Data Models

Data modeling is standard based approach to model configuration and operational data in networking devices. Using data models, customers can automate and simplify network wide visibility and configuration.

Data Models - Programmatic and Standards-based Configuration

Cisco IOS XR software supports the automation of configuration of multiple routers across the network using Data models. Configuring routers using data models overcomes drawbacks posed by traditional router management techniques.

CLIs are widely used for configuring a router and for obtaining router statistics. Other actions on the router, such as, switch-over, reload, process restart are also CLI-based. Although, CLIs are heavily used, they have many restrictions.

Customer needs are fast evolving. Typically, a network center is a heterogenous mix of various devices at multiple layers of the network. Bulk and automatic configurations need to be accomplished. CLI scraping is not flexible and optimal. Re-writing scripts many times, even for small configuration changes is cumbersome. Bulk configuration changes through CLIs are error-prone and may cause system issues. The solution lies in using data models - a programmatic and standards-based way of writing configurations to any network device, replacing the process of manual configuration. Data models are written in a standard, industry-defined language. Although configurations using CLIs are easier (more human-friendly), automating the configuration using data models results in scalability.

Cisco IOS XR supports the YANG data modeling language. YANG can be used with Network Configuration Protocol (NETCONF) to provide the desired solution of automated and programmable network operations.

YANG model

YANG is a data modeling language used to describe configuration and operational data, remote procedure calls and notifications for network devices. The salient features of YANG are:

- Human-readable format, easy to learn and represent

- Supports definition of operations
- Reusable types and groupings
- Data modularity through modules and submodules
- Supports the definition of operations (RPCs)
- Well-defined versioning rules
- Extensibility through augmentation

For more details of YANG, refer RFC 6020 and 6087.

NETCONF and gRPC (Google Remote Procedute Call) provide a mechanism to exchange configuration and operational data between a client application and a router and the YANG models define a valid structure for the data (that is being exchanged).

Protocol	Transport	Encoding/ Decoding
NETCONF	SSH	XML
gRPC	HTTP/2	XML, JSON

Each feature has a defined YANG model. Cisco-specific YANG models are referred to as synthesized models. Some of the standard bodies, such as IETF , IEEE and Open Config, are working on providing an industry-wide standard YANG models that are referred to as common models.

Components of Yang model

A module defines a single data model. However, a module can reference definitions in other modules and submodules by using the **import** statement to import external modules or the **include** statement to include one or more submodules. A module can provide augmentations to another module by using the **augment** statement to define the placement of the new nodes in the data model hierarchy and the **when** statement to define the conditions under which the new nodes are valid. **Prefix** is used when referencing definitions in the imported module.

YANG models are available for configuring a feature and to get operational state (similar to show commands)

This is the configuration YANG model for AAA (denoted by - cfg)

```
(snippet)
module Cisco-IOS-XR-aaa-locald-cfg {

  /*** NAMESPACE / PREFIX DEFINITION ***/

  namespace "http://cisco.com/ns/yang/Cisco-IOS-XR-aaa-locald-cfg";


  prefix "aaa-locald-cfg";

  /*** LINKAGE (IMPORTS / INCLUDES) ***/

  import Cisco-IOS-XR-types { prefix "xr"; }

  import Cisco-IOS-XR-aaa-lib-cfg { prefix "a1"; }

  /*** META INFORMATION ***/
```

```
  organization "Cisco Systems, Inc.";
          ......................
          ...................... (truncated)
```

This is the operational YANG model for AAA (denoted by -oper)

```
(snippet)
module Cisco-IOS-XR-aaa-locald-oper {

  /*** NAMESPACE / PREFIX DEFINITION ***/

  namespace "http://cisco.com/ns/yang/Cisco-IOS-XR-aaa-locald-oper";


  prefix "aaa-locald-oper";

  /*** LINKAGE (IMPORTS / INCLUDES) ***/

  import Cisco-IOS-XR-types { prefix "xr"; }

  include Cisco-IOS-XR-aaa-locald-oper-sub1 {
    revision-date 2015-01-07;
  }

  /*** META INFORMATION ***/

  organization "Cisco Systems, Inc.";
    ......................
    ...................... (truncated)
```

Note A module may include any number of sub-modules, but each sub-module may belong to only one module. The names of all standard modules and sub-modules must be unique.

Structure of Yang models

YANG data models can be represented in a hierarchical, tree-based structure with nodes, which makes them more easily understandable. YANG defines four nodes types. Each node has a name, and depending on the node type, the node might either define a value or contain a set of child nodes. The nodes types (for data modeling) are:

- leaf node - contains a single value of a specific type
- list node - contains a sequence of list entries, each of which is uniquely identified by one or more key leafs
- leaf-list node - contains a sequence of leaf nodes
- container node - contains a grouping of related nodes containing only child nodes, which can be any of the four node types

Data types

YANG defines data types for leaf values. These data types help the user in understanding the relevant input for a leaf.

Name	Description
binary	Any binary data
bits	A set of bits or flags
boolean	"true" or "false"
decimal64	64-bit signed decimal number
empty	A leaf that does not have any value
enumeration	Enumerated strings
identityref	A reference to an abstract identity
instance-identifier	References a data tree node
int (integer-defined values)	8-bit, 16-bit, 32-bit, 64-bit signed integers
leafref	A reference to a leaf instance
uint	8-bit, 16-bit, 32-bit, 64-bit unsigned intergers
string	Human-readable string
union	Choice of member types

Data Model and CLI Comparison

Each feature has a defined YANG model that is synthesized from the schemas. A model in a tree format includes:

- Top level nodes and their subtrees
- Subtrees that augment nodes in other yang models
- Custom RPCs

The options available using the CLI are defined as leaf-nodes in data models. The defined data types, indicated corresponding to each leaf-node, help the user to understand the required inputs.

Supported YANG Models in NCS 1004

Table 1: Feature History

Feature Name	Release Information	Feature Description
New Unified Model for Enhanced IKEv2 Encryption Support	Cisco IOS XR Release 24.1.1	The new model Cisco-IOS-XR-um-ikev2-cfg introduced in this release enhances the IKEv2 encryption. Now IKEv2 encryption complies with RFC 8784, which describes about using postquantum preshared keys (PPK) for IKEv2 encryption. The PPKs are generated with the help of the Cisco Secure Key Integration Protocol (SKIP) which makes the IKEv2 encryption resilient to quantum attacks.

The supported config and oper YANG models for NCS 1004 are listed below:

Config Yang Models	Oper Yang Models
Cisco-IOS-XR-osa-cfg.yang	Cisco-IOS-XR-osa-oper.yang
Cisco-IOS-XR-controller-optics-cfg.yang	Cisco-IOS-XR-controller-optics-oper.yang
Cisco-IOS-XR-pmengine-cfg.yang	Cisco-IOS-XR-pmengine-oper.yang
Cisco-IOS-XR-ethernet-lldp-cfg.yang	Cisco-IOS-XR-ethernet-lldp-oper.yang
Cisco-IOS-XR-ifmgr-cfg.yang	Cisco-IOS-XR-telemetry-model-driven-oper.yang
Cisco-IOS-XR-telemetry-model-driven-cfg.yang	Cisco-IOS-XR-fpd-infra-oper.yang
Cisco-IOS-XR-fpd-infra-cfg.yang	Cisco-IOS-XR-ikev2-oper.yang
Cisco-IOS-XR-ikev2-cfg.yang	Cisco-IOS-XR-otnsec-oper.yang
Cisco-IOS-XR-um-ncs1004-hw-module-osa-cfg	
Cisco-IOS-XR-um-ikev2-cfg	

The supported versions of Open Config model are listed below:

- openconfig-platform.yang
- openconfig-platform-transceiver.yang
- openconfig-terminal-device.yang
- openconfig-interfaces.yang

- openconfig-system.yang

Note openconfig-platform-transceiver.yang model is the augmented model of openconfig-platform.yang model.

gRPC

gRPC is a language-neutral, open source, RPC (Remote Procedute Call) system developed by Google. By default, it uses protocol buffers as the binary serialization protocol. It can be used with other serialization protocols as well such as JSON, XML etc. The user needs to define the structure by defining protocol buffer message types in.*proto* files. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs.

gRPC encodes requests and responses in binary. Although Protobufs was the only format supported in the initial release, gRPC is extensible to other content types. The Protobuf binary data object in gRPC is transported using HTTP/2 (RFC 7540). HTTP/2 is a replacement for HTTP that has been optimized for high performance. HTTP/2 provides many powerful capabilities including bidirectional streaming, flow control, header compression and multi-plexing. gRPC builds on those features, adding libraries for application-layer flow-control, load-balancing and call-cancellation.

gRPC supports distributed applications and services between a client and server. gRPC provides the infrastructure to build a device management service to exchange configuration and operational data between a client and a server in which the structure of the data is defined by YANG models.

Cisco gRPC IDL

The protocol buffers interface definition language (IDL) is used to define service methods, and define parameters and return types as protocol buffer message types.

gRPC requests can be encoded and sent across to the router using JSON. gRPC IDL also supports the exchange of CLI.

For gRPC transport, gRPC IDL is defined in .proto format. Clients can invoke the RPC calls defined in the IDL to program XR. The supported operations are - Get, Merge, Delete, Replace. The gRPC JSON arguments are defined in the IDL.

```
syntax = "proto3";

package IOSXRExtensibleManagabilityService;

service gRPCConfigOper {

    rpc GetConfig(ConfigGetArgs) returns(stream ConfigGetReply) {};

    rpc MergeConfig(ConfigArgs) returns(ConfigReply) {};

    rpc DeleteConfig(ConfigArgs) returns(ConfigReply) {};

    rpc ReplaceConfig(ConfigArgs) returns(ConfigReply) {};

    rpc CliConfig(CliConfigArgs) returns(CliConfigReply) {};

}
```

gRPC Operations

- oper get-config—Retrieves a configuration
- oper merge-config— Appends to an existing configuration
- oper delete-config—Deletes a configuration
- oper replace-config—Modifies a part of an existing configuration
- oper get-oper—Gets operational data using JSON
- oper cli-config—Performs a configuration
- oper showcmdtextoutput

gNOI for BERT

Table 2: Feature History

Feature Name	Release Information	Description
gNOI for BERT	Cisco IOS XR Release 7.3.1	EMS gNOI supports Bit Error Rate Testing (BERT) operations on NCS 1004 for the following remote procedure calls (RPCs): • StartBERT • StopBERT • GetBERTResults gNOI for BERT is a vendor agnostic open configuration method of enabling and testing network links through the Pseudo Random Binary Sequence (PRBS) feature.

gRPC Network Operations Interface (gNOI) defines a set of gRPC-based microservices for executing operational commands on network devices. Extensible Manageability Services (EMS) gNOI is the Cisco IOS XR implementation of gNOI.

gNOI uses gRPC as the transport protocol and the configuration is same as that of gRPC.

From R7.3.1 onwards, EMS gNOI supports Bit Error Rate Testing (BERT) operations on NCS 1004 for the following remote procedure calls (RPCs):

- StartBERT
- StopBERT
- GetBERTResults

Start a New BERT Session

StartBERT

Starts a new BERT operation for a set of ports. Each BERT operation is uniquely identified by an ID, which is given by the caller. The caller can then use this ID (as well as the list of the ports) either to stop the BERT operation or get the BERT results, or can perform both BERT operations.

```
rpc StartBERT(StartBERTRequest) returns(StartBERTResponse) {}
```

Request and response messages

```
message StartBERTRequest {
  // Per port BERT start requests.
  message PerPortRequest {
    // Path to the interface corresponding to the port.
    types.Path interface = 1;  // required
    // The selected PRBS generating polynomial for BERT.
    PrbsPolynomial prbs_polynomial = 2;  // required
    // BERT duration in seconds. Must be a positive number.
    uint32 test_duration_in_secs = 3;  // required
  }
  // Unique BERT operation ID specified by the client. Multiple BERTs run on
  // different ports can have the same BERT operation ID. This ID will be used
  // later to stop the operation and/or get its results.
  // TODO: Investigate whether we can use numerical IDs instead.
  string bert_operation_id = 1;
  // All the per-port BERTs that are considered one BERT operation and have the
  // same BERT operation ID.
  repeated PerPortRequest per_port_requests = 2;
}

message StartBERTResponse {
  // Per-port BERT start responses.
  message PerPortResponse {
    // Path to the interface corresponding to the port.
    types.Path interface = 1;
    // BERT start status for this port.
    BertStatus status = 2;
  }
  // The same BERT operation ID given by the request.
  string bert_operation_id = 1;
  // Captures the results of starting BERT on a per-port basis.
  repeated PerPortResponse per_port_responses = 2;
}
```

The supported values for **prbs_polynomial** on NCS1004 are PRBS7, PRBS13, PRBS23, and PRBS31.

- The **StartBERT** RPC can return an error status in any one of the following scenarios:
  - When BERT operation is supported on none of the ports specified by the request.
  - When BERT is already in progress on any port specified by the request.
  - In case of any low-level hardware or software internal errors.

  The RPC returns an **OK** status when there is no error situation encountered.

Stop and Delete an Existing BERT Session from the Device

Stops an already in-progress BERT operation on a set of ports. The caller uses the BERT operation ID it previously used when starting the operation to stop it.

StopBERT

```
rpc StopBERT(StopBERTRequest) returns(StopBERTResponse) {}
```

Request and response messages

```
message StopBERTRequest {
  // Per-port BERT stop requests.
  message PerPortRequest {
    // Path to the interface corresponding to the port.
    types.Path interface = 1;
  }
  // The same BERT operation ID given when BERT operation was started.
  string bert_operation_id = 1;
  // All the per-port BERTs that need to be stopped. Must be part of the BERT
  // operation specified by the `bert_operation_id` above.
  repeated PerPortRequest per_port_requests = 2;
}

message StopBERTResponse {
  // Per-port BERT stop responses.
  message PerPortResponse {
    // Path to the interface corresponding to the port.
    types.Path interface = 1;
    // BERT stop status for this port.
    BertStatus status = 2;
  }
  // The same BERT operation ID given by the request.
  string bert_operation_id = 1;
  // Captures the results of stopping BERT on a per-port basis.
  repeated PerPortResponse per_port_responses = 2;
}
```

When the **PerPortRequest** field is not configured, then the device stops and deletes BERT sessions on all the ports associated with the BERT ID.

The RPC is expected to return an error status in any one of the following situations:

- When there is at least one BERT operation in progress on a port which cannot be stopped in the middle of the operation (either due to lack of support or internal problems).
- When no BERT operation, which matches the given BERT operation ID, is in progress or completed on any of the ports specified by the request.

The **StopBERT** RPC returns to an **OK** status when there is no error situation is encountered.

Note The BERT operation is considered completed if the device has a record or history of it. Also note that you might receive a stop request for a port which has completed BERT, as long as the recorded BERT operation ID matches the one specified by the request.

Get BERT Statistics for an Existing Session

Gets BERT results during the BERT operation or after the operation completes. The caller uses the BERT operation ID that it previously used when starting the operation to query it. The device stores results for the last BERT based on the required period of time.

GetBERTResults

```
rpc GetBERTResult(GetBERTResultRequest) returns(GetBERTResultResponse) {}
```

Request and response messages

```
message GetBERTResultRequest {
  // Per-port BERT get result requests.
  message PerPortRequest {
    // Path to the interface corresponding to the port.
    types.Path interface = 1;
  }
  // The same BERT operation ID given when BERT operation was started.
  string bert_operation_id = 1;
  // All the per-port BERTs result of which we want to query. Must be part of
  // the BERT operation specified by the `bert_operation_id` above.
  repeated PerPortRequest per_port_requests = 2;
  // If set to true, the results for all the per-port BERTs will be returned.
  // `bert_operation_id` and `per_port_requests` will be ignored will be
  // ignored in that case.
  bool result_from_all_ports = 3;
}

message GetBERTResultResponse {
  // Per-port BERT results/status.
  message PerPortResponse {
    // Path to the interface corresponding to the port.
    types.Path interface = 1;
    // BERT result get status for this port. Only if the status is
    // BERT_STATUS_OK are the rest of the fields meaningful.
    BertStatus status = 2;
    // The ID of the BERT operation running on this port. Since the caller
    // can query the BERT results for all the ports, ID can potentially be
    // different for different ports.
    string bert_operation_id = 3;
    // The selected PRBS generating polynomial for BERT on this port.
    PrbsPolynomial prbs_polynomial = 4;
    // The last time BERT started on this port.
    uint64 last_bert_start_timestamp = 5;
    // The last time BERT results were read for this port.
    uint64 last_bert_get_result_timestamp = 6;
    // Indicate whether BERT peer lock has was established. If false,
    // `bert_lock_lost`, `error_count_per_minute`, and `total_errors` will not
    // be meaningful.
    bool peer_lock_established = 7;
    // Indicate whether BERT peer lock was lost after being established
    // once.
    bool peer_lock_lost = 8;
    // Sequence of bit errors per min since lock was established.
    repeated uint32 error_count_per_minute = 9;
    // Total number of bit errors accumulated since lock was established.
    uint64 total_errors = 10;
  }
  // Captures the BERT results on a per-port basis.
  repeated PerPortResponse per_port_responses = 1;
}
```

When the **per_port_requests** is ignored, then the device returns results and status for all the ports associated with the BERT ID.

The following table lists the descriptions of BERT results and status.

Table 3: BERT Results and Status

Field	Description
interface	Port in **types.Path** format representing a path in the open configuration interface model.
status	• **BERT_STATUS_OK** denotes that the BERT session is active. • **BERT_STATUS_PORT_NOT_RUNNING_BERT** denotes that BERT is not running as the duration has expired.
bert_operation_id	BERT operation ID that the port is associated with.
prbs_polynomial	The PRBS polynomial value that is configured.
last_bert_start_timestamp	Start operation timestamp in form of a 64-bit value UNIX time, which is the number of seconds elapsed since January 1, 1970 UTC.
repeated last_bert_get_results_timestamp	Timestamp of the last GetBERTResults operation in form of a 64-bit value UNIX time, which is the number of seconds elapsed since January 1, 1970 UTC.
peer_lock_established	Current status of peer lock. Note that there could be a 10-second delay in updating this field.
peer_lock_lost	Indicates if the peer lock is lost anytime after a peer lock is established. This field is only meaningful if **peer_lock_established** field is set.
error_count_per_minute	A list of one-minute historical PM buckets containing bit error counts. Historical buckets are maintained since the StartBERT operation started.
total_errors	Cumulative count of bit errors of the StartBERT operation.

The GetBERTResults RPC can return error status in any one of the following scenarios:

- When no BERT operation, which matches the given BERT operation ID, is in-progress or completed on any of the ports specified by the request.
- When the BERT operation ID does not match the in progress or completed BERT operation on any of the ports specified by the request.

The RPC returns an **OK** status when none of these situations is encountered.

Note The BERT operation is considered as completed only when the device has a record of it.