The International Journal of Computational Science, Information Technology and Control Engineering
(IJCSITCE) Vol.5, No.3, July 2018

NETCDL : THE NETWORK CERTIFICATION DESCRIPTION LANGUAGE

Cody Hanson and Kristen R. Walcott

University of Colorado, Colorado Springs

ABSTRACT

Modern IP networks are complex entities that require constant maintenance and care. Similarly, constructing a new network comes with a high amount of upfront cost, planning, and risk. Unlike the disciplines of software and hardware engineering, networking and IT professionals lack an expressive and useful certification language that they can use to verify that their work is correct. When installing and maintaining networks without a standard for describing their behavior, teams find themselves prone to making configuration mistakes. These mistakes can have real monetary and operational efficiency costs for organizations that maintain large networks.

In this research, the Network Certification Description Language (NETCDL) is proposed as an easily human readable and writeable language that is used to describe network components and their desired behavior. The complexity of the grammar is shown to rank in the top 5 out of 31 traditional computer language grammars, as measured by metrics suite. The language is also shown to be able to express the majority of common use cases in network troubleshooting. A workflow involving a certifier tool is proposed that uses NETCDL to verify network correctness, and a reference certifier design is presented to guide and standardize future implementations.

KEYWORDS

Physical Network Testing, Automated Testing, Integration Testing

1. INTRODUCTION

Internet Protocol (IP) networks are complex entities that exist in a chaotic and dynamic environment. A network is comprised of many pieces of advanced equipment, including routers, switches, firewalls, and wireless access points. Other often overlooked components of a network are the physical interconnects between devices, which include copper, fiber-optic cabling, and wireless spectrum. Whether installing a new network, or maintaining and expanding an existing one, ensuring that all devices are configured properly and in compliance with the intended network design is not a trivial task. The network designer must carefully specify every aspect of network construction, including routing protocols, IP subnetting and VLANs, link bandwidth capacities, packet filtering and firewall rules, etc. Because of the myriad configuration options available on modern equipment, it is highly likely that something will become misconfigured during an install or upgrade, or business requirements were not clearly communicated to the installation team. Networks are also subject to entropy as physical cabling degrades, hosts are added and removed, additional routing and switching is deployed, and new traffic patterns

DOI: 10.5121/ijcsitce.2018.5301

emerge. As a result, keeping a network in top shape is a process that takes a large amount of energy and attention from talented IT and networking professionals.

High visibility into the state of a network, and high confidence that the information is accurate, can be a great advantage for any organization. Visibility enables organizations to be more effective in maintaining a highly performant network. Nearly all parts of a modern business rely on network connectivity, and downtime at a site can be a costly loss of productivity. Maintaining a detailed and accurate picture of a network is difficult in practice. As with any large system, complexity invades and employees do their best to `just keep it working'. This introduces risk for the network owner in the form of expensive downtime, poor performance, and difficulty in upgrading and expanding their investment in the network. If an employee is afraid that they will break something by working on an established network component, their effectiveness is diminished. When undocumented and informal `tribal knowledge' about the state of the network grows, teams become less effective as they grow and churn.

There currently does not exist an accessable tool that allows a network administrator to systematically verify that a network or points of connectivity are behaving as intended. Verification of proper network behavior is largely done with ad-hoc testing and improvised tools, or troubleshot only as problems surface. It is common for a network team to plug their laptop or workstation into a problem area and manually diagnose and triage problems. Sometimes misconfigurations may lie unknown for a long time, until after a serious availability or security incident. There are tools and software packages available that attempt to fill this niche, but they often fall short on usability, and become their own costly system to maintain.

Professionals in the software and hardware design industries have been using Domain Specific Languages (DSL's) for many years. From Hardware Description Languages (HDL's) that allow a circuit designer to clearly define how they want their integrated circuit to behave, to a software engineer who uses an automated test suite that can check for defects and regressions, these languages help to encode the intent of a human expert into a format that can be consumed by a machine. The machine can then assist with implementation, or verification of correctness, in a far more efficient manner than a human.

In this work, we present the Network Certification Description Language (NETCDL), and an associated network certification workflow. NETCDL is a DSL designed to allow a user to expressively describe how a network should behave. A user of NETCDL expresses her requirements as a NETCDL specification document, which is a series of statements that describe the desired state of network connectivity. The document is then the input to a NETCDL certifier, the software that can parse the NETCDL language and verify the document's assertions against a live network connection.

The NETCDL language's simplicity and expressiveness are key metrics for evaluation. This is because the intention is for NETCDL statements and certifiers to replace ad-hoc verification methods that are often based on difficult to maintain programs and scripts. Network engineers do not need to be programming experts in order to use NETCDL to check their networks for correctness. NETCDL statements are simple and declarative, which removes the need for complex syntax and program logic, and certifier software abstracts away the specifics of verifying assertions about network states. Certifier design must ensure that certification is quick and reliable, important properties for successful use on a live job site.

This research makes the following contributions:

- Developing a new technique for rigorous certification of network behavior;
- Describing the NETCDL language, and how it enables certification of networks;
- Demonstrating that the language is simple to understand, and supports the majority of common use cases;
- Proposing a reference design for a NETCDL certifier, and how certifiers can become important tools for network engineers.

In this work, the systems described are intended to be a new open standard which can be built upon and extended by others. In order to meet this goal of openness, guides and resources for implementers have been included, such as certifier standards and language grammars.

2. RELATED WORK

There has been a large amount of research in developing techniques for diagnosing problems and improving operations. We will present several prior works that have influenced the creation of the NETCDL language and certification concept.

Hardware Description Languages. Some of the most indispensable tooling used by hardware engineers are Hardware Description Languages (HDL's), such as Verilog and VHDL. These languages allow a hardware designer to assert how they want their circuit to behave. Languages like Verilog [1] describe digital logic. The designer can assert that a block has the properties of an adder, multiplier, memory, and many other fundamental units without needing to be concerned about how they are implemented internally. The HDL file eventually will be input to a synthesizer program which will translate the logical design into a physical circuit that can then be manufactured. It is this high level of abstraction that network engineers could also find useful while designing and maintaining their systems.

Software Testing. For many years the software development industry has enjoyed expressive and useful languages that let engineers and designers describe how their software should behave. Tests for software allow for unlimited flexibility when it comes to creating synthetic input data, mocking and injecting dependencies, and making assertions about the outputs of a function or system. Most common programming languages have their own ecosystem of tools for enabling the user to write tests against the rest of their code.

A standout among these testing tools is one that allows software specifications to be written in natural language. This tool is called Cucumber [14], a Behavior Driven Development (BDD) framework mainly used for integration and feature acceptance testing. It allows product planners and software designers to specify requirements in plain English, which are then matched to sections of executable code that perform the test. Cucumber matches plain English descriptions with the appropriate test definition files via pattern matching and regular expressions. These test definition files are the computer code which actually performs the test.

Another software testing tool that seeks to be expressive and easy to use is Should.js [18], an assertion library for Javascript that lets the user chain together english phrases that make testing clear and easy to understand. Should.js is a clever and useful library that helps to bridge the gap

between computers and their human operators but remains a technical tool that would be difficult for non-programmers to use and understand.

NETCDL aims to bring the expressiveness of Cucumber to a simple imperative language that does not require the user to interact with computer code at all. Cucumber uses natural language in addition to requiring computer code to be written, while NETCDL is only natural language. It is preferrable for our plain language statements to be simply declarative in nature, and that software be allowed to handle the details instead of a programmer. This is the purpose of the NETCDL certifier software, to carry out the certification according to the statements defined by the user.

IT Automation and Behavior Driven Infrastructure. Behavior Driven Infrastructure (BDI) [16] is the notion that rather than performing ad-hoc and incremental changes to get the computing and network equipment into the required state, an administrator should instead describe how it should behave with an IT automation framework. There are many of these frameworks to choose from, and they each have their own dialect with which one can describe how machines and network devices should be configured. The benefit of this approach is that the framework takes care of the details of configuring the infrastructure, in order to conform to the specification. Examples of these BDI frameworks include Chef [6], Puppet [20], SaltStack [24], and Ansible [5]. These have all seen wide use within the system administration, operations, and developer communities.

BDI is an attractive deployment methodology, especially for large deployments, because it automates and standardizes the mechanisms by which machines are managed and configured. If attended to individually, servers might not have the right version of software, network configurations could be incorrect, and distributed systems might not be wired together properly.

BDI and software testing tools can be used together [15]. Like Cucumber, BDI tools often do not hide all of the complexity of what is being described. In order for BDI tools to function properly, computer code must still be written and technical details must be attended to.

Existing Network Certification Tools and Software. Since networks and computers have existed, tools have existed to help troubleshoot their problems. Some of these tools are dedicated hardware. Others are software tools that are deployed on a general purpose computer and are available with either commercial or open-source licenses. Nagios [11], for example, is a long established and popular open-source framework that allows the user to set up monitoring and alerting for the network services that they care about. The project cucumber-nagios [17] combines the network monitoring capabilities of Nagios, with the expressiveness of Cucumber. Cucumber-nagios is a Ruby library that allows the user to write Cucumber statements about how a network resource should behave, and then have the test run with the output being in a format compatible with an existing Nagios system. At the time of this writing, Cucumber-nagios appears to be limited to testing with HTTP servers, and Secure Shell (SSH).

RANCID (the Really Awesome New Cisco Config Differ) [25] is a software system that can monitor the configuration texts of a machine (such as a router) and alert users when something about those configurations has changed. The information RANCID provides enables teams that are better equipped to manage the complexity of a changing and dynamic set of configurations. Changelog diffs and recored history provide extra information on on-the-fly configurations.

Network engineers often need to use interactive command line interfaces when configuring and troubleshooting equipment. The interactive nature of these interfaces makes reliable large scale

configuration automation a challenge. It can be difficult to write a program that can simluate the way a human would use an interactive command line program. Expect [21] is a technique developed to solve this difficulty. An example use case for Expect would be to log in to many devices and add a new firewall rule to all of them at once. This saves time and reduces the chance of errors compared to manual configuration. Expect is a great example of a purpose built language and technique that helps engineers better manage complexity in their networks, i.e., expect scripting.

Some commercially available tools designed to fill the network verification niche include the LinkSprinter [8] and the OneTouch AT [9]. Both of these products partially accomplish what NETCDL seeks to enable. The LinkSprinter has a simple `plug and play' model, where a small number of essential connectivity checks are performed for each port that is tested, and tests results are then sent to a central database for later analysis. The OneTouch AT has the ability to \visually" script various network connectivity checks that can be run quickly in a repeatable way. Available checks on the OneTouch AT include ping, port open, HTTP server availability, performance testing, and many others. A limitation of the OneTouch AT is that complex assertions beyond the tests offered by the manufacturer are not possible. The LinkSprinter is limited by the small number of checks it can perform, making it suitable only for basic verification. NETCDL improves upon these products by giving the design of more complex tests to the user, in addition to being an open standard.

Existing Network and Service Description Languages. The Network Description Language (NDL) [26] uses the Resource Description Framework (RDF) [7] to create a standardized way to model network connectity in IP networks. The NDL has three main entities: locations, devices, and interfaces. Each of these have further properties describing them and how they relate to the rest of the network topology.

Large and detailed XML documents, while a very useful tool for machines and automated systems, are difficult for humans to read, and almost impossible to write correctly without some computer support. The reason that tools like Cucumber have enjoyed success is that they are easy to read and understand. NDL and the Network Markup Language are better suited to detailed structural descriptions of networks, rather than the day to day troubleshooting and verifying of their behavior, which are key use cases that NETCDL enables.

3. THE NETCDL LANGUAGE DESIGN

Certifying that a network conforms to a detailed specification enables network owners to have confidence that their infrastucture will be available, performant, and secure. A key design goal of NETCDL was to make this certification capability accessable to those without a background in computer programming, or other technical languages. This is accomplished by designing the language grammar to resemble English sentences and phrases, and limiting the use of special symbols and other syntax typical of most programming languages. Complex multi-line statements and expressions are also deliberately not included in the grammar.

3.1. Writing a NETCDL Specification

NETCDL statements let users describe how their network should behave. Each statement asserts a single condition. A user collects many NETCDL statements into a NETCDL specification

document. This document then defines the aggregate behavior that will be evaluated by NETCDL Certifier software, which is discussed in Section.

The user can tailor their specification documents to their specific needs. One user might create a specification document per client connection point, such as each wall jack on an office floor, or each WiFi access point in a building. This would afford very targeted and specific certification, customized for each particular client machine. A different user might choose to write a smaller number of NETCDL documents, one for each class of network access, or network device type. These different classes could represent an unpriviledged user, an employee, or a highly priviledged network administrator. Guest connectivity could be verified to be appropriately limited, while a network administrator can be assured that all necessary access is present for them to perform their duties.

Like most computer languages, NETCDL allows comment statements that are ignored by parsing software, as well as the use of whitespace to group and organize statements. The format of a comment is a line prefixed with #, similar to Python. NETCDL documents are great candidates to be stored in a version control system because the history of a document and the associated comments can inform the user of how the network is changing over time, and why.

```
# Specify easy to remember aliases for my home network.
define host 192.168.1.1 as MyRouter
define host 192.168.1.2 as MyPrinter
define network 192.168.1.0/24 as HomeNetwork
```

Figure 1: Example Usage of the Define Statement

Sometimes it is important to verify that a particular network condition can not occur. To enable negative assertions, most NETCDL statements can be negated using the should not phrase. Negative assertions can be used to make sure certain machines cannot connect to each other, or that sensitive network traffic is isolated. This is useful in the case of security auditing where access restrictions need to be verified.

3.2. NETCDL Statements

Define Statements. Networking documentation is often studded with obscure notation, including IP addresses, network ranges, and hostnames. Repeating these can clutter a specification document and make it less maintainable. Like other languages with macros or named variables, NETCDL enables the user to create their own aliases for important hosts and networks for commonly referenced network locations.

Define statements start with the define keyword, followed by the define type: host or network. The rest of the statement aliases two strings to each other. In the example in Figure 1, `192.168.1.1` is defined as `MyRouter`, `MyPrinter` will be translated to `192.168.1.2`, and `HomeNetwork` will be translated to the network prefix `192.168.1.0/24`.

Once a user has defined an alias for a network or host, that alias can be used in any valid grammar context where a domain name or IP address would be allowed, such as the target of a

ping test. Adding a layer of indirection helps to future-proof the certification document against IP address changes.

VLAN and Link Statements. Certain misconfigurations such as VLAN and link mis-configurations are easy to fix, but can prevent all other network operations from proceeding. If these settings are misaligned with what the connecting device expects, performance of the link could suffer, or the transmission of traffic may be interrupted. Any connectivity troubleshooting or verification should begin with checking these fundamental settings. VLAN and link Statements help to verify these basic layer-2 configurations.

Connected link bitrate and duplex refer to auto-negotiated settings for how traffic is transmitted through the transmission medium. Verifying duplex and bitrate are important because they ensure that the link behaves optimally. NETCDL allows the user to specify two duplex modes (full or half), and a bitrate specified in megabits per second (the most common settings being 100 Mb/s and 1000 Mb/s).

VLAN's are logical partitions of an IP network. Modern networking hardware can have a different VLAN assigned to every single port, so it is a common mistake to have a port belonging to the wrong access VLAN. Verifying the access VLAN of a client port is useful because it can have adverse security or connectivity implications. VLAN's are identified by an integer VLAN ID, either in the header of a packet or by special broadcasts sent from routers and switches.

```
access vlan should be 500
link speed should be 1000Mb/s
link duplex should be full
```

Figure 2: Example Usage of Link statements

```
# On home networks, it is common for gateway, DHCP server, and DNS
    server to be the same machine
define host 192.168.1.1 as my_router
dhcp gateway should be my_router
dhcp server should be my_router
dhcp dns should be my_router
# Address assigned to host should be within this network range.
dhcp network should be 192.168.1.0/24
```

Figure 3: Example Usage of DHCP Statements

In Figure 2 we can see that the desired access VLAN ID is 500, and expected duplex and speed are specified as full and 1000 Mb/s. These statements can be easily negated using the should not phrase in place of the keyword should.

DHCP Statements. Dynamic Host Configuration Protocol (DHCP) is commonly used to assign a client machine an IP address, a default gateway (router), and DNS servers. When DHCP is misconfigured or unavailable, these critical settings do not get set on the client, and connectivity fails. DHCP servers are also extremely common (almost every consumer WiFi router contains a

DHCP server), and unauthorized ones can appear in a controlled environment when they are not wanted.

DHCP Statements in NETCDL ensure that DHCP information comes from the correct source, and that it is accurate. Verifying the identity of the DHCP server is important as well, because an unauthorized DHCP server on a network that is responding to DHCP DISCOVER probes can cause networks to behave erratically, and potentially be a security risk.

DHCP statements begin with the dhcp keyword, followed by the type of DHCP information to verify: `gateway`, `server`, `dns`, or `network`. Then the common should or should not phrase asserts the value of the DHCP element.

IPv4 network ranges are specified using the common notation format of network number/bitmask, where bitmask is the number of mask bits in the network mask. IPv6 is not officially supported in the initial version of the NETCDL grammar.

In Figure 3, a common home networking scenario is verified, along with a useful host define statement.

DNS Statements. The Domain Name System (DNS) translates common network and domain names into the underlying IP addresses that end up in the IP headers of packets. If DNS is unavailable or misconfigured, even though the connection to the internet is established, most users would be unable to complete their tasks.

NETCDL DNS statements can ensure that important names resolve, either to any address at all, or to a specific address. This is important to verify that DNS records have propagated correctly throughout the DNS system hierarchy, as well as to verify that the designated DNS servers are reachable by clients.

```
# 8.8.8.8 is a Google public DNS server
domain name google.com should resolve using server 8.8.8.8
domain name notreal.site should not resolve using server 8.8.8.8
domain name devserver.local should not resolve to 192.168.1.144 using
    server 8.8.8.8
domain name devserver.local should resolve to 192.168.1.144 using
    server myRouter
```

Figure 4: Example Usage of DNS Statements

```
MyRouter should be reachable by ping
SecuredServer should not be reachable by ping
```

Figure 5: Example Usage of ping Statements

DNS statements begin with the domain name keyword, followed by the network name that will be resolved using the DNS protocol. Then the user can specify if the name should resolve or not, and optionally specify what they expect the name to resolve to. Finally, the IP, domain name, or

alias of the DNS server to use for the lookup is provided. Examples of the usage can be seen in Figure 4.

Ping Statements. ping (ICMP Echo) is a commonly used technique for checking connectivity between two hosts. The NETCDL ping statements let the user specify the ping target, and if the ping should succeed or not. A user might want to use the ping statement in the negative case to verify that a server is not reachable from an unsecured network. A ping is considered a success if at least one response packet is received from the target. If no response is received, or a message about the traffic being undeliverable or rejected, then ping is considered to have failed. While ping is in some ways inferior to a Port Open test because ICMP Echo packets are often blocked on modern networks, it remains a common tool that is often used by those in the networking field. An example usage of the ping statement is demonstrated in Figure 5.

Port Open Statements. The majority of all network connected software operates using the concept of `ports'. Port numbers direct traffic to the appropriate software listening on a server. Ports are often `closed' by default (meaning that they reject traffic), and are commonly misconfigured on the host machine. Many operating systems have all ports closed by default, as a security practice. The most common protocols that use ports are the transport layer protocols TCP and UDP.

TCP and UDP ports can be tested for connectivity on a server using the Port Open statement. This is useful to verify because even if a server is reachable more generally (for example with an ICMP Echo, or ping), traffic to a given application port may not be possible due to firewalls, or software misconfiguration. Verifying a port is reachable, however, does not imply that the underlying software that uses the port is configured properly. For example, if a host is reachable on port 80, it does not necessarily ensure that the HTTP server functions correctly. Despite these caveats, the Port Open statement is an important step in verifying proper network application and firewall operation.

```
#Allow SSH
DevelopmentServer should be reachable on TCP port 22
#Don't Allow Telnet
DevelopmentServer should not be reachable on TCP port 23
VideoServer should be reachable on UDP port 4000
```

Figure 6: Example Usage of Port Open Statements

```
iperf download from slowserver.com should be at most 30Kbps
iperf upload to slowserver.com should be at most 30Kbps
iperf upload to fastserver.com should be at least 30Kbps
```

Figure 7: Example Usage of iperf Statements

Port Open statements are demonstrated in Figure 6. They specify the destination host, transport protocol, and port number. Transport refers to either the TCP or UDP protocols. The standard should/should not phrase also applies.

Bandwidth Testing Statements. Even if all connectivity is achieved, servers are up and running, and all else is working perfectly, a network can suffer from poor throughput. This is especially harmful for high bandwidth applications such as voice and video. Though available bandwidth can depend on network utilization, router and switch misconfigurations can cause insufficient bandwidth even on an idle network.

Bandwidth testing statements allow the user to specify an iperf [19] test to an Iperf server. iperf is a widely used open-source tool that allows point-to-point bandwidth testing. An iperf test consists of two hosts, one acting as server and the other as client. A client initiates a test by contacting the server and specifying the parameters of the test, such as transport protocol, duration, target bitrate, and direction. While iperf supports testing both TCP and UDP streams, NETCDL does not specify the protocol in the language, so TCP is assumed. The target server is assumed to be running a copy of the Iperf3 software on the default ports, or similar software that conforms to the Iperf3 protocol.

`iperf` statements begin with the `iperf` keyword, followed by the direction of the test, download or upload. Download means that the traffic flows from server to client, and upload means the inverse. NETCDL certifiers are always `iperf` clients. Next, the Iperf server is specified, followed by a should/should not clause. Finally, the expected bitrate of the transfer is specified, in addition to a threshold clause, which specifies if the measured bitrate should be higher or lower than the expected bitrate. Example usage can be seen in Figure 7.

A user might write statements like these to ensure that network performance is adequate for users. In a different situation, such as a guest or shared network, the user might ensure that bandwidth limits are being enforced, to prevent a single user from monopolizing network resources unfairly.

File Fetch Statements. NETCDL File Fetch Statements allow the user to exercise three of the most common file transfer protocols, HTTP, FTP, and TFTP. HTTP and FTP are important protocols to verify the operation of, because they are very popular among end users. TFTP can be important to verify because it is commonly used to bootstrap and upgrade network equipment like routers and switches.

File Fetch Statements begin with the protocol name and the keyword server, followed by the location of the server, either an IP address, host alias, or DNS name. Next is a standard `hould/should not clause. After that, the serve keyword along with the expected filename that should be fetched from the remote server appear. Finally, the port number to connect on can be specified, if desired. If no port is specified the default ports for the specific protocol are used. Example usage can be found in Figure 8.

These statements are important because they can simulate the entire end to end user experience for common network services. Merely verifying that a server is listening on the correct port does not gaurantee that files can be served as expected. Security devices such as packet-inspecting firewalls also can be fully exercised and tested by simulating real application layer traffic.

```
#HTTP
http server at example.com should serve "/index.html"
http server at example.com should not serve "/secure/secretfile.html"
http server at example.com should not serve "/index.html" on port 12345
#TFTP
tftp server at 192.168.1.144 should serve "afile"
#FTP
ftp server at ftpSite should serve "pictures.zip" on port 2121
ftp server at ftpSite should not serve "pictures.zip" on port 21
ftp server at ftpSite should not serve "securedFile"
```

Figure 8: Example Usage of File Fetch Statements

```
traceroute to 10.0.5.5 should traverse [192.168.1.1 10.0.0.1]
traceroute to 10.0.0.1 should traverse [MyRouter]
```

Figure 9: Example Usage of traceroute Statement

Traceroute Statements. One of the more complex parts of computer networking is routing between IP networks. Many routing protocols run on routers to dynamically build the forwarding tables. Ensuring that routing behaves optimally is an important part of network verification. A common tool used for this purpose is traceroute that probes and discovers the path by which the traffic to a given destination travels. A network engineer can then determine if traffic is flowing along the expected pathways.

NETCDL `traceroute` statements enable assertions about the path that a packet takes on its way to a destination. This is useful for verifying that routing tables are configured properly, or to ensure that no extra hops or loops are encountered. To use this statement, the user specifies the traceroute target, and an ordered list of consecutive hops to verify, starting from the first hop. traceroute for NETCDL is based on TCP, using decrementing time-to-live (TTL) counters in the packet headers.

`traceroute` statments begin with the `traceroute` to keywords, followed by the target of the test, a hostname, IP address, or host alias. Next the keywords should traverse are then followed by an ordered, space-delimited list of routing hops, which can be hostnames, IP addresses, or host aliases. Example usage can be seen in Figure 9.

Packet Capture Statements. One of the most powerful (and time consuming) techniques network engineers have in their arsenal is to capture packets from a network and inspect them. This gives a raw and unfiltered look at exactly what happened in a given network. Some common questions that packet inspections can answer are: Did a machine respond with any packets at all? Are certain types of packets detected? Are TCP/UDP port numbers configured properly? Is outgoing application layer traffic properly formed?

```
#IP source and destination assertions, for network ranges and hosts
packets from network DMZ should not be seen
packets to host 10.250.0.1 should not be seen
#Packet and Frame type field assertions
packets with type 0x18 should not be seen
frames with ethertype 0x1F should not be seen
#TCP and UDP port assertions
packets with TCP destination port 544 should be seen
packets with TCP source port 1000 should not be seen
packets with UDP source port 1010 should be seen
packets with UDP destination port 4000 should not be seen
```

Figure 10: Example Usage of Packet Capture Statements

Packet inspection is an indispensible tool for advanced users, but can often be difficult to wield for less experienced users. NETCDL Packet and Frame statements allow the user to make assertions about network traffic captured in a network. Assertions can be made about the presence of:

- IP packet source and destination fields
- IP packet source or destination network ranges.
- IP packet type field values.
- TCP and UDP source and destination ports.
- Ethernet frame ethertype values.

There are four classes of statements: IP Packet Type assertions, Ethertype assertions, IP Packet source/destination address assertions, and Transport protocol port assertions. Example usage can be seen in Figure 10. A packet of a certain type is "seen" if it is present in the array of captured packets and frames. The absence of a packet from a capture does not necessarily guarantee that a packet of that type would never arrive. NETCDL Certifier implementation will determine the packet capture duration. Longer captures are more likely to obtain a representative sample of packets, but could cause certification to take more time.

For a complete overview of the NETCDL grammar and all available language features, please refer to our website.

4. CERTIFIER DESIGN AND IMPLEMENTATION

Most computer languages are intended to be executed or evaluated in some way, whether by a compiler, interpreter, or other software. For the NETCDL language this software is known as a NETCDL Certifier. A certifier is a critical part of the workflow for NETCDL because it is the agent by which the statements in a NETCDL document are evaluated in the real world. A user provides a NETCDL document as input to a certifier that has a link to the network location under test. The certifier parses the document, develops a plan to evaluate the assertions, and then carries them out against the network interface. As certification proceeds, the certifier can render a 'pass/fail' verdict on whether the specification of the input document was met. This workflow is illustrated in Figure 11.

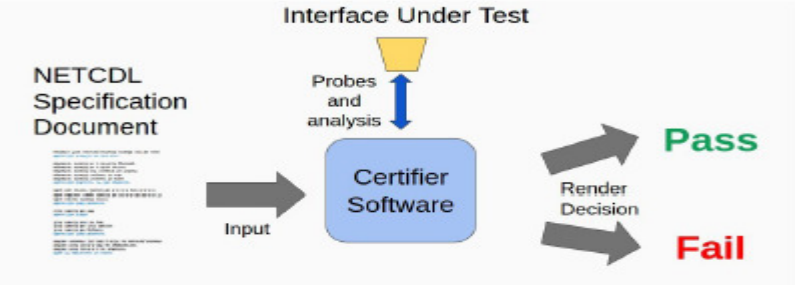

Figure 11: NETCDL Certification Workflow

A well designed certifier should:

- Minimize the time required to evaluate the specification.
- Be as non-invasive as possible to the network under test (i.e. be a good steward of limited resources).
- Provide helpful feedback to the user about network problems detected.

The initial reference implementation of the NETCDL certifier is meant to showcase the use of the NETCDL Language and illustrate the new concept of certifying network connectivity. It is also meant to serve as a guide for future certifier implementations. Others are encouraged to implement their own certifier, taking into account the NETCDL Certifier Standards document available on our website.

4.1. Certifier Operation

The certifier has three main tasks: (1) parse and verify the input as valid NETCDL statements, (2) for each assertion in the input document, render a Pass/Fail result in a time efficient manner, and (3) report the results of certification to the user.

After parsing the input according to the NETCDL grammar, the certifier will have everything it needs to evaluate each NETCDL assertion. As seen in Figure 12, some tests can be considered 'active' while others would be 'passive', or 'non-active'. An example of an active test would be a ping test (derived from the NETCDL ping statement); Traffic must be sent out the network interface to attempt to elicit a response. An example of a passive test would be looking for the presence of a particular type of packet. This passive test is merely searching through data that is already collected. A key enabler for passive tests is the packet capture component. Every packet and frame traversing the network interface under test is collected to be input for the Packet Capture NETCDL statements. It is a smart optimization to listen for packets the entire duration of certification, to gather as much traffic as possible to be data for the packet and frame assertions.

All test results (which are the truth values for the assertions of the specification document) are collected into a final report to be displayed to the user, such as the one seen in Figure 13. In a simple implementation, results may be printed to a console window, with color-coding to correspond to 'green/passing' and 'red/failing'. More advanced implementations could be presented to the user as well, including interactive diagnostic print-outs, or other useful graphical user interfaces.

Another component of Figure 12 worth mentioning is the depiction of multiprocessing. It is a common technique to divide work units in software amongst child processes or threads in order to parallelize the workload, and take advantage of multiple CPU cores.

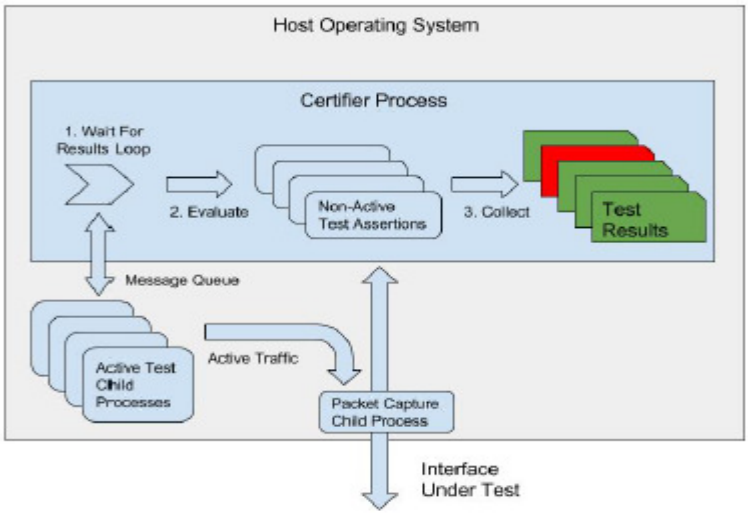

Figure 12: NETCDL Certifier Software Block Diagram

Figure 13: Example Certifier Command Line Output With Failing and Passing Statements

After certification is complete, the software should close all connections, release IP addresses, and relinquish other resources that could be needed by real clients. It should also 'reset' itself in order to be ready to carry out certification again on the next network connection point.

Reference Certifier Design Philosophy. In conjunction with the development of the NETCDL language, this work presents a reference implementation of the first NETCDL certifier software. Reference implementations are important because they provide a starting point for future work to leverage, and a philosophical guide for future designs. The goal of this implementation was to

represent a 'minimum viable product' which sufficiently demonstrates all of the important concepts of NETCDL.

The software was implemented using Python. While Python is cross platform, the initial version of the software targets standard tooling available on the GNU/Linux Operating System. Development of the software was done using Ubuntu 14.04. For full software version notes, see our website.

4.2. Design Challenges

Optimizing Certification Performance. Fast and reliable certification is important for applications in large networks and new installations. The number of connections that need to be certified could be in the thousands. At this scale, if the software were able to reduce certification time by 15 seconds per link, for 1000 links, over four hours of idle time could be recouped. This is especially important if the number of certifiers on a job site is limited, and certification tasks cannot be split up among workers.

Fast certification speed is achieved by executing as many tasks in parallel as possible. This is important because as the software needs to send packets to elicit responses from remote machines, it is possible that we must wait for a timeout in case of no response. If dozens of requests had to time out sequentially, certification would be unacceptably slow. Separate child processes are used, rather than threads, one per Active Test. Multiple processes allow the tasks to fail independently if necessary, which provides resilience. Inter-process message queues are used to move data between the parent and child processes.

Further tuning for speed can be accomplished by minimizing timeout periods for non-responsive servers, or building more advanced heuristics for knowing when a test is guaranteed to fail, and then skip those tests. A test that talks to a web server would be guaranteed to fail for example if our local router was unreachable.

The reference certifier implementation is open source and can be found on Github [12].

5. EVALUATION

The framework is evaluated with respect to two criteria: NETCDL Grammar Complexity, and NETCDL Language Expressiveness.

A key goal of NETCDL is to be an approachable and simple to learn DSL. One way to objectively measure these properties is to examine the language grammar. Grammars can be analyzed by tools that generate standardized metrics which quantify the size, structure, and complexity of a grammar. Because existing alternatives to NETCDL are mostly programming and scripting languages, it is useful to make comparisons between their grammars.

Another important property of any tool is the ability to support common use cases within the target domain and user base. We refer to this property as “Language Expressiveness." An expressive language allows the speaker or writer to easily and fluently encode their ideas. In this domain, these ideas are assertions about network behaviors that are important to network engineers.

5.1. Evaluation of Language Complexity

Prior work in Grammar Engineering [4] show how we can take an objective approach to designing and evaluating computer languages. Taking a quantitative approach to analyzing the NETCDL language grammar and comparing it to other well known computer languages is a good way to estimate how difficult the new language is to read and write for a human. The SdfMetz [3] project provides software that can gather complexity metrics from a grammar expressed in the Syntax Definition Formalism (SDF) [13] format. An SdfMetz environment was built using notes and instructions from this prior research. Then the NETCDL Grammar was re-written using SDF, in order to be compatible with the tools. SdfMetz was then used to analyze this equivalent SDF grammar. To see the SDF version of the NETCDL grammar, please refer to our website.

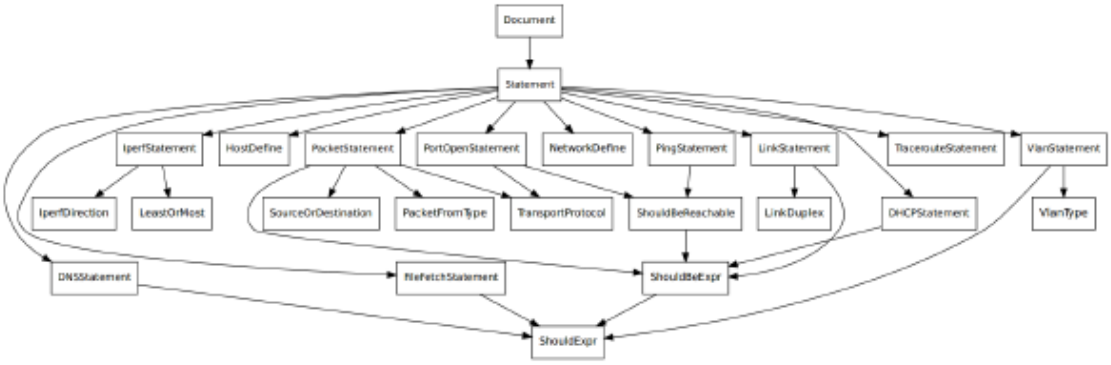

Figure 14: NETCDL Grammar Diagram - Generated by SdfMetz

An initial depiction of grammar structure as evaluated by `SdfMetz` can be observed in Figure 14. This graphic of the NETCDL SDF grammar visualizes which grammar units are available, and which rely on each other (indicated by lines with arrows). For example, the `ShouldExpr` (should expression) is an important part of the language because it is relied on by many other parts of the language, whereas the `TraceRouteStatement` is not relied on by any other parts of the language. The width and height of this tree can also give us a qualitative view of the grammar structure.

The most interesting analysis that SdfMetz can provide us about a grammar are the quantitative metrics. Descriptions of the definitions and practical meaning of these metrics can be found in Figure ??. These metrics were chosen as the ones for comparison because they are well known and have been used historically to describe context free grammars. SdfMetz does support other metrics, which were not used are part of the evaluation. Figure 15 contains the raw output of the SdfMetz tool, including the abbreviations and descriptions of each metric.

To get an understanding of what these numbers mean in a practical sense, it is useful to compare them to metrics from other well-known languages. Also present in the SdfMetz research is a data set for other grammars that were examined by the tool. They include well known languages such as C, C++, Java, PHP, Javascript, and Verilog, among others. The full dataset of 30 grammars, was the standard of comparison for the NETCDL grammar metrics.

Table 1 summarizes the results of this comparison. The languages were ranked out of 31, with a 'lower' ranking denoting a better performance in a particular metric category. Another useful comparison is to look at languages that were similar in score to NETCDL for a particular metric.

This lets us use our experience with these languages to get a sense of the complexity of NETCDL in a qualitative way. For example, for the HEI metric, NETCDL was comparable to BibTex and MatLab. These comparisons are also included in Table 1.

```
Size and Complexity Metrics Created by SdfMetz

Number of unique terminals       (TERM) : 49.0
Number of defined non-terminals  (VAR)  : 24.0
Number of used non-terminals     (UVAR) : 80.0
Number of productions            (PROD) : 42.0
Cyclometric Complexity McCabe    (MCC)  : 29.0
Average RHS per non-terminal     (AVSN) : 5.5
Average RHS per production       (AVSP) : 3.142857

Halstead Metrics

Number of Distinct Operators  (n1) : 6
Number of Distinct Operands   (n2) : 81
Total Number of Operators     (N1) : 137
Total Number of Operands      (N2) : 174
Program Vocabulary            (n)  : 87
Program Length                (N)  : 311
Program Volume                (V)  : 2003.7555
Program Difficulty            (D)  : 6.444444
Program Effort (in thousands) (E)  : 12.913091
Program Level                 (L)  : 0.15517242
Programming Time              (T)  : 717.3939

Structural Metrics

Tree impurity                          (TIMP)   : 3.1620555
Tree impurity after trans. closure     (TIMP2)  : 18.972332
Count of levels                        (LEV)    : 24
Normalized Count of Levels             (CLEV)   : 100.0
Nr of Non-singleton Levels             (NSLEV)  : 0
Size of largest levely                 (DEP)    : 1
Maximum height                         (HEI)    : 6

Ambiguity-related Metrics

Nr of follow restrictions         (FRST)  : 1
Nr of associativity attributes    (ASSOC) : 0
Nr of reject production           (REJP)  : 0
Nr of unique prods in priorities  (UPP)   : 0.0
Nr of preference attributes       (PREF)  : 0
```

NETCDL metrics compared favorably to the majority of languages in the comparison dataset from the SdfMetz research, and consistently ranked in the top 5 least complex languages for a particular grammar. This lets us conclude that the NETCDL grammar achieved the goal of being simpler than most popular programming languages. One reason that the TERM metric was one of the worst performing metrics for NETCDL is due to the lack of use of symbols to denote syntax. The higher number of terminals in NETCDL is due to the fact that NETCDL is almost entirely made of English sentences, rather than relying on curly braces which are more easily `reused' (thus keeping the terminal count low). While a quantitative approach cannot describe everything about how a language feels to a user, by succeeding in minimizing the key indicators of language complexity, NETCDL is in good position to be received as an easy to understand language.

5.2. Evaluation of Language Expressiveness

A language is only useful if it can be used by writers and speakers to convey their ideas. The ideas in this context are the common network conditions that need to be certified. To objectively measure this quality, we can compute the percentage of common use cases that the language supports. To gather common use cases that network engineers and technicians might encounter in the real world, authoritative texts on network design and troubleshooting were surveyed.

Use Case	Description	Sources	NETCDL
Ping	ICMP Echo, verifies layer 3 connectivity	[2], [22]	Yes
TCP Port Open	Verifies that a TCP port on a remote host is open and reachable	[2]	Yes
UDP Port Open	Verifies that a UDP port on a remote host is open and reachable	[2]	Yes
HTTP Get	Verifies that a web server is up and running and can serve a file.	[22]	Yes
FTP Get	Verifies that an FTP server is up and running and can serve a file.	[23]	Yes
TFTP Get	Verifies that an TFTP server is up and running and can serve a file. TFTP is commonly used to bootstrap and update networking and VoIP equipment.	[22]	Yes
Traceroute	Verifies correct order of layer 3 hops, using packets with increasing TTL	[23], [2]	Yes
VLAN Trunking	Verifies that port on a router/switch tags vlans and acts as a "Trunk".	[23]	No
Access Vlan ID	Verifies that a port belongs to the correct access vlan	[23]	No
DHCP Server testing	Verifies that DHCP services are operating properly	[22], [2]	Yes
DNS Server testing	Verifies that DNS services are operating properly	[2]	Yes
Link Speed	Verifies that the network interface negotiates to the correct bitrate	[23], [2]	Yes
Link Duplex	Verifies that the network interface negotiates to the correct duplex	[23], [2]	Yes
Link Power	Verifies that the correct Power Over Ethernet voltage is present	[2]	No
Nearest Switch	Verifies that the next Layer 2 hop is the correct network device.	[2]	No
Network Bandwidth	Verifies that the correct thresholds for upload and download bandwidth for a client can be achieved.	[2]	Yes
Packet Presence Forensics	Examine contents of a packet capture to look for presence of desired packets.	[23]	Yes
Packet Sequencing Forensics	Similar to prescence forensics but ensuring packets arrive in proper order.	[2]	No
Physical Cabling Fitness	Includes verifying wiring order, signal quality, and other characteristics	[2]	No

Figure 16: Selected Grammar Complexity Metrics and Meanings

Table 1: NETCDL Grammar Performance vs SdfMetz Grammar

Metric Name	NETCDL Value	Ranking	Comparable Languages
TERM	49.0	5th	MatLab, XPath
VAR	24.0	3rd (Tie)	C (Grammar Base/SDF)
PROD	42.0	3rd	FORTRAN 77
MCC	29.0	2nd	FORTRAN 77
TIMP	3.162%	1st	
DEP	1	1st (Tie)	BibTex
HEI	6	3rd (Tie)	BibTex, MatLab
E (thousands)	12.913	3rd	MatLab

Metric Name	Description	Practical Impact
TERM	Number of unique terminals	Impacts the size of the vocabulary a user must comprehend
VAR	Number of defined non-terminals	Large VAR can increase program maintenance cost due to cascading effects to rest of grammar
PROD	Number of defined production rules	More production rules imply more rules governing the structure of the grammar
MCC	McCabe's Cyclomatic Complexity. Number of linearly independent paths (or decisions) through a graph. Related to PROD.	Measures cognitive impact on user, due to branch complexity
TIMP	Tree Impurity	Measures to what degree that the parse tree of the grammar is actually a tree. 0% means the graph is a perfect tree, 100% means the graph is fully connected
DEP	Size of largest level	Maximum Number of non-terminals in a level of the parse tree. Higher numbers denote wider trees, which increase grammar complexity
HEI	Maximum Height of Parse Tree	Taller parse trees denote more complex grammar structure
E	Program Effort [24]	Computation that combines frequency of occurance for operators and operands into a single number. Useful for comparing complexity between grammars of different sizes.

Figure 17: Common Network Troubleshooting Use Cases

Two main bodies of networking expertise were referenced while gathering use cases. The first was Interconnecting Cisco Network Devices, Part 1 and 2 [22], [23]. These are core training materials that many network engineers reference while preparing for common industry certifications, such as those offered by Cisco. They cover basics of network design and construction, including theory that applies to networking in general. The second text that was referenced was the Network Maintenance and Troubleshooting Guide [2], a book that represents decades of expertise in network troubleshooting.

Table 17 summarizes the findings from the most common use cases found in the reference network troubleshooting texts. Each row represents a common networking use case, and includes a description, sources, and most importantly, whether the NETCDL Grammar as initially designed supports it. A use case was considered supported if the language grammar could exprese the case in addition to the reference certifier being able to evaluate it. For example, the 'Ping/ICMP Echo' use case is enabled by the NETCDL Ping statement and can be carried out by the certifier.

By examining column 4 of Table 17 it can be seen that the NETCDL language was able to cover 68% (13/19) of the common use cases identified. The reason that some use cases were not able to be supported was in part due to time limitations in development time, and in part due to hardware limitations for the reference certifier. For example, it is difficult to conduct a full 'Physical Cabling Fitness' test with consumer hardware. These measurements require complex and expensive time-domain reflectometry devices, such as those produced by Fluke Networks [10]. Overall, NETCDL was able to cover a majority of the important networking tasks that network engineers use in their daily work.

6. CONCLUSION AND FUTURE WORK

In this work, we develop the Network Certification Description Language (NETCDL), which applies software systems that are well-known for their ease of use and adapts them for network needs. In addition to the language, a reference design for a NETCDL certifier was presented, along with guidelines and patterns for future implementers to leverage. We demonstrate that the language meets the goals of simplicity and expressiveness. Objective measures of the language grammar complexity compared favorably with the grammars of other well known computer languages. These results support the claim that NETCDL was designed to be easy for humans to read and write by minimizing relevant metrics. NETCDL was shown to support a majority of common use cases as defined by well regarded network troubleshooting texts and guides, thus supporting the claim of high language expressiveness.

In the tradition of other computer languages, advanced tooling may be developed that utilizes NETCDL, such as automatic router and switch configuration. More broadly, as computers continue to be critical to the lives of every person, natural language computing languages could expand into other domains such as home automation and smart devices.

REFERENCES

[1] Ieee standard for verilog hardware description language. IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001), pages 1{560, 2006.

[2] N. Allen. Network Maintenance and Troubleshooting Guide: Field-tested Solutions for Everyday Problems, 2nd Edition. Addison Wesley, Upper Saddle River, NJ, USA, 2010.

[3] T. L. Alves and J. Visser. Sdfmetz: Extraction of metrics and graphs from syntax definitions. on Language Descriptions, Tools, and Applications, page 101, 2007.

[4] T. L. Alves and J. Visser. A case study in grammar engineering. In Software Language Engineering, pages 285{304. Springer, 2008.

[5] Ansible. Ansible. http://www.ansible.com/home, October 2014.

[6] I. Chef Software. Chef - automation for web-scale it. http://www.getchef.com/chef, October 2014.

[7] W. W. W. Consortium. Resource description framework - w3c spec. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/, February 2014.

[8] F. Corporation. Linksprinter. http://www.linksprinter.com/, October 2014.

[9] F. Corporation. Onetouch at. http://www.flukenetworks.com/enterprise-network/network-testing/OneTouch-AT-Network-Assistant, October 2014.

[10] F. Corporation. Versiv dsx-5000. http://www.flukenetworks.com/datacom-cabling/Versiv/DSX-5000-Cableanalyzer, January 2017.

[11] N. Enterprises. Nagios - the industry standard in it infrastructure monitoring. http://www.nagios.org/, November 2014.

[12] C. Hanson. Netcdl reference certifier. https://github.com/netcdl/netcdl, January 2017.

[13] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdfreference manual. ACM Sigplan Notices, 24(11):43{75, 1989.

[14] A. Hellesy. Cucumber: behaviour driven development with elegance and joy. http://cukes.info/, October 2014.

[15] L. Holmwood. Behaviour driven infrastructure through cucumber. http://fractio.nl/2009/11/09/behaviour-driven-infrastructure-through-cucumber/,November 2009.

[16] L. Holmwood. Behaviour driven infrastructure. http://www.slideshare.net/auxesis/behaviour-driven-infrastructure, January 2011.

[17] L. Holmwood. Cucumber-nagios. https://github.com/auxesis/cucumber-nagios/, November 2014.

[18] T. Holowaychuk. Should.js. https://github.com/shouldjs/should.js, November 2016.

[19] E. B. N. Laboratory. iperf3. http://software.es.net/iperf/, November 2016.

[20] P. Labs. Puppet - automate it. http://puppetlabs.com/puppet/puppet-open-source, October 2014.

[21] D. Libes. expect: Curing those uncontrollable fits of interaction. In USENIX Summer, pages 183{192, 1990.

[22] S. McQuerry. Interconnecting Cisco Network Devices, Part 1 (ICND1). Cisco Press, Indianapolis IN, USA, 2008.

[23] S. McQuerry. Interconnecting Cisco Network Devices, Part 2 (ICND2). Cisco Press, Indianapolis IN, USA, 2008.

[24] Saltstack. Saltstack - fast, scalable and exible systems management software for data center automation, cloud orchestration, server provisioning, configuration management and more. http://www.saltstack.com, October 2014.

[25] I. Shrubbery Networks. Rancid - really awesome new cisco config differ. http://www.shrubbery.net/rancid/, January 2014.

[26] J. J. Van der Ham, F. Dijkstra, F. Travostino, H. Andree, and C. T. de Laat. Using rdf to describe networks. Future Generation Computer Systems, 22(8):862{867, 2006.