Automating the Testing of RESTCONF Agents

Alberto Gonzalez Prieto, Alfred Leung
Cisco Systems
San Jose, CA, USA
{albertgo, alfleung}@cisco.com

Kevin Rockwell
Purdue University
West Lafayette, Indiana, USA
Kevinrockwell@gmail.com

***Abstract*— RESTCONF is a management protocol, based on REST principles, currently under design at the IETF. In this paper, we present our solution for automating conformance testing of RESTCONF agents. The solution takes as input the set of YANG modules supported by the agent and automatically generates test cases based on the YANG definitions. Test cases are consumed by a test execution environment, which issues requests to an agent instance and determines its conformance based on the obtained responses.**

Keywords—RESTCONF, yang, automation, testing.

I. INTRODUCTION

RESTCONF is a new management protocol, currently under design at the IETF [1]. It is based on REST principles [2]. It can be roughly described as a simplified Netconf [3] over HTTPS for accessing data defined in YANG. RESTCONF agents expose a programmatic interface supporting CRUDX operations (i.e., create, read, update, delete, and execute) on YANG modules. More specifically, they support the following operations: get state (running configuration and operational state), edit the running configuration, execute YANG-defined rpc's, and issuing YANG-defined notifications (i.e. asynchronous event reporting).

While still under design, RESTCONF is attracting a lot of attention. For instance, the OpenDaylight controller [4] includes a RESTCONF agent that is exposed to management applications on its northbound interface.

A key part of the implementation of management agents (and software development in general) is testing. Testing aims at guaranteeing the quality of the software, including functionality, performance, and security. In this paper, the focus is on conformance testing, where the goal is guaranteeing that a RESTCONF agent implementation meets the specification requirements.

Generally, the coverage achieved by testing (i.e., the degree to which the agent is tested) increases with the number of tests. Ideally, testing would cover all conceivable use cases. However, generating a large number of tests cases is costly for it is, commonly, a manual effort. Commonly, each test case is designed (and coded) manually by an expert in the technology to test. This generation is an effort-intensive task and this influences the trade-off between using few resources in test case generation and achieving high coverage. In order to reduce the coding efforts, test designers try and identify a subset of tests that are representative of the software quality. This identification is a difficult task. Errors in this identification can lead to bug hits in deployments.

II. RESTCONF

This section provides a brief introduction to RESTCONF. First, we introduce REST architectural principles, upon which RESTCONF builds. We also briefly introduce the YANG modeling language, the other pilar of RESTCONF.

A. The REST Architecture Style

REST stands for **Re**presentation **S**tate **T**ransfer and it is a software architecture style for designing networked applications. The key adopter of REST is the World Wide Web. REST was defined in Roy Fielding's doctoral dissertation [2]. Next, we include a brief introduction to REST based on that dissertation.

The REST architectural style consists of a set of architectural constraints, aimed at achieving a set of desirable properties. Note that REST focuses on architectural constraints and does not mandate implementation details or protocol syntax. Those constraints are:

- **Client-server.** Use a client-server architecture. The rationale for this constraint is separation of concerns.
- **Stateless communication.** Each client request must contain all the information required by a server to understand the request. In other words, each request is independent from other requests. The rationale for this constraint is visibility, reliability, and scalability.
- **Cachable data.** All data in a response to a client must be marked as cacheable or non-cacheable. The marking can be implicit. The rationale for this constraint is efficiency.
- **Uniform interface.** The interface should be generic. The rationale for this constraint is simplicity.

There are two more constraints, i.e., use a layered system, and support code-on-demand.

A key concept in REST is the **resource**. Informally, a resource is any information that can be named. More formally, a resource R is a temporally varying membership function MR(t), which for time t maps to a set of entities, or values, which are equivalent. The only thing that is required to be static for a resource is the semantics of the mapping, since the semantics is what distinguishes one resource from another. A resource has an associated identifier, i.e., an URL (e.g., /restconf/data/art:top-level/name, for the leaf defined in line 7 in Figure 1).

```
 1: module art {
 2: namespace "urn:cisco:params:xml:ns:art";
 3:   prefix "art";
 4:   revision 2014-08-01 { }
 5:
 6:   container top-level {
 7:     leaf name { type string;}
 8:     leaf number { type uint32;}
 9:
10:     list table {
11:         key index;
12:        leaf index { type uint32;}
13:         leaf text { type string;}
14:     }
15:  }
16: }
```

Figure 1: YANG module example

B. *The YANG Modeling Language*

YANG [5] is a **domain-specific modeling language**. It allows for multi-level data tree hierarchies. The key constructs are: the container, the list, and the leaf. While the language goes well beyond these constructs, we limit ourselves to these for the sake of space. The interested reader is referred to the YANG RFC [5].

A **container** is an interior node in the data tree used to group related nodes. An example is a container under which all bgp-related data is organized. A **list** is an interior node in the data tree used to denote entity set. An example is the list of interfaces on a device. A **leaf** is a leaf node in the data tree, representing a scalar property. An example is the MTU of an interface.

Figure 1 shows a simple example of YANG module that we use in the rest of the paper to illustrate our solution. It is composed of a single container with two leaves and a list. Each list entry contains two leaves.

C. *The RESTCONF Protocol*

RESTCONF is a novel management protocol. It allows **accessing management data defined in YANG**, using NETCONF datastores, i.e., conceptual places to store and access information.

Next, we provide a brief introduction to the protocol. The interested reader is referred to the Internet Draft [1]. First, we present the RESTCONF framework model, which summarizes the design guidelines for RESTCONF. Then, we introduce the RESTCONF operations and functionality.

1) The RESTCONF Framework Model

The RESTCONF framework model is defined as a set of models, touching different aspects. Next, we summarize those models:

- **Message model**. RESTCONF uses HTTPS. Each protocol message performs a single operation on a single resource, e.g., retrieving a counter. (The exception is the "yang patch" method, briefly described below.)
- **Resource model**. A resource represents a manageable component. It can contain child nodes that are nested resources or fields. A resource has its own media type identifier, described by the "Content-Type" HTTP header.

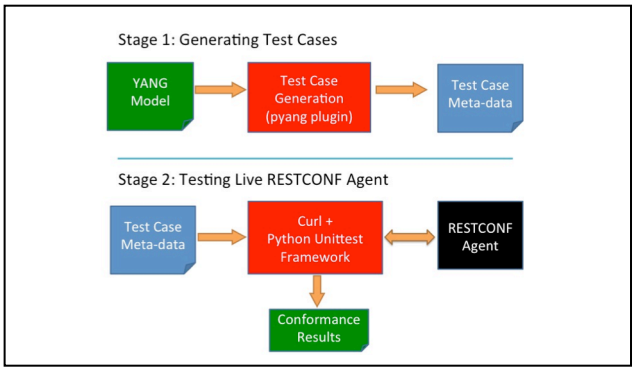

Figure 2: Two-stage Approach

- **Datastore model**. A single conceptual datastore is used. It is the union of the running configuration and the operational state of the device.
- **Content model**. The supported YANG modules are accessible at the URI "/restconf/modules/module"
- **Transaction and editing model**. RESTCONF defines a simplified transaction model, whereby one resource (and sub-resources) is edited at a time. However, it also defines a "YANG patch" method [6] that allows richer edit operations that can be applied to multiple resources. There is no mechanism to lock a datastore in RESTCONF. Edit collision detection mechanisms are provided though.
- **Defaults model**. The server must return default values when in use.

2) RESTCONF Operations

RESTCONF operations are encoded in HTTP requests. Each operation includes a HTTP method and a URI, which identifies a target resource. For instance, /restconf/data/art:top-level/name refers to the leaf in line 7 in Figure 1.

Each YANG data node is accesible as a separate resouce. Read operations on resources are performed using the GET method. For instance to retrieve the leaf "name" in the example above, the request to issue and a possible response are:

```
GET /restconf/data/art:top-level/name HTTP/1.1

--
HTTP/1.1 200 OK

{"name": "the leaf name"}
```

Write operations on configuration data nodes are performed using the DELETE, PATCH, POST, and PUT methods. The input to configuration operations is included in the HTTP body. An operation to set the leaf "name" may contain:

```
PATCH /restconf/data/art:top-level/name HTTP/1.1

{"name": "a new name"}
```

If sucessful, the response would contain:

```
HTTP/1.1 204 No Content
```

Operations can be refined using query parameters in the URI and HTTP headers. For instance, If-Match headers can be used to make requests conditional.

III. Solution Overview

We tackle the problem of automating conformance testing of RESTCONF agents by **splitting the problem into two sub-problems** (see Figure 2). First, the generation of test cases. Second, the execution of test cases and result reporting. The rationale for the split is **enabling the re-use of the test generation in different testing frameworks** with different mechanisms to send requests to a server and process its responses.

Given a YANG module, the test generation stage produces a set of tests, i.e., a number of requests and corresponding expected responses.

We define the test space using the following dimensions: the target resource (specified in the URI), the HTTP method, the query parameters, the HTTP headers, and the YANG data node substatements. Note that we do not include the HTTP body as one more dimension. We treat it differently due to its complexity in terms of design space, i.e., how many nodes to include, and the configuration values for them. Without loss of generality, the reader can assume the HTTP bodies in the test cases contain a single node with a random value. We discuss HTTP bodies in more detail in Section V.A.

Test case generation is driven by the target resource dimension (included in the URI). Our tool navigates a YANG module top-down and for each YANG node (i.e., target resource), it enumerates tests across the remaining dimensions. Each test we generate covers a single RESTCONF request. That is, it corresponds to a single data point in the multi-dimensional space.

The second stage, i.e., the test execution, takes as input the test cases generated in the previous stage. It issues requests against an agent and evaluates its conformace based on its responses.

The design goals for our solution are:

- **Standalone tool**: The tool assumes no support from any other tool or environment setup. That is, it requires no access to the device under test beyond the RESTCONF interface. It does not require a controlled environment beyond guaranteeing that the device configuration is not altered while the tests are being run. The rationale for this is simplicity.
- **Reusability.** The tool should be modular to allow for reusability in different contexts.
- **Tests independence**. Tests should be runnable in isolation. This facilitates isolating the root cause of a defect.

A non-goal of the tool (at this point) is accepting input that is not formally defined in a YANG module. This excludes two types of information. First, constraints on data that are not included in the module. This includes cases where the agent does not accept some data values, but there is no reference to that in the YANG module. Second, constraints on data that are only included in the node descriptions. For example:

```
leaf foo {
  type uint32;
  description "Value must be greater than 10";
}
```

While the module contains the information, and a human can extract the constraint information, this is not supported by our tool for it requires interpreting human language.

IV. Describing Test Cases

This section presents how test cases are described in our solution. For illustrating purposes, we will consider a test case for the following RESTCONF request:

```
PATCH /restconf/data/art:top-level/name HTTP/1.1
{"name": "bar"}
```

The operation should update the leaf "name", but it should not create it if it does not exist. In this case, we want to test that it gets properly updated.

Each test case involves a number of RESTCONF operations. Each operation consists of a request and a template for the expected response. For the operation to be successful, the response obtained from the agent must match the template. We split operations into four phases.

A. Four Phases

1) Pre-requisite Phase.

This phase is responsible for setting the test environment. Aiming at test independence, all tests assume a blank configuration datastore as starting point. Some tests need to modify the datastore before the test can be run. Consider a positive test case for patching a list entry, we must guarantee that before invoking the PATCH operation, the list entry already exists. This phase can be composed of zero, one or multiple operations. No operation is needed in this phase for tests on non-configuration data. For the example above, we must create the leaf "name", before we can patch it. The operation we use is:

```
POST /restconf/data/art:top-level/name HTTP/1.1
{"name": "dummy_value"}
```

2) Request under test Phase.

This phase consists of exactly one operation, corresponding to one data point in the test case space.

3) Validation Phase.

For tests on configuration data, we need a validation phase after we have executed the operation in the previous phase. A response matching the template of the expected response is a necessary condition to guarantee the correctness of the agent. However, it is not a sufficient condition. Consider the PATCH operation above. The expected response should include status code "200 OK" or "204 No Content". However, a faulty agent may return such a status code without actually modifying the configuration datastore. This phase aims at detecting such defects, i.e., to gain test coverage. In this phase we poll the configuration state we intended to update and validate that the

current configuration state is the intended one. For the example above, the operation we use in this phase is:

```
GET /restconf/data/art:top-level/name HTTP/1.1
```

The response should include:

```
HTTP/1.1 200 OK

{"name": "bar"}
```

Note that, while it is conceivable that the agent returns the expected state, but this one does not correspond to the contents of the datastore. Note that since the tool is designed to be a standalone, we do not use alternative methods to validate the data we obtain from the RESTCONF agent.

4) Post-requisite Phase.

It is responsible for resetting the test environment. It is required for tests that modify a configuration datastore. That is, PUT/PATCH/POST test. The goal is guaranteeing that the state of the configuration datastore is the same as it was before the test run. That is, this phase must undo what was done in first and second phases. The rationale for this is achieving test independence. For the example above, the operation we use in this phase is:

```
DELETE /restconf/data/art:top-level/name HTTP/1.1
```

The response should include:

```
HTTP/1.1 204 No Content
```

Note that any operation at any phase can fail, i.e., the obtained response does not match the template. If any operation in phase 1 fails, the test is aborted and deemed inconclusive. If all operation in phases 2 and 3 pass, the test is deemed successful. Otherwise, the test is deemed failed. If phase 4 fails, all subsequent tests on the same resource or its children are deemed inconclusive. The reason is that the failure can impact subsequent tests for descendant resources to the one for which the operation failed.

Figures 3-5 describe the model we use for an operation. Figure 3 shows that an operation is mainly composed of a request (lines 4-24) and an expected response (lines 25-35).

B. The Request

The request contains the elements of a RESTCONF request presented in Section II. To support the description of negative tests cases, we explicitly differentiate between valid and invalid HTTP methods (lines 6-7 and 8-9 respectively). We also differentiate explicitly valid and invalid HTTP headers (lines 18-19 and 20-21). Note that the header value can be of two types (line 22), a literal value or an expression including variables. All variables refer to fields in responses to previous operations in the current test case. This is used, for example, for conditional requests, e.g., those including If-Match headers, whereby a request is processed if and only if, the etag provided in the If-Match header corresponds to the one identifying the configuration state version. We obtain the etag value from the agent during the pre-requisite phase. For phase #2, the header-value-type (line 22) is “reference” (vs. “literal”) and the value (line 23) refers to the etag obtained in the previous phase.

The input tree (line 24) is composed of a set of connected nodes. The model for those nodes is shown in Figure 4. Each

```
 1: +--ro operation*
 2:    +--ro resource              string
 3:    +--ro description           string
 4:    +--ro request
 5:    |  +--ro (method)
 6:    |  |  +--:(valid-http-method)
 7:    |  |  |  +--ro http-method       identityref
 8:    |  |  +--:(invalid-http-method)
 9:    |  |     +--ro invalid-http-method  string
10:    |  +--ro URI
11:    |  |  +--ro entry               string
12:    |  |  +--ro resource            leafref
13:    |  |  +--ro query_parameters?   string
14:    |  +--ro HTTP_version           string
15:    |  +--ro request-headers
16:    |  |  +--ro header*
17:    |  |     +--ro (header-type-choice)
18:    |  |     |  +--:(valid-header-type)
19:    |  |     |  |  +--ro header-type? identityref
20:    |  |     |  +--:(invalid-header-type)
21:    |  |     |     +--ro invalid-header-type? str
22:    |  |     +--ro header-value-type  enumeration
23:    |  |     +--ro header-value            string
24:    |  +--ro input-tree
25:    +--ro expected-response
26:       +--ro return-code*             uint16
27:       +--ro response-headers
28:       |  +--ro header*
29:       |     +--ro must-be-present    boolean
30:       |     +--ro header-type?       identityref
31:       |     +--ro expected_values
32:       |        +--ro expected_value*
33:       |           +--ro type         enumeration
34:       |           +--ro output-value expectedval
35:       +--ro output-tree
```

Figure 3: Operation Model

```
1:  +--ro input-tree-node
2:     +--ro identifier           string
3:     +--ro (scope)?
4:     |  +--:(module)
5:     |  |  +--ro module-name?   string
6:     |  +--:(namespace)
7:     |     +--ro namespace?     string
8:     +--ro input-value          string
```

Figure 4: Mode for a Node in the Input-Tree

```
 1: +--ro output-tree
 2:    +--ro identifier             string
 3:    +--ro (scope)?
 4:    |  +--:(module)
 5:    |  |  +--ro module-name?       string
 6:    |  +--:(namespace)
 7:    |     +--ro namespace?         string
 8:    +--ro must-be-present          boolean
 9:    +--ro expected_values
10:       +--ro expected_value*
11:          +--ro yang_metadata      yang_md
12:          +--ro type               enumeration
13:          +--ro output-value       expected-value
```

Figure 5: Model for a Node in the Ouput-Tree

node has its YANG identifier (line 2). It may contain its module name (line 4) or namespace (line 6), depending on whether format used is json or XML.

C. The Expected Response

The expected response (lines 25-35) includes a list of expected return codes (line 26). Note that we have to accommodate for different valid values. For instance, a

successful PATCH operation can get in the response either "200 OK" or "204 No Content". The expected response also includes a list of headers. Note that we cannot determine (at this stage) with certainty the exact set of headers we will get when testing the device. The reason is that the RESTCONF specification indicates that some headers SHOULD or MAY be included. That is why we include a "must-be-present" flag (line 29). Also note that for some header values we cannot determine (at this stage) the exact value we will get during testing. Therefore, we have three possible value types (line 33): "literal", "reference" (with the semantics described above), and "regex" for regular expression describing the template the value must conform to. "Regex" is used for "Date" headers.

The output tree (line 35) is composed of a set of connected nodes. The model for those nodes is shown in Figure 5. This tree is similar to the input tree. There are two key differences though. First, for non-configuration data, we cannot determine (at this stage) the exact set of nodes we will get during testing. The reason is that some nodes are not labeled in the YANG module as mandatory, so they may or may not be included in the response (line 8). Second, in the general case, we cannot determine the values of operational data. While this could be achievable integrating our tool with systems that control the testing environment, we have decided to not do it and design a standalone tool for the sake of simplicity. Still, we know the value must conform to its YANG definition (line 11) and we validate the returned data against it.

V. Selecting a Test Case Subspace

As discussed before, the coverage achieved by testing increases with the number of tests. However, covering the full test space is unfeasible for it is vast. Consider how, for a simple YANG leaf of type string, the number of PATCH requests that can be generated is unbounded as it is the number of combinations of characters of unbounded length. Furthermore, for requests on containers, the number of possible requests on a container grows exponentially with the number of leaves in it.

While automating test case generation leads to a much lower cost per generated case (with respect to manual generation), still, generating all possible test cases is unfeasible. We need to select what test cases to generate and which ones to exclude.

Our solution restricts the generated test case subspace based on three types of factors: domain knowledge, positive vs. negative test cases, and user input. We analyze each set of factors next.

A. Knowledge-based Test Case Generation

Based on our knowledge of how agents are commonly implemented, we select test cases that we deem representative of a range of tests. Note that this implies making assumptions on the agent implementation. While we do not claim universality for these assumptions, we believe that they commonly hold. We align them in two areas: value representativeness, and test correlation.

First, we condider **value representativeness**. For each dimension in the test case space, the cardinatlity of possible values can be very high, and in some cases unbounded, as discussed above. For some of those dimensions, we choose a small subset of values (based on the YANG definition) that we deem representative for a large set of tests.

Let us consider configuration operations, and focus on the new configuration they carry. Consider a leaf of type int8 [5] (i.e., an 8-bit integer). The range of valid values is [-128, 127]. For such a leave, we deem that a representative set of test values would be: -129, -128, 0, 127, 128. That is, both positive and negative values; values that are on both sides of the range limits; and zero. If these five tests pass, we consider that tests with different values have a low the probability of failing.

Similarly, for leaves of type string with a constraint on its length, we test strings with a length matching that in the constraint and one exceeding by one character. Pattern constraints on strings impose a different challenge for it is hard to determine, in the general case, what tests would give a higher success in detecting software defects beyond a random valid value and a random invalid value (i.e., what values are more representative).

Another example is the value for the query parameter "depth", which controls the descendant levels to include in the reply to a GET request. We deem that a representative set of values is: 1, "unbounded" and a random value between 1 and the maximum depth for that node. "Unbounded" and 1 are special cases. That is why we single them out. All other values are expected to be treated similarly. That is why we deem a single additional value to be representative of multiple tests.

We consider **test correlation** next. The key idea in test correlation is identifying tests with highly correlated results. I.e., one of them passing indicates a high probability of another one passing, and viceversa. One example of correlated tests are tests that differentiate in the order of the query parameters. We assume that varying the order in which the query parameters is unlikely to detect additional defects. Therefore, we use a single ordering in all requests, reducing the number of generated tests (1 vs. O(q!), where q is the number of supported query parameters).

Another example is configuration tests with multiple leaves and with only one leaf. Consider a container with three leaves (e.g., "a","b", "c"). If tests for setting each single leaf pass, we assume that tests where two or more leaves are set are unlikely to hit additional defects. This permits limiting the number of tests for a container to O(n), n being the number of container child nodes (vs. O(n!)). There are two caveats to this assumption. First, we need one additional test to guarantee that the agent can parse incoming HTTP bodies with more than one leaf. Note we need one additional test per test suite (vs. per container or list). Second, for agent implementations that are closely related to CLI interfaces, for leaves that correspond to arguments in a single CLI command, this assumption may not hold.

The tool can be extended should the assumptions above not hold for a particular agent implementation.

B. Positive vs. Negative tests

A positive test case is one where the request sent to the agent is a valid RESTCONF request as defined by its

specification and the YANG modules supported by the agent. Examples of negative test cases include using a non-supported URI, using incorrect headers (e.g., using a Content-type that does not correspond to the media type of the request body), and a malformed request body. Negative tests aim to guarantee that the agent responds gracefully to invalid input. I.e., the agent correctly identifies the request as invalid and returns the appropriate error per the specification.

Arguably, during agent development, it is more critical to guarantee that the agent works properly for valid requests. Furthemore, the test space for negative test cases is broader than the positive test case space. This is why, so far, we have focused on positive test cases. Currently, we generate negative test cases only for invalid values (e.g., out of range).

C. User Input

Our design allows users to control the generated test space. The tool is designed to permit restricting it. We find this useful during the development of the agent, when some features are still not available. Similarly, it is also useful for agents that deviate from the specification.

We envision four types of restrictions. First, limiting the dimensions considered when generating test cases. For instance, if a RESTCONF query parameter is not supported, tests including that parameter are not generated. Second, limiting the valid values in a given dimension for generating test cases. For instance, if an agent does not support JSON media types, a user can exclude all tests that use media types of type "application/*+json". Third, excluding target resources based on their properties. E.g., exclude nodes whose existence depends on a feature being supported by the device under test. Forth, exclude nodes in a given subtree. This permits excluding subsets of a device features that are still under development.

VI. Implementation

This section describes our work-in-progress implementation of the presented solution.

A. Test Case Generation

Our test case generation tool is a pyang [7] plugin that builds on top of the tree plugin, which is part of the pyang distribution.

The generated test cases are stored in json format following the data model presented in figures 3-5. Test cases are organized after the YANG module. We create a file hierarchy matching the YANG module hierarchy. E.g., we create a directory per container, and a child directory per container's child YANG node. That is, each resource is assigned a directory, containing the metadata for the test cases associated to that resource. Our RESTCONF developers consider this organization more intuitive than others.

Our plugin is written in Python (as all plugins in the pyang distribution are). Developing in Python has allowed us to generate the tool very quickly. In terms of performance, a potential drawback of Python, the test generation is extremely fast. Generating test cases for a model representative of production models (in terms of size and complexity) takes less than two seconds. Another advantage of using Python is the support its standard library provides for manipulating data in json format in a simple way using high-level operations.

B. Running Tests Against The Agent

We are developing a separate tool for realizing the second stage. This tool is also written in Python. We use the Python curl library [9] for generating and issuing requests and receive and parse responses. The testing framework we use is unittest [8]. A limitation of this framework we hit is the lack of support for defining generic tests that are data-driven. We had to provide a mechanism for generating unittest test cases based on our metadata.

Unittest considers three phases. We mapped our pre-requisite phase to the unittest's "setup" phase. Our post-requisite phase is mapped to the unittest's "teardown" phase. The rest of our phases are included in unittest's "test" phase.

VII. Concluding Remarks

In this paper, we have described our 2-stage solution for automated conformance testing of RESTCONF agents. We have used the tool against a RESTCONF agent under development and our experience has been very positive for it has been able to detect defects at a very low cost. It is well known that the cost of a defect depends on the development cycle stage where it is detected. The later in the cycle, the higher the cost for the project. Our tool is used by developers, at zero development cost for them. This allows them running unit tests with extensive coverage, detecting defects before the agent goes to the next software production phase (i.e., development testing). This early detection of defects at the development stage vs. (say) system testing leads to significant additional savings.

Our future work includes extending the tool to gain coverage for negative test cases and YANG node substatement not supported yet, like "choice", and "when".

References

[1] A. Bierman, M. Bjorklund, K. Watsen, and R. Fernando, "RESTCONF Protocol". [Online]. Available: https://tools.ietf.org/html/draft-ietf-netconf-restconf-01

[2] Roy T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures", Ph.D. dissertation, Dept. Inform. and Comput. Sci. University of California, Irvine, CA, USA, 2000.

[3] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network Configuration Protocol (NETCONF)", RFC 6241.

[4] OpenDaylight [Online]. Available: http://www.opendaylight.org

[5] M. Bjorklund, Ed., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020.

[6] A. Bierman, M. Bjorklund, K. Watsen, and R. Fernando, "YANG Patch Media Type". [Online]. Available: http://tools.ietf.org/html/draft-bierman-netconf-yang-patch-00

[7] Pyang (An extensible YANG validator and converter in python) [Online]. Available: https://code.google.com/p/pyang/

[8] The Python Standard Library. Unit Testing Framework. [Online]. Available: https://docs.python.org/3/library/unittest.html

[9] PycURL. [Online]. Available: http://pycurl.sourceforge.net