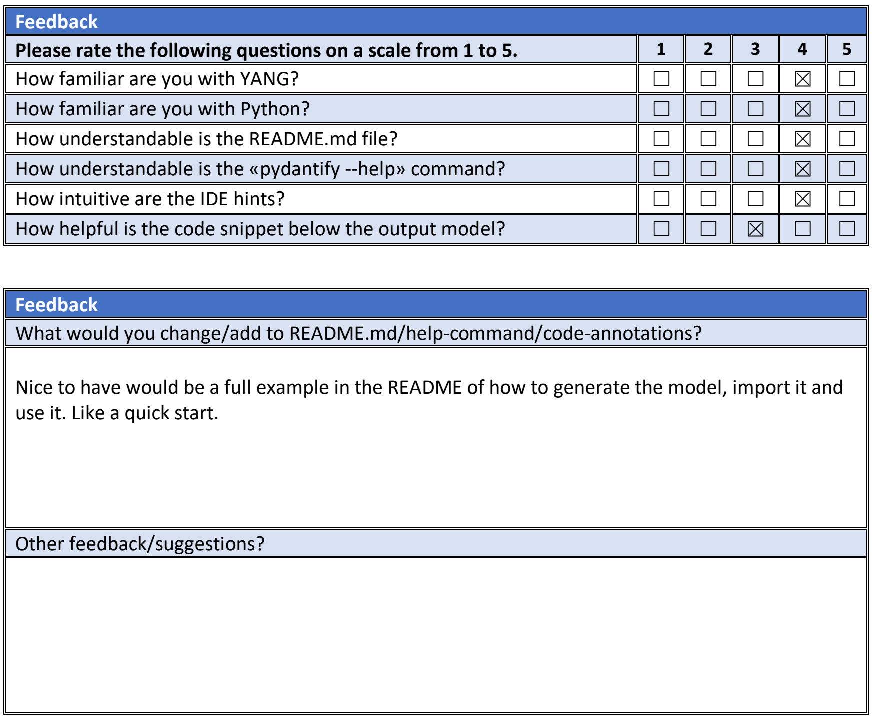
1-Installation	
Instructions	Install Pydantify according to the instructions in the README.md file.
Hint	[redacted]
Feedback	Was easy

2-Hello World	
Instructions	Use the «pydantify» command to convert «hello-world/model.yang» into a Pydantic data structure. Use «pydantify --help»
Hint	[redacted]
Feedback	Worked well. Would be nice to have the option to change the output name. For example, if someone needs multiple models

3-Initializing	
Instructions	Open the Python file generated in the previous step (located in the «out» folder). Navigate to the bottom and fill the «Model» constructor with values. Follow the hints given by the IDE. Run the file as a regular python file. You should see no errors or exceptions.
Hint	[redacted]
Feedback	Works well for someone with pydantic know-how.

4-Added complexity	
Instructions	Use the «pydantify» command to convert only the «openconfig-interfaces/interfaces/interface/config/» branch of «complex/openconfig-interfaces.yang» into a pydantic structure into a directory named "openconfig" (all in one command).
Hint	«pydantify -t openconfig-interfaces/interfaces/interface/config -o openconfig complex/openconfig-interfaces.yang»
Feedback	Easy

Figure B.11: Usability Test Form 2



Feedback					
Please rate the following questions on a scale from 1 to 5.	**1**	**2**	**3**	**4**	**5**
How familiar are you with YANG?	☐	☐	☐	☒	☐
How familiar are you with Python?	☐	☐	☐	☒	☐
How understandable is the README.md file?	☐	☐	☐	☒	☐
How understandable is the «pydantify --help» command?	☐	☐	☐	☒	☐
How intuitive are the IDE hints?	☐	☐	☐	☒	☐
How helpful is the code snippet below the output model?	☐	☐	☒	☐	☐

Feedback
What would you change/add to README.md/help-command/code-annotations?
Nice to have would be a full example in the README of how to generate the model, import it and use it. Like a quick start.
Other feedback/suggestions?

Figure B.12: Usability Test Form 2

Bibliography

[1] A. Bierman, M. Bjorklund, and K. Watsen, “Restconf protocol,” Jan 2017, last time accessed: 28/10/2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8040#section-4

[2] M. Bjorklund, “Yang - a data modeling language for the network configuration protocol (netconf),” Oct 2010, last time accessed: 28/10/2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6020.html#section-4

[3] ClausHolbechArista, “Use schema description to populate field docstring,” Sep 2022, last time accessed: 21/11/2022. [Online]. Available: https://github.com/koxudaxi/datamodel-code-generator/issues/857

[4] S. Colvin, “Model config - pydantic,” Oct 2022, last time accessed: 28/10/2022. [Online]. Available: https://pydantic-docs.helpmanual.io/usage/model_config/#options

[5] ——, “Pydantic v2 plan - pydantic,” Nov 2022, last time accessed: 19/11/2022. [Online]. Available: https://pydantic-docs.helpmanual.io/blog/pydantic-v2/#strict-mode

[6] P. V. B. et al., “Xml schema part 2 Datatypes second edition,” Oct 2004, last time accessed: 1/12/2022. [Online]. Available: https://www.w3.org/TR/2004/REC-xmlschema-2-20041028/#regexs

[7] kwatsen, “yangson/test_model.py at master cz-nic/yangson,” Jan 2021, last time accessed: 4/12/2022. [Online]. Available: https://github.com/CZ-NIC/yangson/blob/1c632c7a4f76956f02173b5f269ccf1c16df9153/tests/test_model.py#L1124

[8] L. Lhotka, “Introduction - yangson 1.4.15 documentation,” Mar 2016, last time accessed: 4/12/2022. [Online]. Available: https://yangson.labs.nic.cz/introduction.html

[9] C. Lunsford, “pyang is not executable when using the windows command prompt as your shell,” Apr 2017, last time accessed: 28/11/2022. [Online]. Available: https://github.com/mbj4668/pyang/issues/310

[10] A. Moore, “regex - combine regexp? - stack overflow,” May 2009, last time accessed: 19/11/2022. [Online]. Available: https://stackoverflow.com/a/870506

[11] D. Walther, “Fix deprecation warnings around ”copy_on_model_validation”,” Nov 2022, last time accessed: 21/11/2022. [Online]. Available: https://github.com/koxudaxi/datamodel-code-generator/pull/927

[12] ——, “Fix indents for multiline docstrings,” Dec 2022, last time accessed: 23/12/2022. [Online]. Available: https://github.com/koxudaxi/datamodel-code-generator/pull/938

[13] ——, “Implement field descriptions as docstrings,” Nov 2022, last time accessed: 21/11/2022. [Online]. Available: https://github.com/koxudaxi/datamodel-code-generator/pull/918