High Level Abstraction Modeling for Network Configuration Validation

Khalid Elbadawi
School of Computing
DePaul University
Chicago, IL 60604
Email: badawi@cdm.depaul.edu

James Yu
School of Computing
DePaul University
Chicago, IL 60604
Email: jyu@cdm.depaul.edu

Abstract—This paper presents our design and implementation of a Configuration Validation System (ConfVS) which uses a high-level language to help system operators verify network configurations based on formal requirements. In contrast to many existing solutions, ConfVS uses a comprehensive model to formalize the network specification and utilizes the NETCONF protocol to collect configuration data from network devices. In addition, ConfVS uses binary decision diagrams to model the behavior of network devices and provides a prototype to query the device configuration. Requirements and specifications are written in the Erlang language, a general-purpose concurrent programming language. To validate ConfVS, we present a case study to show the features and the expressiveness of ConfVS by performing different types of reasoning with network requirements.

I. INTRODUCTION

Modern networks are becoming increasingly complex to control and manage since these networks deploy many different network devices manufactured by various vendors. As network operators need continuously to conduct configuration tuning in order to satisfy new customer requirements, the possibility of configuration inconsistency is very high. Several surveys [1], [2] show that the major cause of network downtime is due to operator configuration errors.

There is a growing consensus on the need for formal models and programming frameworks to diagnose and validate network configurations. A framework provides high level abstractions of devices variety and a unified language to interact with the underlying devices. However, obtaining such a framework that defines the semantics of network architecture is not a trivial task. This is because of not only the variety of the underlying devices, but also the complexity of configuration tasks that need to be accomplished. These include configuration synthesis, configuration error diagnosis, configuration error fixing, reconfiguration to meet some requirements, or reconfiguration as devices are added or deleted [3].

In the past few years, many researchers have attempted to address various challenges in network configuration management, most of which have focused either on particular communication aspects [4] or achieving particular end-to-end requirement [5], [6]. These works provide limited analysis that covers certain network devices such as firewalls or routers. Few attempts have proposed models that provide network-wide analysis. The work in [3] proposes a model that is based on first-order logic to describe end-to-end requirements using Alloy language. The main disadvantages are its complexity and lack of scalability. In [7], the authors use SQL-like language to automatically configure network devices. However, the language is not suitable to describe policy-based network devices such as routers or firewalls. The work in [8] uses model checking to achieve global end-to-end verification. Model checking has the limitation of the state-explosion problem, particularly when used to verify large networks.

The paper proposes a new system called ConfVS that enables system administrators to describe network requirements and to formalize network specifications. Unlike previous work, ConfVS provides a comprehensive end-to-end network configuration analysis to model all network devices such as routers, firewalls, NAT, IPSec gateways, etc. It has the advantage of achieving verification for networks of arbitrary size and providing a high-level language that is easy to understand and debug. It models network specifications using predicate logic. In addition, it models the behavior of each policy-based device using binary decision diagrams (BDD). Both models are encoded by using the Erlang language. Our approach does not synthesize configuration files or determines whether the set of configuration parameters are configured with optimal values. The major contributions of this paper are: (1) building a system that allows network operators to formalize requirements of network elements or service using high-level abstractions, (2) providing an easy-to-use mechanism to query the operation parameters, (3) applying the latest effort of IETF which is the NETCONF protocol to retrieve configuration data from managed devices, and (4) demonstrating the system with a real case.

The paper is structured as follows. Section II gives a detailed description of our model. In section III, we describe the implementation of our approach. Section IV provides a case study to evaluate the system expressiveness. We make some concluding remarks and propose some future works in section V.

II. COMPREHENSIVE CONFIGURATION MODELING

As the complexity and the size of networks increases, the gap between high level configuration requirements and low level devices implementations also increases. To bridge
devices manufactured by different vendors would be extremely difficult unless there is a unified model to describe the data. Recently, there were some attempts in IETF to come up with a uniform data model [11], [12]. However, none of these efforts succeeded to provide a comprehensive data model. Nowadays, the current effort of IETF is focused on a new language called YANG which was designed specially for NETCONF protocol [13]. YANG has not been standardized yet, but it is expected to be the standard data model for the content layer of NETCONF.

YANG is a conceptual schema for data configuration. It provides a concise description of data types, relationships between data configuration, and integrity constraints that ensure the correctness of uploaded values during configuration process. In YANG, data configuration is partitioned into modules in which each module contains multiple hierarchies. A hierarchy contains a set of nodes that correspond to configuration data, state data, RPC operations, and notifications.

List 1 shows an example of a simple YANG module to define some parameters of RIP configuration. The module router consists of one hierarchy tree called rip that consists of three nodes version, network, and neighbor. The node version is a simple integer value that can take either 1 or 2 where its default value is 1. The node network is a list of structure where the structure is composed of two nodes: address and mask. The keyword key is used to uniquely identify the entries in network list. The last node in rip tree is neighbor node which represents a list of IP addresses.

An important feature of YANG is its direct map with XML schema. For example, a RIP configuration that conforms to LIST 1 is shown in List 2.
B. ConfVS design

The basic principle of ConfVS's runtime system is that each real world element can be formally specified. The abstract model is expressed in the Erlang language and it is based on predicate logic as well as binary decision diagrams.

1) Primitives: The basic unit of expressing a value or literal in ConfVS is the term. Terms are comprised of simple or complex values. Simple terms can be atoms or numbers. Complex terms are either tuples or lists. A tuple consists of fixed number of values surrounded by braces {} and elements of the tuple are separated by commas. A list consists of variable number of values surrounded by brackets []. A predicate \( Q(x_1, x_2, ..., x_n) \) is either a tuple of the form \( \{Q, x_1, x_2, ..., x_n\} \) with \((n + 1)\) arguments or a function symbol that returns true or false. Patterns have the same structure of terms, with the addition that they include variables. Variable names begin with an upper letter, while literal constants (atoms) begin with a lower letter.

2) Network topology description: To have global view of a network, ConfVS must have the information about network topology. A network topology is described by using four primitive predicates: device, service, interface, and link.

a) device: Device is a tuple that abstracts logically a network component and it is characterized by a name, class, parent, group and agent. For example, \{device, ro1, router, as1, access_routers, undefined\} is a predicate for a device named ro1 of type router. Its parent device is as1 where as1 is a logical device that represents an autonomous system of area 1. ro1 belongs to a group of devices which are access_routers, and it has no agent. The meaning of agent will be explained in section II-C.

b) service: Service is a tuple that represents an activity or a network service running on a device that needs to be managed and controlled. A service is characterized by a name, device, class, namespace, rootdm where device represents its device name and class defines its type. Usually, a service is associated with a YANG module. If it is associated, then namespace and rootdm represent the module’s namespace URI and the root node, respectively.

c) interface: ConfVS enables network operators to create interfaces using either of two methods: explicitly from ConfVS runtime library, or from interface data module retrieved from a managed device. Each interface is characterized by a number, name, and class.

d) Link: Link is an abstraction for any type of connectivity between two adjacent devices. ConfVS allows to build a hierarchy of connectivity between two adjacent devices. For example, router A is physically connected to router B using 1000Base-T Ethernet cable. The routing table of router A indicates that router B can be its next hop. Thus, router A and router B are logically connected. However, the logical connectivity cannot exist unless there is a physical connectivity between the two routers. To express this scenario, ConfVS

\[
\{\text{node, rip,} [\{\text{node, version, ["1.1"]}\}, \\
\{\text{node, network,} [\{\text{node, address, ["192.168.1.0"]}, \\
\{\text{node, mask, ["24"]}\}])
\}
\]

List 3: XML document represented in ConfVS

defines a link where its existence relies on its parent link. link is characterized by a name, a type, and a parent link. If a link has no parent, then its parent link is undefined.

C. Configuration Database

Network topology and configuration variables are stored in a database DB (knowledge base) that consists of multiple tables. If \( T \) is a table with \( k \)-arguments, then the table \( T \) has a set of tuples (predicates) of the form \( \{T, x_1, x_2, ..., x_k\} \)

The collection of all predicates in DB constitutes configuration database.

Configuration variables are also stored in configuration database. ConfVS uses agent field in the device tuple to communicate with a managed object using the NETCONF protocol and uses namespace and rootdm fields to determine which module and target root node that need to be accessed. It issues NETCONF’s <get> request to the managed device and the managed device replies with an XML document (similar to List 2) that contains configuration variables under the root node. Once ConfVS receives the XML document, it converts the document to a list of tuples without loosing the structure of XML document. For example, the corresponding term of List 2 is shown in List 3.

D. Modeling network behavior

The network behavior depends on the behavior of each network device when a traffic flow passes through them. Each of these devices operates based on protocol and locally configured access control policy [8]. The device determines the appropriate action (such as permit, deny, encrypt, decrypt, etc) on incoming traffic based on its configured filtering policy. The followings describe precisely how to model the behavior of a network device.

1) service behavior: We model service behavior using binary decision diagrams (BDDs). BDDs have been used extensively for modeling firewall and IPSec policies [4, 8]. Unlike [4], we model access control policy using a single Boolean expression instead of having a Boolean expression for each action. Let \( P \) be a list of access control policy for a network service where \( P \) contains a finite set of rules. Let \( R_i = \{f_1, f_2, ..., f_k, a_i\} \) be a tuple with \((k + 1)\)-arguments that represents a filtering rule where \( f_j \) is a constraint field in \( R_i \) and \( a_i \) is an action that will be taken when an incoming packet matches \( f_1, f_2, ..., f_k \). To generate a single Boolean
expression that exhibits multiple actions, we encode each individual rule $R_i$ as:

$$F_i = f_1^i \land f_2^i \land \cdots \land f_k^i \land a_i$$  \hspace{1cm} (1)

The above expression can be understood as follow. Let us imagine that there is a function $\delta(f_1 \times f_2 \times \cdots \times f_k \times a \rightarrow \{\text{true}, \text{false}\})$ such that $\delta(\bar{x})$ evaluates to true if $\bar{x} = (f_1^i, f_2^i, \cdots, f_k^i, a_i)$.

Encoding the entire policy $P$ to a Boolean expression depends on how a specific rule is selected. For example, most firewalls search the matching rule sequentially; however, routers choose the next hop based on the length of network prefix. Longer prefixes are always preferred over shorter ones when forwarding a packet. We construct a Boolean expression $P$, for a routing table as follows. For each network prefix of length $m$, we construct $P_m$ such that

$$P_m = \bigvee (\neg F_i^m \land \neg F_2^m \land \cdots \land \neg F_{i-1}^m \land F_i^m)$$  \hspace{1cm} (2)

where $F_i^m$ is a Boolean expression for the $k$th rule with network prefix of length $m$. Then

$$P = P_{32} \lor (\neg P_{32} \land P_{31}) \lor (\neg P_{32} \land \neg P_{31} \land P_{30}) \lor \cdots$$  \hspace{1cm} (3)

Equation 1 allows us to query a service using a single Boolean expression, $\Psi = \pi \times P \rightarrow A$ where $\pi$, $P$, and $A$ are Boolean expressions. The function $\Psi$ takes as parameter the set of network traffics that satisfies $\pi$ together with the policy $P$ and returns the set of actions that satisfies $A$. Similarly, we define the inverse of the projection function, $\Psi^{-1}$, such that

$$\Psi^{-1} : A \times P \rightarrow \pi$$  \hspace{1cm} (4)

2) device behavior: We model device behavior as a finite state machine (FSM). A (deterministic) FSM $M$ can be described as a set of relations of the form $\text{State}(S) \times \text{Event}(E) \rightarrow \{\text{Action}(A), \text{State}(S)\}$ where $S$, $E$, and $A$ are finite and nonempty sets of states, events, and actions, respectively.

The set $S$ consists of three predefined states (in, out, and sink) as well as states that represent services running in a device. The set $E$ is determined based on the current state. Basically, $E$ describes the expected actions taken by the service at state $S$. For example, if the device $D$ has two services firewall (expressed by state $S1$) and routing table (expressed by state $S2$), then we can express the FSM logically as $M = [\{\text{in}, \text{in}, S1\}, \{S1, 1, S2\}, \{S1, 0, \text{sink}\}, \{S2, \text{out}\}]$.

The expression $M$ states that when device $D$ receives incoming traffic $\pi$ at interface $i$, the implementation of FSM starts at state $\{\text{in}, i, \_\}$. State $\text{in}$ has no action so the traffic $\pi$ will be forwarded to state $S1$. At state $S1$, the function $\Psi^{-1}$ will be used to determine the set $E$. Let assume for simplicity that the set $E$ consists of one element which is \{1\}. Then the system enters state $S2$ and the forwarded traffic will be recalculated as $\pi(\text{new}) = \pi(\text{old}) \land \Psi^{-1}(1, P_s)$ where $P_s$ is the policy expression of the service at state $s$. The implementation of FSM keeps running until the system reaches either the state $\text{sink}$ or $\text{out}$.

E. Requirement specification

A requirement $R$ is expressed as a clause (or rule) of the form $r \rightarrow p_1, p_2, \ldots, p_n$. Where $r$ is referred to as the head of the clause and $p_1, p_2, \ldots$ and $p_n$ are expressions that constitute the body of the clause. The value of the clause is determined by the value of the last expression $p_n$. This implies that each expression in the clause body is evaluated sequentially.

In the clause body, usually we would like to select all predicates from configuration database that are related and satisfying certain criteria. To perform such selection, we use special construct called list comprehension. List comprehension is an expression of the following form [14]:

$$[X || \text{Qualifier1, Qualifier2, \ldots}]$$

where $X$ is an arbitrary expression, and each qualifier is either a generator or filter. Generators are written as Pattern $<=$ L where L must be an expression that evaluates to a list of terms. Filters are either functions that return true or false or boolean expressions. For example, $[X || X <$ L, X $>$ 10] is equivalent to the set notation $\{X | X \in L, X > 10\}$.

III. ConfVS IMPLEMENTATION

ConfVS is implemented by using the Erlang language. Erlang [14] is a general-purpose concurrent programming language that supports real-time and fault-tolerant distributed systems. It has been used in many mission-critical applications such as the AXD 301 ATM switch [15].

An Erlang program consists of a set of modules. Each module basically contains a list of function definitions (rules). Each function consists of a set of Erlang expressions. A function which has a name fun_name defined in the module mod_name with arity $N$ is referred as mod_name:fun_name/N.

In Erlang, a tuple can be represented by using record. For example, $X = \{\text{device}, \text{rol, router, as1, access_routers, undefined}\}$ can be expressed as \#device[name=rol, class=router, parent=as1, group=access_routers, agent=undefined} where device is the record name. If $X$ is record of type device, then \#device.name and X#device.parent are equivalent to rol and as1, respectively.

ConfVS contains of several Erlang modules as shown in Figure 2. The following is a brief description of each module.

ConfVS module: This is the main module. This module is responsible for performing system initialization such as database initialization and link-graph initialization.

Agent module: This module is responsible for communicating to each managed device defined in the specification using the NETCONF protocol.

The software has more than half a million lines of Erlang code.
DB module: This module is responsible for handling all requests that involve database operations such as reading, writing, creating a table, etc.

ConfVS Library: This is a set of modules that provide a set of utilities for network operators. For example, devices, services, interfaces, and links are stored in tables. To manipulate these tables, ConfVS provides a set of functions for this purpose. For instance, device:new(ro1, router, as1, access_routers, undefined) defines a new device tuple that will be stored in device table and select:device(all) returns (as a list of tuples) all defined devices in the system. In ConfVS, each device or service has a class type and each class has an equivalent Erlang module. These modules encode the behavior (semantic) of the class type by providing unified set of functions such as projection function and the inverse of projection function. Having a class module enables us to extend ConfVS if a new behavior has been added to the system.

BDD module: This module provides a set of functions to perform BDD operations. BDD operations are not executed in BDD module, but rather in another C library that supports BDD. BDD module links to that C library and provides a set of APIs to C library in Erlang language.

ConfVS creates an Erlang process for each device tuple. This allows ConfVS to compute BDD of several devices concurrently. In large-scale network, ConfVS can create multiple BDD modules where a group of Erlang processes is linked to a particular BDD module.

IV. CASE STUDY

We now illustrate how to diagnose and debug network configuration using ConfVS. First, we run ConfVS in a simple network. Then, we demonstrate other capabilities of ConfVS in validating network configuration.

The setup of our experiment consists of four Pentium machines, all running OpenSuSE with kernel 2.6.19. Three machines have quagga version 0.99 and ConfD version 2.9. Quagga is a routing software that supports many routing protocols such as RIPv1, RIPv2, OSPFv2, etc. ConfD [16] is an XML-based software product to support the NETCONF protocol. We configured these machines as routers running RIPv2. The connectivity between these machines is shown in Figure 3. The fourth machine has ConfVS, and it is connected to each router via a switch.

A. Topology description

The topology description of Figure 3 is specified as:

topology() ->
device:new(mynet, network, undefined, undefined),
A=device:agent("192.168.1.3",2023,"tcp","admin"),
device:new(r1, router, mynet, A),
service:new(rip, routing, r1, "http://cdm.depaul.edu/ns/router", rip),
interface:new([0,1,2], eth, ethernet, r1),
device:clone(r2,r1,A#agent{hostname="192.168.1.4"}),
device:clone(r3,r1,A#agent{hostname="192.168.1.5"}),
link:new(l1, phy, ’100Mbps’, [0,r1], [0, r2]),
...

B. Configuration consistency requirements

It is necessary to guarantee that all routers are configured consistently. For example, we would like to ensure that all routers are configured with the same RIP timers or all are running the same RIP version. The following rule validates that all routers are running the same RIP version:

req_1() ->
% which devices has 'rip' as service
Devs = [X#service.device || X <- select:service(rip, all) ],
% for each device D, get its rip version
Vers = [X#node.value || D <- Devs, 
X <- data:get_value("version", [rip, D]) ],
assert:all(Vers).

Similarly, we may need to ensure that no non-equal interfaces are placed on the same IP subnet for each router. To formalize this requirement, we check if there are two non-equal interfaces on the same subnet:

Infs = [[X#interface.id, Y#interface.id] ||
X <- select:interface(Device),
Y <- select:interface(Device),
X#interface.id != Y#interface.id,
interface:subnet(X) == interface:subnet(Y)]

By iterating each router, as we did in req_1, we can identify which two interfaces are placed on the same subnet. In general, each router should be connected to at least two links. The following rule shows how to formalize this requirement:

req_3() ->
Devs = [X#device.name || X <- select:device(all), X#device.class == router 1],
Nlinks = [length(graph:out_neighbours(phy, D)) || D <- Devs],
assert:all(Nlinks, fun(X) -> X >= 2 end).

C. Connectivity Requirements

A typical IP traffic flow consists of protocol, destination IP, destination prefix, source IP, source pre-
fix, destination port, source port. In our system, a traffic flow is expressed as a tuple of type flow. For example, the flow \#flow{dstIP=<<192,168,1,1>>, dstPxn=<<255,255,255,255>>, dstPort=80} is the set of ALL packets where destination IP is 192.168.1.1 and destination port is 80. Notice that the remaining fields – such as srcIP – are not specified. Their values are set to the default value which is typically zero. The value of zero has the meaning of “don’t care”.

Let us assume that there is a physical path \( P \) from node \( u \) to node \( v \) such that any node in the path is either a router or a firewall. Then, node \( u \) can reach node \( v \) via path \( P \) if the following Boolean expression\(^3\)

\[
F \rightarrow \left( \bigwedge_{x \in (P \setminus \{v\})} \Psi(x, I_x) \right)
\]

is true where \( F \) is the Boolean expression of the flow \#flow{dstIP= IP(v), srcIP = IP(u)} and \( I_x \) is the output interface to the next node in the path \( P \). For instance, to express there is a path from Domain 1 to Domain 2 as in Figure 3, we write:

\[
\begin{align*}
device_bdds(_, [], []) & \rightarrow []; \\
device_bdds(U, [V | P], []) & \rightarrow \text{Out} = \text{interface:port(U, V)}, \\
F & = \text{device:project(Out, U)}, \\
[I | device_bdds(V, P), P] & \rightarrow \text{conn(P)} \\
Bdd & = \text{bdd:andset}(device_bdds(P)), \\
Res & = \text{bdd:imp}(F, Bdd), \\
\text{Ret} & = \text{if Res=1 \rightarrow true; true \rightarrow false end}, \\
\text{bdd:delrefs}(Bdd, Res), \text{flow:delete}(P), \text{Ret}.
\end{align*}
\]

\[\text{conn(P)} \rightarrow \text{Out} \rightarrow \text{F} \rightarrow [I | \text{device_bdds(V, P), P}].\]

\[\text{Out} = \text{interface:port(U, V)}, \]

\[F = \text{device:project(Out, U)}, \]

\[I = \text{device_bdds(V, P), P}.\]

\[\text{conn(P)} \rightarrow \text{Bdd = bdd:andset}(device_bdds(P)), \]

\[\text{Res = bdd:imp}(F, Bdd), \]

\[\text{Ret} = \text{if Res=1 \rightarrow true; true \rightarrow false end}, \]

\[\text{bdd:delrefs}(Bdd, Res), \text{flow:delete}(P), \text{Ret}.\]

D. Security Requirements

System administrators would like to ensure high-availability for their networks, such as creating a redundant path between two nodes. It is necessary to ensure that the security properties on the original path (for example, denying certain traffics) are also maintained on the redundant path, in case of link failure. Because, in many situations, the correct placement of firewall nodes can be tricky. Let \( P \) be the original path and \( Q \) be the redundant path to node \( v \). Then, the two paths have the same security properties when a flow \( F \) passes through them if the following Boolean expression

\[
\left( F \land \left( \bigwedge_{x \in (P \setminus \{v\})} \Psi(x, I_x) \right) \right) \leftrightarrow \left( F \land \left( \bigwedge_{y \in (Q \setminus \{v\})} \Psi(y, I_y) \right) \right)
\]

is true, where \( F \) is the Boolean expression that represents a flow.

Security analysis has been further investigated in a previous paper [8], and we can formalize all of the previous work in our system. 

\[\text{In BDD, the truth of } \pi \rightarrow \eta \text{ is equivalent to say that } \pi \text{ is a subset of } \eta.\]

V. CONCLUSION AND FUTURE PLAN

ConfVS is an attempt to address the complexity of configuration management by abstracting from system variety and providing a comprehensive model for specifying network infrastructures as well as configuration requirements. We demonstrate how ConfVS can successfully collect configuration data using the NETCONF protocol and model the data as a set of facts (tuples). ConfVS relies on the Erlang language to specify the network requirements, and we demonstrated that ConfVS provides an expressive framework to model the infrastructures in Erlang.

There are different tasks for configuration management and this paper addresses a single task of configuration verification. As future plan, we extend our model to support other tasks such as configuration synthesis.

Acknowledgment

The authors would like to thank Mr. Carl Moberg from Tail-f System for providing the ConfD software. Mr. Moberg also provided valuable comments for the ConfVS project and recommended using Erlang for requirement specifications.

References