Supporting multiple management interfaces through YANG model transformation

NIKLAS SEMMLER
RAMKUMAR RAJAGOPALAN

Master Thesis
Stockholm, Sweden September 2014

TRITA-EE 2006:666
Supporting multiple management interfaces through YANG model transformation

Advisor: Mr. Jan Lindblad, Tail-f
Examiner: Prof. Rolf Stadler, KTH

NIKLAS SEMMLER
RAMKUMAR RAJAGOPALAN

Master Thesis
Stockholm, November 2014
Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av ????????????????? 06.11.2014 i A528.

© Niklas Semmler & Ramkumar Rajagopalan, September 2014

Tryck: Universitetsservice US AB
Management interfaces for network devices have evolved from SNMP over various CLIs and Web GUIs to the new standard NETCONF. Experts have been trained and systems have been developed, for any combination of these interfaces. Network devices that are to be integrated into existing systems require the support of multiple management interfaces. Developers of these devices face the challenge of mapping data models used by supported management interfaces (e.g. MIB for SNMP) to the devices’ internal configuration database.

This challenge is split into two problems. First, mappings from each virtual data model (used by a management interfaces) to the base data model (employed by the configuration database) have to be declared. Second, incoming requests from management interfaces have to be transformed (according to the model mappings) to the device’s internal format.

In this thesis, all data models are defined in the YANG modeling language to simplify the declaration of mappings. A domain model is constructed from an analysis of existing solutions. For the brief and consistent expression of mappings, a Domain-specific Language (DSL) is synthesized. To transform incoming requests to a standard format, an Erlang application is developed. Finally, the solution’s coverage of the problem domain is evaluated.

The work is part of a project at Tail-f Systems in Stockholm. Contributions of this work are a DSL for defining YANG model mappings, a Transform Application for transforming configuration change requests and a domain model.
Acknowledgments

We would like to thank our supervisor at Tail-f, Mr. Jan Linblad, for his constant support and motivation. We would also like to thank our examiner at KTH, Prof. Rolf Stadler for offering his valuable guidance and feedback throughout the thesis project. Finally we thank our family and friends who have always been a source of inspiration during our academic career.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xi</td>
</tr>
<tr>
<td>Listings</td>
<td>xii</td>
</tr>
</tbody>
</table>

1 Introduction 1
   1.1 Motivation 2
   1.2 Problem statement 3
   1.3 Approach 3
   1.4 Contributions 4
   1.5 Outline and authorship 5

2 Background 7
   2.1 YANG 7
   2.2 ConfD 11
   2.3 pyang 14
   2.4 Domain-specific Languages 15
   2.5 Compiler construction tools 18

3 Earlier projects at Tail-f 21
   3.1 7-transform 21
   3.2 MetaCLI 24
   3.3 genet 25
   3.4 Summary 26

4 Design of the DSL 29
   4.1 Scope of the DSL 29
   4.2 Requirements for the DSL 34
   4.3 Language design of the DSL 37

5 Implementation of the DSL 45
## List of Figures

1.1 Solution overview ...................................................... 2
1.2 System overview ...................................................... 4
2.1 ConfiD architecture .................................................... 12
2.2 UML sequence diagram of a Callback in ConfiD .................. 13
2.3 Compiler end-to-end ................................................... 18
3.1 Virtual Management Information Base (MIB) and Base Command-line interface (CLI) models side-by-side .................. 23
3.2 Feature diagram of MetaCLI ......................................... 24
3.3 genet example: Virtual and Base models ......................... 26
4.1 Arities of mapping .................................................... 31
4.2 Mapping of keys ..................................................... 33
4.3 Feature diagram of the Domain .................................... 34
4.4 Entity-Relation diagram of the Domain-specific Language (DSL) environment ......................................................... 38
5.1 Conversion of the DSL ................................................ 45
5.2 UML sequence diagram of the pyang plugin ....................... 48
5.3 UML sequence diagram of the DSL Compiler ...................... 48
5.4 Diagram of the package dependencies of the DSL Compiler .. 49
5.5 UML class diagram of the pyang plugin .......................... 50
5.6 UML class diagram of the Parser & Lexer ......................... 51
5.7 UML class diagram of the Abstract Syntax Tree ................. 52
5.8 UML class diagram of the type system .......................... 53
5.9 UML class diagram of the Node Visitor .......................... 54
5.10 UML class diagram of the Symbol Table ......................... 54
5.11 UML class diagram of the output model ......................... 56
6.1 Architecture of the Transform Application (UML Component diagram) ..................................................... 58
6.2 A Simple Transformation: Virtual and Base models ............. 60
6.3 Base model data for the simple transformation .................. 60
6.4 Virtual Model data for the simple transformation ............... 61
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.5</td>
<td>Sequence diagram of operations in the Transform Application</td>
<td>63</td>
</tr>
<tr>
<td>6.6</td>
<td>UML class diagram of the Transform Application</td>
<td>66</td>
</tr>
</tbody>
</table>
# List of Tables

2.1 Possible advantages and disadvantages of the use of a DSL . . . . . . . 16  
2.2 DSL design patterns [21] . . . . . . . . . . . . . . . . . . . . . . . . . . . 17  
2.3 DSL implementation patterns [21] . . . . . . . . . . . . . . . . . . . . . 17  
3.1 DSL statements in MetaCLI . . . . . . . . . . . . . . . . . . . . . . . . . 25  
4.1 ConfD Callbacks and the direction of mapping . . . . . . . . . . . . . . 30  
4.2 Different arities . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31  
5.1 Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53  
6.1 Arguments and return values for the Callbacks . . . . . . . . . . . . . . 63  
7.1 Characteristics of mappings supported by the Transform Application . . 72  
7.2 Grammar comparison to genet . . . . . . . . . . . . . . . . . . . . . . . . 74  
7.3 Grammar comparison to other DSLs (from [16]) . . . . . . . . . . . . . 74  
7.4 Analysis of DSL Compiler's code . . . . . . . . . . . . . . . . . . . . . . 75  
7.5 Code Comparison to genet . . . . . . . . . . . . . . . . . . . . . . . . . . 76
# Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Leaf example</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Container example</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>List example</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Typedef Example</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>YANG extension example</td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td>Example of pyang and the tree plugin</td>
<td>15</td>
</tr>
<tr>
<td>2.7</td>
<td>Function call</td>
<td>19</td>
</tr>
<tr>
<td>2.8</td>
<td>PLY Tokenizer</td>
<td>19</td>
</tr>
<tr>
<td>2.9</td>
<td>PLY Yacc</td>
<td>19</td>
</tr>
<tr>
<td>2.10</td>
<td>AST for function call</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of including a DSL statement</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Part of the example model</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>Erlang function used as lookup table</td>
<td>46</td>
</tr>
<tr>
<td>5.3</td>
<td>Erlang record</td>
<td>47</td>
</tr>
<tr>
<td>6.1</td>
<td>An example of hardcoding Callbacks</td>
<td>63</td>
</tr>
<tr>
<td>6.2</td>
<td>The Mappings Record</td>
<td>64</td>
</tr>
<tr>
<td>6.3</td>
<td>General algorithm for Callbacks</td>
<td>66</td>
</tr>
<tr>
<td>6.4</td>
<td>MAAPI cursor for iterating through a list</td>
<td>68</td>
</tr>
<tr>
<td>6.5</td>
<td>Algorithm for the get_next Callback</td>
<td>68</td>
</tr>
</tbody>
</table>
ABOUT

This thesis work has been performed at Tail-f Systems in Stockholm, Sweden, for five months (February till June 2014). During this time the authors were supported by a scholarship from the Masterschool of the EIT ICT Labs, a body of the European Union and by a success-oriented payment from Tail-f Systems.

All the material presented in the thesis is–if not stated differently–the result of the authors’ efforts. Graphics used in this document have been designed with TikZ, TikZ-ER, TikZ-UML, Inkscape and pyreverse from the pylint package. Code that has been produced by the thesis work is property of Tail-f Systems.
Several methods exist to manage network devices. Traditionally, Simple Network Management Protocol (SNMP) has been used to manage network devices. But, SNMP turned out to be quite difficult to use for network management, although it is still the most widely deployed protocol for network monitoring and fault handling. Network device vendors often ship their own management interfaces with their devices. All the major vendors, such as Cisco and Juniper, have their own styles of Command-line interfaces (CLIs), their own web-based GUI etc. This leads to a lot of wasted effort in developing the interfaces and a lot of unlearning and relearning for the users. RFC 3535 lists the features of a network management system that most network operators identified as desirable. SNMP and most of the proprietary solutions do not support many features desired by the operators, such as transactions and rollbacks in configurations, separation of state data and configuration data and so on. Network Configuration Protocol (NETCONF) was defined to address these shortcomings.

NETCONF runs over SSH and talks to the devices using a Remote Procedure Call (RPC) mechanism. It supports features like transactions and rollbacks in configurations, configuration validation, network-level configuration, etc. RFC 6241 describes the protocol in detail. In order to model a device configuration, with all the configuration and state data, a modeling language called YANG has been defined to supplement NETCONF. YANG is a hierarchical, extensible data modeling language. Together, NETCONF and YANG are out to be the new standard solution in the field of network configuration and management. They have been specifically designed to solve the shortcomings of previous solutions.

A big problem NETCONF and YANG face is compatibility. Old systems are hard wired to use certain management interfaces and many experts are trained to configure their devices via CLIs or vendor specific GUIs. Hence it is not possible to abolish the old management interfaces. Instead, NETCONF has to coexist with them.
CHAPTER 1. INTRODUCTION

1.1 Motivation

The Swedish company Tail-f Systems contributed greatly to the specification of NETCONF and YANG. Their device management product ConfD can provide various management interfaces (CLI, NETCONF, ...) for any device or application. The device’s configuration is modeled in YANG. Using this model, ConfD can render multiple management interfaces automatically.

From top to down: Agents send requests to the configuration data via different management protocols (NETCONF, SNMP & CLI). Daemons contained in the network device (box with rounded corners) accept the requests and evaluate them according to their respective model. (It is assumed, that all models are defined in YANG). The model that is used by the configuration database is the base model. Requests to the base model can access the configuration database directly. Requests for all other models, called virtual models, are transformed at run-time. For this transformation, mappings from each virtual model to the base model are required. These mappings are compiled from user-specified annotations to the virtual model.

This is a good solution for new devices that are being provisioned. In this case, the operator can model the device as desired. Existing devices already use some management interface and hence they expect the configuration data to be in a particular format. It will be beneficial to the operators if the device can be managed with a data model that is easy to use and understand. So, it is desirable for the
device to expose a data model different from its native data model. In other words, it should be possible to store the configuration of a device using one data model and access it using a different data model. This will be useful if the current data model of the device is unintuitive or is machine-friendly, and the operator wishes to use a different data model that is more intuitive or straightforward. For example, a device might store its configuration information in the form of Management Information Bases (MIBs) (the data structures used by SNMP), while the operator might want to manage the device using a Cisco-style CLI.

We call the data model in which the configuration is stored in the device the base model. Any other model, which is used to manage the device’s configuration, without any data storage associated with it, is the virtual model. What we need now is a way to translate operations done in the virtual model to equivalent operations in the base model.

The goal of the authors in this project is to develop a general way of defining mappings between YANG models (the base model and one or multiple virtual models). The construction of a Domain-specific Language (DSL) for a brief expression of mappings is at the heart of this endeavor. Further to the DSL, a Transform Application has to be developed that uses the results from the DSL to actually realize the mappings. Figure 1.1 shows a scenario where the configuration is stored in the device using the format defined in a YANG model, but is accessed using some other data models (using, say, SNMP and a web GUI). Here, the look-up tables and the functions that are necessary for the mapping are produced by the DSL. The Transform Application takes requests to the virtual model and transforms them into requests to the base model.

1.2 Problem statement

The problem tackled in this work is twofold. First, mappings from the virtual model (the model not connected to any data store) to the base model (the model connected to the data store) have to be specified. A Domain-Specific language is needed to specify the mappings between the YANG models. The DSL should take the specified mappings and generate some mapping data structures to be used by the Transform Application.

Second, a Transform Application that interfaces with ConfD APIs has to be developed. This application, along with ConfD, forms the run-time system, that actually realizes the transformation. It should take requests to the virtual model, and using the mapping data structures generated by the DSL, transform them into requests to the base model. Since ConfD is written in Erlang, this transformation application is also required to be implemented in Erlang.

1.3 Approach

The following is the approach used in this project:
1. Previous attempts at Tail-f to solve this mapping problem are analyzed. They are run, and their code is thoroughly read.

2. The mapping process is modeled. General characteristics are extracted from an analysis of previous work. A DSL is designed using these characteristics.

3. The designed language is implemented with the help of a compiler-compiler and a YANG model parser.

4. An Erlang application is designed and implemented. It uses the output from the DSL and realizes the transformations.

5. The characteristics of the grammar of the DSL and the code of the DSL compiler are analyzed.

1.4 Contributions

The DSL has been developed in conjunction with a Transform Application. An overview of the system is given in Figure 1.2. The DSL compiler generates a look-up table containing the mappings. The Transform Application uses it to transform incoming requests from an agent into requests for the model in the configuration database. The resulting language has been evaluated against the requirements.

![Figure 1.2: System overview](image)

During compile-time the virtual model and its mappings defined as DSL statements are compiled to a lookup table. During run-time the lookup table is used to transform incoming requests for any virtual model to requests to the base model. Responses are transformed back to the format of the virtual model.

All parts of the document are the work of the authors unless noted differently.¹

¹Examples presented in Chapter 3 are taken from the respective work presented.
1.5 Outline and authorship

This thesis document is structured as follows:

- Chapter 2 presents an overview of the technologies and tools used in the project. It introduces YANG, ConfD, PLY and pyang. Both authors contributed equally to this chapter.

- Chapter 3 describes earlier attempts to this problem at Tail-f. This chapter is authored by Niklas Semmler.

- Chapter 4 and Chapter 5 presents the design and implementation decisions behind the DSL. These chapters are authored by Niklas Semmler.

- Chapter 6 explains the design choices and implementation details of the Transform Application. This chapter is authored by Ramkumar Rajagopalan.

- Chapter 7 describes the evaluation of the project and evaluates them. This chapter is authored by both the authors.

- Chapter 8 suggests potential improvements in the project and concludes the thesis report. This chapter is authored by both the authors.
This chapter explains the background of the technologies used in the project. Especially Section 2.1 and 2.2 of this chapter are required to understand the descriptions in the following chapters. In the later sections, alternative technologies are compared and choices made in this project are highlighted.

2.1 YANG

YANG is a data modeling language [6]. It was developed to serve as a complement to the Network Configuration Protocol (NETCONF) [11]. Where the NETCONF protocol defines how to ‘install, manipulate, and delete’ configuration data, the YANG modeling language specifies the format of the configuration data, the data model. YANG follows a hierarchical organization. Data, on the broadest level, is structured into modules (which can import other modules) and submodules (can be included into modules). Every module is assigned a unique namespace.

On a lower level, the most common data structures in YANG are lists, containers and leafs. A brief discussion of those structures is given in the following.

Leaf

Leafs are the basic elements. They have no children (hence the name) and denote a single value. The leaf statement in YANG must specify the following:

name  The name of the leaf

type  The type of the data in the leaf (integer, string, etc.). The type can either be a built-in type that is defined in YANG or a derived type that is specified by the user.

Optionally, a leaf can also have other information relevant to the data model, like the following:
default The default value for the leaf. When configuring, if this leaf is not explicitly set, it will take the default value.

mandatory Its a boolean. If true, the leaf should be explicitly configured. If false, the leaf may or may not be configured.

constraints The type statement can optionally contain substatements that impose some constraints on the values the leaf can accept. These are used to specify the range of the values the leaf can take, the string pattern that a leaf’s value should match against and so on.

```yang
leaf example_leaf {
  type uint32 {
    range "1..5000";
  }
  default 60;
}
```

Listing 2.1: Leaf example

The Listing 2.1 shows an example of a leaf in a YANG model. The name of the leaf is example_leaf, and it has the type of a 32-bit unsigned integer. The leaf can take values from 1 to 5000, and the default value of the leaf is 60.

A special case is a leaf with type empty. In this case, the leaf only holds binary information: its presence or absence. There are many other optimal substatements. More information about leafs is explained in the RFC section 7.6 [6].

Container

A container is used to group a number of child nodes. A container can have leafs, lists and even other containers as children (See Listing 2.2). Containers only organize the data. They hold no data by themselves, and are deleted, when they do not contain any child nodes. Presence containers are special in that their sheer existence holds information. Presence containers are marked with a presence substatement.

```yang
container example_container {
  leaf leaf1 {
    type string;
  }
  leaf leaf2 {
    type uint16;
  }
}

container ssh_enabled {
  presence "If this container exists, ssh is enabled";
  leaf ssh_protocol {
    type string;
  }
}
```

Listing 2.2: Container example
2.1. YANG

Listing 2.2: Container example

The Listing 2.2 shows two containers. The first container is of name `example_container` and it has two leafs defined inside. The second container is a presence container. If this container is present in the configuration, it might mean that ssh is enabled in that device, for example.

Containers are explained in detail in the RFC section 7.5 [6].

Lists

A list groups child nodes and can contain leafs, containers and other lists as children. The difference between lists and containers is that a list can have many instances. A list instance is called a `list entry`.

The `key` statement specifies the keys of the list. Keys are special leafs that are used to identify a particular list entry.

```ylang
list example_list {
  key "key1 key2";
  leaf key1 {
    type string;
  }
  leaf key2 {
    type uint16;
  }
  leaf data1 {
    type uint64;
  }
  leaf data2 {
    type string;
  }
}
```

Listing 2.3: List example

Listing 2.3 shows a list defined in YANG. The name of the list is `example_list`. The list has two leafs as keys, `key1` and `key2`. For each list entry in the configuration, the combination of `key1` and `key2` should be unique. Since keys are used to uniquely identify a list instance, those leafs, which are keys, cannot have default values.

There are many other sub-statements that make lists the most utilized grouping construct in YANG. RFC section 7.8 [6] explains lists in much more detail.

Types

YANG specifies several built-in data types, like uint8, uint16, int8, int16, decimal64, binary, bits, string etc. The user can impose additional restrictions on the data,
using keywords like range, pattern etc. (See Listing 2.1). Besides this, the user can also define completely new types from derived types, using the `typedef` statement. The type can be used in the model by specifying the name used in the typedef.

```yab
typedef Validity {
    type enumeration {
        enum notAvailable;
        enum partial;
        enum complete;
    }
    default complete;
}

leaf ipValidity {
    type Validity;
}
```

Listing 2.4: Typedef Example

Listing 2.4 shows how to define a derived type. After this definition, `Validity` becomes a separate type, that can be used in leafs, just like the built-in types. So, the leaf `ipValidity` is of type `Validity`, and it can take one of the three values `notAvailable`, `partial` or `complete`.

**YANG extensions**

The **YANG** language can be extended with additional statements, using the `extension` keyword. **YANG** modules can make use of extensions defined in some other modules, by importing those modules. The extensions consist of a keyword and an argument (see section 6.3.1 in [6]).

```yab
module dsl {
    extension path {
        argument modelpath;
    }
    extension fun {
        argument function_name;
    }
}

module virtual {
    import dsl {
        prefix dsl;
    }
    container main {
        dsl:path "*/base/model";
        list servers {
            key "address port"
            leaf address {
```
2.2. CONFD

Listing 2.5 shows two YANG modules. In the module dsl, the two extensions path and fun are defined. The dsl module is imported to the module named virtual, using the import statement. Now the virtual module can use the extensions defined in the dsl module. When using constructs from imported modules, the imported module should be specified along with the extensions. So, when using an extension from the module dsl, it should be used as dsl:path and dsl:fun. This is an example of how YANG can be extended and how those extensions can be used.

These extensions are the first step in the creation of the Domain-specific Language (DSL). The choice of the number and naming of extensions becomes the macro structure of the DSL. Of course the meaning of those extensions depends on how they are used by the YANG and DSL compiler. But, as the names should be intuitive to the user, the meaning cannot stray too far from the naming.

Adding an extension in this way has the restriction that double quotes cannot be used in the code as they are already used by YANG as delimiters.

2.2 ConfD

ConfD is a device configuration toolkit [12] developed by Tail-f Systems. It is deployed on top of network devices (either a physical device or a software daemon) and automatically renders multiple interfaces to agents for interaction with the device. These interfaces include Simple Network Management Protocol (SNMP), Web GUI, NETCONF, Command-line interface (CLI), etc. Hence, the device can be configured and managed using any of these interfaces, or a combination of several interfaces.

ConfD talks to the devices and to the agents both via a socket API. The management interfaces are rendered using the data model of the device’s configuration, modeled in YANG (see Section 2.1). A user of ConfD has to implement the callbacks in their software and provide a data model of the device (again see Section 2.1). ConfD will access the device/software via the callbacks and allow read and write access to the configuration data according to the requests received via the configuration interface. A structural description of ConfD follows.
CHAPTER 2. BACKGROUND

Architecture of ConfD

The architecture of ConfD consists of three layers (see Figure 2.1). Each layer is separated from the others by a socket API. Hence each layer could in principle be replaced by a custom one. The lowest layer of ConfD is a database that stores the configuration data. ConfD ships with a hierarchical fault-tolerant XML database called Configuration Database. Devices read their configuration via a dedicated API from the Configuration Database.

On top of the Configuration Database sits the Management Backplane. The Management Backplane has a hierarchical view on the configuration. It takes care of authentication, authorization and accounting and is responsible for processing transactions over the configuration data. It connects to the Configuration Database and any sort of plugins through the Data Provider API (DPAPI).

The different configuration interfaces (NETCONF, SNMP, Web GUI, CLI, etc) access the configuration (possibly interface-specific views of the configuration) through the Configuration Database. Those interfaces interact with ConfD through the Management Agent API (MAAPI).

Paths

To access any configuration data, the corresponding YANG structure (leaf, list or container) has to be accessed. The position of this structure is expressed as a path...
starting from the root structure. There are two ways to specify such paths in \texttt{ConfD}.

The notation used for display purposes (when showing output or interacting with the user) looks very similar to the Unix file path. The path starts with a slash and structures are separated with a slash as well. For example, to refer to the leaf port in the model shown in Listing 2.5, the path should be \texttt{/virtual:main/servers \{addr, port\}/port}, where \texttt{virtual} is the module name. The keys of lists in the path are specified inside curly braces. Since the list \texttt{servers} has two keys, they are specified as \texttt{\{addr, port\}} in the path.

Internally in \texttt{ConfD}, the path is expressed as a list. The order of the elements is reversed, in that the element that is referenced is placed first, and the root of the module is placed last in the list. For example, to refer to the same path as above, the list used in \texttt{ConfD} will be \texttt{[port, \{addr, port\}, servers, main, virtual]}. Paths used inside \texttt{ConfD} are explained in detail in section 6.2.

\section*{Callpoints and Callbacks}

A callback is a function or function reference that is passed to a program, so that this program can execute the function on a given event. \texttt{ConfD} allows external applications to register callback functions. Once registered, they are invoked when some required conditions are met. Callpoints are the means by which \texttt{ConfD} knows when to invoke the callbacks. Callpoints are indicated in the data model using annotations (Tail-f specific \texttt{YANG} extensions). Application can register themselves as data provider for any callpoints, thereby letting \texttt{ConfD} know where the callback function is defined. A data provider can register for multiple callpoints. Whenever the part of the data model, the callpoint is responsible for, is accessed, the callback functions are invoked.

In Figure 2.2, the \texttt{Transform Application} (\texttt{TransformApp}) registers (shortened to ‘reg’ in the figure) a callback function (\texttt{fun_callback}) to a callpoint id (\texttt{id\_callpoint}). Whenever a request comes in for which the callpoint is responsible the callback

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{sequence_diagram.png}
\caption{UML sequence diagram of a Callback in \texttt{ConfD}}
\end{figure}
function is called with the request as its argument and the result is returned to the calling agent (Agent).

There are a lot of events for which Applications can register callbacks. Applications can register as data providers for subscription callbacks, used for notifying changes to a part of the configuration. For example, using the model in Listing 2.5, if an application subscribes to the path /virtual:main/servers and registers a subscription callpoint, whenever there is a change in that part of the configuration, i.e. whenever a new servers instance is added, or an existing one is deleted or modified, Confd invokes the registered callback functions relevant to that callpoint. Similarly, there are validation callbacks for validating the values of particular leaves in the configuration, type callbacks for converting between user-defined types and so on.

The callbacks relevant to this project, are the transformation callbacks. For them to be invoked by Confd, an application should register itself as a data provider to some transformation callpoints. It is done using the tailf:transform extension in a YANG model. Once it is used in a model, Confd knows that requests to that model should be transformed to requests to the model stored in the configuration database. Hence, whenever there is a request to the model, which has registered a transformation callpoint, Confd invokes the callback methods in the data provider, which transforms the requests into requests to the base model.

The callbacks necessary for a transformation application are as follows:

get_elem It is invoked, when there is a request to read a leaf in the virtual model.

set_elem It is invoked, when a leaf is set in the virtual model.

get_next It is invoked, when it is required to iterate over a list in the virtual model.

create It is invoked, when a list, presence container or an empty leaf is created in the virtual model.

delete It is invoked, when a list, presence container or an empty leaf is deleted in the virtual model.

Transformation Callbacks are described in more detail in section 6.2.

2.3 pyang

From the project’s website [26]: pyang is a YANG validator, transformator and code generator, written in Python. pyang is an open source project using the New BSD license. In the context of the thesis project, it has been mostly used to process annotations in YANG files through plugins.

pyang is a command-line tool that operates over YANG files. Through the development of plugins, the user of pyang can gain access to its internal data structures.
2.4 Domain-specific Languages

In this project, a domain-specific language (short DSL) has been developed to simplify the expression of mapping between two models. In this section, a short introduction to the field of domain-specific languages will be given.

Introduction

Users who dare not write a ‘real’ program, are happily coding macros in their Spreadsheets. Mathematicians naturally use software such as MATLAB or Mathematica. Administrators use various shell scripts and Unix mini-languages (sed, grep, ...) without ever writing ‘proper’ software. Once we start looking for the uses of DSL, we can find them in various places.

The concept of the DSL is defined in contrast to the General-purpose programming Language (GPL). While the exact definitions vary across literature most definitions agree on a few key concepts: A DSL is a codified language having as its target a single kind of problem (the problem domain). A very specific notation is a defining factor for a DSL as is its limited expressiveness. A DSL can, but does not necessarily have to, be executable. The uses of DSLs range from data extraction and data processing to modeling purposes and more.

Often the main goal of introducing a DSL is to arrive at a better fit between the logic as written in the program code (from here on referred to as process logic) and...
the logic in the minds of the domain experts (from here on referred to as domain logic). Theoretically a DSL should be easily understandable and even writable by the domain experts. Even though this is seldom achieved the reduced notation of the DSL makes it at least easier to discuss problems with domain experts. Additionally, a reduced notation leads to less lines of codes and hence less possible bugs.

But, it should not be ignored that a DSL can induce a great deal of costs [10]. On the one hand, there is the cost for the development of the DSL and on the other, the cost for training users to use it. Even after the initial development and training, maintenance and extensions will introduce more costs. GPLs have the advantage that they can distribute the costs throughout a huge user community, but DSL are most often used in smaller settings and hence the costs are higher for fewer people. A list of general advantages and drawbacks of the use and development of a DSL are listed in Table 2.1.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ readable by domain experts</td>
<td>✗ costs of implementation and design</td>
</tr>
<tr>
<td>✓ (possibly) writable by domain experts</td>
<td>✗ difficulty of finding the right scope</td>
</tr>
<tr>
<td>✓ shorter, hence less bugs</td>
<td>✗ language can suffer from isolation</td>
</tr>
<tr>
<td>✓ enable conservation of domain knowledge</td>
<td>✗ requires additional training</td>
</tr>
<tr>
<td>✓ validation at domain level</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Possible advantages and disadvantages of the use of a DSL

Phases of DSL development

The development of a DSL follows four phases: Decision, Analysis, Design, Implementation. The first phase is nothing but the evaluation of the feasibility of DSL development followed by a decision. (For help on deciding for or against the development of a DSL, refer to the decision guidelines in [21].)

The analysis phase comes close to the field of domain engineering. The domain and its borders have to be identified and the knowledge has to be aggregated. As a prerequisite for designing a DSL it is necessary to have extensive knowledge of the domain. The understanding of legacy systems has a particular important role. To reach an understanding of the domain three options are available. Either an informal approach is used, that is the domain is simply described. Or in a formal approach, the domain is researched with a domain analysis methodology. Or, finally, the domain knowledge is mined from existing code bases.

In the design phase, the knowledge is formalized to create notions and operations. Those are used to design the language. The steps taken differ according to whether a new language is invented or an existing one is exploited. In the former case, there are nearly no constraints on the format of the language other than the expectancy of familiarity by the users. In the latter, the DSL needs to follow the existing language. In Table 2.2, the common DSL design patterns are listed.
2.4. DOMAIN-SPECIFIC LANGUAGES

**Language exploitation**

Piggyback  The DSL is implemented in an existing language. (e.g. as a collection of interwoven objects)
Specialization  The existing language is restricted for use as a DSL.
Extension  The existing language is extended with libraries, overloading of operators, etc.

**Language invention**

Table 2.2: DSL design patterns [21]

In the implementation, the chosen DSL is realized in code. Depending on the choice in the design phase one of several DSL implementation patterns can be used. See Table 2.3. Still most DSLs end up as libraries of functions and objects [29].

Table 2.3: DSL implementation patterns [21]

The authors made the following decisions in the development of the DSL. In this project, an informal analysis was chosen, followed by language invention and the development of the compiler. Three legacy systems have been analyzed for the creation of the DSL (see Chapter 3).

Language invention was chosen for the design phase. The language evolved in parallel with new insights into the domain and problem. From the beginning it was planned to deliver a very simple language closely connected to the domain. Therefore, language exploitation with its lesser development cost was discarded in favor of more freedom in the design of the language.
In the implementation phase, it was chosen to build a compiler. The reason being that the product of the DSL, a data structure created from a template, was easier to construct with a compiler (one step translation) than with an interpreter (stepwise translation).

An alternative would have been to exploit the host language, Python, using the piggyback approach and implement the DSL via embedding. This approach would require less effort in compiler development, but would restrict the final syntax of the DSL to encapsulated Python objects and function calls. This would be an impediment on the reading and writing for the user.

### 2.5 Compiler construction tools

In the scope of the project, a custom language had to be created including a compiler \[24\]. The pipeline of a typical compiler is depicted in Figure 2.3. A string is segmented into basic units (tokens) and parsed into an Abstract Syntax Tree (AST). From there an intermediate representation or output model is devised, which finally leads to an output format (most often code). Of the many stages of the compiler pipeline the development of the parser is probably the most time consuming. Compiler construction tools (also called compiler-compiler) have been developed in many programming languages to ease the development of both the parser and the lexer (also called tokenizer).

![Diagram](image.png)

**Figure 2.3: Compiler end-to-end**

The most used compiler-compiler is the combination of YACC and LEX. YACC (Yet Another Compiler Compiler) is a LALR parser generator written by Stephen C. Johnson at AT&T in 1970. LEX (LEXical Analyzer) is a tokenizer written by Mike Lesk and Eric Schmidt in 1975 and is used in combination with YACC.

In this project, the compiler was required to interface with pyang through a pyang plugin (see Section 2.3 and Chapter 4). For this reason, only compiler-compilers producing compilers in Python were relevant. Among the many compiler-compilers with Python targets, three stand out: PLY and Pyparsing are recommended in the Python literature (e.g. in \[4\]). ANTLR is a Java parser generator, which can generate parsers for Python that are beyond the typical capabilities of a parser generator \[23\].

In the end, PLY was chosen for reasons of maintainability. It is the only one that follows the YACC and LEX paradigm, which is the one most known to computer
scientists everywhere and it was also the one that was used previously at Tail-f.

**PLY**

PLY stands for ‘Python Lex-Yacc’ and this is exactly what it is. It tries to mimic the functionality of lex and yacc as close as possible while maintaining its own syntax. It was conceived by David Beazley in 2001 for an introductory course to compilers [3].

To give the reader an idea on how the code looks like, a small practical example is provided. The two function calls in Listing 2.7 is parsed and a basic AST generated.

```python
do_stuff(param)
res = do_stuff(param1, param2)
```

Listing 2.7: Function call

First a tokenizer is written in Listing 2.8. Every token is simply designed as a regular expression. PLY automatically reads all variables from the current context that begin with ‘t_’.

```python
from ply import lex
t_ID = r'[a-zA-Z][a-zA-Z_0-9]*'
t_LPAREN = r'[ ]{'
t_RPAREN = r'[ ]}'
t_COMMA = r'[ ]'
t_EQ = r'[ ]'
t_ignore = r'[ ]'
tokens = [ 'ID', 'LPAREN', 'RPAREN', 'COMMA', 'EQ' ]
lexer = lex.lex()
```

Listing 2.8: PLY Tokenizer

Next a parser is developed in Listing 2.9. The parsing rules are written as functions. Again PLY automatically reads all functions this time starting with a ‘p_’. The documentation string of the functions contains the grammar rule and the body of the function defines the way of processing. In this particular case it is defined that a valid expression can either be a bare function call or a function call including an assignment. Tuples are used to create the AST.

```python
from ply import yacc
def p_root(p):
    '*** root : assign
     | expr ***
    p[0] = p[1]
def p_assign(p):
```

Listing 2.9: PLY Parser
**CHAPTER 2. BACKGROUND**

```python
*** assign : ID EQ expr ***
p[0] = ('assign', p[1], p[3])

def p_expr_plus(p):
    *** expr : ID LPAREN args RPAREN ***
p[0] = ('function', p[1], p[3])

def p_args_basecase(p):
    *** args : ID ***
p[0] = [p[1]]

def p_args(p):
    *** args : ID COMMA args ***
p[0] = [p[1]] + p[3]

parser = yacc.yacc()
```

Listing 2.9: PLY Yacc

When we use the parser on the example given in Listing 2.7, we receive a tuple and nested tuple respectively as shown in Listing 2.10.

```python
('function', 'do_stuff', ['param'])
('assign', 'res', ('function', 'do_stuff', ['param1', 'param2']))
```

Listing 2.10: AST for function call

Of course the lexer and parser in the actual project are far more complex. For example, instead of using tuples for the AST dedicated classes are used. Still this example gives a good idea on how the compiler is organized.
CHAPTER
THREE

EARLIER PROJECTS AT TAIL-F

This project stands in the legacy of three previous projects at Tail-f. As a preparation in the development of the current project the code of each of these previous projects was analyzed. Here the results of the analysis are presented and the reader is introduced in a piecemeal fashion to the evolution of the solution.

Each of these approaches share the same process flow: A request to the virtual model is received via an API callback (see Section 2.2) and is then translated into a request for the base model. If the process succeeds, the result (on a read request), a confirmation (on a write request) or an error is returned. (See the ConfD user guide [12] chapter 10, section 3 for more information on transformations.) The three projects differ in the complexity of their solution. The first project 7-transform does not include a Domain-specific Language (DSL), but simply uses a Transform Application with hardcoded mappings. Next, MetaCLI introduces seven DSL statements, yet they are simply used to call dedicated functions. Only the last project, genet, actually implemented a real DSL and is a direct ancestor of our work. It did, however, never leave the state of a prototype.

3.1 7-transform

The first project done by Tail-f ships with ConfD as an example of model transformations. The example scenario features two management interfaces. A Command-line interface (CLI) is set up as the base model and a Simple Network Management Protocol (SNMP) interface using a writable Management Information Base (MIB) is used as the virtual model. In the following, the differences between the two models and how they are transformed will be briefly discussed.

Differences between the models

The example in Figure 3.1 presents a mapping of the representation of network interfaces. It serves as a good motivation for the necessity of model transformation.
The different YANG models exhibit structural and semantic differences. The CLI model follows a hierarchical organization, while the MIB model is organized in multiple tables connected by a single key.

While the elements connected with arrows marked ① and ② in Figure 3.1 feature a linear mapping, the arrow ③ points us directly to the different organization. The MIB model shown is built around multiple tables which all include the ifIndex (or ipAddressIfIndex) at some point. The CLI model in contrast features a hierarchical organization where a single list holds all the information related to the interfaces.

Marker ④ and ⑤ show us two extensions to the generic YANG modeling language by Tail-f that are relevant to our project. ④ points to a transformation callpoint. Transformation callpoints are used by ConfD to allow external application to be called when part of a model is accessed. A transformation callpoint is a callpoint where the application is not simply called once the part is accessed, but it takes responsibility to answer all requests (see also Section 2.2). It is the essential element that distinguishes a virtual from a base model. Every Transform Application is connected to a YANG model via such a transformation callpoint.

⑤ points to an extension that helps in the case of an asymmetry between models. Here the MIB model has the element ifIndex that is not part of of the CLI. To store the value of ifIndex in the CLI model without changing it’s representation to the outside world, a hidden leaf is created. This leaf will not appear to northbound interfaces, but can be used by ConfD internally.

A simple Transform Application

The transformation application is shipped as a program written in C. It initializes with ConfD and registers a number of callbacks (see Section 2.2 for information on callbacks which are used in the transformation). Its point of contact with the base and virtual model reside in the callpoints which are defined in the YANG models.

In 7-transform all mappings are hard coded in the C program. When any entry of the virtual model is written or read, the callbacks are called. The code of the function associated with the callback proceeds through the following steps.

1. Identify which entry is actually called: The path of the entry, an argument of the callback is compared with a compiled list of symbols.

2. (Optional:) Test if any condition holds in the base model.

3. Set, get, remove or create any entry in the base model by calling the ConfD Management Agent API (MAAPI).

4. Return either the value (in the case of a get_elem or get_next operation) or a state (success CONFD_OK or error CONFD_ERR).
For a briefer representation, leafs in the base model are generally depicted without further information (type, etc.) and a number of leafs are not depicted.
CHAPTER 3. EARLIER PROJECTS AT TAIL-F

Conclusion

This first approach has no separation of process logic and domain logic. The YANG models include only direct contact points to the application. The application is written in C and thereby makes the understanding of the involved domain logic hard. This obfuscation is reflected in numerous comments throughout MetaCLI’s code base.

3.2 MetaCLI

The MetaCLI project is a ConfD model transformation that has been developed for use with the Metaswitch [22] router. It enabled this router to render both SNMP and CLI as northbound interfaces. In this scenario, the size of the model made a layer of abstraction indispensable. The domain was analyzed into distinct features. Figure 3.2 shows the result in the form of a feature diagram. The features are separated hierarchically. Optional features are marked with an empty circle and mandatory ones with a full circle. Mutual exclusive features are connected with a curve. (For more information see [15].)

As can be seen from the Figure 3.2 every YANG statement has a path and a type. The type can either be list, leaf, or empty leaf and presence containers. Lists have keys, leaves have a type and a value, and empty leaves and presence containers either exist or not (see Section 2.1). For all that does not fall in this clear distinction, the feature other is used.

![Figure 3.2: Feature diagram of MetaCLI](image)

The features are reflected in the YANG model itself as an annotation and in the transformation application as a category of functions. The YANG model is extended by the module view that contains the respective annotations. An example of an annotation is given in Listing 3.1.
3.3. GENET

leaf some {
    view :pathmap "path/in/base/model";
    view :typemap "typemap_to_string_function";
    type uint8;
}

Listing 3.1: Example of including a DSL statement

Of all the annotations, the pathmap is the only annotation that accepts a path string as an argument. This string is either an absolute path in the other model, a relative path that uses the path further up in the hierarchy of the model or a dot signifying that this path is defined in the program code. Every mapping includes a pathmap. All other annotations receive a function name as an argument. A list of the annotation and their meaning is given in Table \ref{dsl}.

<table>
<thead>
<tr>
<th>Name</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathmap</td>
<td>path</td>
</tr>
<tr>
<td>keymap</td>
<td>keys</td>
</tr>
<tr>
<td>typemap</td>
<td>type</td>
</tr>
<tr>
<td>elementmap</td>
<td>type and value</td>
</tr>
<tr>
<td>existmap</td>
<td>existence</td>
</tr>
<tr>
<td>register-hook</td>
<td>anything else</td>
</tr>
</tbody>
</table>

Table 3.1: DSL statements in MetaCLI

At runtime the Transform Application responds to requests by looking up the function that is associated with the requested path in the virtual YANG model and calls the respective function.

MetaCLI improves over 7-transform by introducing feature-oriented mappings and references to functions in the YANG model. Hereby the mappings gain in visibility for the reader of the model. But, parametrization of the functions or definition of domain logic directly in the YANG model, is still missing.

3.3 genet

genet is an initial prototype that has been developed by Jan Lindblad to research the effect of a greater level of abstraction on the problem. Again the YANG model is extended with DSL annotations. This time however, the extension takes not only a single word but a limited amount of DSL code.

For each YANG statement, the DSL code is extracted by a pyang plugin and parsed through a compiler written with PLY. A data structure is constructed in the process. It is used by the Transform Application to respond to any requests at runtime. In contrast to both MetaCLI and 7-transform, the Transform Application of genet is entirely written in Erlang instead of C.

\begin{footnote}
Jan Lindblad is the Principal Solutions Architect at Tail-f Systems
\end{footnote}
The power of adding a compiler is shown in Figure 3.3. Even without the arrows it should be clear what each DSL statement means. First the list virtual is mapped to the list base, meaning that their keys are mapped onto each other. The DSL statement src of the genet extension is used for this. Then each of the elements x, y and v are mapped. Each of them uses the DSL statement map. The mapping of leaf x might seem surprising. The DSL denotes an addition, but the arrow is annotated with a \(-1\). The reason here is that the relation is described from the base model side (T:a stands on the left, not the right of the assignment) so the mapping from the virtual model to the base model is actually an inverse of the one depicted, hence a subtraction.

```yarn
list virtual {
    genet:src "plain_list(T=/base)";
    key "x";
    leaf x {
        genet:map "T:a = x + 1"; type uint8;
    }
    leaf y {
        genet:map "y = T:c"; type uint8;
    }
    leaf v {
        genet:map "v = T:b + 1"; type string;
    }
}
list base {
    key a;
    leaf a {
        type uint8;
    }
    leaf b {
        type string;
    }
    leaf c {
        type uint8;
    }
}
```

Figure 3.3: genet example: Virtual and Base models

Of course this is just a simple example, but it already shows the power of introducing a DSL. Instead of hiding the actual logic in a separate code file it can be represented directly in the YANG model.

### 3.4 Summary

In this journey through prior work, several characteristics of the problem and possible solutions have been touched. In 7-transform the reason for model transformation was explained on a practical example. 7-transform also featured the simplest Transform Application which we used as an inspiration when developing our own. MetaCLI was an example on how the problem could be segmented into features and how those features could subsequently lead to separation in the code base. Finally, genet is probably the closest our project has to an ancestor. The power of
introducing a [DSL] is illustrated and the code base of the [Transform Application] is greatly reduced by using [Erlang] instead of C.

Starting from this point we will perform our own feature analysis of the domain and create a language based on it. We will create a generic [Transform Application] and finally compare our results back to the *genet* project.
The design of a Domain-specific language is a delicate task. Multiple concerns have to be traded off against each other. First of all, the Domain-specific Language (DSL) should fit the problem domain as closely as possible. Usually some kind of semantic model is constructed beforehand and the details of the particular problem are fed to it. Second, the language has to fulfill expectations of the users. For an audience that mostly consists of programmers, the created language should not differ too much from established languages. Finally the language has to be embedded in a system, giving way to further constraints.

In this chapter, the design of the DSL will be described in three sections. First the domain will be analyzed. Second, general concerns surrounding the language will be considered. Third and finally, the language will be specified. The description of the implementation is left to the next chapter.

4.1 Scope of the DSL

Before a DSL can be designed the domain has to be modeled. In this section, the domain will be described. In the process of the description, a feature set of the domain will be constructed.

Receiving a callback

In the first step, a single YANG statement in the virtual model is accessed via a ConfD callback. (See Section 2.2 for an overview of the callbacks in ConfD.) Only the three YANG statements discussed in Section 2.1 are of relevance (leaf, list, and container). Of those three, containers can be mostly ignored, as only presence containers actually contain information in themselves and they can be mapped in the same way as empty leafs.

For leafs the value they contain and its type has to be mapped. Lists are mapped by mapping their keys (again leafs) individually.
Direction of mapping

The mapping of the different callbacks can be filed into two categories by taking their ‘direction’ into account. For leaves, a set callback requires a mapping from the virtual to the base model. In contrast a get callback requires a mapping from the base to the virtual model. create and delete callbacks are only applicable to the special case of empty leafs and are ignored in this prototype. The get_next callback only applies to lists. Similarly for mapping lists—and thus their key leafs—only the callbacks create, delete, and get_next are relevant. Of those the create and delete callback requires a mapping from the virtual to the base model and the get_next requires a mapping in the other direction. Table 4.1 gives an overview.

Broadly, mappings can be categorized into two categories a forward mapping from the virtual model to the base model and a backward mapping from the base model to the virtual model. In this thesis, the first is referred to simply as ’mapping’ and the other as ’inverse mapping’ or simply ’inverse’.

<table>
<thead>
<tr>
<th>YANG statement</th>
<th>callbacks</th>
<th>inverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>container</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>presence container</td>
<td>create, delete</td>
<td>exists</td>
</tr>
<tr>
<td>leaf</td>
<td>set</td>
<td>get</td>
</tr>
<tr>
<td>empty leaf</td>
<td>create, delete</td>
<td>exists, get</td>
</tr>
<tr>
<td>list</td>
<td>create, delete</td>
<td>get_next</td>
</tr>
</tbody>
</table>

Table 4.1: ConfD Callbacks and the direction of mapping

Arities of mapping

For each leaf, the mapping between the virtual model and the base model can be categorized based on its arity. Figure 4.1 gives a visualization of how leafs in the two models can be related. The mapping is either one-to-one, one-to-many, many-to-one, or many-to-many. In the following, all of the possibilities are listed and they are also summarized below in Table 4.2.

**One-to-One** There is exactly one corresponding leaf in the base model. Different forms of correspondence are possible:

- **Path-only** the corresponding leaf has exactly the same format and type. In this case, only the path has to be mapped. (E.g. both models have the leaf *IP address* but at different places in the model.)

- **All** the corresponding leaf differs in the format, type or in the addition/subtraction of a constant value. (E.g. both models use a counter, but one starts at 0 and one at 1)
4.1. SCOPE OF THE DSL

One-to-Many  The value of the leaf to be mapped is distributed over multiple nodes. For a mapping the value has to be split up among those, for an inverse the value has to be joined from the corresponding ones. (E.g. the string tcp80 has to be partitioned into tcp and 80.)

Many-to-One  In the opposite case mapping a leaf statement from the virtual model is not possible without accessing other leaves from the virtual model. (E.g. in the base model the type and number of a network interface are stored as one value eth0, while in the virtual model those are stored as two entries ethernet and 0.) In this case, the values have to be stored, either in a data structure in the Transform Application or in hidden entries in the YANG model, till the rest of the required information is available. The inverse mapping can be solved by extracting for each virtual model leaf the respective part of the base model leaf.

Many-to-Many  This case requires a combination of the above.

<table>
<thead>
<tr>
<th>arity</th>
<th>mapping</th>
<th>operations inverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:1</td>
<td>reference, cast, format, add or subtract constant</td>
<td></td>
</tr>
<tr>
<td>1:N</td>
<td>split</td>
<td>join</td>
</tr>
<tr>
<td>M:1</td>
<td>store &amp; load</td>
<td>extract part</td>
</tr>
<tr>
<td>M:N</td>
<td></td>
<td>all of the above</td>
</tr>
</tbody>
</table>

Table 4.2: Different arities

Path translation

Before the value or type of a leaf can be mapped, first its path has to be mapped. In the simplest case the path is just a combination of containers followed by a final leaf. Then, the path can be expressed as a pure string. Unfortunately this is seldom
the case. In almost all cases, the leaves in the virtual model are nested in one or more lists and map to similarly nested leaves in the base model. Before a leaf can be mapped, the keys of each list this leaf is nested in have to be mapped. The keys are mapped to the corresponding keys of lists in the base model.

The difficulty of mapping keys from one path to the other depends first of all on the number of keys and the number of lists in each path. Apart from the usual forms of mappings for leaves, which holds for keys too (keys are after all leaves), there are the following variations of key mappings (see also Figure 4.2):

**Default** In the default case a key in the virtual model corresponds to a key in the base model on the same level of the path.

**Encapsulation** means that the keys of the virtual model have to be put into a list to map to the base model. A list maps to a nested list.

**Decapsulation** is the inverse of encapsulation. That is, a nested list maps to a list.

**Reordering** In case of reordering, the keys in the virtual model have to be reordered to match the base model.

**Duplication** refers to the process of duplicating keys and lists of the virtual model. A list maps to multiple lists.

**Unification** is the inverse of duplication. It leads to the merge of multiple lists of the virtual model into one in the base model. Multiple lists map to a single list.

**Promotion** is the scenario, where a leaf in the virtual model is used to set a key in the base model. This leads to an asymmetry between the two models, as only a single list entry in the base model is accessible from the virtual model.

**Demotion** is the opposite of promotion. A key in the virtual model refers to a leaf in the base model. This situation should normally not occur, as the multiple list entries of the virtual model will be lost when transformed to the base model. (After all the virtual model itself cannot not be stored.)

For the resulting language this entails that there must be a convenient way to map keys and distribute them over multiple lists and paths. It should also be possible to specify paths consecutively.

**Conditionals and Loops**

At this stage conditionals and loops are not considered for the DSL. They can be used in the Erlang library of DSL functions, but they cannot be used in the DSL.
4.1. SCOPE OF THE DSL

Figure 4.2: Mapping of keys

The boxes represent lists. To map a leaf (white circle), first the keys (black circle) have to be mapped. The keys need to be restructured according to patterns a-h to allow the mapping.

itsell. That said, next iterations of the DSL would definitely need to consider to include at least the conditionals.

A usage pattern of conditionals in the MetaCLI’s code base is to make the mapping of one leaf dependent on the value contained in a second leaf. Or that a cast only occurs if a value of a leaf is in a certain format and not another. A usage pattern of loops in MetaCLI is aggregation of the contents of a list.

For a first prototype, both conditionals and loops are not essential. They have the potential to make the DSL harder to understand and divert the attention from the essential features. However for realizing all mappings the support of loops and conditionals are necessary.

Summary

In summary, mappings are defined on a per-leaf basis. There are two directions of mappings: A mapping denotes a translation from the virtual to the base model and an inverse describes the opposite relation. For the mapping of leafs functions have
to be available. Those functions can be distinguished by the arity of their relation (1:1, 1:N, M:1, M:N). Before a leaf can be mapped, first the path it maps to has to be established. This is achieved by translating the path string and mapping all keys of all lists.

Figure 4.3 visualizes the features of the domain. It is a representation used by the feature description language (FDL). Features are organized in hierarchical order. Mandatory and optional features are distinguished by full and empty circles respectively. The empty partial circle connecting multiple edges denotes mutual exclusion between the elements at the end of these edges. For more information see [15].

The current project aims to create a prototype of a DSL for specifying mappings. In the limited scope of a prototype certain cases cannot be dealt with. Most important mappings of the arity M:1 and M:N are not covered. The YANG constructs empty leafs and presence containers are ignored. Loops and conditionals are not covered as well.

4.2 Requirements for the DSL

Independent of the domain, general language concerns have to be met. These concerns will be discussed in the present section. To simplify the design of the mappings for the users – the mapping designers – the expression of mappings was required to follow the principles below. (In the following, the language specification of the DSL will simply be referred to as DSL while the actual instances of DSL code will be referred to explicitly as DSL code. The person that will actually write the mappings is either named YANG model engineer’ or simply ‘engineer’.)

Compactness One of the big justifications of the use of any DSL is the reduction in code size it introduces. Less lines of code has been correlated to less bugs in the past [20]. To reduce the numbers of lines as far as possible the notation has to be very concise.
4.2. REQUIREMENTS FOR THE DSL

**Abstraction** Redundancy can and should be avoided by introducing levels of abstraction.

**Extensibility** For very complex mappings the user should be able to define the mapping in the General-purpose programming Language (GPL) Erlang.

**Inference** Instead of letting the DSL engineer specify every detail of the mapping, as much as possible should be inferred by the DSL Compiler. This holds especially for the specification of the mapping and its inverse.

**Integration** The DSL needs to be close to the YANG files, both in the format of the language and in the storage location of the actual code. So, it would help if the DSL syntax is similar to YANG and the DSL code can be specified in the YANG file itself.

**Error handling** The feedback given by the DSL implementation should help the engineer debug the DSL code. So, the feedback should be in terms of helpful error statements instead of just crash dumps or stack traces.

To meet those requirements the following design decisions had to be made.

**Compactness**

External DSL (see Section 2.4 as language invention) implementation allows the definition of languages without restrictions on their syntax. In principle, the language could be reduced to operators and one-letter variables, but of course that would make the language hard to understand. To make the language brief, it has been segmented into several parts. Constants are defined in one, functions in another and the core mapping logic is contained in a third segment.

**Abstraction**

Another gain of the language is the possible introduction of domain-specific abstractions. Paths and constants can be reused across the hierarchy of the YANG model. Specific functions for mappings can be defined once and reused for multiple leafs. The DSL allows abstractions in the form of DSL functions.

**Extensibility**

For a domain as broad as this one it is hard if not impossible to find a DSL that covers all the cases of potential mappings. As a consequence the developed DSL should only cover the most common cases and leave the other ones to GPL code. As the Transform Application is developed in Erlang, custom code must as well be implemented in this language.

To include custom code into the DSL all that has to be done is to implement functions in Erlang and add them to the library of DSL functions. With little effort they can be directly used in the DSL instructions.
Inference

Another potential area of reducing the amount of code is by inferring some of the details that are required to do the mapping. In the analysis it was established that mappings have two directions. As mappings are usually symmetrical, one direction of the mappings can often be inferred from the other. To this end the mapping has to be reversible. That is the mapping can be turned to its inverse by reversing all of its operations.

The split function can for example be reversed to join and adding a constant can be reversed to subtracting the same constant. Other functions such as a cast cannot be inversed. Casting a value to an integer gives little information on how to reverse the effect. It could be that the reverse is a cast to a string or a double or something completely different. (Of course this could be inferred from the \texttt{YANG} statements type, but this is another level of difficulty.)

In the cases where a reverse can automatically be produced, the \texttt{YANG} model engineer must be freed from the burden of specifying it. While for the cases where it cannot be automatically inferred, there must be a way to define it explicitly.

Error handling

A criteria most relevant to the person actually defining the \texttt{YANG} models and their mappings is the feedback from the \texttt{DSL Compiler}. The more expressive the feedback is, the less time the engineer has to spend on debugging.

The error feedback was an early goal for the language. It has been later dismissed in favor for further development of the \texttt{DSL} itself. Even though a user-friendly error handling is essential, it is not required for the purpose of a prototype.

Integration

The integration of the \texttt{DSL} into the host application is an often overlooked problem of \texttt{DSL} development [14]. In this project, the \texttt{DSL} has to be integrated at the same time with the \texttt{YANG} model and with the \texttt{Transform Application} that will do the actual transformations at runtime. Due to the importance of the matter it will be discussed in further detail.

\texttt{DSL} code and the \texttt{YANG} model

There is no technical requirement that would make an integration of the \texttt{DSL} code and the \texttt{YANG} model necessary. In contrast, the \texttt{DSL} code could be specified in an independent file. There are two disadvantages to this approach. First, in any mapping of the \texttt{DSL} the source path has to be specified. When the code is already co-located with the source leaf, the source path can be inferred. Second, it complicates the understanding of the mapping by adding a third source of information (the base model, the virtual model and the mappings). Instead, the \texttt{DSL} code can be co-located with the virtual model. It simply has to be defined as an extension
(see [19] for details). The DSL then takes the form of an annotation that can appear at any place in the model.

Co-locating the DSL statements with the model has the potential to increase the size of a YANG model enormously and thereby obfuscate the domain logic. To resolve this issue, the definition of mapping functions can be decoupled from the actual calls to those functions. The latter is normally a one-liner, while the former can span several lines. In that case, the functions can either be specified in the DSL library in Erlang or at any other place of the model.

**DSL and the Transform Application**

The role of the DSL is the simplification of the specifications of mappings. The engineer should always have the possibility to resort to specifying the mappings directly in Erlang. The DSL is just a short cut to this end. In this project, the contact point between the DSL and the Transform Application is a data structure (see Section 6.2). It is used by the Transform Application to perform the mappings at runtime and can be generated from DSL code via templating.

**Conclusion**

Figure 4.4 summarizes the results. The DSL will be implemented as a YANG extension containing several statements (see Section 4.3). Those statements as a whole will provide the mapping to the base model, which in turn is used as a schema by the database. From those mappings an Erlang data structure will be generated. For an explanation of the concrete implementation please refer to Chapter 5.

### 4.3 Language design of the DSL

The language consists of three parts. First, language elements that enable the specification of mappings of leaves and keys, including the path elements. Second, further language elements allow the construction of mapping functions right in the YANG file. Third and last, a library of custom DSL functions written directly in Erlang make it possible to use the full power of a GPL to create building blocks for the DSL.

Reuse in this DSL is based on the propagation of custom functions, constants (integers and strings) and path elements throughout the tree of the YANG model. Each of those three is available at the place of the definition and further down the tree (a hierarchical scope).

In the following each of the three parts of the language will be explained. The first two parts contain DSL statements. The DSL statements are separated into first-order statements and second-order statements. Statements of the first-order are elements of the YANG extension. They take as their argument a string containing second-order elements. Similarly, the explanation always starts with the
first-order statement and then goes on to explain the respective supported second-order statements.

To ease the reading of the chapter, the format of a language specification is mimicked. Every statement in the first two parts is presented with its appearance in the model as an annotation, its general structure and an example. In the following listings are not referred to via number, but are embedded in the surrounding text. (Similarly, as to what would be expected from a language specification.) In all DSL listings pointy braces represent attributes that have to be filled in. They are either variable or function names, or complete instructions. The full example can be found in the appendix.

### Specification of mappings

dsl::key "<keymappings>";

This statement can be used for specifying mappings of keys, so that they can be reused for different leafs. The key has to be mapped on the list level:

\[
<\text{new_key1}, \text{new_key2}, \ldots > = \text{function}(<\text{keyname}>)
\]

A number of new keys are created by running a function on the key identified by the name given to the key in the [YANG] file. Multiple of these key definitions can be specified, but must be separated by semicolons. The key can then be defined with the [DSL] path statement.

Figure 4.4: Entity-Relation diagram of the [DSL] environment
4.3. LANGUAGE DESIGN OF THE DSL

list e {
    key 'x';
    dsl: key 'kx = subone(x)';
    dsl: path 'L = */tm/t{kx} *;
    ...
}

In this example, the key x is mapped to the new key kx, which happens to be the key subtracted by one. This new key is used to define the path to list L in the base model.

In the current stage, the DSL can not generate mappings of keys as there is no generic way to handle them in the Transform Application’s data structure.

dsl:path "<pathassignments>";

Path assignments have two purposes. The obvious one is to define path variables that can be used throughout the tree. The not so obvious one is that they relate keys as defined above to a base path. They are essential for the mapping of lists.

<path1> = *<path-string>*

Stars are used to delimit path strings. Multiple assignments are separated by semicolons.

<path2> = <path1> + *<path-string>*

Apart from directly defining variables (the stars are used as delimiters), paths can also be combined via concatenation.

<path1> = */root/list{<key1>, <key2>, ...}/leaf*

Keys are included in curly braces right after a list.

list e {
    key 'x';
    dsl: keys 'kx = subone(x)';
    dsl: path 'L = */tm/t{kx} *;
    py = L + *b*;
    pv = L + *c*;
    ...
}

In this example, the variable L is created with the key kx. The variables py and pv refer to leafs in the list L of the base model.

dsl:map "<mappings>";

Here the actual mapping happens.

<base_path1>, <base_path2>, ... = <function>({self})
Similar to the key expression above, the mapping statement is a simple function call. Only a single function call can be specified. The keyword `self` stands for the virtual path itself.

```plaintext
dsl::keys 'kx = subone(x)';
dsl::path 'L = */tm/t{kx}*/';
  py = L + *b*;
  pv = L + *c*;
leaf x {
  type uint8 {
    range 1..255;
  }
}
leaf y {
  dsl::map 'py = addfoo(self)';
  type string;
  default 'default';
}
leaf v {
  dsl::map 'pv = muladd(self)';
  type uint8;
  default 255;
}
```

In this example, the path variables defined earlier are used to map a leaf from the virtual to the base model. So, variable `py` is used to map leaf `y` and variable `pv` to map leaf `v`. The mappings are executed when a `set` operation is performed on the leafs `v` and `y`. Note how the leaf `x` does not include any mapping logic. As this is the key, the mapping is done on the list level.

```plaintext
dsl:inv "<inverse-mappings>";
```

The inverse mapping follows the exact same structure as the mapping.

```plaintext
self = <inv-function>(<path1>, <path2>, ...)
```

The arguments and results are reversed and the mapping function is inversed reflecting the nature of this statement. When the function used in the mapping statement can be inversed this statement can be omitted.

In the example, in the description of `map` the inverse was not stated and as such has to be inferred. The inverse mapping would be executed by the `Transform Application` during a `ConfD get` callback.

**Specification of custom DSL functions**

```plaintext
dsl:def "<constant-definitions>";
```
4.3. LANGUAGE DESIGN OF THE DSL

Similar to the definition of paths, constants can be defined.

\[
\texttt{<var> = <var/constant>}
\]

\[
\texttt{<var> = <var/constant> <operator> <var/constant>}
\]

Multiple constant definitions are separated by semicolons. Constants can either take simple values (strings or numbers), values referred to by variables or values and constants combined by operator.

\[
\texttt{dsl: def 'more = '_foo'";}
\]

Here the variable more is set and can be used. It can also be concatenated with other variables or strings through the use of the + operator.

\[
\texttt{dsl: fun "<function-definon>";}
\]

Function definitions are the most complicated construct of the [DSL]. They are employed to construct functions out of multiple library functions (chains). In the best case, such functions are using only library functions that can be automatically inversed. In these cases, the inverse function is automatically inferred. Function definitions are also the only part of the first-order [DSL] statements that can appear more than once at any level of a [YANG] model.

\[
\texttt{<function-name> : <multiple-instructions>}
\]

The function name is the first statement of the function definition and is separated from the rest of the logic by a single colon.

\[
\texttt{<fun-name> : @1, @2, ..., = <lib-fun(<consts>), $1, $2, ...)}
\]

Input variables and output variables are written as a $ and a @ sign followed by a number respectively. The variables are numbered according to the number of arguments and return parameters. The numbering scheme starts at one.

\[
\texttt{<fun-name> : <outs/tmps> = <lib-fun>(<consts>, <ins/tmps>)}
\]

Temporary variables, an underscore sign followed by any string, can be used to chain more than one function. Constants are defined by their name as defined in the definition part of the [DSL]. They should not begin with any special characters.

\[
\texttt{dsl: def 'more = '_foo'";}
\]
\[
\texttt{dsl: fun 'addfoo : @1 = append(more, $1)}
\]
\[
\texttt{dsl: fun 'muladd : _x = add(1, $1)}
\]
\[
\texttt{dsl: fun 'muladd : _x = add(1, $1); @1 = mult(2, _x)}
\]

Two examples are shown. The first function addfoo simply adds the string ‘_foo’ to the input variable. The second function adds one to the input variable and stores the result in the temporary variable _x. The value of this variable is then multiplied by two and returned.
To inverse the function the DSL Compiler exchanges the input arguments and return values. Additionally the order of statements is reversed.

### Specification of external functions

Mapping functions that can be used in the map, inv and key statement (discussed above) can be defined in two ways. Either they can be constructed via the fun statement right in the DSL or in the Erlang code of the library of DSL functions. In both cases they can use a number of library functions as building blocks. Those functions will be discussed below. The library contains two types of functions, building blocks for functions defined in the DSL and ready-made for direct use in the map statement. The difference is that functions used in the map statements cannot use constants, but only paths.

\[
\begin{align*}
<res> &= \text{add}(<\text{integer}>, <\text{path}>) \\
<res> &= \text{sub}(<\text{integer}>, <\text{path}>) \\
<res> &= \text{mult}(<\text{integer}>, <\text{path}>) \\
<res> &= \text{div}(<\text{integer}>, <\text{path}>)
\end{align*}
\]

These are basic integer functions. They should be used only in the function definition. The operation is addition, subtraction, multiplication and division respectively.

\[
\begin{align*}
<res> &= \text{append}(<\text{string}>, <\text{path}>) \\
<res> &= \text{rstrip}(<\text{string}>, <\text{path}>) \\
<res> &= \text{prepend}(<\text{string}>, <\text{path}>) \\
<res> &= \text{lstrip}(<\text{string}, <\text{path}>)
\end{align*}
\]

These building blocks can be used for simple string operations. A string constant is appended or removed from the right or left end of the string. Apart from this, the function is similar to the add and other integer functions.

\[
\begin{align*}
<res> &= \text{join}(<\text{string}, <\text{path}1>, <\text{path}2>) \\
<res1>, <res2> &= \text{split}(<\text{string}, <\text{path}>)
\end{align*}
\]

These two functions realize the 1:N relationship between leafs. The string constant plays the role of a separator. Take care that only the first occurring separator is used.

\[
\begin{align*}
dsl: \text{def} \ '\text{comma} = ', ';' \\
dsl: \text{fun} \ '\text{joincomma} : @1 = \text{join}(\text{comma}, @1, @2)' \\
dsl: \text{fun} \ '\text{splitcomma} : @1, @2 = \text{split}(\text{comma}, @1@)$\'
\end{align*}
\]

leaf xyz {
    dsl:map \ 'pa, pb = splitcomma(self)' \\
    dsl:inv \ 'self = joincomma(pa, pb)';
}

This example might lead to unforeseen consequences when the value of \(pa\) (definition of \(pa\) and \(pb\) are not shown) contains a string, which already has a
comma (e.g. ‘foo,fubar’). In this case, the splitcomma operation would lead to unexpected results.

\[
\text{<res> = cast('int', <path>) <res> = cast('str', <path>)}
\]

To change the type, a cast is necessary. So far, the casting of integers and strings is supported. The function identifies the current type of the value held by the path reference and uses the respective internal cast mechanism.

\[
\text{<res> = same(<path>)}
\]

This function is a mapping function and not a building block function. Its use is simply in the cases where a mapping is not necessary because the two values in the models correspond to each other.
In this chapter, the implementation of the **Domain-specific Language (DSL)** will be described. First, an overview will be given by highlighting the format of input and output of the **DSL Compiler**. Further down the internal workflow of the **DSL Compiler** will be explained. Finally, the structure of the **DSL** will be analyzed. Class and function names are highlighted with italics. (The same highlighting is also used when design patterns are referred to.) Where actual code would obfuscate the explanation, **Unified Modeling Language (UML)** diagrams or similarly structured diagrams have been chosen as replacement.

### 5.1 Overview

In this section, the implemented system will be presented by its input and output. On the broadest level, the system could be described as a conversion (see Figure 5.1). For each leaf including **DSL** statements in the virtual **YANG** model (first part of the figure), a single transformation rule is created (third part) by the **DSL Compiler** (second part of figure). This transformation rule specifies how a request (in the form of a **ConfD** callback) to the virtual model can be satisfied via requests to the base model.

![Figure 5.1: Conversion of the DSL](image)

---

45
CHAPTER 5. IMPLEMENTATION OF THE DSL

Input

Input to the DSL Compiler is a virtual model extended by DSL statements. Every YANG structure that is mapped contains at least a single DSL statement. For any such structure the DSL statements are jointly compiled via the DSL Compiler. The compilation takes place during a tree traversal. The tree is traversed starting at the top-most YANG structure. During the traversal the symbol table is built up.

To visualize this process, the reader is advised to take a look at the example model in Listing 5.1. Thanks to the top down traversal a simple form of scoping is implemented. The variable $L$ defined in the list can be reused in the specification of the leafs (see highlighted area). The variables defined in the leafs in contrast will not influence any variables defined in a later YANG structure. The same goes for the functions. (Again this does not differ from scoping as used in programming languages.)

```erlang
container tm {
    dsl:path  'root = */tm*';
    list e {
        key  'x';
        dsl:key  'xkey = ksub(x)';
        dsl:path  '*L = root + *xkey*';
        leaf x {
            type uint8 { range 1..255; }
        }
    }
    leaf v {
        dsl:path  '*pv = L + *c*';
        dsl:map  'pv = muladd(self)';
        type uint8;
        default 255;
    }
}
```

Listing 5.1: Part of the example model

Output

The output format of the DSL is the erlang file `ec_mappings.erl`. Its main use is the function `get_mappings`, basically a lookup table. As its argument it takes the path of any structure in the virtual model. This path is looked up in a gigantic switch statement (in erlang a `case` statement). The switch (see Listing 5.2) returns for any path an erlang record (similar to a dictionary in Python) simply named `mapping`.

```erlang
get_mappings(Path) ->
case Path of
    [e,tm,?TL1HIGH] -> mappings{...};
```

5.2. INTERNAL WORK FLOW

```
[x,X,e,tm,?TL1HIGH] when is_tuple(X) -> #mappings
  (...);
end .
```

Listing 5.2: Erlang function used as lookup table

These records (see Listing 5.3) and corresponding paths contain information required to process any callback. First, any resource used from the base model is contained in the basepaths list. If the virtual path refers to a list, then virt2basekeys and base2virtkeys define how the keys have to be mapped in both directions. Finally, the record contains an entry for any of the main callbacks. (The names correspond to the following as follows: get → get_elem, set → set_elem, create → create, delete → delete, next → get_next).

```
-record(mappings, {
  basepaths = [],
  virt2basekeys = fun(),
  base2virtkeys = fun(),
  get = fun(),
  set = fun(),
  create = fun(),
  delete = fun(),
  next = fun()
}).
```

Listing 5.3: Erlang record

Additionally ec_mappings.erl contains functions that are used inside of the records. Adding more functionality can simply be done by adding a function to the template used by the DSL. For more information on the Erlang records please refer to Section 6.2.

5.2 Internal work flow

It is time to look inside the black box. Below a procedural overview of the pyang plugin (in Figure 5.2) and internals of the DSL Compiler (in Figure 5.3) is presented.

Traversing the YANG module

pyang has several uses (see the background Section 2.3). Here, pyang’s plugin system is exploited. pyang will call any given plugin through an emit function. The plugin will be called with an internal data structure containing the modules of YANG models that were chosen as input parameter to pyang. The plugin is forwarding the first module to the special traversal class CompilerTraverse. For any statement, CompilerTraverse scans the statement for DSL code, extracts it and sends it to the DSL Compiler MyCompiler. All mappings encoded in the DSL statements across
CHAPTER 5. IMPLEMENTATION OF THE DSL

The YANG model are aggregated and returned to the pyang plugin. In a final step the templating engine Mako constructs an output file from all the mappings.

Inside the Compiler

As presented in the last section, the DSL Compiler receives a single set of DSL statements and returns the encoded mappings. Internally the DSL statements are at first passed to the parser. The parser MyParser receives a token stream from the input by the Lexer and applies the grammar to the input. (See Section 2.5 in the background for a detailed description.) While parsing the input, an Abstract Syntax Tree (AST) is constructed. The AST is returned to the DSL Compiler.
5.3. STRUCTURE OF THE DSL

front-end and from there used by a Node Visitor. The Node Visitor walks over the AST and creates a data structure. The Symbol Table (shown as table) is used by the Node Visitor and stores all variable declarations for further use down the YANG hierarchy. The class Output is used by the Node Visitor to construct a data structure that can be handled by the template.

5.3 Structure of the DSL

![Diagram of the package dependencies of the DSL Compiler]

Next to be considered is the structure of the actual code. An overview is given in the diagram of the module dependencies depicted in Figure 5.4. The pyang plugin yangparser includes references to the DSL Compiler front-end and the Symbol Table (dsl_symbol_table) which is shared among the different occurrences of DSL statements. The Parser (dsl_parser) and Node Visitor (dsl_visitor) make use of the AST (dsl_ast). References to the DSL type system (dsl_types) can be found in all these classes.

Apart from the Python standard library, two external dependencies exist. PLY (see Section 2.5 in the background) was used as a compiler-compiler and the Mako templating engine was used to populate a template with data.

In the following, a structural overview will be given. Classes are explained through their UML diagrams. Where possible, classes will be connected with general implementation patterns as specified in [24].

It should be noted that the Python parser for C code the pycparser served as a reference implementation for this DSL Compiler.
**pyang plugin**

![Figure 5.5: UML class diagram of the pyang plugin](image)

In Figure 5.5, the classes of the pyang plugin are shown. YangParserPlugin interfaces with pyang and manages the control flow.

The focus is on the traversal. The class TraverseObject implements a traversal over the YANG hierarchy that is used by other classes. In particular it is used by CompilerTraverse which handles the interaction with the DSL Compiler. PrintStatements prints the structure of the AST for debugging purposes.

The class FindObject is used to find YANG statements in the base model. This can and was supposed to be used for a check of correctness by comparing the user’s input with the actual base model. While there were some initial successes, changes to the Node Visitor broke the functionality multiple times and further development was left for a time when the DSL Compiler would be more stable. EnumTypeParser was created for parsing the custom YANG types (see Section 2.1), so that they could be used by reference to the name in the DSL Compiler. Further work on this approach was abandoned for reasons similar to the ones mentioned above.

**Parser and Lexer**

Parser (MyParser) and Lexer (MyLexer) (see Figure 5.6) follow the typical structure of DSL Compiler implemented using PLY. Token specifications in terms of regular expressions are implemented both as variables and functions. Parsing rules are specified as functions. In both cases, the doc string is abused for the sake of specifying the grammar of the domain-specific language. The Lexer is tested via unit testing in TestLexer. MyCompiler works as the main interface for the DSL Compiler. See Section 2.5 for more details on PLY.
5.3. STRUCTURE OF THE DSL

Figure 5.6: UML class diagram of the Parser & Lexer
AST

The AST is an abstract syntactic structure of the DSL code. The classes in Figure 5.7 follow a combination of pattern 10 and pattern 11 of [24]. All direct children of the Node class follow pattern 10 Normalized Homogeneous AST. In this pattern multiple node types exist, but children are represented as a homogeneous list. This is useful for nodes having only mandatory children.

Descendants of StructuredNode follow pattern 11 instead. Pattern 11 Heterogeneous Irregular AST refers to the use of multiple node types but with the difference that children of the node are named (dictionary in this case). This pattern is used for the case of multiple optional parameters.

The implementation patterns are explained in depth in [24].

DSL Types

Types (see Figure 5.8) are generated while the parser walks the input and constructs the AST (in a single pass). In contrast, in most implementations of bigger languages, types are created in an additional traversal over the AST.
5.3. STRUCTURE OF THE DSL

Types are directly inferred from the input: Pure numbers are represented as numbers, a quoted string as a string and so on (see Table 5.1). A close resemblance to pattern 20 Computing Static Expression Types can be found.

The operations over any constant or variable is implemented as an operator method on the class of the type of the first constant or variable. The second constant or variable becomes an argument to this method. Variables are resolved before with the aid of the Symbol Table. This conversion from a DSL operator to an operator method of a type class occurs during the visitation of the AST by the Node Visitor. The corresponding pattern is pattern 22 Enforcing Static Type Safety.

<table>
<thead>
<tr>
<th>Name</th>
<th>Examples</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>var, fOO</td>
<td>Any non-quoted string of letters and underscore</td>
</tr>
<tr>
<td>String</td>
<td>12, abc!</td>
<td>Anything contained in single quotes</td>
</tr>
<tr>
<td>Integer</td>
<td>12, -3</td>
<td>Any pure number with or without negative sign</td>
</tr>
<tr>
<td>Path</td>
<td><em>/high/low</em></td>
<td>Any string nested in a path constructor</td>
</tr>
</tbody>
</table>

Table 5.1: Types

Node Visitor

The Node Visitor (see Figure 5.9) used in this DSL follows pattern 13 External Tree Visitor. Two Node Visitors are implemented. The simple Show just prints the AST for debugging purposes and the Generator builds an Output Model (see below) from the AST that is used in the template. It is also responsible for the type checking and for populating the Symbol Table.

The logic of the Node Visitor is distributed among several functions (named visit_. . . . ) that contain the respective logic for an AST node. The advantage over pattern 12 Embedded Heterogenous Tree Walker lies in the aggregation of the logic which ease oversight and understanding.
This project uses a very simplified Symbol Table. (see Figure 5.10). All DSL
statement that belong to one YANG statements are parsed in a single scope. It is
not possible to create new types or create structures with additional scopes (like for
example the struct in C).

For most purposes, the Symbol Table can be understood as a simple Python
dictionary. Symbols and their values are stored and accessed as key, value pairs
in an internal dictionary. (Important are the hidden methods __getitem__ and
__setitem__ which are not shown in the graphic).

However, every time the traversal of the YANG model reaches a deeper level
(e.g. a list in a list) the Symbol Table creates new layer via nesting. This way the
same symbols can be used with different values on different levels.
5.3. STRUCTURE OF THE DSL

When a symbol cannot be resolved in the current level’s scope, the scope belonging to the YANG statement one level up in the hierarchy will be searched. Similarly, only if a function definition cannot be found in the current scope, the scope of the upper levels will be searched.

In this way, the Symbol Table follows a simplification of pattern 17: Symbol Table for Nested Scope of \[24\], even though only a fraction of this pattern’s functionality is used.

Output Model

To make the task easier for the Mako templating engine, an Output Model (see Figure 5.11) is created by the DSL Compiler. For this purpose, specifications in the format of comments are read from a library of DSL functions. Two types of DSL functions exist in this library. First, functions that can be used as instructions (Instruction) in the function definitions (FunDef) and second, functions that can be used in the mapping and inverse mapping statements (FunCall). During the compilation of DSL code, whenever a function call or definition is encountered it is added as a FunCall or FunDef to a LookUpTableEntry basically a transformation rule, for the particular leaf. (The classes Var and Vars and their children are used for the variables of the function definitions.) The Renderer class takes care of most of the formatting and is unit tested by separate class (not shown). Exceptions in the format of class OutputExceptions are thrown whenever an unexpected input is handled either from the DSL code or from the function specifications.

This Output Model OutputModel contains all the transformation rules and is stored by the CompilerTraverse class of the yangparser module. From the same module, the class YangParserPlugin calls the function make_template with the OutputModel and a Erlang template file as arguments. The data structure of the Transform Application is created by iterating over the multiple components of the OutputModel.
Figure 5.11: UML class diagram of the output model
CHAPTER
SIX

THE TRANSFORM APPLICATION

This chapter describes the design and implementation details of the Transform Application. The architecture of the Transform Application is explained. Later, a detailed description of the most important callback functions for the Transform Application is given. Later, the implementation of the callbacks is described. Finally, the limitations of the current implementation are considered briefly.

6.1 Design of the Transform Application

In this project, the DSL is the compile-time component. The mappings are specified using the DSL. The Transform Application is the run-time component, which actually does the transformation. The Transform Application interfaces with the ConfD daemon via a socket interface. It takes queries to the virtual model from ConfD and transforms them into queries to the base model. Finally, it does the necessary manipulations to the results from the base model and returns it in the format expected by the virtual model, back to ConfD. For this, the Erlang Records generated by the compile-time component, the DSL Compiler, are used. The mappings between the virtual model and base model are specified in those records.

There are several different approaches to realize such transformations, depending on the requirements on the base and virtual models. For example, in [7], the authors classify such transformations based on what part of the model is transformed. If the transform is applied to the whole model in one stretch, it is called a ‘one-time bulk transformation’. If it is applied only to some parts of the model depending on user needs, it is called the ‘call-by-need’ transformation. In our case, the transformation we are interested in is a call-by-need transformation, since we are only interested in the transformation of the part of the base model that corresponds to some particular part of the virtual model.
CHAPTER 6. THE TRANSFORM APPLICATION

Architecture of the Transform Application

Figure 6.1 shows the overall architecture of the Transform Application along with the Erlang API for ConfD.

The Transform Application consists of two modules. One of them, the ec_mappings module is, generated from the DSL Compiler. It specifies the mappings between the base model and the virtual model, i.e., which set of properties in the base model correspond to each property in the virtual model. It also contains methods needed for the transformation, in doing things like converting between different types of values (e.g. from integer to float, 8-bit integer to 16-bit integer ...), string concatenation/splitting etc. They can also contain user-specified methods used in transforms. Lastly, the ec_transform module implements the callbacks that realizes the actual transformation.

ConfD API’s and Callbacks

ConfD exposes several API’s for use by external applications. From the ConfD User Guide [12], the main API’s are:

MAAPI Management Agent API

It is used by the application to manage transactions, read and write configuration data to the data store, etc.
6.1. DESIGN OF THE TRANSFORM APPLICATION

DPAPI Data Provider API

Applications that act as data providers (applications that provide some kind of data to Confd) register themselves as data providers and register callbacks related to some events. When the registered events happen, the callbacks are invoked by Confd and the application code gets executed.

CDB API Configuration DataBase API

Applications can access the database, read and write configuration data using this API.

The Erlang API for Confd (Fig 6.1) provides programmatic access to these socket API’s, in the Erlang programming language. The modules most relevant to the Transform Application in the API are `econfd`, which provides methods for registering data callbacks, and `econfd_maapi`, which provides methods for accessing the data in the Configuration Database through the management agent using Management Agent API (MAAPI). `econfd` also has methods to create sockets to connect to Confd, start and stop transactions and so on. Other important modules include `econfd_schema`, that has methods to access the schema information of the YANG models loaded into Confd, `econfd_cdb` that provides methods to access the Configuration Database, and has methods for registering for subscription and notifications.

Of these, applications that perform transformations need access to MAAPI and Data Provider API (DPAPI). The application registers some callbacks using DPAPI, and when those callbacks are invoked, the application accesses the base model data using MAAPI. The flow of the application can be better understood using an example transformation.

A simple transformation

Consider the two YANG models shown in Figure 6.2. The base model that is stored in the database is shown on the right, and the virtual model that will be exposed to the user is shown to the left.

Here, the leaf `ipPort` in the virtual model corresponds to the concatenation of the leaves `ip` and `port` in the base model. Since the key of the list is same in both models (the leaf `server` of type string), there is no need to transform the keys of the lists between the base and the virtual models. In real cases, often, the keys will be different, and they too have to be mapped between models.

In Figure 6.3, the base model is populated with some values. Now, when there is a query for data in the virtual model, say, to show the full configuration, the result must be as per the format of the virtual model. So, Confd invokes the necessary callbacks on the Transform Application which gets the relevant information from the base model, manipulates them as necessary and gives the result back to Confd in the format expected by the virtual model. Finally, the results will look like Figure 6.4 in the virtual model.
CHAPTER 6. THE TRANSFORM APPLICATION

container virtualModel {
  list server {
    key name;
    dsl: key 'kn = same(name)';
    dsl: path *
      L = */baseModel/server{kn}/*;
      bip = L + *ip*;
      bnm = L + *netmask*;
      bport = L + *port*;
    leaf name {
      type string;
    }
    leaf ipPort {
      dsl: map 'bip, bnm = splitcolon(self)';
      type string;
    }
    leaf netmask {
      dsl: map 'bnm = same(self)';
      type string;
    }
    dsl: fun 'splitcolon :
      @1, @2 = split(': ', $1)';
  }
}

class container baseModel {
  list server {
    key name;
    leaf name {
      type string;
    }
    leaf ip {
      type inet:ip-address;
    }
    leaf netmask {
      type inet:ip-address;
    }
    leaf port {
      type inet:port-number;
    }
  }
}

Figure 6.2: A Simple Transformation: Virtual and Base models

admin@Master> show configuration baseModel
server s1 {
  ip  1.2.3.4;
  netmask 10.20.30.40;
  port  99;
} server s2 {
  ip  192.168.1.1;
  netmask 255.255.255.0;
  port  12345;
} server s3 {
  ip  10.10.10.1;
  netmask 255.255.0.0;
  port  123;
}

Figure 6.3: Base model data for the simple transformation
Hence, for creating a Transform Application, all that is needed is to implement the necessary callbacks, and then register them to Confd. The basic callbacks needed to implement a fully working transform are get_elem, set_elem, get_next, create and delete callbacks. The exact functionality of these callbacks and how they are implemented will be explained in Section 6.2.

6.2 Implementation of the Transform Application

This section explains the implementation details of the Transform Application. The Transform Application is written in Erlang [1]. Erlang is a general-purpose, functional programming language.

While the previous approaches in Tail-f Systems, such as the 7-Transform and the MetaCLI framework are written in C, in this project, we chose to use Erlang to write the application. The following are the reasons for choosing Erlang:

- Confd itself is written in Erlang and hence it runs inside an Erlang Virtual Machine. Now, if the Transform Application is in a language other than Erlang, there will be a lot of context switches between the Erlang VM and the other language for every callback invoked. If the Transform Application is also in Erlang, all those context switches will be saved. Hence, from a performance point of view, Erlang is beneficial.

- Database transformation is a very high-level problem (in terms of abstraction). We need to be able to specify associations and mappings easily. A language as low-level as C, is not the optimal choice to deal with this problem.

An overview of Callbacks

As mentioned in the previous section, for creating a Transform Application, the necessary callbacks must be implemented. An overview of the most important callbacks is as follows:
get_elem This callback is invoked, when there is a request for a particular leaf in the virtual model. For example, in the example given in Figure 6.2, if there is a query for the leaf /virtualModel/server{s1}/ipPort, the get_elem callback will be invoked with the virtual path as the argument. It is then the job of the callback function to fetch the relevant information from the base model, (which in this case is in /baseModel/server{s1}/ip and /baseModel/server{s1}/port), manipulate them (for example, concatenate), convert them to the relevant type if necessary, and return the result in the format that the virtual model expects.

set_elem This callback is invoked, when a particular leaf in the virtual model is set to a value. For example, in Figure 6.2, if the leaf /virtualModel/server{s1}/netmask is set to the value 10.10.10.10, the set_elem callback will be invoked with the virtual path and the value as the arguments. The set_elem callback takes the value supplied, and writes them to the relevant leaves in the base model, after some manipulations (In this case, it converts the value from string representation into an ip-address representation).

create This callback is invoked if there is a new list instance to be created in the virtual model. For example, if a new server instance is created in the virtual model, create callback is invoked, with the virtual path as the argument. It creates the corresponding server instance in the base model. If there are key mappings, they are taken into account as well.

delete This callback is invoked if an existing list instance is to be deleted in the virtual model. For example, if /virtualModel/server{s1} is deleted in the virtual model, delete callback is invoked with the virtual path as the argument. It deletes the corresponding /baseModel/server{s1} instance in the base model. If there are key mappings, they are taken into account as well.

get_next This callback is invoked to iterate over the instances of a list in the virtual model. For example, if there is a show configuration request on a list in the virtual model, (say /virtualModel), this callback is invoked. Repeated invocations of the get_next callback returns successive instances of the list, until all the instances are exhausted. After the last entry, get_next invocation returns {ok, done}.

These are the callbacks necessary for a minimal functional Transform Application. The Table 6.1 lists the arguments and the return values of the callbacks.

Implementation of the Callbacks

Figure 6.5 shows the sequence of operations, when the Transform Application is running. The first step is registering the callbacks. Once all the callbacks are registered, the application is ready to receive requests from ConfD. When there is a query to the virtual model, ConfD invokes the relevant callbacks in the Transform Application.
6.2. IMPLEMENTATION OF THE TRANSFORM APPLICATION

### Table 6.1: Arguments and return values for the Callbacks

<table>
<thead>
<tr>
<th>Callback</th>
<th>Arguments</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_elem</td>
<td>Virtual path</td>
<td>virtual model value</td>
</tr>
<tr>
<td>set_elem</td>
<td>Virtual path and virtual value</td>
<td>ok</td>
</tr>
<tr>
<td>create</td>
<td>virtual path</td>
<td>ok</td>
</tr>
<tr>
<td>delete</td>
<td>virtual path</td>
<td>ok</td>
</tr>
<tr>
<td>get_next</td>
<td>virtual path and cursor</td>
<td>next keys and next cursor</td>
</tr>
</tbody>
</table>

In the implementation of the callbacks, the aim is to come up with a generic method for implementing them, so that they work automatically for as many mappings as possible between any two YANG models. Of course it might not be possible to specify all possible mappings using the DSL comfortably, or it might be difficult to generate the mapping records correctly. In such cases, it should be possible to extend the Transform Application easily with user-written code, that covers the complex mappings that are not automatically generated.

```plaintext
get_elem(Tctx, [Leaf, Key, List, _Path]) ->
  ...
  case lists:member(Key, Keys) of
    true ->
      case Leaf of
        ipPort -> ...
```

**Application** The callbacks are executed, and the information in the database manipulated and returned in the appropriate format to ConfD. This is the sequence of operations followed in the transformation. Here, except for the registering of the callback, all the other steps together form a transaction. Each new query to the virtual model is handled as new transaction.

In the implementation of the callbacks, the aim is to come up with a generic method for implementing them, so that they work automatically for as many mappings as possible between any two YANG models. Of course it might not be possible to specify all possible mappings using the DSL comfortably, or it might be difficult to generate the mapping records correctly. In such cases, it should be possible to extend the Transform Application easily with user-written code, that covers the complex mappings that are not automatically generated.

```plaintext
get_elem(Tctx, [Leaf, Key, List, _Path]) ->
  ...
  case lists:member(Key, Keys) of
    true ->
      case Leaf of
        ipPort -> ...
```
In the initial attempts on the Transform Application, the callbacks were implemented using a case statement, where all possible paths were handled separately. That is, the application was hard-coded to handle some particular mappings between particular YANG models. An example callback implementation is shown in Listing 6.1. Since the mappings are handled in the case-statement itself, there was no need for the mappings to be specified in a separate file as well. This worked very well, but it was limited to the particular YANG models that were being mapped. If we need to use another YANG model or a different mapping, then the application has to be hand-written again, mapping the paths of the new models.

This level of generality is obviously not enough. Hence, in the next iteration, it was decided to use a separate module to specify the mappings, and to get the mappings from that module, when the callbacks are invoked.

The Mappings Record

The data structure that can be used to specify the mappings must be an associative array or a dictionary. Then, we can have many properties for the mapping, and for each property, the corresponding value can be stored in the dictionary. In Erlang, the data structures that can be used for storing an associative array are Erlang records [8] or ETS (Erlang Term Storage) tables [9], which is Erlang’s inbuilt database system. Both of these structures use named tuples [10] internally.

```
-record(mappings, {  
  basepaths = [], %% [econfd:tagpath()]  
  virt2basekeys, %% fun()  
  base2virtkeys, %% fun()  
  get = none, %% fun()  
  set = none, %% fun()  
  create = none, %% fun()  
  delete = none, %% fun()  
  next = none, %% fun()  
}).
```

Listing 6.2: The Mappings Record

If we are using ETS tables, the mappings should first be pushed into the database. And then for every callback invoked, there will be a database query. However, using a full-fledged database system is an overkill for just specifying such mappings. Moreover, if plain records were used, there would be no need to execute
6.2. IMPLEMENTATION OF THE TRANSFORM APPLICATION

code to put the data into a database. Hence we decided to specify the mappings using records. The record that is used in the application is as shown in Listing 6.2.

The record consists of one list, `basepaths`, and seven `fun’s`, which are either anonymous functions or named functions defined in the `ec_mappings` module. The `basepaths` element is a list of lists, which contains the paths of all the elements in the base model that corresponds to the given element in the virtual model. Next follow seven fields, whose values are functions, that can be passed as parameters to other functions which can eventually execute them.

The field `virt2basekeys` holds the function that transforms the keys of a list in the virtual model to the keys of the corresponding list(s) in the base model. The reverse of the function, i.e., a function to transform the keys of a list in the base model to that in the virtual model is held in the field `base2virtkeys`. These two fields must be specified for all elements in the virtual model, be it lists, leafs or containers. The other fields are not mandatory for all types of elements.

The fields `get` and `set` are required only for leafs, and not for lists. The values that are taken from the base model must be modified in some way to make it suitable for the virtual model. This is done by the function in the `get` field. Similarly, before setting a value in the base model, it must be modified to suit the requirements of the base model. This is done by the function in the `set` field. The other fields in the record are required only for lists, and not for leafs. The `create` field holds a function which creates new list(s) in the base model, when a corresponding list is created in the virtual model. So, the `delete` field holds a function to delete lists(s) in the base model, when a corresponding list is deleted in the virtual model. The `next` field has a function that iterates through a list in the base model, and transforms the keys of that list into the keys of the corresponding list in the virtual model. These are the fields in the record, and these fields together specify a mapping from each element in the virtual model to the base model.

Since the development of the DSL Compiler and the Transform Application were going on in parallel, the mappings records were initially hand-written, since the complete records could not be auto-generated at that time. The application was developed and tested using hand-written records.

Implementation of `get_elem`, `set_elem`, `create` and `delete`

The UML class diagram for the Transform Application, along with the parts of ConfD that it uses the most, is shown in Figure 6.6. Since Erlang is a functional programming language, concepts pertaining to object oriented programming like classes, objects and inheritance are not available. An Erlang application is composed of modules. So, this diagram just indicates the important methods, which modules they are present in and which module calls them.

The important methods that the application module `ec_transform` uses are:

- `register_*_cbs` methods, which are used to register the data provider callbacks to ConfD
- `get_mappings` method, which is used to get the mappings records that are auto-generated by the DSL Compiler
- `ecofdl_maapi:*` methods, which are used to access the configuration data of the base model, from the database

```
ecofdl

record confd_trans_cbs
record confd_data_cbs
record confd_db_cbs

start() : ok
init_daemon() : Pid
register_trans_cb(p:Pid, t:confd_trans_cbs) : ok
register_data_cb(p:Pid, d:confd_data_cbs) : ok
register_db_cb(p:Pid, d:confd_db_cbs) : ok
register_done(p:Pid) : ok
```

```
econfd

start(): ok
init(): ok
finish(): ok
get_elem(): ok
set_elem(): ok
get_next(): ok
create(): ok
delete(): ok
```

```
e_transform

record mappings

start(): ok
init(): ok
finish(): ok
get_elem(): ok
set_elem(): ok
get_next(): ok
create(): ok
delete(): ok
```

Figure 6.6: UML class diagram of the Transform Application

An algorithm for implementing the `get_elem` callback is described in Listing 6.3. As the callback is invoked, the first thing to do is to get the mappings record corresponding to the virtual path, using the `get_mappings` method in the `ec_mappings` module. Once we have the record, we pattern match the fields to get the fields that we are interested in. For example, for the `get_elem` callback, we will need the `basepaths`, `virt2basekeys` and `get` fields. For the `set_elem` callback though, we will need the `set` field instead of the `get` field.

1) Get the mappings record corresponding to the virtual path
2) Get the keys of the list in the virtual path
3) Transform them into keys to the corresponding list(s) in the base model
4) Build the base model paths with the above keys
5) Check if the paths constructed above actually exist in the base model
6) If they do, get the values from the base model using the MAAPi
6.2. IMPLEMENTATION OF THE TRANSFORM APPLICATION

7) Apply the transform function to transform the value
8) Transform the value to the required type

Listing 6.3: General algorithm for Callbacks

In ConfD, the paths in the models are expressed as lists, with the keys in the lists denoted as tuples. For example, the path /baseModel/server{s1}/ip is represented internally in ConfD as [ip,{s1},server,baseModel]. The order is reversed for performance reasons, as the leaf and the nodes near the leaf are referenced much more often than the root of the tree.

Now, given a virtual path, the algorithm gets the keys of the list in that path. From the mappings record that we got earlier, the base model paths are retrieved. Then the keys transform function (virt2basekeys) is applied to the keys of the virtual list to get the keys of the list in the base model. Using these, the actual path(s) in the base model are constructed. Then the application checks if there is data actually stored in that path in the base model. If it is, MAAP calls are invoked to get the values in the base model. Then the transform function (from the get field) is applied to the values to transform them into the format required by the virtual model. If there is any type transformation to be done, that has to be taken care of as well. Finally, we have the value in the virtual model format, and we can return the value to ConfD. If there is no data corresponding to the base model paths, then the application returns {ok, not_found} as the result to ConfD.

The implementation of the other callbacks are very similar and follow the same steps. Only the fields being pattern matched differ. So, in the set_elem callback, instead of the get field, we match the set field. In create, we match the create field, to do additional tasks when creating new list(s) in the base model. Similarly, in the delete callback, we match the delete field, to do the necessary tasks when deleting list(s) in the base model.

Mapping lists between models

Section 4.1 discusses the types of mappings that can occur when mapping lists in virtual models to lists in base models. Since lists can have multiple instances, when there is a query to a list in the virtual model, the application not only has to figure out which list in the base model the request maps to, but also which particular instance of the list. Mapping lists essentially means being able to transform the keys of the list in one model into the keys of the corresponding list in the other model. In other words, given the keys of a list in one model, the application should be able to construct the keys of the relevant list in the other model. There can be multiple keys for each list, and lists in the two models can have different number of keys. Taking all these into account, mapping lists correctly is quite a difficult problem.
Implementation of \textit{get\_next}

\begin{verbatim}
-record(maapi_cursor, { n = -1, secondary_index = '',
  preterm = first,
  ikp, isrel, socket, thandle,
  cursor_id}).
\end{verbatim}

Listing 6.4: MAAPI cursor for iterating through a list

The only callback that is significantly different from all the others is the \textit{get\_next} callback. Since this involves iterating through the instances of an already existing list, the algorithm is a little different. Steps to be followed for this callback are as shown in Listing 6.5. \texttt{ConfD} keeps track of the instances of a list using a data structure called the cursor. \texttt{ConfD} uses the record shown in Listing 6.4 as the cursor. It contains information like the current instance’s number, the keys of the previous instance, the path of the list, socket used to connect to \texttt{ConfD} etc. In this, the only information useful to us is the previous instance’s keys, denoted by \texttt{preterm}. When the callback is invoked for the first time, \texttt{ConfD} sets the value of the cursor to be -1. In this case, we have to initialize the cursor using the method \texttt{econfd\_maapi\_init\_cursor}. This returns a cursor having the keys of the first instance. For the second time, the \textit{get\_next} callback is invoked with the cursor that was returned the last time. By successively invoking the callback with the last returned cursor, we can get the keys of all the instances in the list. Once all instances are exhausted, the next call returns \{\texttt{ok, done}\} as the result. From this, we can know that the list is fully iterated.

\begin{verbatim}
A. When the cursor is -1, 
  1) Get the mappings record corresponding to the virtual path
  2) Get the base model list(s) corresponding to the virtual list
  3) Initialize the cursor to the base model list(s)
  4) Get the next cursor and the keys of the base model list(s) using MAAPI
  5) Transform into keys of the base model list(s) into keys to the virtual list
  6) Return keys and the next cursor

B. When the cursor is not -1, 
  1) The same steps as above, except initializing the cursor
\end{verbatim}

Listing 6.5: Algorithm for the \textit{get\_next} Callback
For example, using the data in Figure 6.2, invoking the get_next callback the first time, the returned value of prevterm will be \{1, \{"s1"\}\}, where 1 is the instance number and s1 is the actual key. Invoking the callback again, with this cursor will result in another cursor with the prevterm set as \{2, \{"s2"\}\}. This goes on, until the whole list is traversed. After the last key is returned, invocation of the callback with the last cursor will return \{ok, done\} as the result, which indicates that the list is fully traversed.

In the implementation of the get_next callback, the difference is that, instead of virt2basekeys, we need the field base2virtkeys. That is, in all other callbacks, ConfD would provide us with the virtual list’s keys, and we need a method to transform them into keys to the list(s) in the base model. However, in get_next callback, we would not be supplied with the keys in the virtual list. After invoking the MAAPI call, we would have access to the base lists’ keys, and we now need a method to transform them into keys to the list in the virtual model. Hence, only for this callback, we need the base2virtkeys method to be specified. Other than that, the steps in the implementation are similar to the other callbacks.

Limitations of the current implementation

In the current implementation, the fields base2virtkeys and virt2basekeys accept a single function. The design allows only one argument to the function. Hence, both the fields take a function that has a single list of all the keys in the path as the argument. This was enough for the simple mappings between lists. But for some complex mappings of lists between the two models, this approach is clearly not expressive enough. Currently, the more complex mappings between lists can only be handled by user-written code specific to the YANG models being mapped. The characteristics of the mappings supported by the application are explained in Section 7.1.
The following chapter describes the results of the project. The characteristics of the YANG model mappings supported by the Transform Application and capabilities of the DSL grammar and the DSL compiler are discussed and evaluated.

7.1 Characteristics of mappings supported by the current application

YANG models are organized into modules, which are structured using containers, lists and leaves. Since containers by themselves have no significance on the configuration, they are easy to map. Mapping containers between models is just a matter of specifying the appropriate paths in the basepaths field in the mappings record. Hence, there is no problem in mapping containers from one model to the other.

In section 4.1, the ways in which leaves in the virtual model can be mapped to leaves in the base model are discussed. Of these, the Transform Application supports the first two, i.e., one-to-one and one-to-many mappings from the virtual to the base model. Many-to-one (M-to-1) and many-to-many (M-to-N) mappings require temporary storing of the virtual model values, before committing. Only when all the M values in the virtual model are available, they can be committed. In order to keep the application relatively simple, this feature was not implemented. Hence, many-to-one and many-to-many mappings for leaves are not handled by the transform application.

Lists are the most complex structures to map. Some of the mappings described in Section 4.1 are fully handled by the application. Some others require user-written code in addition to the generated mappings to do the transformations. Some mappings are not handled at all.

Of the mappings in Section 4.1, the default case (simple list in virtual model mapped to simple list in base model), reordering (keys in virtual model reordered to match keys in the base model) and duplication (one list in virtual model maps to multiple different lists in base model) are fully handled, meaning, the generated code
alone is enough to transform the models. Nested lists are not handled fully, since the current implementation of the key mapping functions are not expressive enough. Hence, *encapsulation* is not handled by the application, although the transformation can be achieved by including user-specified mapping records and the required functions for those lists. It’s not an easy task, but can be done, nonetheless.

On the other hand, the cases of *decapsulation* and *unification* are not handled by the application, even with user-written code. The reason for this is that these kinds of mappings are similar to the many-to-one case of the leaf mappings. They require storing some values in memory and waiting for the necessary elements to be configured before they can be committed. For example, in the *unification* case, two lists in the virtual model correspond to one list in the base model. Hence, when configuring the base model, the changes cannot be committed until both lists in the virtual model are configured.

The cases of *promotion* and *demotion* are also not handled. These cases require additional constraints on the lists in the models, in order to make any sense. Since these cases map a leaf in one model to a list in the other, the lists must be constrained somehow. Lists can have multiple instances, while leaves can only have a single value. So, when a list corresponds to a leaf, only the first instance of the list will be relevant. So, there should be constraints on configuring such a list (have max-elements to be 1, for example), or some other strategy to deal with the list (like ignoring everything but the first element). These cases are not handled in the current implementation of the transform application. Table 7.1 summarizes the characteristics of the supported mappings.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Fully Handled</th>
<th>Handled with user-written code</th>
<th>Not handled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaves</td>
<td>One-to-One</td>
<td>-</td>
<td>Many-to-One</td>
</tr>
<tr>
<td></td>
<td>One-to-Many</td>
<td></td>
<td>Many-to-Many</td>
</tr>
<tr>
<td>Lists</td>
<td>Default</td>
<td>Encapsulation</td>
<td>Decapsulation</td>
</tr>
<tr>
<td></td>
<td>Reordering</td>
<td>Unification</td>
<td>Unification</td>
</tr>
<tr>
<td></td>
<td>Duplication</td>
<td>Promotion</td>
<td>Promotion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Demotion</td>
<td>Demotion</td>
</tr>
</tbody>
</table>

Table 7.1: Characteristics of mappings supported by the Transform Application

Further to the above, mappings in which there are some elements in the virtual model that are not mapped to any elements in the base model cannot be handled by the transform application. This is because, to handle such cases, we have to create the corresponding elements in the base model as hidden entries, so that they are not visible in the base model, but still available to the virtual model when it reads data from the base model. This is currently not possible in the application.

In most real-world modules, nested lists and many-to-one/many-to-many mappings are fairly common. As of now, when mapping two models in a real application, most of the mappings will have to be hand-written. As this is just a prototype, and
not a full-fledged application, this shortage is to be expected. The focus is on the approach to the problem, and not in making the solution market-ready. Hence, in a practical sense, this prototype is still in its infancy.

7.2 Evaluation of the Grammar

The grammar of the mapDSL was evaluated based on metrics used in [16] and derived from [25]. The term mapDSL is used here to refer to the grammar developed for the DSL Compiler. To measure the complexity of parsers, most of these metrics have been derived directly from software metrics. A script that automatically evaluates a grammar according to the chosen metrics was implemented in python. It can read grammar implemented as a PLY directly and will output values of the metrics described below.

**Number of non-terminals (VAR)** A higher number of non-terminals is assumed to correlate with greater maintenance effort, as change to any definition of a non-terminal may impact others as well.

**McCabe cyclomatic complexity (MCC)** The metric equals the number of alternatives in the grammar. Any alternative could lead to a potential parsing conflict.

**Halstead effort metric (HAL)** A measure of the effort required to understand the grammar.

**Average of right hand side size (AVS)** An estimate of the number of symbols to be expected on average on the right-hand side of a grammar rule. A high AVS correlates with low readability and possibly impact the parser performance. It should be considered with the total count of the number of non-terminals.

**Number of grammar rules (PROD)** The number of rules in the Grammar influences the lines of code used in the compiler-compiler and has an impact on the ease of maintenance.

The comparison with the grammar of genet (see 3.3 for details) is shown in Table 7.2. In general, the grammar of genet is smaller (around half the size in terms of TERM, VAR and PROD) than the one used in this project, simply because genet covers comparatively few cases and does not have the same functionality. In contrast to genet, this project’s grammar grew more complex (MCC and HAL) and yet is easier to read (AVS).

In Table 7.3, the grammar used in this project was also compared to the grammars presented in [16]. The aim of this comparison was to achieve a more generic evaluation. Data of DSLs defined in [16] were used for this comparison. A description of the DSLs from the original article follows (citations from the original article are omitted):
CHAPTER 7. EVALUATION

One way of determining how representative, is to compare it against other DSLs. The following DSLs were chosen for comparison with FDL:

- Production Grammars (PG) for software testing,
- A DSL that allows experimentation for the different regulation of traffic lights (RoTL) and supports the domain-specific analysis of junctions,
- Context-Free Design Grammar (CFDG), designed for generating pictures from specifications. DSL domain is similar to Elliot’s work,
- GAL, a well-known DSL used to describe video device drivers, and
- Extended BNF (EBNF), a standard notation used to unambiguously define language grammar.

Against the other grammars it occurs that the number of non-terminals and production rules are rather high (VAR) and the number of alternatives denoted by the MCC value is a bit higher than the other DSLs but otherwise it performs well. The reading complexity (AVS) is on one level with other DSLs and the effort required for understanding (HAL) is unusually low.

It can be concluded that the grammar used in this project is fairly complex, and yet remains readable.

7.3 Evaluation of the Compiler

For the analysis of the DSL Compiler and the comparison to genet (see Section 3.3 for details), the static code analyzer Radon [17] was chosen. Radon implements var-
7.3. EVALUATION OF THE COMPILER

In this section, various code metrics [18] are evaluated. Among those, there are several different measures regarding the number of code lines. A short explanation follows.

**lines of code (LOC)** The total number of lines of code.

**logical lines of code (LLOC)** Every logical line consists of a single statement.

**source lines of code (SLOC)** Similar to LOC but excluding blank lines.

**Cyclomatic Complexity (CC)** The average of the number of possible paths through every function or method in the code.

**Maintainability Index (MI)** From [18]: 'The maintainability index is calculated as a factored formula consisting of SLOC (Source Lines Of Code), Cyclomatic Complexity and Halstead volume. It is used in several automated software metric tools, including the Microsoft Visual Studio 2010 development environment, which uses a shifted scale (0 to 100) derivative.'

Table 7.4 reflects a more detailed analysis of the code. Of these listings, the results for the parser should be considered with caution. It is unlikely that the static code analyzer can follow the metaprogramming tricks used in the compiler-compiler PLY.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>LOC</th>
<th>LLOC</th>
<th>SLOC</th>
<th>CC</th>
<th>MI</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>92</td>
<td>74</td>
<td>79</td>
<td>1.2</td>
<td>55.9</td>
</tr>
<tr>
<td>Lexer</td>
<td>171</td>
<td>124</td>
<td>142</td>
<td>1.4</td>
<td>57.7</td>
</tr>
<tr>
<td>Main Module</td>
<td>24</td>
<td>19</td>
<td>20</td>
<td>1.0</td>
<td>100.0</td>
</tr>
<tr>
<td>Node Visitor</td>
<td>171</td>
<td>148</td>
<td>160</td>
<td>1.6</td>
<td>53.1</td>
</tr>
<tr>
<td>Parser</td>
<td>253</td>
<td>224</td>
<td>240</td>
<td>1.0</td>
<td>53.4</td>
</tr>
<tr>
<td>Types</td>
<td>110</td>
<td>85</td>
<td>92</td>
<td>1.2</td>
<td>57.0</td>
</tr>
<tr>
<td>Symbol Table</td>
<td>91</td>
<td>62</td>
<td>77</td>
<td>2.0</td>
<td>70.4</td>
</tr>
<tr>
<td>Outp. Model</td>
<td>749</td>
<td>558</td>
<td>658</td>
<td>1.6</td>
<td>34.1</td>
</tr>
<tr>
<td><strong>YANG</strong> Parser</td>
<td>304</td>
<td>215</td>
<td>260</td>
<td>2.6</td>
<td>41.3</td>
</tr>
</tbody>
</table>

Table 7.4: Analysis of DSL Compiler’s code

From this table, it is visible what part of the DSL Compiler required the most work. That has been without question the Output Model, the YANG Parser, the Parser, and the Node Visitor. The Output Model suffered from a great number of cross references between classes and with the Node Visitor together contained most of the logic, hence complexity, of the DSL Compiler. The Parser in turn contained the complex grammar.

In Table 7.5, the code characteristics of the developed DSL Compiler is put into contrast with those of the genet project. To this end the values used in Table 7.4 are aggregated as follows. LOC, LLOC, and SLOC are simply summed. For CC the code analyzer provided a mechanism to generate an average. Only for the MI
no solution was available. In this case the lowest score from all the modules in Table 7.4, the Output Model, was chosen.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>LOC</th>
<th>LLOC</th>
<th>SLOC</th>
<th>CC</th>
<th>MI</th>
</tr>
</thead>
<tbody>
<tr>
<td>genet</td>
<td>1377</td>
<td>1023</td>
<td>1253</td>
<td>1.7</td>
<td>9.3</td>
</tr>
<tr>
<td>mapDSL</td>
<td>2019</td>
<td>1552</td>
<td>1776</td>
<td>1.5</td>
<td>34.1</td>
</tr>
</tbody>
</table>

Table 7.5: Code Comparison to genet

The increase of features lead to a growth in lines of code. For the MI even with the pessimistic aggregation function, it is clearly visible that the *mapDSL* has a greater ease of maintainability than *genet*. It is likely that this difference stems from the greater modularization of the code. As can be seen from Chapter 5, the code of this project is spread over multiple modules, each assigned to a particular stage of the [DSL Compiler]. The CC is roughly the same for both projects.
8.1 Contributions

The contributions of this project are manifold. We provided two functional software packages: The DSL Compiler and the Transform Application. More important is the analysis of the problem. When the project started, the description of the problem existed only in the code base of previous projects (see Chapter 3). Now, the projects are brought into a common context and their evolution is documented. Furthermore, a systematic analysis of the problem has been worked out in Chapter 4. Even though the solution here has its limitations, as described in Chapter 7, the analysis of the problem of the project give an overview of the possible extensions. In short, the biggest contribution is that a problem with unknown boundaries has been transformed into a well-understood problem.

The described work has expanded previous work. While the hard coding of any aspect of the transformations is still supported, as it is in 7-transform, it has become greatly unnecessary due to the introduction of the Domain-specific Language (DSL). In contrast to MetaCLI, which would only allow the specification of function names in the YANG model, it is now possible to define functions purely in the DSL or to add constants and change them throughout the YANG model. Where MetaCLI and 7-transform would use C for the Transform Application, this project implemented it in Erlang. By using Erlang, the size of the Transform Application shrinks and the code can be integrated with ConfD in one Erlang Virtual Machine, making context switches unnecessary. genet is the closest to a predecessor the presented project has, still due to genet being an incomplete prototype a comparison is hard. In its current form, expressions in genet can be expressed more brief than in the DSL presented in this project. The reason behind this is that genet can only express simple operations (additions, subtractions, etc.). Those operations are represented by using operators. The presented DSL does not allow the use of operators for purposes of mapping. Instead function calls have to be used. However, in contrast to genet, reuse through scoping and the definition of more complicated functions is
possible. Additionally, the shorter notation using operators could still be introduced when the [DSL] reaches maturity.

8.2 Future work

During the development of both the [DSL Compiler] and the [Transform Application] a number of features became visible that would enrich the product. Only a fraction of those could actually be realized in the scope of the project. The others are documented below.

Future work for the [DSL]

There have been several limitations that restrict the number of mappings that are expressible in this [DSL]. In principle, this is expected. It does not seem possible at this point to cover all mappings. This is the reason why the extensibility has been discussed in Chapter 4. Still the coverage of the [DSL] could be greatly extended by adding the following options:

All the limitations mentioned in Section 4.1 are valid starting points for improvements of the [DSL]. First of all, the restriction of covering 1:N cases only should be loosened. To that end a smart way of storing incomplete information has to be developed either using hidden [YANG] entries or a data structure in the transformation engine. Next, conditional statements and loops should be implemented. This is the major difference that makes it difficult to compare the [DSL] to the MetaCLI (see Section 3.2) project.

In the future, it might be interesting to review how the mappings are defined. Instead of using simple text files, it could be easier to use a sort of [Integrated Development Environment (IDE)] to visually construct the mappings (similar to the way they are shown in Chapter 3).

Future work for the [Transform Application]

The [Transform Application] currently handles some types of mappings between YANG models, and for some others, it requires user-written code. There are many areas where the application can be substantially improved.

The foremost focus should be on supporting more kinds of list mappings. The [Transform Application] could handle many more mappings, if it was capable of supporting many-to-many mappings. For instance, enabling the application to store values in the virtual model that are not yet ready to be committed, will help in handling a lot more list mappings.

The next important feature is the handling of nested lists. The application can be modified, so that the functions [base2virtkeys] and [virt2basekeys] could be specified for mapping keys for nested lists. Adding these two features alone would make the [Transform Application] much more stable and capable of handling more mappings.
Another important area for consideration is the type system. Currently, ConfD has some support for converting between built-in data types and their string representations. However, there is no easy way to automatically convert between derived types like enumerations, unions, and other complex types. Making it easier to convert between data types in the Transform Application would allow for even more versatile mappings, as the mappings are not restricted by data types.

8.3 Experience

In the following, the authors will voice their perspective on the project. Problems of the project will be explained and possible alternatives shown.

Niklas Semmler

The challenge of this project lay not so much in mastering a single piece of software or understanding a single subject in depth, but in understanding diverse fields to varying degrees. The problem is connected to various concepts. First, it had to be reverse engineered from the existing code. Domain modeling techniques had their use in segmenting the problem domain into features (see Figure 4.3). The Language Design of the DSL had to be developed and then implemented via DSL Compiler construction. The implementation required both the use of Python for the DSL Compiler and the functional programming language Erlang for the Transform Application. And the Transform Application had to be developed to interface with ConfD with little help from the documentation.

Given that I am originally from the field of Computer Networks, there were major gaps of knowledge that had to be filled. If I had the chance to redo the project with the existing knowledge, I would follow a stricter bottom-up approach. Instead of developing the DSL Compiler and the Transform Application side-by-side, it would have made sense to concentrate first on a semantic model in Python that can generate the data structure for the Transform Application. It could then have been easier to proceed first with an internal DSL (or language exploitation) before working on the tedious development of an external DSL (or language invention). Compiler development has the big disadvantage that changes made in some component propagate through the whole pipeline.

In this way the development speed of the project and hence the scope of the project could be increased, but probably only with the knowledge that is now compiled in this document. Finally, it should not be underestimated how much a language to describe a problem helps. It makes it possible to talk about specific cases without wasting time on drawing the same diagrams over and over on a white board. With this document, I hope we have introduced that language.
Ramkumar Rajagopalan

This thesis was a very valuable learning experience for me. It gave me the opportunity, to learn many new technologies that I was not familiar with, including YANG modeling, Network Configuration Protocol (NETCONF) protocol and the Erlang Programming language.

I spent a lot of time reading the code of the existing solutions to the problem and analysing them. But the insight gained from the analysis was not significant enough, compared to the time invested in the effort. Also, I developed a prototype of the Transform Application by hard-coding the transform of a simple YANG model. The model that I chose initially might have been too simple. Since the design decisions in this hard-coded prototype were carried to the general solution, I might have been myopic about the consequences of some design choices, like the key-mapping functions. As a result, some mappings like nested lists are not handled at present.

So, given a chance to do the thesis again, and armed with the current knowledge, I would spend much less time on analyzing the previous solutions. I would also choose a slightly more complicated model for my initial hard-coded prototype. That would have eventually produced a general solution that can handle more mappings than what is currently handled.

Final words

Finally we want to thank Jan for his support during the project. He was never too busy to answer our questions or to show us the best lunch places in Stockholm. We also want to thank everyone at Tail-f, even though they will most likely never read this, for their support at the coffee machine and the great entertainment this project has been.
BIBLIOGRAPHY


81


**GLOSSARY**

**AST** Abstract Syntax Tree.

**AVS** Average of right hand side size.

**CC** Cyclomatic Complexity.

**CLI** Command-line interface.

**ConfD** A device configuration toolkit developed by Tail-f.

**data model** From [25]: ‘[A datamodel is] a collection of conceptual tools for describing data, data relationships data semantics, and consistency constraints.’ The term is used throughout the ConfD User Guide [12].

**DPAPI** Data Provider API.

**DSL** Domain-specific Language.

**DSL Compiler** The compiler developed to process the DSL.

**Erlang** A functional programming language specialized for parallelization, developed by Ericsson.

**GPL** General-purpose programming Language.

**HAL** Halstead effort metric.

**IDE** Integrated Development Environment.

**LLOC** logical lines of code.

**LOC** lines of code.

**MAAPI** Management Agent API.
management interface  In this thesis a management interface refers to a daemon
listening for requests in a certain management protocol (SNMP; NETCONF, ...
) at the protocol’s default port.

MCC  McCabe cyclomatic complexity.

MI  Maintainability Index.

MIB  Management Information Base.

NETCONF  Network Configuration Protocol.

PLY  Python version of Lex and Yacc.

PROD  Number of grammar rules.

pyang  A YANG validator, transformer and code generator.

Python  A easy readable general-purpose programming language.

SLOC  source lines of code.


Transform Application  Application interfacing with ConfD (via DPAPI and MAAP
for the purpose of performing transformations between two models.

UML  Unified Modeling Language.

VAR  Number of non-terminals.

YANG  The YANG modeling language.
EXAMPLE MODEL

Below the full virtual model used in the thesis is depicted. The [DSL] code is highlighted in a dark brown color. The corresponding base model is not shown. For an explanation of [DSL] statements used in the model, please view the chapters on the [DSL].

```plaintext
module virtualmodel {
    import dsl {
        prefix dsl;
    }
    container tm {
        list e {
            key 'x';
            dsl: def "more = '_foo'";
            dsl: fun "subone : @1 = sub(1, $1)";
            dsl: fun "addfoo : @1 = append(more, $1)";
            dsl: fun "muladd : _x = add(1, $1);
                @1 = mult(2, _x)";
            dsl: keys 'kx = subone(x)';
            dsl: path 'L = */tm/t{kx}*;
                py = L + *b*;
                pv = L + *c*';
        }
        leaf x {
            type uint8 { range 1..255; }
        }
        leaf y {
            dsl: map "py = addfoo(self)";
            type string;
            default "default";
        }
    }
}
```
leaf v {
  dsl:map "pv = muladd(self)";
  type uint8;
  default 255;
}
Niklas Semmler has taken part in the EIT ICT Masterschool. His study revolved around the double degree of Innovation & Entrepreneurship and Internet Technology & Architecture. He has written scripts and software for sensor networks, wireless and wired networks. During his work as a student worker at the Start-Up BENOCs he has helped to conceive a virtual network environment, that was used for the emulation of content delivery networks. His skill set includes advanced scripting in Python and various forms of network engineering including vulnerability assessment. He acquired his Bachelor’s degree in the field of Artificial Intelligence at the University of Amsterdam where he created a simulation of the slime mold physarum polycephalum. The fascination with this organism’s ability to spontaneously create reliable networks without any oversight has driven his passion towards networking.

Ramkumar Rajagopalan has taken part in the Masters Program under EIT ICT Labs Masterschool, where he focused on the double degree track of Innovation & Entrepreneurship and Internet Technology & Architecture. Previously, he has worked in Cisco Systems India for two years, on Charging and Policy interfaces for the Diameter Protocol. During his time there, he was working in the quality assurance of the mobile networks for CDMA/eHRPD architectures. He acquired his Bachelor’s degree in the field of Electronics and Communication Engineering from College of Engineering Guindy, India. He is skilled in network engineering and programming, mostly in Python. He is passionate about programming in general, and Software Defined Networking. He has recently acquired an interest for Functional Programming, especially in Erlang.