Document Number	
Based on Template	
Created By	Zhenggen Xu

SONiC configuration management design

Modification History

Revision	Date	Originator	Comments
0.1	12/5/2018	Zhenggen Xu	Initial Version
0.2	5/7/2019	Zhenggen Xu	updated

Table of Contents

Introduction

SONiC historically used static configuration file (e,g, minigraph.xml) to manage the configuration for different applications, and over time, the configuration was moved to a central configuration database (i,e, configDB) to support incremental configuration etc. However, the configuration management system still has a lot of challenges to resolve:

- There was no formal schema definition in a model language.
- Configuration syntax and dependency checks were missing
- No transactional and rollback options.
- No programmable interface for outside to manage the configuration
- Hard to extend CLI

This design document is to address the above challenges, this includes how the design fit the current architecture, what trade-offs it made and how to extend the system to different external management system etc, It also includes some examples of how the model is defined and used in the system. To make the change gracefully, the document also provides a first phase approach to address some key challenges.

Scope

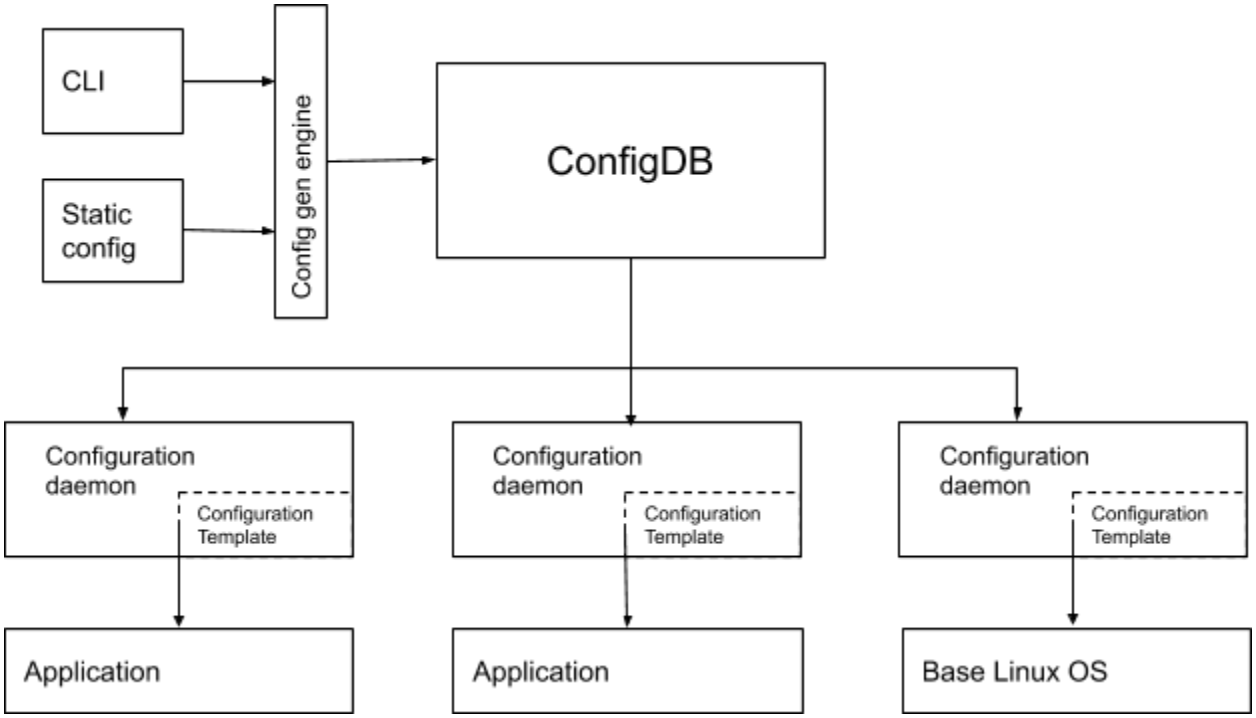
This document is mostly focused on configuration management part of the SONiC system, the SONiC backend interaction between different applications, feedback mechanism from bottom to up are out of scope of this document. Also, the document only covers the configuration aspect not the state data from SONiC.

Design

Current architecture

SONiC today has static configurable file : minigraph.xml (deprecated) or config_db.json for the system initial configurations. And it also has python click based CLI and/or use incremental config files to add/change/delete some configurations dynamically in configDB. The configuration changes will be picked up by each application (e,g, BGP, SNMP, VLAN, LAG, BaseOS etc) configuration daemon, and the changes will be applied directly or by template to the applications.

Current configuration management system architecture:



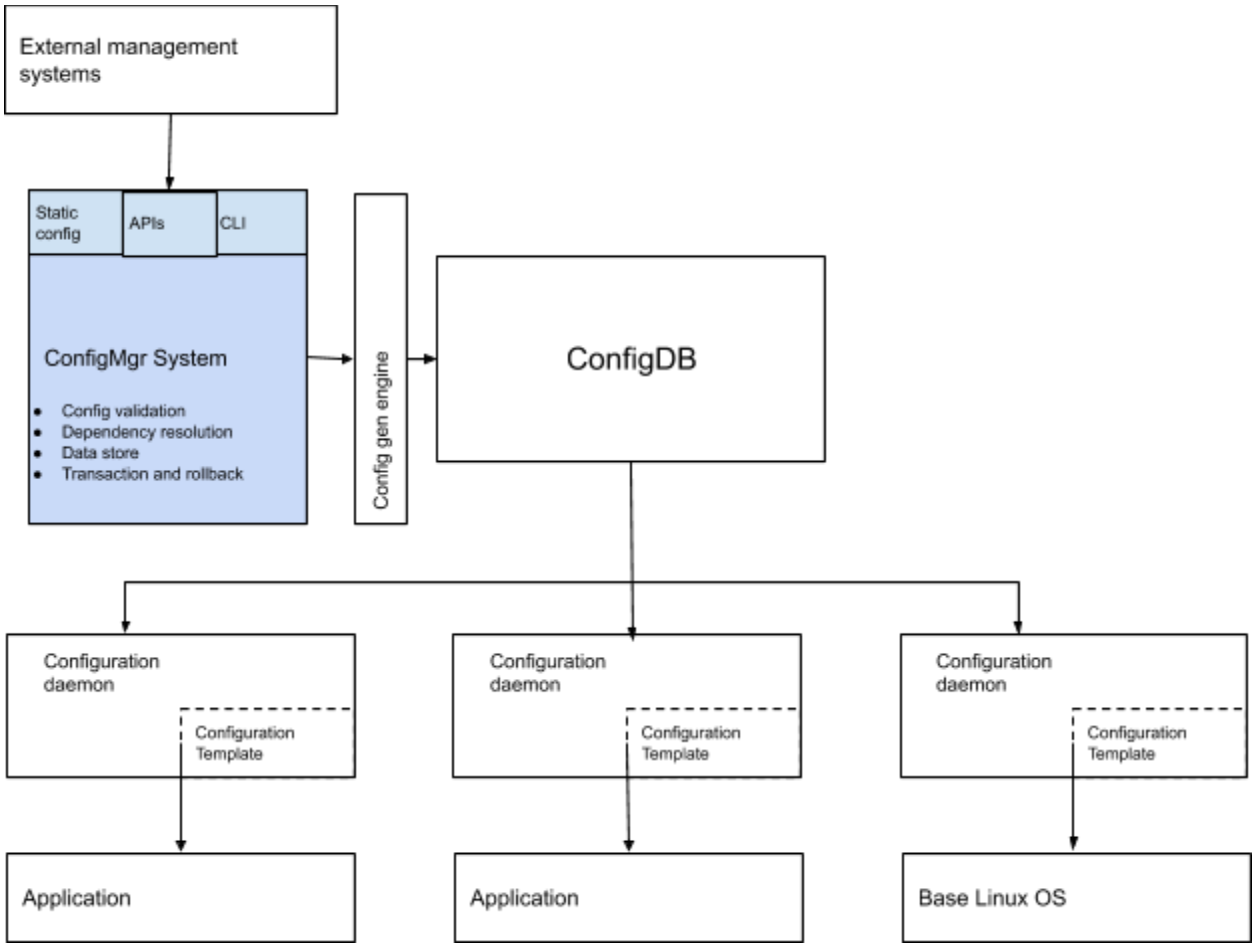
There wasn't any schema modeling in this design. The CLI or config file essentially writes the key/value pairs into configDB. No syntax validation nor dependencies checks before configuration was pushed to configDB. This leaves invalid configurations on the system which the backend have to deal with unnecessarily and this can cause unwanted behaviour and make the system in an inconsistent state. Lacking of the schema makes it very hard for external management system to use SONiC too, no formal contract is defined between external and internal systems.

Also, the current design was based on CLI commands and configuration files, that is not adequate in scalable networks. Network scripting using expect and screen scraping techniques is too primitive and unreliable to be used in large-scale networks.

Proposed architecture

To address the above issues, a model-driven management architecture is proposed. A separate configuration management system (service) is provided for external management system to use. The model defined on the system provides a formal contract between SONiC configurations and external management entities. Once the model is defined, the validation, dependency check and unifying north-bond interface became possible and extendable. On top of this, this system here will add data store for candidate/committed configurations, also support transactional commit and rollback.

See the proposed architecture for configuration management system on SONiC:



ConfigMgr System Design

To support programmable interface, we are leveraging gNMI protocol here : gNMI

The protobuf definition of gNMI is maintained in the openconfig/gnmi GitHub repository, but gNMI protocol can be used for any data with the following characteristics:

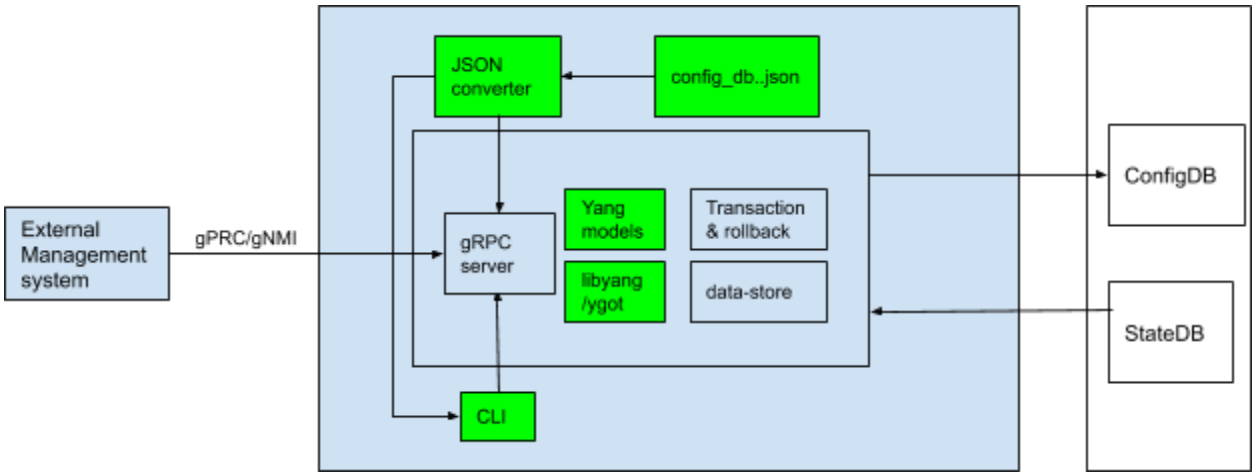
1. structure can be represented by a tree structure where nodes can be uniquely identified by a path consisting of node names, or node names coupled with attributes;
2. values can be serialised into a scalar object.

Currently, values may be serialised to a scalar object through encoding as a JSON string or a Protobuf type - although the definition of new serialisations is possible.

A gPRC server is built on the system to interact with external management system. To have a common backend code, configuration files and CLI will be talking to the gPRC server as well, this could be done via gNMI CLI interface.

Yang models, yang libraries, transaction/rollback and data-store components are built in the system, the system will write the key/value pair into the SONiC configDB system, also it could read the stateDB for some special case where we need to initialize the config data instance.

See ConfigMgr System internal architecture:



Yang models

YANG is a language originally designed to model data for the NETCONF protocol, but it could support other protocols like RESTCONF, gNMI etc. A YANG module defines hierarchies of data that can be used for the operation, including configuration, state data, RPCs, and notifications. This allows a complete description of all data sent between a client and server.

Instead of writing code tied to any particular user interface, YANG allows us to write API-agnostic code (in the form of callbacks) that can be used by any management interface. As an example, it shouldn't matter if a set of configuration changes is coming from a gPRC session or from a CLI terminal, the same callbacks should be called to process the configuration changes. This model-driven design ensures feature parity across all management interfaces supported by SONiC.

Yang models has a lot of tools/libraries available, which make it appealing for our design:
Yanglint : validation and conversion of the schemas and YANG modeled data.

Pyang / pyangbind Validate YANG modules for correctness, to transform YANG modules into other formats, and to generate code from the modules.
PyangBind is a plugin for Pyang that generates a Python class hierarchy from a YANG data model. The resulting classes can be directly interacted with in Python. Particularly, PyangBind will allow you to:

- Create new data instances - through setting values in the Python class hierarchy.
- Load data instances from external sources - taking input data from an external source and allowing it to be addressed through the Python classes.
- Serialise populated objects into formats that can be stored, or sent to another system (e.g., a network element).

Libyang: Parsing (and validating) schemas in YANG format, Parsing, validating, printing and manipulating instance data in XML/JSON format, also Support for YANG extensions. The software is BSD licensed.

Goyang: YANG parser and compiler for Go programs.goyang uses the yang package to create an in-memory tree representation of schemas defined in YANG and then dumps out the contents in several forms.
Ygot: Uses goyang as backend, Generates a set of Go structures and enumerated values for a set of YANG modules, with associated helper methods.

- Validate the contents of the Go structures against the YANG schema (e.g., validating range and regular expression constraints).
- Render the Go structures to an output format - such as JSON, or a set of gNMI Notifications for use in a deployment of streaming telemetry.

In this design, we will leverage the Yang language to model the configuration schema.

One decision needs to be made when using YANG is which models to implement. In general, using standard models is preferable over using custom models because the applications based on standard models can be reused for all network nodes that support those models.

However, the reality is that, not all network nodes using the same standard (IETF vs OpenConfig), not to mention that some of them do not support yang based models at all. Even they are supporting the same standard (e,g, OpenConfig), it is not guaranteed that it is compatible between them (e,g, major OpenConfig version changes between products).
Also, from an implementation point of view, it's challenging to adapt the existing code base to match standard models given that the codebase for SONiC is not small and code is already complex enough. It is also a challenge to switch to the configuration models gracefully if we are switching to standard model.

In the design, we are going to modeling the system in another way:
Describing the current SONiC schema in Yang model. This means we will figure out the current SONiC configuration schema and leverage the tools for parsing, validating, printing, and manipulating the model data. This solution has below pros and cons:
Pros:

- No or very little backend changes needed inside SONiC. The configuration management changes is somewhat isolated from other systems.
- Yang tools can still be leveraged.
- gRPC/gNMI can be used as transport

Cons:

- Programmable but not standard interface for north-bond

Even though having SONiC customized YANG models might not be the ideal solution, it is a big step forward for SONiC to have a model-driven management architecture, to support different type of north-bond interfaces and support configuration transactions. This is considered as an intended trade-off based on pros and cons mentioned above.

On the other hand, Yang module translator can be implemented to allow users to translate to and from standard YANG models by using translation tables. The Yang model translator can be used either outside of the box or on the box itself. However, extensive and complete research has not been done yet, so it is not very clear what can be achieved through the translation and if that could be a long term solution for supporting standard models.This tanslator details is out of scope of this document.

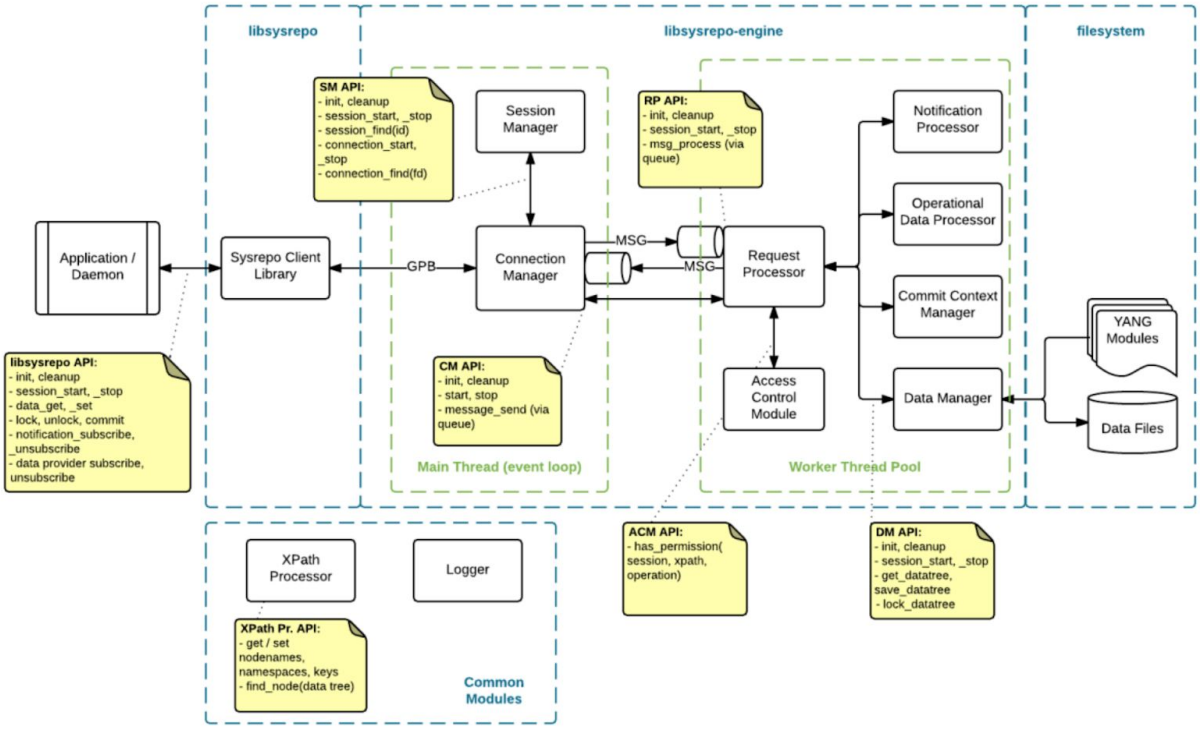
Data-store, transaction and rollback

In the current system, every command entered in the CLI is applied immediately. This could make the network in an inconsistent state until all planned changes are configured. Also, a series configurations could fail in the middle, it will require recovery if the configurations are related to each other.

In the new design, we are introducing the concept of candidate configuration. The candidates is a copy of running configuration at the beginning. It can be changed and shown before we consider it ready and commit it. The changes done in the candidate configuration are validated and applied to the running configuration. If the validations failed, we can simply rollback the changes in the candidate configurations.

The configuration changes will be transaction based, I,E, the changes will be applied all or nothing, and also the changes will be processed in a predefined order, for example, there is no difference if we configure two objects in one order (A then B) or the other (B then A).

To support this, we are leveraging open source project: [Sysrepo]



Active/Inactive configuration (Phase 2)

For some features, we would like to introduce an inactive configuration. E,g, We can have configurations (ACL, VLAN settings etc) defined on a port where the port does not exist yet.

To support this, we will leverage the data structure (e,g lyd_node) from libyang, as it can hold private data. The data made it possible to mark configuration commands or sections as active or inactive. With this feature, the CLI/API users can disable parts of the configuration without actually removing the associated commands, and then re-enable the disabled configuration later when necessary.

For user configurations, the configDB.json is defined as this:

```
 "Ethernet0": {
          "alias": "Eth1",
          "speed": "100000",
          "lanes": "65,66,67,68"
       },
       "Ethernet1": {
          "alias": "Eth1/2",
          "speed": "10000",
```

```
    "lanes": "66",
    "inactive": "true"
},
"Ethernet2": {
    "alias": "Eth1/3",
    "speed": "10000",
    "lanes": "67",
    "inactive": "true"
},
"Ethernet3": {
    "alias": "Eth1/4",
    "speed": "10000",
    "lanes": "68",
    "inactive": "true"
},
```

This configuration along with other configurations (e,g, ACL, VLAN etc) will be loaded through configuration management system, it will leverage the Yang model private data, and keep the configurations in the configMgr system but it won't be pushed to SONiC configDB itself. This includes any configurations that applies to the non-existing port, for example, ACL, LAG, VLAN configurations. By doing this, no changes need to be made to SONiC backend for the inactive configurations. The configurations save commands would be adjusted so that the inactive configurations would be saved to the configuration file too.

Some special validation

There were special validation logic that is not part of Yang models, we need to do such validation for such configurations. For example, if we create a port with number of lanes, where the lanes were used/shared already by other ports, the create of port should fail. It might not be straightforward to define those checks in the Yang model, for such checks, it should be put into the callbacks of those entities.

Also, some of the data or constraints might not be part of user configuration, but rather they are system information based on platform dependencies. E,g, the port speed capabilities.
For such objects, we can still model them with Yang, but the instance data would be initialized by reading from system instead of configuration data.

Overall workflow

For initial configuration:

- config_db.json was provided on the box through provision system e,g, ZTP

- CLI will load it and convert it to yang based json, this will be used as input to gRPC server CLI.
- In the backend, this is considered as candidate configuration, and validation will be done through the extended validation libraries
    - If it passes, configuration will become running configuration and pushed to configDB.
    - If it fails, initialization would be stopped

During the run time:

- CLI or gPRC APIs will talk to gRPC server with the incremental changes
- Backend will validate the changes as candidate configuration, the validation for candidate configuration will be performed by extended validation libraries: syntactic and semantic.
    - If validation passes, it will go to next step
    - If validation fails, errors will be printed, candidata configuration will be rolled back.
- Calculate a delta between the running configuration and candidate configuration
- Create a list of operations to be performed in order to apply this delta.
- Push to configDB

SONiC Yang Examples

The below examples is for PORT TABLE and VLAN TABLE on SONiC:
The SONiC document for these two tables is as below, this is not a formal model nor it was enforced by any applications:

PORT TABLE:

```
;Configuration for layer 2 ports
key             = PORT|ifname ; ifname must be unique
                  across  PORT,INTF,VLAN,LAG
                  TABLES
admin_status = "down" / "up" ; admin status
lanes           = list of lanes ;
mac             = 12HEXDIG ;
alias           = 1*64VCHAR ; alias name of the port
                  used  by LLDP and SNMP, must be
                  unique
description     = 1*64VCHAR ; port description
speed           = 1*6DIGIT ; port line speed in Mbps
mtu             = 1*4DIGIT ; port MTU
fec             = 1*64VCHAR ; port fec mode
autoneg        = BIT ; auto-negotiation mode
```

VLAN TABLE:
;Defines VLANs and the interfaces which are members of the vlan ;
key = VLAN_TABLE:"Vlan”vlanid ; DIGIT 0-4095
with prefix "Vlan"
admin_status = "down" / "up" ; admin status
oper_status = "down" / "up" ; operating status
mtu = 1*4DIGIT ; MTU for the IP interface of the
VLAN
description = vlan description
members = list of ports

And here we are going to present the schema with Yang models:
PORT TABLE:
ports
|-- port
|--- ifname
|--- admin_status
|--- lanes
|--- alias
|--- description
|--- speed
|--- mtu
|--- fec
|--- autoneg

```
module sonic-module {
  yang-version 1.1;
  namespace "urn:ietf:params:xml:ns:yang:sonic-module";
  prefix tm;
  import ietf-inet-types {
    prefix inet;
  }
  organization "organization";
  description
    "example yang module";
   container ports {
       list port {
           config true;
           key "name";
            leaf name {
              type string;
           }
           leaf lanes {
```

```
            type uint8;
          }
          leaf fec {
            type string;
          }
          leaf mtu {
            type uint16;
            description
              "Set the max transmission unit size in octets
               for the physical interface.  If this is not set, the mtu is
               set to the operational default -- e.g., 1514 bytes on an
               Ethernet interface.";
          }
         leaf admin_status {
                  type enumeration {
                           enum UP {
                            description
                            "Ready to pass packets.";
                           }
                            enum DOWN {
                            description
                                    "Not ready to pass packets and not in some test
                                    mode.";
                            }
                  }
        }
          leaf alias {
             type string;
          }
          leaf speed {
            type string;
            units "bits/second";
        }
     }
}

VLAN TABLE:
vlans
   |-- vlan
          |--- vlanid
          |--- admin_status
          |--- oper_status
          |--- mtu
```

```
        |--- description
        |--- members
                |----- ifname

  container vlans {
    list vlan {
       config true;
       key "name";
       leaf name {
          type string;
       }
       leaf vlanid {
          type string;
       }
      leaf admin_status {
          type string;
      leaf description {
         type string;
      }
       leaf mtu {
          type uint16;
          description
            "Set the max transmission unit size in octets
             for the physical interface.  If this is not set, the mtu is
             set to the operational default -- e.g., 1514 bytes on an
             Ethernet interface.";
      }
      leaf-list members {
          type leafref {
              path "../../../ports/port/name";
          }
       }
    }
  }
```

With this model, tools (validation libraries) can automatically validate the instance data for syntax and semantic checks. For example, if the PORT does not exist while VLAN is referring to it, the dependency checks will fail.

The instance data can be generated as JSON file, see below, the translation from/to SONiC existing config_db.json to/from the yang based JSON is very generic, a small utility could be

written for such translation. This step would be necessary if some backwards compatibility is required, also we may want to leverage the existing utilities to write/read from configDB.

SONiC existing JSON format:


```
 "PORT": {
        "Ethernet73": {
        "alias": "Eth3/1",
        "speed": "25000",
        "lanes": "73"
        },
        "Ethernet74": {
        "alias": "Eth3/2",
        "speed": "25000",
        "lanes": "74"
        },
        "Ethernet122": {
        "alias": "Eth31/2",
        "speed": "25000",
        "lanes": "123"
        },
"VLAN": {
        "Vlan100": {
        "vlanid": "100",
        "admin_status": "up",
        "description": "Data Traffic",
        "members": [
                "Ethernet8",
                "Ethernet9"
        ],
```

```
            "mtu": "9100"

            },
```

SONiC Yang based JSON:

```
"sonic-module:ports":
   {"port":[{
            "name": "Ethernet8",
            "alias": "Eth3/1",
            "speed": "25000",
            "lanes": 73
        },
        {
            "name":"Ethernet9",
            "alias": "Eth3/2",
            "speed": "25000",
            "lanes": 74
        },
        {
            "name":"Ethernet122",
            "alias": "Eth31/2",
            "speed": "25000",
            "lanes": 123
        }
    ]
   },
 "sonic-module:vlans":
   {"vlan":[
        {
          "name": "Vlan100",
          "vlanid": "100",
          "admin_status": "up",
          "description": "data traffic",
          "members": [
               "Ethernet8",
               "Ethernet9"
           ],
          "mtu": 9100
        }
```

Phase 1 implementation

To limit the scope, in the phase 1, we will keep gPRC programmable interface part out, and the data-store and transaction/rollback is also left out.

So basically, we will be modeling the SONiC schema by YANG language with its own schema. A small JSON converter is used to translate from/to Yang generated JSON to/from SONiC current config_db.json, this is to keep backwards compatible with the existing configuration. Since the Yang model is defined based on the current SONiC schema, the translation should be generic and not case-by-case.

For initial config reload, CLI will read the config_db.json and translate it to Yang based JSON, and initiate the YANG instance data, then call the validation libraries to validate the syntax and dependencies.

Any config load or incremental commands, CLI will read the config from configDB and translate it to Yang based JSON, then issue the incremental changes, then call the validation libraries to validate the syntax and dependencies.

In case validation didn't pass, the error would be printed, this includes the syntax errors or dependency errors. List of dependencies would be printed if they are not met.

Once everything is validated the configuration will be pushed to configDB through existing config-gen engine, the data from configDB will be consumed by the application backend daemons.