CM model view transformations

To support runtime forward/backward compatibility

Sebastian Nödtvedt
Abstract

CM model view transformations to support runtime forward/backward compatibility

Sebastian Nödtvedt

The task to implement a solution of handling updates and version discrepancies within a testbeds Configuration Management. The Ericsson 5G testbed is built to support deployment of higher layer functions in a cloud environment.

The benefits of using cloud deployments is mainly that it enables elastic application that can grow and shrink its footprint in runtime to adjust the capacity according to the traffic load.

Schema data associated with different versions of a document-oriented database within a cloud environment provides dynamic properties but what remains static and cumbersome is updating parts of the system.

If one can resolve this is then newer versions of functions can be instantiated in runtime and in parallel with older versions which partially can remove the need for application and system upgrades.

However, this puts completely new demands on the architecture and how support functions are designed. One such support function is the configuration management function which in the current 5G testbed system is seen as an infrastructure function that can be replaced and upgraded independently of other running traffic applications.

This requires handling of forward and backwards compatibility between the configuration management function and traffic functions that consume the configuration data.

In this report a prototype was constructed and tested, the prototype consists of mainly two core components. Firstly a Wizard which handles two different versions of a model and generates a transformation schema, this is then passed to the Transformation which does the needed data transformation for compatibility. The Wizard starts by ensuring the required data is compatible and additionally acts as a interactive tool for an operator, providing an overview and insight into the data transformation.

A solution within the frames of being a proof of concept was successfully implemented and demonstrated, inherent limitations where taken into account in the design.

In conclusion a feasible solution is possible to implement for resolving version management within a system like the 5G testbed, which reduces a otherwise slow and error prone manual process.
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>A example of such a instance with the type rbu</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>CLI menu</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>A segment of nr-cells schema</td>
<td>15</td>
</tr>
<tr>
<td>A.1</td>
<td>Transform schema</td>
<td>22</td>
</tr>
<tr>
<td>A.2</td>
<td>Objects within a view instances</td>
<td>24</td>
</tr>
<tr>
<td>A.3</td>
<td>The result of the transformation</td>
<td>26</td>
</tr>
<tr>
<td>A.4</td>
<td>amf example</td>
<td>28</td>
</tr>
<tr>
<td>A.5</td>
<td>vl example</td>
<td>28</td>
</tr>
<tr>
<td>A.6</td>
<td>vl example</td>
<td>28</td>
</tr>
<tr>
<td>B.1</td>
<td>Upgrade value function within the transform implementation</td>
<td>29</td>
</tr>
<tr>
<td>B.2</td>
<td>Finds the instances to a corresponding type</td>
<td>30</td>
</tr>
<tr>
<td>B.3</td>
<td>The parse function within the Wizard</td>
<td>31</td>
</tr>
<tr>
<td>C.1</td>
<td>The delta between two model versions</td>
<td>33</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Moving from traditional computation to cloud computing and the use of *OS-level virtualisation*, also called containers, in the combination of *container-orchestration* that provides automation of application deployment and lastly scaling and management.

This provides flexibility and elastic runtime footprint of the system, but undesirable properties still remains. One of these properties are rolling out updates to a running system, within the project this entails updates to what is called *configuration model*. This model, which the applications reads is called a *view*, usually provides a subset of the objects and their relations between each other. The model that resides within Ericsons 5G testbed.

When a new update makes the previous running instance versions incompatible this results in the need to bring down the live applications to be updated, to ensure compatibility with the new model.

The solution is to transform the data sent between the database and the application to be compatible with the new model. This will lead to backwards compatibility and the instance of the application can be kept running.

The development took place under the Technology and Research Department, which is responsible for soft and hardware concepts and standardisation’s and lastly testbeds and prototypes at Ericsson.

The project was conducted at the telecommunication company Ericsson. It revolves around implementing additional functionality to a existing 5G testbed. The main goal is to develop and test additional functionality to a so called *view* (seen in Figure 1.1) to the testbed *Configuration Management*. This is done with data transformation, which will grant modularity and compatibility between versions.

This functionality is something that doesn’t currently exist within the testbed and would provide the possibility to update applications during runtime.

Traditionally adding a feature or a minor change is done by a system upgrade which involves updating references and properties across the whole system. The desired functionality should provide the flexibility to roll out updates while keeping the system up and running, while providing compatible data.

Alternatively, one can use a Configuration Management service which can be upgraded where the configuration model consists of different fragments of the applications "owns". This means that an upgrade of an application also upgrades it’s corresponding model fragment.
1.1 Background

The Ericsson’s 5G radio testbed is a platform used for innovation and testing of both soft and hardware solutions. This enables operators to base trials on a live network environment. A overview of the architecture of the CM service (Configuration Management) can be seen in Figure 1.1.

1.1.1 CM Service

The CM service is a system service implemented as a container that includes both the CM back-end and CM front-end together with one or more models and optionally one or more views. The CM service container requires a external database for configuration object storage.

1.1.2 CM back-end

The CM back-end exposes three north-bound RESTful APIs for managing the configuration stored in the CM service. The CM back-end loads one or more models which define what configuration objects that can be configured and their contents. The CM back-end is implemented as part of the ”oam-backend” application.

1.1.3 CM front-end

The CM front-end is a service that provides a south-bound RESTful API towards system internal services and functions. The CM front-end loads one or more view specification to provide a set of views that allows services and functions to read configuration data relevant to that service or function without having to resolve references and access objects individually. The CM front-end is implemented as part of the ”oam-backend” application.

1.1.4 Models

A model is a description of allowed object types and their relations. The model also contains schemas that define attributes that can be configured on each object type. Models are loaded into the CM back-end and decides what objects and attributes are possible to configure through the northbound interfaces. There is no requirement for a single root object in a model, therefore objects instances created in the scope of a model instead from one or more graphs.

1.1.5 Views

A view contains rules that describes how a graph of object instances should be transformed into a tree view rooted at a selected object instance. The CM front-end uses the views to give services and functions in the system possibility to read a set of related configuration objects in a single request by using the rules defined in the views to resolve references between objects and expand those objects into a single view.
1.1.6 OAM-backend

The oam-backend is the application that implements both the CM back-end and CM front-end.

1.1.7 Database

The CM back-end uses an external database as is its back-end object store, written in MongoDB.

1.2 Purpose

The task at hand is to implement and test view transformation within the testbeds Configuration Management. The view will manage the output data from the front-end to be compatible with the current version of an application.

When the model (seen in Figure 1.1) has been updated during runtime, objects using the views API will run into compatibility issues. If required properties has been altered in certain ways, this could be address fields or data-types.

The Ericsson 5G testbed is built to support deployment of higher layer functions in a cloud environment. The benefits of using cloud deployment is mainly that it enables an elastic application that can grow and shrink its footprint in runtime to adjust the capacity according to the traffic load. An disaggregated and elastic application can also be built to better cope with application and infrastructure failures, by using geographic redundancy, fault resilience can also be improved.

Another benefit is that newer versions of functions can be instantiated in runtime in parallel with older versions which partially can remove the need for application and system upgrades.

However, this puts completely new demands on the architecture and how support functions are designed. One such support function is the configuration management function which in the current 5G testbed system is seen as an infrastructure function that can be replaced and upgraded independently of other running traffic applications.

This requires handling of forward and backwards compatibility between the configuration management function and traffic functions that consume the configuration data.
The back-end component provides the REST interface which is accessible for the owner of that system, usually called an operator. A Management system built to work with this interface could be anything from a CLI interface to a user interface in a web-browser.

The model decides what configuration objects which can be configured through the "northern" REST interface, what attributes a object consists of, what limitations or values a attribute has, and relationships between different objects.

The REST interface that the front-end provides to the applications gives the possibility for the applications to read the configuration through the view. Through the view, the application can specify how the objects in the model can expand to a composite view so the application does not need to retrieve a object to retrieve its configuration.

Instead it can make a request and get the complete configuration that is relevant to the application.

The issue with updating the CM back-end on a live system is providing compatibility as previously mentioned. This will have the effect of a front-end which provides incompatible views to live instance that relies on reading the configuration. Here a implementation which will take a incompatible view and transform it to a compatible view. Firstly the database which holds the objects will receive a update, from here a delta between the schemas will arise. The delta can then be used to figure out how to provide a compatible view if possible.

### 1.3 Requirements

The first requirement will be the ability to apply a transformation to a view instances. This will be done by parsing and applying the needed changes for compatibility. The parsing and transformation is affected by the model within the testbed, that is specified in the JSON-Schema (JavaScript Object Notation Schema) \[4\] \[5\].
The second requirement is to use *Golang* \[6\] \[7\] for the implementation. The reason is because parts of the system are written in *Golang* within the testbed. This makes it reasonable to keep a consistent *code-base*, for future development and compatibility.

Lastly a way to generate a *schema* by looking at two different models which the previously mentioned implementation will apply to an instance. A difference from the transformation implementation is that it will be written in *Python* \[8\] \[9\]. The reason is that the external library support and a lower development time for a language such as Python makes it suitable.

This can be broken down into two major parts which are illustrated in Figure 2.1. These are a ”Wizard (software)” for making the process easier and quicker for the operator, and applying the transformation generated by the wizard onto the view.

### 1.4 Delimitation

Since this project is intended to be a proof of concept the requirement is not having a implementation to work with a live system and applying real-time view transformations.

The data which the implementation will be tested against is user generated test, and not data from a live deployment. Field testing is considered out of scope for this project.

Automatic generation of a transformation schema comes with limitations within the scope of this project as well. The values for some keys cannot be automatically generated and set while providing function. These cases will require manual input from an operator before applying the transformation, such cases could be address fields or unique *ID* fields. These fields will need manual input from the operator who is rolling out the update.
Chapter 2

Design

The User or otherwise known as a operator within Figure 2.1 will supply the project solution with a updated model. The two models are then compared to one and another to spot differences that needs to be addressed to ensure compatibility within the Wizard. The comparison and the parsing of the JSON Schema model (seen in A.1), which all the objects within the testbed follows. This will dictate later how the parsing is done within the implementation. The functionality to automatically generate this schema by looking at two different versions of the model would reduce the time for manual input which is a time consuming and error prone process. The section of interest is the schema section of each JSON object which describes the properties of a object, while the objects section describes the relations to other objects the object in question has. Automatic schema generation primarily looks into a objects required fields and its corresponding properties, these are expected by the live application. By ensuring that these fields are provided to the live instance to take a running instance down. Knowing this the generator can be designed with presumptions and look for changes within certain keys and corresponding values of interest within a objects schema. That makes the generation of a transformation schema feasible.

Additionally by looking at the revision history of the model where the following core changes, which I chose to describe as the following (can also be seen in Figure 3.1). New additions to the model, Remove a existing property within the model and lastly either a Value or Key change of a pre-existing property.

With the specification of the transformation model done and the generation of the transformation schema from the Wizard finished, the next step simply called Transform starts. The model is passed and parsed by the Transform and applies necessary changes needed to the corresponding View instances.
Chapter 3

Implementation

This chapter will go more into the previously shown overview (Figure 2.1) and details around the implementation. Which consists of two major parts, the Transform 3.1 and the Wizard 3.2. The software design and within the implementation objects is described and illustrated with UML (Unified Modeling Language) diagrams, snippets of code which either shows parts of the solution or the JSON objects and schemas.

The first part of the implementation is to define the structure of what will be called the transformation object. The transformation object will be read from a file written in a JSON Schema and be known as the transformation schema. This will be parsed into a transformation object, the structure of this is based on the findings of the commits and changes done to the model during its development.

The figure 3.1 illustrates how the model will look internally within the Golang transform implementation, a corresponding JSON representation is seen in Appendix A.2.

Figure 3.1: Overview of the transformation objects within the Golang implementation.
The most important datatype is what is seen simply as Change, which holds either what is to be removed or added and lastly changed. This is all returned from the parseTransformation function which have parsed the transformation schema generated from the Wizard.

3.1 Transform

The first step before applying the transformation is done by parsing the schema generated by the Wizard. The schema is parsed into a corresponding Golang data-structure. Once the transformation schema is parsed its only a matter of applying these changes onto the view instance.

The changes that will be applied on the incoming instances depends on the resulting object structure which is returned by the parseObject function.

![Figure 3.2: The program structure for the Transform implementation.](image)

The program will iterate through each Object and each Change that the object has, and add these changes to the Golang struct. If an object is either defined to be added or removed then the parsing ends and returns the parsed object. The other outcome is that the object is neither a new addition or a object up for removal but rather an existing object that has been changed.

Each type of change such as value or key change is checked and added to the respective objects transformation, which is applied after there is no more objects or changes to be parsed.

What ended up being used extensively within the transformation is the use of Golangs interfaces. The interface is a so called method set within Golang, which provides:

“A variable of interface type can store a value of any type with a method set that is any superset of the interface.” [10].

With this functionality at hand one can use the properties of evaluating interface variations. Such variations are []interface which is used in the context of check in runtime if a key holds a value which is a list of values. Other cases where its a matter to determine
if a schema is nested or not where checking if its a interface or map[string]interface

This can for example be seen with the upgradeValue in Appendix B.1 function. The function firstly checks if the upgrade will be applied to an instance or a model. Within this function one can see three different types of if statements, where a interface is checked for its properties.

The ability to upgrade models and not only instances by using the transform as implemented but no wizard for it was created. It would exceed the scope and time of the project, but it was left out for future development.

Finding each instance and applying the changes is done by pattern matching with a regular expression. The reason is that instances are described by a type and a unique ID. Several instances of the same type can be running within a system simultaneously and hence the need to label these instances with a unique ID.

Listing 3.1: A example of such a instance with the type rbu

```
"rbu/5a7d68b7f2916415c74f2f05": {
    "ip_address": "10.10.22.4",
    "name": "indoor-ki09-1"
},
"rbu/5a7d68b7f2916415c74f2f07": {
    "ip_address": "10.10.23.4",
    "name": "indoor-ki09-2"
},
"rbu/5a7d68b8f2916415c74f2f09": {
    "ip_address": "10.44.39.4",
    "name": "outdoor-ki09-1"
},
```

The IDs compose of 24 characters consisting of letters and numbers. The function findInstances will look for one or more character until a white-space character appears, which will be a quotation mark within this context. All the instances of a certain type is matched and added to a array of the type string, and the implementation for finding all the instances of a type can be seen in Appendix B.2.

With all instances of the correct type gathered the changes which has been parsed from the transformation schema can now be applied.
3.2 Wizard

The purpose of the wizard is to automate the process of figuring out the differences between model versions and removing the need to write a transformation schema by hand to handle a model updates.

The wizard will firstly display a simple CLI (Command Line Interface) menu, where the user can chose to generate a transformation schema by comparing by two models of different versions. The ability to preview and edit a generated schema by the wizard can be chosen. Which enables the operator to fill out the values which couldn’t be automatically generated by the wizard. Such values could be address fields or name fields which needs to be unique and correctly assigned to function correctly. The last menu option allow the operator to exit the wizard.
When the first menu option is selected two models will be compared to each other, if there is no difference then no transform schema will be generated. If there is a difference between these two models it will be a matter to firstly check if this delta is a schema change and not a object relation change. We are interested in object properties changes, not the object relations within the testbed.

What comes after this is, if a property has been deleted, or a new property has been added. Then an existing property has been updated that is then passed to the parse function.

The first changes of interest are deleted properties and newly added properties to the object, secondly changes to existing fields are parsed.

The functions within this implementation of interest is the parse function. It will firstly look into updating a objects property schema. This will be used to do look ups if anything within required has changed and needs updating. This will then be added to the objects transformation schema.

The parse function which is one of the core functions of the wizard implementation works by parsing in the delta of which consists the changes of a objects properties and required field.

Within the parse function (Appendix B.3) the difference (Appendix C.1) of the objects properties is parsed. From here changes in the required field is of interest. The reason is that everything within the required field is, as the name implies required. This means that when an application sends a GET. Those fields needs to be within the view instance.
The difference Appendix C.1 displays a scenario where the objects \textit{rhu}, \textit{vim} and lastly \textit{vnf} are all updated. The tags \texttt{$insert$} and \texttt{$delete$} implies that the old model needs to add and remove certain properties within these objects.

What can be seen is that removals is usually within the \texttt{required} field, an exception is seen under \textit{vnf} where the property \texttt{vim} is removed.

What was used extensively throughout the wizard is the use of pattern matching. The following example simply assigns the variable \texttt{up} with the value of the key \texttt{$update$}.

```python
if match(delta, {'$update': _}, lambda update: update, default=False):
    up = (match(delta, {'$update': _}, lambda update: update))
```

Chapter 4

Result

The resulted project was successfully implemented and delivered according to the design seen previously in Figure 2.1. The test cases where handwritten and ran on a local machine, which means the implementation was not field tested with real-time instances.

The intention of the project was a proof of concept, to show the possibilities as well the inherent limitations of a solution to a problem like this. The focus was on the core functionality, which is taking a transformation schema and applying it to a view instance.

4.1 Testing

Testing was conducted continuously during development, each function had their own written unit tests and lastly a system test which runs through the whole implementation. Each unit test works by mock up objects and mock up schemas which are then asserted to an expected output.

Establishing early on how each function and unit tests should work eased the development and functional correctness, by being able to quickly test and verify.

The demonstration conducted a unit test which covered all possible changes that the transformation schema and wizard could handle. A sample demonstration from can be seen in Appendix A.3.

A example of these changes are the following:

Listing 4.1: A segment of nr-cells schema.

```
"nr-cell": {
    "$schema": "http://json-schema.org/draft-04/schema#",
    "$id": "3gpp/nr-cell",
    "title": "NR Cell",
    "description": "Object representing a NR cell group in the system ..."
}```
Type: A type change is when one changes in this case \texttt{nr-cell} seen on the very first line to \texttt{nr.Cell}.

Key: A key change is simply when a certain key is changed, for example if "description" in Listing 4.1 is change to "desc".

Value: Similarly to a key change a value change is where the right hand side of a key and value pair is changed. A example where "title":"NR Cell" is changed into "title":"NR-Cell".

New: Constitutes something added to a existing schema, which is something with a key and a corresponding value.

Remove: The removal of a property within a schema, this could be any key with its corresponding value.

NewObject: Adding a new object to the schema with all of its corresponding properties.

RemoveObject: Removing a existing object from the schema.

4.2 Evaluation

Lastly the differences that the transformation applied was shown with a diff tool, in this case \texttt{vimdiff} was used.

The ability to handle updates with more ease and keeping a running system alive was something that didn’t previously exist within the 5G testbed.

Finding a solution which would remove the slow and error prone process to and answering the question if a practical solution could be done. How strong are the limitations before it becomes unpractical to use or infeasible to implement? Where the key questions.

Being able to handle version updates with more ease and without halting a running system was something that previously didn’t exist within the 5G testbed. The unanswered questions if a practical solution could be done. And how strong are the limitations before it becomes unpractical to use or infeasible to implement?

The project was written to match the existing implementation and to leave the possibility to have future development or additional features added to it.

Updates can be released with a transform schema along side it, or an operator can use the Wizard to quickly generate an outline for a transformation schema.

Which satisfied the intention of the project, by firstly showing that there is a practical way to handle model changes. Secondly that there is a way to generate the needed transformation schema, and lastly applying all of this onto an view instance.
Chapter 5
Related Work

A way of handling versions of a relational database schema can be seen in the work of Leela S. Tamma and Krishna Vitaldevara called "Relational database schema version management" [15]. Where different techniques for handling versions within a relational database are demonstrated. Laws of set theory was used to identify in which order the installation scripts should be executed to create a full install of a certain version of the database. The functionality to upgrade between several versions was left out as future work for this project. And the focus was aimed at providing a tool that handles updates in a graceful manner to a live system. The reason for finding a way to automate is to ease the process of updating a database which traditionally is done manually. By providing either automation for updating or tools for keeping up time in an elastic system reduces the otherwise slow and error prone process of doing it manually. Comparatively to this project the focus was to transform a view that is requested to be compatible instead of updating a whole database. Where in this project the aim was to support certain containers to be either backwards or forward compatible in a live system.

The generation of a view schema graph and supporting multiple views titled "Automatic view schema generation in object-oriented databases" [16] written by Elke A. Rundensteiner and Lubomir Bic. Where a solution was sought out for view generation within an object-oriented database (OODB), and the concept of MultiView is created. The problem arises where a schema has to be consistent with semantics of each respective class and the relationships amongst these classes. Manual typing can introduce errors, errors being the view schema implying incorrect property inheritance between classes which are not related in the global schema. Introducing automatic generation with validity checks removes a error prone process form the system. Two algorithms are presented the first with linear complexity but assumes that the global schema has a tree structure. Lastly the second algorithm which does not rely on a assumption but at the cost of a higher computational complexity.
In this project a solution for handling model changes within a real time system was implemented. A back-end solution for applying the changes was created, along with a front-end solution to provide the necessary changes for compatibility.

Since the time frame of the project was limited, actual field testing was left out, instead unit testing was done to confirm functional correctness. Several limitations where inherently from the start, but was handled with a best effort solution. The situation, where no conventional program can predict or parse certain values within the model schema, where handled by providing default values of the correct type.

This ties into writing software which is able to understand context, the concept touches upon fields relating to AI (Artificial intelligence) [17] and the sub fields such as NLU (Natural-language understanding) [18] and lastly NLP (Natural language processing) [19].

If one where to implement a solution which would roll out updates, by parsing in this case the model difference, but also parse the commit message and to gain context from it. This would quickly tip the projects size and implication into something far greater with questionable practicality.

The requirement of a large data set if one has any hopes to apply a statistical model, instead of a rule-based solution would require that the model would go through a multitude of changes. This would be unlikely within the testbed, to see major changes at a frequency that would lead to a any useful amount of data.

This is far beyond a feasible solution for a project of this scope and time-span.

This wasn’t obvious at first glance and came to light after a completed technical design was done, and reflecting around possible solutions was done. Several scenarios could not be solved by traditional software, but required a solution which understands the context and intention of an update.

So the solution of setting default values of a certain type was instead implemented, leaving it up to the operator to fill in what the wizard could not. This leaves it at a practical middle ground.
Chapter 7

Future Work

The purpose of the project was to see if there was a convenient way to handle model changes within this particular system, and if that was the case, what are the limitations?

A more extensive solution would provide a wizard for the schema that would flag if there were updates which requires the operator to provide the expected values. This would further improve the chance of rolling out a compatible transformation schema and thus keeping the application compatible.

Additional testing would require a live system with incoming view instances, and requests with running applications and applying real-time model transformations. This would further prove the possible appliance of this implementation.

A possible addition would be incorporating a diff tool into the wizard. This would show the difference of the current instance and what will be applied to it. With this a operator would get a clearer overview over what will be done and being able to evaluate correctness of the generated schema.

Further refinement would allow an operator to have version control functionality within the CLI menu which the wizard came with. This functionality could be the ability to see commit history and logs for the current model, which provides context to the changes.

Smaller additions such as being able to pick and choose models and run through a comparison and generation directly in the wizard could also be implemented given more time.
Appendix A

JSON-Schema Examples

A.1 Model example

```json
{"$model": "http://ericsson.net/5g-...",
"name": "3gpp",
"version": "0...",
"description": "Ericsson 5G NR testbed configuration model",

"objects": {
"example": {
    "abstract": false,
    "implements": [],
    "parents": [],
    "children": [],
    "references": {},
    "schemas": {"$ref": "/schemas/example"}
        },
    ... 
},
"schemas": {
    "example": {
    "example": {
    "valid JSON schema according to draft... 
},
    ... 
} 
}
```
A.2 Transform example

```json
{
  "object": {
    "type": "example",
    "version": 1.0,
    "removeobject": false,
    "newobject": false,
    "schemaorobject": false,
    "instances": false,

    "changes": {
      "type": "example",
      "key": [{
        "newkey": "new",
        "oldkey": "old"
      }],
      "value": [{
        "key": "test",
        "value": "test"
      }],
      "remove": [{
        "key": "test"
      }]
    }
  }
}
```
A.3 Demo example

Listing A.1: Transform schema.

```json
{
    "objects": [
        {
            "vnf": {
                "changes": [
                    {
                        "new": [
                            {
                                "key": "parameters",
                                "value": "object"
                            }
                        ],
                        "remove": [
                            {
                                "key": "vim"
                            },
                            {
                                "key": "vnf_id"
                            }
                        ]
                    }
                ],
                "instance": true,
                "newobject": false,
                "removeobject": false,
                "schemaorobject": false,
                "type": "vnf"
            }
        },
        {
            "rbu": {
                "changes": [
                    {
                        "new": [
                            {
                                "key": "password",
                                "value": "string"
                            }
                        ]
                    }
                ],
                "instance": true,
                "newobject": false,
                "removeobject": false,
                "schemaorobject": false,
                "type": "rbu"
            }
        }
    ]
}
```
"newobject": false,
"removeobject": false,
"schemaorobject": false,
"type": "rbu"
}

"vim": {
  "changes": [
    {
      "value": [
        {
          "key": "id",
          "value": "integer"
        }
      ]
    }
  ],
  "instance": true,
  "newobject": false,
  "removeobject": false,
  "schemaorobject": false,
  "type": "vim"
}
}
Listing A.2: Objects within a view instances.

```
{
    "instance": "ns/5a7d68b8f2916415c74f2f0f",
    "objects": {
        "ns/5a7d68b8f2916415c74f2f0f": {
            "name": "cu-1",
            "pnfs": [
                "rbu/5a7d68b7f2916415c74f2f05",
                "rbu/5a7d68b7f2916415c74f2f07",
                "rbu/5a7d68b8f2916415c74f2f09",
                "rbu/5a7d68b8f2916415c74f2f0b",
                "rbu/5a7d68b8f2916415c74f2f0d"
            ],
            "vnfs": [
                "vnf/5a7d68b8f2916415c74f2f13",
                "vnf/5a7d68b8f2916415c74f2f17",
                "vnf/5a7d68b8f2916415c74f2f1b",
                "vnf/5a7d68b8f2916415c74f2f1f",
                "vnf/5a7d68b8f2916415c74f2f23"
            ]
        },
        "rbu/5a7d68b7f2916415c74f2f05": {
            "ip_address": "10.10.22.4",
            "name": "indoor-ki09-1"
        },
        "rbu/5a7d68b7f2916415c74f2f07": {
            "ip_address": "10.10.23.4",
            "name": "indoor-ki09-2"
        },
        "rbu/5a7d68b8f2916415c74f2f09": {
            "ip_address": "10.44.39.4",
            "name": "outdoor-ki09-1"
        },
        "rbu/5a7d68b8f2916415c74f2f0b": {
            "ip_address": "10.10.24.4",
            "name": "lab-ki09-1"
        },
        "rbu/5a7d68b8f2916415c74f2f0d": {
            "ip_address": "10.10.25.4",
            "name": "lab-ki09-2"
        },
        "vim/5a7d68b7f2916415c74f2f03": {
            "auth_url": "https://cic01.vran5g.rnd.ki.sw.ericsson.se:5000/v2.0",
            "name": "dev-cloud"
        }
    }
}
```
"password": "abc123",
"project": "test",
"user": "test"
},
"vnf/5a7d68b8f2916415c74f2f13": {
 "name": "vnb-1",
 "type": "vnb",
 "vim": "vim/5a7d68b7f2916415c74f2f03"
},
"vnf/5a7d68b8f2916415c74f2f17": {
 "name": "vcell-1",
 "type": "vcell",
 "vim": "vim/5a7d68b7f2916415c74f2f03"
},
"vnf/5a7d68b8f2916415c74f2f1b": {
 "name": "vue-1",
 "type": "vue",
 "vim": "vim/5a7d68b7f2916415c74f2f03"
},
"vnf/5a7d68b8f2916415c74f2f1f": {
 "name": "vpp-1",
 "parameters": {
 "scaling_policy": "contexts"
 },
 "type": "vpp",
 "vim": "vim/5a7d68b7f2916415c74f2f03"
},
"vnf/5a7d68b8f2916415c74f2f23": {
 "name": "vpp-2",
 "type": "vpp",
 "vim": "vim/5a7d68b7f2916415c74f2f03"
}
Listing A.3: The result of the transformation.

```json
{
    "instance": "ns/5a7d68b8f2916415c74f2f0f",
    "objects": {
        "ns/5a7d68b8f2916415c74f2f0f": {
            "name": "cu-1",
            "pnfs": [
                "rbu/5a7d68b7f2916415c74f2f05",
                "rbu/5a7d68b7f2916415c74f2f07",
                "rbu/5a7d68b8f2916415c74f2f09",
                "rbu/5a7d68b8f2916415c74f2f0b",
                "rbu/5a7d68b8f2916415c74f2f0d"
            ],
            "vnfs": [
                "vnf/5a7d68b8f2916415c74f2f13",
                "vnf/5a7d68b8f2916415c74f2f17",
                "vnf/5a7d68b8f2916415c74f2f1b",
                "vnf/5a7d68b8f2916415c74f2f1f",
                "vnf/5a7d68b8f2916415c74f2f23"
            ]
        },
        "rbu/5a7d68b7f2916415c74f2f05": {
            "ip_address": "10.10.22.4",
            "name": "indoor-ki09-1",
            "password": "string"
        },
        "rbu/5a7d68b7f2916415c74f2f07": {
            "ip_address": "10.10.23.4",
            "name": "indoor-ki09-2",
            "password": "string"
        },
        "rbu/5a7d68b8f2916415c74f2f09": {
            "ip_address": "10.44.39.4",
            "name": "outdoor-ki09-1",
            "password": "string"
        },
        "rbu/5a7d68b8f2916415c74f2f0b": {
            "ip_address": "10.10.24.4",
            "name": "lab-ki09-1",
            "password": "string"
        },
        "rbu/5a7d68b8f2916415c74f2f0d": {
            "ip_address": "10.10.25.4",
            "name": "lab-ki09-2",
            "password": "string"
        }}
```
"vim/5a7d68b7f2916415c74f2f03": {
    "auth_url": "https://cic01.vran5g.rnd.ki.sw.ericsson.se:5000/v2.0",
    "id": "integer",
    "name": "dev-cloud",
    "password": "abc123",
    "project": "test",
    "user": "test"
},
"vng/5a7d68b8f2916415c74f2f13": {
    "name": "vnb-1",
    "parameters": "object",
    "type": "vnb"
},
"vng/5a7d68b8f2916415c74f2f17": {
    "name": "vcell-1",
    "parameters": "object",
    "type": "vcell"
},
"vng/5a7d68b8f2916415c74f2f1b": {
    "name": "vue-1",
    "parameters": "object",
    "type": "vue"
},
"vng/5a7d68b8f2916415c74f2f1f": {
    "name": "vpp-1",
    "parameters": "object",
    "type": "vpp"
},
"vng/5a7d68b8f2916415c74f2f23": {
    "name": "vpp-2",
    "parameters": "object",
    "type": "vpp"
}
A.4 Transform Schema example

The following shows an example of a Transform Schema object called vl, this would then apply to one or several instances of the type vl. The transformation that would be applied in this case would be the addition of a new key and value called id and string.

Listing A.4: amf example

```json
{
  "objects": [{
    "vl": {
      "changes": [{
        "new": [{
          "key": "id",
          "value": "string"
        }]
      },
      "instance": true,
      "newobject": false,
      "removeobject": false,
      "schemaorobject": false,
      "type": "vl"
    }
  }
}
```

Listing A.5: vl example

```
"vl/5a7d68b7f2916415c74f2f03": {
  "group_name": "id",
  "name": "test"
}
```

Thus the new instance will have the additional field after the applied transformation. Which is then sent to the receiving application, which expects the object to have certain required fields.

Listing A.6: vl example

```
"vl/5a7d68b7f2916415c74f2f03": {
  "id": "string",
  "group_name": "id",
  "name": "test"
}
```
Appendix B

Implementation

Listing B.1: Upgrade value function within the transform implementation.

```go
func upgradeValue(in map[string]interface{}, obj *Object) {
    if obj.Instance == true {
        objMap, _ := in["objects"].(map[string]interface{})
        targets := findInstances(objMap, obj.Type)

        for _, i := range targets {
            if targetObject, ok := objMap[i].(interface{}); ok {
                if object, ok := getValue(targetObject).Interface().(map[string]interface{}); ok {
                    for _, v := range obj.Changes {
                        for _, v := range v.Value {
                            object[v.Key] = v.Value
                        }
                    }
                }
            }
        }
    } else {
        objMap, err := SchemaOrObject(in, obj)
        check(err)

        if targetObject, ok := objMap[obj.Type].([]interface{}); ok {
            for _, v := range targetObject {
                if objects, ok := getValue(v).Interface().(map[string]interface{}); ok {
                    for _, v := range obj.Changes {
                        for _, v := range v.Value {
                            objects[v.Key] = v.Value
                        }
                    }
                }
            }
        }
    }
}
```
Listing B.2: Finds the instances to a corresponding type

```go
func findInstances(in map[string]interface{}, key string) (ret []string) {
    var b []byte
    var n int

    b = make([]byte, len(in))
    n = -1

    exp := `\w+\S`
    rexp := key + exp
    re, _ := regexp.Compile(rexp)

    for k, _ := range in {
        k = k + "\n"
        b = append(b, k...)
    }

    res := re.FindAll(b, n)
    ret = make([]string, len(res))

    for k, v := range res {
        for _, j := range v {
            ret[k] += string(j)
        }
    }

    return ret
}
```
Listing B.3: The parse function within the Wizard.

```python
def parse(tr_obj, p_obj, model_1, model_2):
    changes = Change(None, [], [], [], [])
    obj = Object(p_obj['type'], None, False, False, False, True, changes)

    obj.Type = p_obj['type']
    c = match(p_obj, {'changes': _}, lambda changes: changes)

    if match(c, {'$update': _}, lambda update: update, default=False):
        c = match(c, {'$update': _}, lambda update: update)

    if match(c, {'properties': _}, lambda properties: properties, default=False):
        p = match(c, {'properties': _}, lambda properties: properties)

    if match(p, {'$delete': _}, lambda delete: delete, default=False):
        d = match(p, {'$delete': _}, lambda delete: delete)

        for r in d:
            rm = Remove(None)
            rm.Key = r
            changes.Remove.append(rm)

    if match(p, {'$update': _}, lambda update: update, default=False):
        up = match(p, {'$update': _}, lambda update: update)

        for k, v in up.items():
            val = Value(k, None)

            if parse_update(v):
                val.Value = parse_update(v)
                changes.Value.append(val)
            else:
                pass

    if match(c, {'required': _}, lambda required:
```
required, default=False):
    target = model_1[obj.Type]

    if match(target, {'properties': _}, lambda properties: properties, default=False):
        prop = match(target, {'properties': _}, lambda properties: properties)

        parse_schema = model_2[obj.Type]
        prop_schema = parse_properties(obj, prop, parse_schema)

        req = match(c, {'required': _}, lambda required: required)
        ret = parse_required(obj, req, model_1, prop_schema)

        changes.New.extend(ret.New)
        changes.Remove.extend(ret.Remove)

# Reformating of the Transform-Schema.
obj = add_key(obj)
tr_obj.Objects['objects'].append(obj)

ret = jsonpickle.encode(tr_obj.Objects, unpicklable=False, warn=True)
ret = dict(json.loads(ret))

for k, v in ret.items():
    for i in v:
        del_none(i)
        changes_list(i)

    ret = lower_keys(ret)

return ret
Appendix C

Delta

Listing C.1: The delta between two model versions.

DELTA:
{
    "$update": {
        "$schemas": {
            "$update": {
                "$rbu": {
                    "$update": {
                        "properties": {
                            "$insert": {
                                "password": {
                                    "description": "A super secret password",
                                    "type": "string"
                                }
                            }
                        }
                    }
                }
            },
            "required": {
                "$insert": [
                    [2,
                     "password"
                    ]
                ]
            }
        },
        "$vim": {
            "$update": {
                "properties": {
                    "$update": {
                    
                }
            }
        }
    }
}
"id": {
    "$update": {
        "type": "integer"
    }
},
"vnf": {
    "$update": {
        "properties": {
            "$delete": [
                "vim"
            ],
            "required": {
                "$delete": [
                    1
                ],
                "$insert": [
                    [
                        1,
                        "parameters"
                    ]
                ]
            }
        }
    }
}
Bibliography


