THE NETWORK CERTIFICATION DESCRIPTION LANGUAGE

by

CODY HANSON

B.S., University of Wisconsin - Madison, 2011

A thesis submitted to the Graduate Faculty of the

University of Colorado at Colorado Springs

in partial fulfillment of the

requirements for the degree of

Master of Science

Department of Computer Science

2017

This thesis for the Master of Science degree by

Cody Hanson

has been approved for the

Department of Computer Science

by

_____

Dr. Kristen Walcott-Justice

_____

Dr. C. Edward Chow

_____

Dr. Jugal Kalita

_____

Date

Hanson, Cody (M.S., Computer Science)

The Network Certification Description Language

Thesis directed by Professor Kristen Walcott-Justice

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 thesis, 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.

Table of Contents

CHAPTER

List of Figures

List of Tables

CHAPTER I
INTRODUCTION

Modern 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 electro-magnetic 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. This is because the network designer must carefully specify every aspect of network construction, including routing protocols, IP subnetting and VLAN's, link bandwidth capacities, packet filtering access control lists and firewall rules, and more. 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 perhaps 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 emerge. Whatever the situation, keeping a network in top shape is a process which takes a large amount of energy and attention from talented IT and networking professionals.

Having a high amount of visibility into the state of a network, and the confidence that the information is accurate, can be a great advantage for any organization. Visibility is important because it enables businesses and organizations to be more effective in maintaining a highly available and 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 to do 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 does not currently exist an accessable tool that will allow a network or server administrator to systematically verify that their network and 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, perhaps until after a serious availability or security incident has already occurred. There are products 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, or a software engineer that has written 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 working alone.

The central thesis of this work is that given an expressive and easy to use DSL that can describe network connectivity, and the associated software to evaluate the specification, those in the networking field can benefit from the same design and test-

ing efficiencies that other engineers already enjoy. Imagine that a network engineer could create a precise functional specification for how each part of the network should behave. This would be a breakthrough in making network maintenance a more quantifiable and reproducible discipline. Given a tool that can automatically test the validity of a real world network port against its specification, the network team can be confident that what they have implemented is faithful to requirements. This would help to reduce costly delays and rework for networking projects, and would enable a network engineer to efficiently share domain expertise with a team.

This thesis presents the Network Certification Description Language (NETCDL), and the associated network certification workflow. NETCDL is a DSL designed to allow a user to expressively and fluently describe how a network should behave. A user of NETCDL expresses their requirements as a NETCDL specification document, which is a series of statements that describe the desired state of network connectivity. Once the user has crafted their specification, the document becomes input to a NETCDL certifier. The certifer is software that can parse the NETCDL language and verify the assertions in the document against a live network connection.

NETCDL language 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 state. Certifier design must ensure that certification is quick and reliable, important properties for successful use on a job site where many network links are being tested.

In this thesis the work presented:

- Develops a new technique for rigorous certification of network behavior
- Describes the NETCDL language, and how it enables certification of networks
- Shows that the language is simple to understand, and supports the majority of common use cases
- Discusses a reference design for a NETCDL certifier, and how certifiers can become important tools for network engineers

Another important facet of this work is that the systems described in this thesis 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.

CHAPTER II
PREVIOUS WORK

Computer networks are a relatively new and fast changing technology. Despite this fact, there has already been a large amount of research and effort put into developing techniques for diagnosing problems and improving operations. In this chapter, we will present several prior works that have influenced the creation of the NETCDL language and certification concept. These works include hardware description languages, software testing languages, existing networking DSL's, IT automation frameworks, and troubleshooting tools for networks.

Hardware Description Languages

Some of the most indispensable tooling used by hardware design and verification engineers are Hardware Description Languages (HDL's), such as Verilog and VHDL. These languages are purpose built to 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. Modern semiconductor designs are often so complex that the use of the synthesizer is the only way to meet all design constraints and produce a working system.

In Figure 1, it can be seen how high-level expressions can make hardware design and testing a much easier task. This Verilog code defines the transfer of an input signal into some 'storage' mechanism named 'q', at the rising edge (transition from low voltage to high voltage) of a signal called 'clock'. The engineer is not concerned with how this construct is exactly evaluated and laid out into hardware, merely that

the semantics of the design are maintained. This separation of concerns allows for an abstraction of the underlying circuit and for users of the HDL to focus on more intricate tasks, such as high-speed datapath design, or other delicate tasks that cannot be designed automatically by the synthesizer tools. It is this high level of abstraction that network engineers could also find useful while designing and maintaining their systems.

```
//Verilog code that defines a D Flip-flop.
//including an asynchronous reset signal
reg q;
always @(posedge clk or posedge reset)
  if(reset)
    q <= 0;
  else
    q <= d;
```

Figure 1: Verilog Hardware Description Language code

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 are often written in the same language as the software under test, and 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. Two major strategies for software testing are 'unit' testing, and 'integration' testing. Unit tests isolate software components and ensure that they operate correctly on their own. Unit tests are often useful because they run quickly and isolate regressions to a single software module. Integration testing takes the entire software system (or combinations of subsystems) as a single module and exercises it. This is a more faithful recreation of how the software will operate in a production setting, but integration testing can often be complex to maintain, and

slow to execute. 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 [23]. Cucumber is a Behavior Driven Development (BDD) framework that was originally written in the Ruby programming language, but has since been adapted to many other languages. It is 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. When implementing a new feature in software, the designer first writes the 'feature definition' in the Gherkin language (Cucumber's DSL), and then writes the tests which would pass if the code for the feature was already implemented. The tests will fail and be 'red' initially. Finally, the code is implemented to make the tests pass and appear 'green' ("like a cucumber", as the creators of the software put it). Cucumber is able to match these 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.

```
Feature: Accumulator addition
    In order to increase the value of the Accumulator
    As a user of the class
    I want to be able to add an integer to an Accumulator object

  Scenario: Add positive integer
  Given I have constructed a new Accumulator with the parameter 1
  When I add 5 to the Accumulator
  Then the value of the Accumulator should be 6
```

Figure 2: Cucumber Feature Definition

In Figure 2, an example of a Cucumber feature definition can be seen. One thing to notice is the ease with which anyone can read this specification. The common 'Given, When, Then' pattern is employed. This specifies the initial state of the situation, an

action taken by the user, and then asserts the expected behavior. Figure 3 shows the corresponding test step definition file that executes the tests.

```
#Test Step definitions for the Accumulator class
Given /^I have an initial value of (\d+)$/ do |arg1|
  $starting_value = arg1.to_i
end

Given /^I have constructed a new Accumulator with the parameter (\d+)$/ do
    |arg1|
  $accumulator = Accumulator.new(arg1.to_i)
end
When /^I construct a new Accumulator with parameter (\d+)$/ do |arg1|
  $accumulator = Accumulator.new(arg1.to_i)
end

Then /^the value of the Accumulator should be (\d+)$/ do |arg1|
  $accumulator.get_value == arg1.to_i
end

Given /^I have constructed an Accumulator with the parameter (\d+)$/ do
    |arg1|
  $accumulator = Accumulator.new(arg1.to_i)
end

When /^I construct a new Accumulator with the starting value$/ do
  $accumulator = Accumulator.new($starting_value.to_i)
end

Then /^the value of the Accumulator should be the starting value$/ do
  $accumulator.get_value == $starting_value.to_i
end

When /^I add (\d+) to the Accumulator$/ do |arg1|
  $accumulator.add(arg1.to_i)
end
```

Figure 3: Cucumber Test Step Definition

Another software testing tool that seeks to be expressive and easy to use is Should.js [27]. Should.js is an assertion library for the Javascript language that lets the user chain together english phrases that make testing clear and easy to understand. In Figure 4 an example test can be seen with expressive 'should' predicates. 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.

```
function exampleUnitTest() {
    var result = returnsThreeFives(); //return value: [5, 5, 5]
    result.should.be.instanceOf(Array).and.have.lengthOf(3);
    result[0].should.be.exactly(5);
    result[1].should.be.exactly(5);
    result[2].should.be.exactly(5);
}
```

Figure 4: Should.js assertions

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

Behind every software service or application there is a machine that must be configured properly for everything to run correctly. It could be a server in a company's datacenter, or increasingly common in modern times, a virtual machine that is rented from a service provider such as Amazon Web Services. Everything about these machines must be controlled and verified including software versions and security patch levels, cryptographic certificates, access control rules, and more. For a small number of machines, a System Administrator may manually log into each one and ensure that it is configured properly, but this approach does not scale and is not sustainable as the demands on the software grow.

Related to the idea of describing the 'state' and behavior of network connectivity is the idea of Behavior Driven Infrastructure (BDI) [25]. BDI is the notion that rather

than performing ad-hoc and incremental changes to get your computing and network equipment into the required state, you 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 you can describe how your machines and network devices should be configured. The benefit of this approach is that the framework takes care of the details of configuring your infrastructure for you, in order to conform to the specification that you wrote. Examples of these BDI frameworks include Chef [8], Puppet [32], SaltStack [41], and Ansible [5]. These have all seen wide use within the system administration, operations, and developer communities.

An example of the Ansible framework ensuring that a Redis database server is properly configured and running on a host can be seen In Figure 5. The simplicity of this excerpt shows the power of these IT automation frameworks and their ability to hide complexity and detailed implementation. This simple declarative notation replaces the many lines of complex scripting normally required to install and run the database server.

```
---
- name: Ensure redis is installed.
  yum: pkg=redis state=installed enablerepo=epel

- name: Ensure redis is running.
  service: name=redis state=started enabled=yes
```

Figure 5: Ansible Task - Redis Server

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 [24]. 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 that you can hold in your hand or be installed as a rack-mounted appliance in a data center. Others are software tools that are deployed on a general purpose computer and are available with either commercial or open-source licenses. Nagios [17], 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. One of the interesting things about Nagios is the 'plugin' architecture, which allows the user to write arbitrary code to perform monitoring checks against hosts on the network. Nagios checks are generally run from the Nagios server itself, and not at the edge of the network where users are connecting from, which can be a limitation. To combat this limitation, there exists the concept of a Nagios 'Agent', which is a piece of software that can be run on an arbitrary host. An example of this agent concept is the NSClient++ software [37]. These remote agents are noteworthy because it solves the problem of getting testing data from various connection points on the network, such as different VLAN's and IP subnets. They are also programmable, so that a user can design a test to fit their exact use case. By connecting at the access layer of the network (where end users are likely to connect), Nagios agents enable more realistic testing environments. Once again, this tool differs from NETCDL in the sense that the user is required to program their assertions into the tool using computer code, rather than making high level assertions about the desired behavior of the system.

The project cucumber-nagios [26] is noteable because it is a tool that 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 - Really Awesome New Cisco Config Differ

RANCID (the Really Awesome New Cisco Config Differ) [42] 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. This is extremely valuable to a network engineering team for three reasons:

- There is a record of the history of the configuration on a device, which can help with problem forensics.
- High visibility of configurations running in production settings discourages risky and untested changes.
- Members of the engineering team can be held accountable for changes that cause problems.

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 recorded history make sure that on-the-fly configuration changes don't catch them off guard. In Figure 6 the RANCID tool informs the user that a gigabit ethernet card was removed from slot 6 on a network device. As is common with "diff" tools, a line prefixed with "-" denotes a line removal, and a line prefixed with a "+" denotes an addition.

```
From: rancid <rancid@example.com>
To: rancid-example@example.com
Subject: example router config diffs
Precedence: bulk

Index: configs/dfw.example.com
===================================================================
retrieving revision 1.144
diff -u -4 -r1.144 dfw.example.com
@@ -57,14 +57,8 @@
  !Slot 2/MBUS: hvers 1.1
  !Slot 2/MBUS: software 01.36 (RAM) (ROM version is 01.33)
  !Slot 2/MBUS: 128 Mbytes DRAM, 16384 Kbytes SDRAM
  !
- !Slot 6: 1 Port Gigabit Ethernet
- !Slot 6/PCA: part 73-3302-03 rev C0 ver 3, serial CAB031216OL
- !Slot 6/PCA: hvers 1.1
- !Slot 6/MBUS: part 73-2146-07 rev B0 dev 0, serial CAB031112SB
- !Slot 6/MBUS: hvers 1.2
- !Slot 6/MBUS: software 01.36 (RAM) (ROM version is 01.33)
  !Slot 7: Route Processor
  !Slot 7/PCA: part 73-2170-03 rev B0 ver 3, serial CAB024901SI
  !Slot 7/PCA: hvers 1.4
  !Slot 7/MBUS: part 73-2146-06 rev A0 dev 0, serial CAB02060044
@@ -136,11 +130,8 @@
  boot system flash slot0:
  logging buffered 32768 debugging
  no logging console
  enable secret 5 1$73Y1$ab1133R
```

Figure 6: RANCID Router Configuration Diff Email

Expect Scripting

Network engineers often need to use command line interfaces that are interactive in nature when configuring and troubleshooting equipment. The software on the router or switch shows a prompt, and based on the context, the user's commands take different effects. 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 [33] is a program and 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. Figure 7 contains an example of using the Expect program to automate the configuration of several machines in an automated fashion.

Dedicated Hardware Tools

Some commercially available tools that are designed to fill the network verification niche include the LinkSprinter [13] and the OneTouch AT [14], both created by Fluke Networks (pictured in Figure 8). 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 at the touch of a button in a repeatable way. Available checks on the OneTouch AT include ping, port open, HTTP server availability, performance testing, and many others. The OneTouch AT also has the notion of a 'red/failing' test, and a 'green/passing' test. 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.

```
#!/bin/sh
# \
exec tclsh "$0" ${1+"$@"}
package require Expect
# Define variables
set username "cisco"
set password "cisco"

# Define all the devices to be configured
set devices "10.0.0.1 10.10.0.1 10.20.0.1"

# Main loop
foreach device $devices {
    # connect to a device
    spawn plink -telnet $device

    # log in programmatically
    expect "Username:"
    send "$username\r"
    expect "Password:"
    send "$password\r"

    # expect to be shown the root prompt
    expect "#"

    # Enter global configuration mode
    send "conf t\r"
    expect "(config)#"

    # Change the hostname of the machine
    send "hostname NEWHOSTNAME\r"
    expect "(config)#"

    # Return to privilege EXEC mode
    send "exit\r"
    expect "#"

    # Exit Telnet session
    send "exit\r"
    expect eof
}
```

Figure 7: Using Expect scripting to automate configuration of network devices

Figure 8: LinkSprinter (left) and OneTouch AT (right) by Fluke Networks

Existing Network and Service Description Languages

The Resource Description Framework (RDF) is a W3C Specification [12] that outlines a standard format for describing resources on the World Wide Web. At its core, the RDF is a graph data-model that seeks to enable the 'Semantic Web', a movement to make the Internet and its content machine-consumable by standardizing the way information is presented.

The Web Services Description Language (WSDL) [10] is based on the RDF, and is used as a specification for how a client should connect to a web service. WSDL manifests itself as an XML document that is to be used by software such as a Simple Object Access Protocol (SOAP) client [11]. The document describes the various functions available within the web service, as well as the inputs and output formats of the system.

The Network Description Language (NDL) [43] uses the RDF 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. The format that this language takes is an XML document, similar to WSDL. In figure 9 a simple network connection between a host and a network switch is described using NDL.

UserPC is connected to Switch1 via interfaces eth0 and port1 respectively. The authors of NDL also have published some more recent work on something called the Network Markup Language [19], which is an evolution of the original NDL.

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:ndl="http://www.science.uva.nl/research/air/ndl#">
    <ndl:Location rdf:about="#SampleNetwork">
        <ndl:name>SampleNetwork</ndl:name>
    </ndl:Location>
    <ndl:Device rdf:about="#UserPC">
        <ndl:name>UserPC</ndl:name>
        <ndl:locatedAt rdf:resource="#SampleNetwork"/>
        <ndl:hasInterface rdf:resource="#UserPC:eth0"/>
    </ndl:Device>
    <ndl:Interface rdf:about="#UserPC:eth0">
        <ndl:name>UserPC:eth0</ndl:name>
        <ndl:connectedTo rdf:resource="#Switch1:port1"/>
    </ndl:Interface>
    <ndl:Device rdf:about="#Switch1">
        <ndl:name>Switch1</ndl:name>
        <ndl:locatedAt rdf:resource="#SampleNetwork"/>
        <ndl:hasInterface rdf:resource="#Switch1:port1"/>
    </ndl:Device>
    <ndl:Interface rdf:about="#Switch1:port1">
        <ndl:name>Switch1:port1</ndl:name>
        <ndl:connectedTo rdf:resource="#UserPC:eth0"/>
    </ndl:Interface>
</rdf:RDF>
```

Figure 9: Network Description Language - Host/Switch connection. Adapted from [43]

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.

CHAPTER III
NETCDL LANGUAGE DESIGN

The NETCDL Language

Certifying that a network conforms to a detailed specification enables network owners to have confidence that their infrastucture will be available, performant, and secure. The copper and fiber-optic cabling that connects network machines has been certified for decades, and is considered an indispensible part of network installation. To date, certification of higher-order network functionality has been ad-hoc, disorganized, and often not considered a priority. The NETCDL Language was created to address this gap, and to formalize and standardize the various troubleshooting and verification techniques that good network engineers have always used.

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 because they add unecessary complexity.

The result is a language that looks as if it were not intended for a computer to consume at all (although it is!). NETCDL statements would not look out of place in an engineer's notebook, documenting ideas about how an upcoming project might be designed. Familiar English sentence structure means that as someone begins writing NETCDL, it is easy for them to express their thoughts without having to constantly refer back to the grammar reference. Another consequence of this is that NETCDL is easy to read and understand. The intent of a NETCDL statement should be immediately apparent to anyone familiar with the networking domain.

In the rest of this chapter, the idea of writing a specification for network connectivity is introduced, along with an introduction to the structure of the NETCDL Language. The different classes of NETCDL statements will be explored in detail and usage of each type of statement with be showcased using realistic examples. Discussion about why specific features of the language were designed, and how they help the user to write a good specification, will also be included. For the detailed and complete NETCDL Grammar definitions, please see the Appendix.

Writing a NETCDL Specification

NETCDL statements let users describe how their network should behave. Because the language grammar is deliberately simplistic, each statement asserts a single condition. When a user begins to write a specification for their network, they collect 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 Chapter IV.

The user can tailor their specification documents to their specific needs. One user might create one 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 or guest, 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 the '#' symbol, similar

to Python. Comment statements allow for extra information to be added to the specification document, and turn it into a 'self documenting' specification. 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.

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 especially useful in the case of security auditing where access restrictions need to be verified.

Define Statements

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

Define statements are an important part of NETCDL that help to improve readability and comprehension of the document. Host and Network define statements allow the user to specify aliases for commonly referenced network locations. Defines 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 10, '192.168.1.1' is being defined as 'MyRouter'. This means that the string 'MyRouter' can be used anywhere in the rest of the document, and it will be translated to '192.168.1.1' by the certifier software. Similarly 'MyPrinter' will be translated to '192.168.1.2', and 'HomeNetwork' will be translated to the network range

```
# 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 10: Example Usage of the Define Statement

'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. This is valuable because it helps the certification document to be easier to read at a glance. Well known names are easier to scan then complex IP addresses in most cases. Adding a layer of indirection also helps to future-proof the certification document against IP address changes. For instance, if the user references the IP address of MyRouter only in the define statement, when that address changes, the define statement can be updated easily, rather than every reference to the IP address.

VLAN and Link Statements

Certain misconfigurations are easy to fix, but can prevent all other network operations from proceeding. This is certainly true of VLAN and link misconfigurations. If these settings are misaligned with what the connecting device expects, performance of the link could suffer, or the transmission of traffic may not be possible at all. 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 is behaving 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, such as a user having too much access, or not enough. VLAN's are identified by an integer VLAN Id, either in the header of a packet or by special informational broadcasts sent from routers and switches.

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

Figure 11: Example Usage of Link statements

In Figure 11 we can see that the desired access VLAN ID is 500, and expected duplex and speed are specified as full and 1000 megabits per second. These statements are easily negated using the common 'should not' phrase, in place of the keyword 'should'.

DHCP Statements

Dynamic Host Configuration Protocol (DHCP) is one of the most prevalent mechanisms for distributing connectivity settings to a new client. 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 is coming 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.

```
# 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 12: Example Usage of DHCP Statements

In Figure 12 a common home networking scenario is verified, along with a useful host define statement.

DNS Statements

The Domain Name System (DNS) is a critical piece of infrastructure for most commmon network services. This is because end users typically remember and type well known names, such as 'amazon.com', instead of entering specific numeric addresses. If DNS is unavailable or misconfigured, even though the connection to the internet is established, most users would be unable to complete their tasks.

DNS translates common network and domain names into the underlying IP addresses that end up in the IP headers of packets. 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.

DNS statements begin with the 'domain name' keywords, 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 13.

```
# 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 13: Example Usage of DNS Statements

Ping Statements

Ping (ICMP Echo) is one of the most basic and commonly used techniques for checking connectivity between two hosts. The NETCDL Ping statement makes this check easy, both in the affirmative and negative cases. The Ping statement is very simple, with the user needing to specify only the ping target, and if the ping should succeed or not. A user might want to use the Ping statment 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 recieved from the target. If no response is recieved, or a message about the traffic being undeliverable or rejected, then the Ping is considered to have failed. While Ping is in some ways inferior to a Port Open test (ICMP Echo packets are often blocked on modern networks), it

remains a common tool that is often used by those in the networking field.

Example usage of the Ping statement is demonstrated in Figure 14.

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

Figure 14: Example Usage of DNS Statements

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 best 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 mean that the HTTP server listening is functioning correctly. Despite these caveats, the Port Open statement is an important step in verifying proper network application and firewall operation.

Port Open statements are simple to use, and are demonstrated in Figure 15 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 is also employed here.

```
#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 15: Example Usage of Port Open Statements

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 having poor throughput. This is especially harmful in the age of high bandwidth applications such as voice and video. That is why certifying that the appropriate bandwidth is available is an important part of network testing. It is true that available bandwidth can depend on network utilization, but router and switch misconfigurations can cause insufficient bandwidth to be available even on an idle network.

Bandwidth testing statements allow the user to specify an Iperf[31] 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 16

```
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 16: Example Usage of Iperf Statements

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 'should/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 17.

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 lis-

```
#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 17: Example Usage of File Fetch Statments

tening 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.

Traceroute Statements

One of the more complex parts of computer networking is routing between IP networks. There are many automated routing protocols which are commonly run on routers in order to dynamically build the forwarding tables. In a large environment, ensuring that routing is behaving optimally is an important part of network verification. A common tool used for this purpose is 'traceroute'. Traceroute programs probe and discover the path by which traffic to a given destination travels. By doing this, a network engineer can determine if traffic is flowing along the expected pathways or not.

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. Traceroutes for NETCDL are 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 traceroute hops, which can be hostnames, IP addresses, or host aliases. Example usage can be seen in Figure 18.

```
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 18: Example Usage of Traceroute Statement

Packet Capture Statements

One of the most powerful (and time consuming) techniques network engineers have in their arsenal is to capture packets from the wire and inspect them. This gives a raw and unfiltered look at exactly what was happening on a given link. 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?

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 on the wire. Assertions can be made about the presence of:

- IP packet source and destination fields, from or to specific hosts.
- 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 19.

```
#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 19: Example Usage of Packet Capture Statements

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, see Appendix A.

CHAPTER IV
NETCDL CERTIFIER DESIGN

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 20.

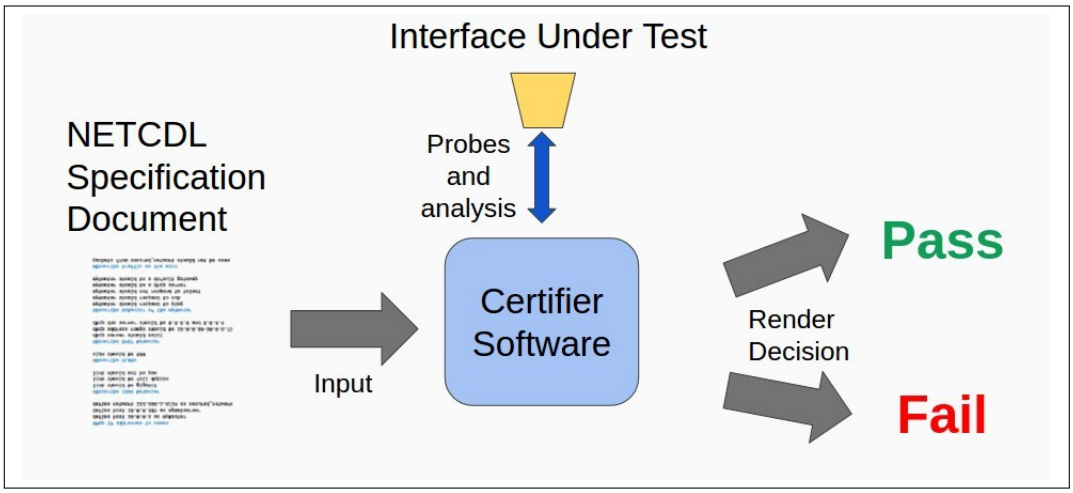

Figure 20: NETCDL Certification Workflow

A well designed certifier should have the following properties:

- It will minimize the time required to evaluate the specification.
- It will attempt to be as non-invasive as possible to the network under test (i.e. be a good steward of limited resources).

- It will provide helpful feedback to the user about network problems where detected.

The initial reference implementation of the NETCDL certifier presented in this thesis 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 (Appendix D).

In the next section the mechanics of a certifier will be examined at a high level. This chapter will also discuss techniques and design used to implement the first certifier, including approaches to dealing with implementation challenges. These challenges include parsing the NETCDL Language, dealing with low-level packet manipulation, and improving certification speed and reliability.

Certifier Operation

The certifier software has three main tasks:

- Parse and verify the input as valid NETCDL statements.
- For each assertion in the input document, render a Pass/Fail result in a time efficient manner.
- 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 begin to evaluating each NETCDL assertion. As can be seen in Figure 21, some tests can be considered 'active' while others would be considered '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 being 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 time efficient 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.

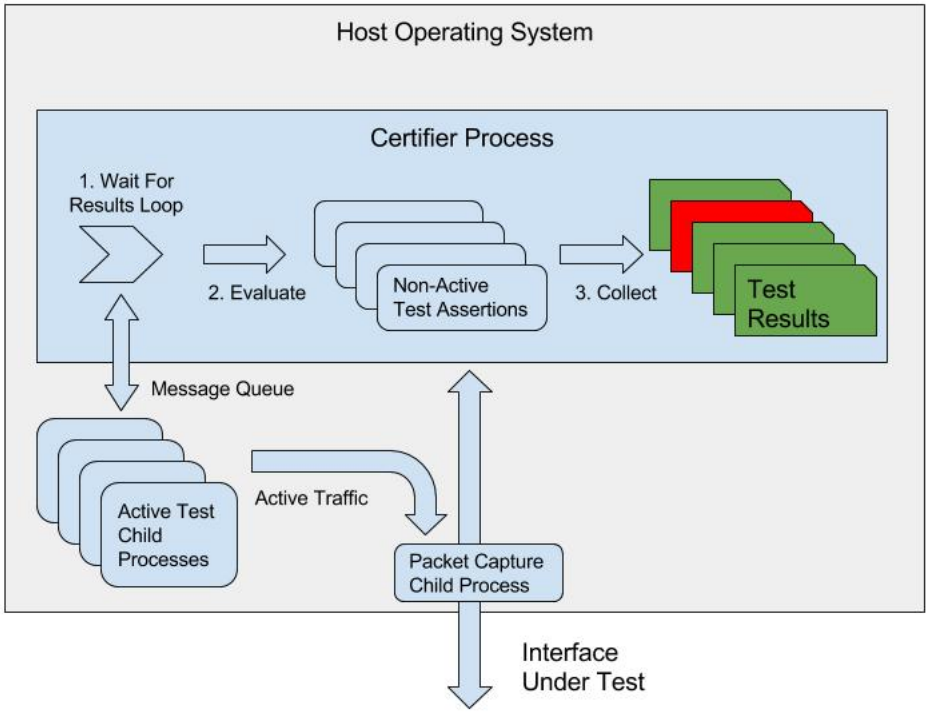

Figure 21: NETCDL Certifier Software Block Diagram

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 22. 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.

Figure 22: Example Certifier Command Line Output With Failing and Passing Statements

Another component of Figure 21 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.

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, in the use case of mass link certification.

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 implementa-

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

The software was implemented using the Python Programming Language [18]. It was chosen due to its emphasis on developer productivity, and for its rich open-source software library. While Python itself 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 version 14.04. For full software version notes, please see Appendix E.

Design Challenges: Low-level packet manipulation

NETCDL Certifiers must examine packets and frames for detailed header information, as well as conduct a wide variety of probing tasks using many protocols. Most software does not have a need to operate at the packet or frame level of the TCP/IP stack, which can make writing low-level networking code a challenge, especially in a high-level language such as Python. Thankfully there exists a very useful Python library for just this task called Scapy [6], which was leveraged in this implementation to make certain tasks easier.

Scapy is a somewhat unique tool in that it allows the user to craft arbitrarily complex IP packets and other traffic with minimal effort. It can be considered a DSL for creating and sending network traffic using the Python language. While many users of Scapy simply use the interactive command prompt, this implementation uses the Python classes directly for tasks such as capturing packets and frames, performing pings and traceroutes, and more. The ability for Scapy to make low level network programming exceptionally easy to wield for the Python programmer is what makes it a good fit for this project. An example usage of Scapy to form and send an ICMP Echo packet can be seen in Figure 23. Using Scapy to perform tests like Ping and Traceroute is preferable to invoking system commands, because it improves portability of the software.

```
ans, unans = scapy.sr(s.IP(dst='google.com') / s.ICMP())
```

Figure 23: Example Scapy Usage - ICMP Echo

What would require many lines of C code, Scapy can do in a single Python statement. As the capabilities and demands of the NETCDL language evolve, Scapy would easily be able to support new and advanced use cases for examining network traffic.

Design Challenges: Language Parsing

Because the NETCDL Language grammar is optimized for humans instead of computers, the software must carefully parse each statement into a programmatic representation for execution. Thankfully this is made easy in Python by a library called TextX [16].

TextX enables the easy creation of parsers for Domain Specific Languages. The user first defines their language grammar in the TextX syntax, and then can write a program that uses the generated Python classes. Using TextX to accomplish parsing in the reference certifier was a clean and elegant solution that helped automate the translation of the NETCDL grammar into executable Python code.

In Figure 24 the definition of the PortOpenStatement can be found as an illustration of TextX usage, The Python code in Figure 25 illustrates how the information from a parsed TextX statement is loaded into a NETCDL Test class. Each of the properties of the 'statement' variable correspond to the assignments in the TextX grammar definition. For example, in the PortOpenStatement, the value of 'protocol' is restricted to be wither 'TCP' or 'UDP', and the value will be assigned to a property called 'protocol' on the instantiated PortOpenStatement object. Similarly for 'port', which matches any valid integer. If TextX is unable to match an input statement, an exception is raised and the user of the Certifier is alerted that their NETCDL statements may have invalid syntax.

```
NonWhiteSpace:
    /[^\s\n,\[\]]+/
;

Should:
 /should(\s+not)?/
;

ReachabilityClause:
 host=NonWhiteSpace should=Should 'be reachable'
;

PortOpenStatement:
 reachable=ReachabilityClause 'on' protocol=/TCP|UDP/ 'port' port=INT
;
```

Figure 24: TextX Definition for PortOpenStatement

```
class PortOpenTest(ActiveTest):
    def __init__(self, statement):
        self.port = statement.port
        self.protocol = statement.protocol
        self.host = str(DefineMap.lookup_host(statement.reachable.host))
        self.should_pass = statement.reachable.should == 'should'
        self.timeout = 10
        should_str = "should" if self.should_pass else "should not"
        self.display_str = "{0} {1} be reachable on {2} port
            {3}".format(self.host, should_str, self.protocol,
                                                        self.port)
        super(PortOpenTest, self).__init__(statement)
```

Figure 25: Partial Class Definition for PortOpenStatement

A full TextX representation of the NETCDL grammar was created, and is the sole text parsing engine for the reference certifier. The rest of the software was then written against the generated TextX classes. To see the full TextX grammar for NETCDL, refer to Appendix C.

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 [21] licensed under the GNU Public License version 3.0.

CHAPTER V
EVALUATION

The work presented in this thesis was evaluated with respect to three things:

- NETCDL Grammar Complexity
- NETCDL Language Expressiveness
- Reference Certifier Software Quality

A key goal of NETCDL was to be an approachable and simple to learn DSL. One way to objectively measure these properties of a language 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. In this thesis 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.

Because the reference implementation of NETCDL was intended to be a guide for future implementers, it is important that the software be of high quality, and sound design. In software engineering, these goals are often accomplished by minimizing the amount of code needing to be written and maintained, and pairing the software with an automated test suite. Another important software property is extensibility, or how easily the software may be modified to support future use cases. The reference implementation will be examined and evaluated with respect to these software quality goals. Throughought the rest of this chapter, each of these evaluation vectors will be examined.

Evaluation of Language Complexity

Some computer languages are notoriously difficult to learn and understand, while others have a reputation for being easy to pick up and master. This can be attributed to the fact that the nature of a language's grammar defines the amount of keywords, structures, idioms, and syntax that need to be learned. Some language grammars simply have fewer intricacies and thus make them more likely to be adopted by users. A language like NETCDL that seeks to be simpler to use than the alternatives would benefit from having a simpler grammar.

Works in Grammar Engineering [4] show how we can take an objective approach to designing and evaluating computer languages. Taking a quantititative approach to analyzing the NETCDL language grammar and comparing it to other well known computer languages is a good way to get an idea of 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) [22] 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 Appendix B.

An initial depiction of grammar structure as evaluated by SdfMetz can be observed in Figure 26. 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.

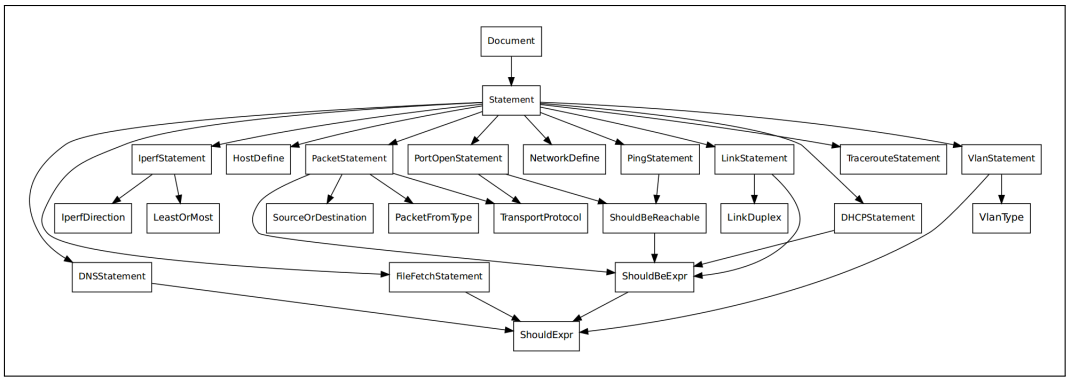

Figure 26: NETCDL Grammar Diagram - Generated by SdfMetz

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 Table 1. 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 27 contains the raw output of the SdfMetz tool, including the abbreviations and descriptions of each metric.

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 - Sometimes referred to as Halstead Effort [39]	Computation that combines frequency of occurance for operators and operands into a single number. Useful for comparing complexity between grammars of different sizes.

Table 1: Selected Grammar Complexity Metrics and Meanings

```
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
```

Figure 27: NETCDL Grammar Complexity Metrics - Gathered from SdfMetz

In order 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 computer languages. Also present in the SdfMetz research is a data set for other grammars that were ex-

amined 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, included in Figure 28, was the standard of comparison for the NETCDL grammar metrics.

Language	Origin	Type	TERM	VAR	PROD	MCC	NPATH	E	TIMPi	TIMP	LEV	CLEV	NSLEV	DEP	HEI
XPath	GCC	Bison	47	28	86	58	86	7.8	2.3	93.7	9	32.1	1	20	4
BibTeX	GB	SDF	20	6	16	21	32	8.9	20.0	100.0	6	100.0	0	1	6
MatLab	DK	Bison	44	34	92	58	92	13.0	4.0	62.3	15	44.1	2	16	6
Fortran 77	GB	SDF	41	16	41	32	58	18.9	4.8	42.9	15	93.8	1	2	7
C	P&M	BNF	86	65	-	149	-	51	-	64.1	22	33.8	3	38	13
C	ALR	ANTLR	122	67	67	197	285	59.4	1.4	94.9	27	40.9	3	37	13
Java 1.3	ALR	ANTLR	104	68	68	171	263	74.0	2.1	86.1	24	35.3	1	45	9
EcmaScript	SD	DMS	144	90	344	254	344	80.8	1.6	83.7	29	32.2	3	57	10
Ada	HF	Bison	93	238	458	220	458	87.4	0.8	63.3	156	65.5	4	42	24
Java	P&M	BNF	100	149	-	213	-	95	-	32.7	89	59.7	4	33	23
PHP 5	SD	DMS	159	112	418	306	418	96.5	1.6	76.1	55	49.1	2	37	15
Java 1.5	ALR	ANTLR	108	102	102	245	450	132.2	1.9	93.1	24	23.5	2	73	9
SDL	GB	SDF	89	91	174	170	273	132.5	1.1	39.8	76	83.5	2	13	14
Java	SD	DMS	99	144	460	316	460	137.9	1.3	93.9	41	28.5	2	87	17
C	GB	SDF	102	24	148	162	196	146.4	5.9	75.9	15	62.5	1	10	10
C++	P&M	BNF	116	141	-	368	-	173.0	-	85.8	21	14.9	1	121	4
EcmaScript	ALR	ANTLR	185	198	198	406	612	175.6	0.6	43.5	104	52.5	4	89	10
DB2	SIG	SDF	214	98	292	311	478	185.0	1.0	42.6	69	71.1	1	29	16
C#	ALR	ANTLR	133	219	219	408	660	202.3	0.7	55.6	159	72.9	4	30	27
PL/SQL	ALR	ANTLR	196	157	157	449	4683	215.5	1.5	45.6	119	76.3	6	15	21
C#	P&M	BNF	138	245	-	466	-	228	-	29.7	159	64.9	5	44	28
VDM-SL	TA	SDF	143	71	227	232	316	247.6	2.8	78.7	35	49.3	3	27	13
Java 1.5	GB	SDF	105	122	327	318	616	265.4	2.1	79.8	53	43.4	2	63	10
Ada	ALR	ANTLR	99	209	209	326	537	295.4	1.1	56.9	143	68.4	5	36	19
VB.net	JV	SDF	170	206	446	466	1554	390.0	1.0	42.9	154	74.8	6	25	26
Veriog 2001	SD	DMS	232	488	1248	760	1248	455.7	0.5	46.1	266	54.5	10	117	19
SDL	ALR	ANTLR	174	461	461	822	2043	708.9	0.6	35.4	357	77.4	6	43	28
PL/SQL	SIG	SDF	456	499	1094	888	1564	710.9	0.3	24.5	434	87.0	2	38	29
Cobol	GB	SDF	479	774	1330	1122	2194	909.4	0.2	22.2	627	81.2	7	70	26
C++	SD	DMS	173	436	2122	1686	2122	1300.6	0.7	96.3	42	9.6	2	393	10

GB = The online Grammar Base [15].
SIG = Software Improvement Group [16].
SD = Semantic Designs [14].
GCC = GNU C Compiler [13].
ALR = ANTLR web site [12].
JV = Joost Visser.
TA = Tiago Alves.
DL = Danny Luk.
HF = Herman Fischer.
P&M = Power and Malloy [3].

Figure 28: Sampled grammars, with complexity metrics [3]

Table 2 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 2.

Metric Name	NETCDL Value	Ranking out of 31 (lower is better)	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

Table 2: NETCDL Grammar Performance vs SdfMetz Grammar Dataset

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 can let us conclude that the NETCDL language 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 recieved as an easy to understand language.

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. In order 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.

Two main bodies of networking expertise were referenced while gathering use cases. The first was Interconnecting Cisco Network Devices, parts 1 and 2 [35][36]. 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, not just products offered by Cisco. The second text that was referenced was the Network Maintenance and Troubleshooting Guide [2] by Neal Allen. Mr. Allen was a long-time member of the Technical Assistance Center (TAC) at Fluke Networks, and his book represents decades of expertise in the network troubleshooting space.

Use Case	**Description**	**Sources**	**NETCDL Support**
Ping	ICMP Echo, verifies layer 3 connectivity	[2][35]	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.	[35]	Yes
FTP Get	Verifies that an FTP server is up and running and can serve a file.	[36]	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.	[35]	Yes
Traceroute	Verifies correct order of layer 3 hops, using packets with increasing TTL	[36][2]	Yes
VLAN Trunking	Verifies that port on a router or switch is tagging vlans and acting as a “Trunk”.	[36]	No
Access Vlan ID	Verifies that a port belongs to the correct access vlan	[36]	No
DHCP Server testing	Verifies that DHCP services are operating properly	[35] [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	[36][2]	Yes
Link Duplex	Verifies that the network interface negotiates to the correct duplex (full or half)	[36][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.	[36]	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 physical characteristics	[2]	No

Table 3: Common Network Troubleshooting Use Cases

Table 3 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 in row one of the table is enabled by the NETCDL Ping statement and can be carried out by the certifier.

By examining column 4 of Table 3 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 [15]. Overall, NETCDL was able to cover a majority of the important networking tasks that network engineers use in their daily work.

Evaluation of the Reference Certifier

Having a quality reference implementation for a new project is important because it can serve as a guide for future implementers. While software quality can be assessed in many ways, a few established evaluation methods were chosen for this research. One of these techniques is measuring code coverage via unit tests. It is generally understood that a higher code coverage percentage correlates to higher quality software. Another way to improve software quality is to simply have less of it. By using efficient languages and design patterns the number of lines of code can be minimized, which in turn helps to reduce the ongoing maintenance and development cost. Finally, steps

that were taken to improve extensibility and enable future work will be discussed.

An automated unit test suite was created to accompany the reference certifier, using the PyTest framework [30]. PyTest automatically generates a coverage report while running the unit and integration tests defined. In Figure 29, an example unit test can be seen. Best practices for unit testing include testing a single piece of functionality at a time, such as a function, class, or module. Assertions are made about the expected values of properties or function return values. If an assertion is violated, the test is marked as failed. It is common to use techiques such as 'mocking' to isolate code from its dependencies. This is done in order to keep the subject of the test focused on the module in question, but also is useful to avoid having to make expensive operations while testing, such as running a query against a database, or conducting network traffic. Instead a mocked dependency can simply return fake data that can exercise the software unit under test.

```
def test_link_control_dhcp(mocker):
    #Use a mock to stub out dependencies that are not under test.
    mocker.patch('scapy.all.sendp')

    #construct module under test
    lc = LinkControl.LinkControl('eth0')

    #Make value assertions
    assert lc.iface == 'eth0'

    #invoke function under test
    lc.dhcp()

    #Make assertions about the mocked dependencies
    scapy.all.sendp.assert_called()
```

Figure 29: Example Python Test using pytest, with mocking and assertions

```
Code Coverage Report Generated by py.test
Name                      Stmts   Miss  Cover
---------------------------------------------
netcdl/ActiveTest.py         14      4    71%
netcdl/Certifier.py          97     22    77%
netcdl/DHCPTest.py           69      0   100%
netcdl/DNSTest.py            30      4    87%
netcdl/DefineMap.py          10      0   100%
netcdl/EthtoolParser.py      16      0   100%
netcdl/FileFetchTest.py      52      0   100%
netcdl/FrameTypeTest.py      17      0   100%
netcdl/IperfTest.py          37      0   100%
netcdl/LinkControl.py         7      0   100%
netcdl/LinkDuplexTest.py     18      0   100%
netcdl/LinkSpeedTest.py      19      0   100%
netcdl/PacketCapture.py      25      0   100%
netcdl/PacketFromTest.py     28      0   100%
netcdl/PacketPortTest.py     29      0   100%
netcdl/PacketTypeTest.py     20      0   100%
netcdl/PingTest.py           19      0   100%
netcdl/PortOpenTest.py       45     11    76%
netcdl/Report.py             18      0   100%
netcdl/Test.py               16      0   100%
netcdl/TestResult.py         11      0   100%
netcdl/TraceRouteTest.py     25      0   100%
netcdl/__init__.py            0      0   100%
netcdl/netcdl.py             32     16    50%
---------------------------------------------
TOTAL                       654     57    91%
```

Figure 30: Reference Certifier Code Coverage Report

It can be seen from the report in Figure 30 that 91% statement coverage was obtained. A high coverage percent is a valuable asset that can help to defend against regressions as the software evolves. This is especially true of an interpreted language such as Python, because there is no compiler that can catch mistakes.

Also visible from Figure 30 is that the size of the software is a modest 654 Python statements. The compact nature of the implementation should prove approachable for others who wish to study the techniques and patterns presented. The leveraging of powerful frameworks such as TextX and Scapy was key to minimizing the amount of code that needed to be developed.

The architecture of the certifier software is highly extensible, in line with the stated implementation goals. This was accomplished by using good Object Oriented Programming practices, and a scalable multiprocessing architecture which makes it easy to plug in new types of network assertions and tests.

An area where the certifier software is limited (besides lacking the hardware support needed for certain advanced tests), is that the output provided to the user is rather basic, and merely denotes a pass or a fail for a given assertion. There is no indication about why the assertion failed. This is definitely a capability that would be required before commercial adoption of a NETCDL Certifier. Failure diagnostics would help to make engineers and troubleshooters more efficient as they certify each link. The reference implementation presented in this thesis provides a good foundation for and demonstration of the NETCDL concept. Through futher development and refinement, certifiers could be considered ready for field deployment in a large network installation.

CHAPTER VI
DISCUSSION

This chapter will explore and expand on the ideas already presented in the thesis. Threats to validity, key assumptions, and limitations will be discussed, as well as thoughts on how NETCDL can evolve and expand into new use cases.

Threats to Validity

One of the key assumptions from the Evaluation chapter is that lower grammar complexity (as measured by a grammar analyzer) correlates to an qualitatively simpler language, as experienced by real people. This was attempted to be addressed by giving a table of comparable languges which had similar values for grammar metrics. With this table, the reader may be able to bring some qualitative judgement to bear on the metrics.

Another key assumption of this work is that network engineers will be willing to maintain specification documents that describe their networks. In order for this to be true, the value of certification and having a formal specification must outweigh the maintenance costs of the upkeep of the documentation.

Finally, it is also assumed that network automation does not advance to the point such that mistakes are no longer prevalent. In the world we currently live in, where much network configuration is still done manually, or only rudimentary automation is used, the certification properties of NETCDL remain valueable. In the potential future scenario where all chance for faults has been automated out of existence, then NETCDL would serve less of a purpose. However, it is hard to say if this perfect automation will ever be a reality or not.

Language and Software Extensions

An important facet of computer language design is the ability for users to extend and enhance the features of the language to meet their needs. These new features

could be things like custom software modules designed to examine something about the network, new kinds of assertions, or enhanced language macros. This could be accomplished by a plugin or package system, similar to those used by LaTeXand most other programming languages. It is also likely that the NETCDL system could be extended through some official channel, such as a form of open governance that oversees similar open-source projects.

It is also possible that in the future the reference implementation of the NETCDL Certifier will be augmented by various other implementations, all supporting the same language standard. An example of a situation like this is with the Python programming language, and the various Python interpreters available such as CPython (the default implementation), IronPython[9], Jython[29], and PyPy[40]. The benefit of having multiple implementations is that it enables competition which can drive improvements, or enable an implementation to be tailored to specific use cases or requirements.

Improved NETCDL Language Tools

There are several things that could be done to make NETCDL easier to user for the writer and reader. One common computer language feature is to support syntax highlighting in popular text editors and viewers. This helps the user to visually differentiate important parts of language statements and keywords. Another common feature of modern computer languages is automatic syntax check software (sometimes known as "linting" software). This would help the writer to know if they are writing invalid statements that are not allowed according to the NETCDL language grammar. Examples of software that do this include JSHint[38] and Pylint[34].

Advanced Hardware Capabilities

Due to the use of commodity PC hardware during the development of NETCDL, certain physical layer and high-speed use cases were not able to be demonstrated. Two

noteable advances that could be made by using more advanced hardware would be Power over Ethernet detection, and full line-rate capture at high speeds (10/40/100 gigabits per second). Power Over Ethernet (POE) modes are able to be detected and reported to software by Ethernet interfaces designed to accept power from them. This would lead to the design of a new "Power Over Ethernet Voltage" assertion. POE is an important quantity to verify because many common devices require it to function, such as VoIP phones, security cameras, and Wifi access points. The addition of line-rate capture technology would ensure that no packets or frames are dropped during the packet capturing portion of a certification. At high bitrates it is possible that portions of the traffic would be missing, which could affect the certification. This is because most commodity network cards found in computers do not have the high-speed digital circuitry required to capture every single frame on a saturated link. Servers and tools that can do this often utilize custom ASIC's or FPGA's for high speed capture. The confidence that no traffic was dropped by the measurement device is a useful piece of information for a troubleshooter to have.

Wireless Protocols

Wireless technologies (primarily IEEE 802.11, or Wi-Fi) are a critical method of connecting that was not addressed in this initial implementation. Wireless networks are plagued by many of the same configuration and performance issues that wired LAN's are, and could similarly benefit from NETCDL statements that would help to verify correct behavior.

Some items to verify include:

- Client association with the correct base station
- Usage of the appropriate channels
- Testing for presence of various WLAN frame types (beacons, de-auths)
- Check keys to ensure correctness, or compliance with key revocation lists

- Check for hidden or unauthorized SSID's

Expanded Real-world Trials

It is said that no battle plan survives contact with the enemy, and in the case of NETCDL the enemy is the entropy and dynamic nature of real world networks. Due to the incalculable number of network configurations, hardware platforms, and software versions available, it is impossible to test and prepare for every situation. An advanced test for the usefulness of NETCDL will need to be conducted by real users performing their daily tasks with the language, and providing feedback to the maintainers. Even well established computer languages such as C and C++ have undergone many improvements and extensions, over a period of decades.

A few vectors of testing that will need to be conducted in the real world include:

- User ergonomics - Does it make users more productive?
- Certifying networks at scale - Is it too costly or slow?
- Certifer software resilience in diverse networks - are unknown conditions handled gracefully?

While the objective evaluation methods we have employed in this thesis are useful and gave hopeful results, the true measure of success for a technology hinges on whether users enjoy using it. This aspect of NETCDL will remain unknown until it is in the hands of users in the wild. All materials presented in this thesis will be published under an open-source license on netcdl.org [20].

The other two items of real-world testing that we are interested in relate to performance of the certifier software and the underlying technique in a production network setting. It is possible that running the required checks for NETCDL certification could be too slow or disruptive to run for large scale deployments, where there are many devices to verify connectivity to, or thousands of ports to verify. The other

concern is that certifier software will need to handle diverse network hardware and software configurations. Each network machine has quirks of its own, inherent to its implementation, despite the attempt to follow documented standards. These differences in implementation or interpretation of standards has always been a challenge in computer networking, and could influence the outcome of certification. Despite these concerns, we contend that it is these large and important networks that could benefit the greatest from NETCDL certification.

CHAPTER VII
CONCLUSION AND FUTURE WORK

This thesis proposed the new idea of network certification above the physical layer. In conjunction with this, an innovative domain specific language was developed and presented, the Network Certification Description Language (NETCDL)[1]. NETCDL took inspiration from other forms of automation in the computer world, as well as software systems that are well-known for their ease of use. A guide to the language was presented that showcased the ease with which it can be used to make useful assertions about networks, and help network engineers to formalize their approach to troubleshooting problems. In addition to the definition of the language, a reference design for a NETCDL certifier was presented, along with guidelines and patterns for future implementers to leverage.

In the evaluation of the work, the language was shown to meet the goals of simplicity and expressiveness. Objective measures of the language grammar complexity compared favorably with the grammars of other well known computer languages and NETCDL consistently placed in the top five out of thirty-one language grammars in the sample. These results support the claim that NETCDL was designed to be easy for humans to read and write by minimizing relevant metrics such as McCabe's Cyclomatic Complexity. 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. The certifier design principles discussed were shown to support the goals of being performant and extensible.

There is much future work possible in the domain of formalized network testing, and network certification tools. A few top priorities for future NETCDL research would be to collect qualitative feedback from real users in order to support the quantitative claims that the language is approachable, easy to use, and ultimately useful.

[1]The project homepage for NETCDL can be found at netcdl.org[20]

Once these base assumptions are verified, targeted improvements to the capabilities of the NETCDL language can be proposed, focusing on expanding must-have use cases, and building in extra features that ease pain points for users. 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] Neal Allen. *Network Maintenance and Troubleshooting Guide: Field-tested Solutions for Everyday Problems, 2nd Edition*. Addison Wesley, Upper Saddle River, NJ, USA, 2010.

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

[4] Tiago L Alves and Joost 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] Philippe Biondi. Scapy. http://www.secdev.org/projects/scapy/, November 2016.

[7] S. Bradner. Key words for use in rfcs to indicate requirement levels. RFC 2119, IETF, March 1997. URL https://www.ietf.org/rfc/rfc2119.txt.

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

[9] Iron Python Community. Ironpython: The python programming language for the .net framework. http://ironpython.net/, November 2016.

[10] World Wide Web Consortium. Web services description language - w3c spec. http://www.w3.org/TR/2001/NOTE-wsdl-20010315/, March 2001.

[11] World Wide Web Consortium. Soap - w3c spec. http://www.w3.org/TR/soap12-part1/, April 2007.

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

[13] Fluke Corporation. Linksprinter. http://www.linksprinter.com/, October 2014.

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

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

[16] Igor Dejanovi. Textx. http://igordejanovic.net/textX/, November 2016.

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

[18] The Python Software Foundation. The python programming language. https://www.python.org/, November 2016.

[19] Grid Final Draft GFD. Network markup language base schema version. *Network*, 2013.

[20] Cody Hanson. The network certification description language. http://netcdl.org/, November 2016.

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

[22] Jan Heering, Paul Robert Hendrik Hendriks, Paul Klint, and Jan Rekers. The syntax definition formalism sdfreference manual. *ACM Sigplan Notices*, 24(11): 43–75, 1989.

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

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

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

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

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

[28] ISO/IEC. Iso/iec 14977 - information technology - syntactic metalanguage - extended bnf. http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip, December 1996.

[29] Barry Warsaw Jim Hugunin. Jython: Python for the java platform. http://www.jython.org/, November 2016.

[30] Holger Krekel. pytest: helps you write better programs. http://doc.pytest.org/en/latest/, November 2016.

[31] ESnet/Lawrence Berkely National Laboratory. iperf3. http://software.es.net/iperf/, November 2016.

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

[33] Don Libes. expect: Curing those uncontrollable fits of interaction. In *USENIX Summer*, pages 183–192, 1990.

[34] Logilab. Pylint: Star your python code! https://www.pylint.org/, November 2016.

[35] Steve McQuerry. *Interconnecting Cisco Network Devices, Part 1 (ICND1)*. Cisco Press, Indianapolis IN, USA, 2008.

[36] Steve McQuerry. *Interconnecting Cisco Network Devices, Part 2 (ICND2)*. Cisco Press, Indianapolis IN, USA, 2008.

[37] Michael Medin. Nsclient++. http://www.nsclient.org/about/, November 2014.

[38] Rick Waldron Caitlin Potter Mike Pennisi Luke Page. Jshint, a static code analysis tool for javascript. http://jshint.com/, November 2016.

[39] James F Power and Brian A Malloy. A metrics suite for grammar-based software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6): 405–426, 2004.

[40] Armin Rigo. Pypy. http://pypy.org/, November 2016.

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

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

[43] Jeroen J Van der Ham, Freek Dijkstra, Franco Travostino, Hubertus Andree, and Cees TAM de Laat. Using rdf to describe networks. *Future Generation Computer Systems*, 22(8):862–867, 2006.

APPENDIX A. NETCDL Grammar - EBNF

Grammar definition using the Extended Backus-Naur Format[28]

```
#Document Structure
<document> ::= <line> | <line> <document>
<line> ::= <definition> | <statement> | <blank-line> | <comment>
<comment> ::= <whitespace> "#.*"
<whitespace> ::= " " <whitespace> | "\t" <whitespace> | ""
<blank-line> ::= <whitespace> "\n"
<nonwhitespace> :: = <nonwhitespace><nonwhitespace> | "a-z" | "A-Z" |
    ".~!@$%^&*()_-"
<statement> ::= <dhcp-statement> | <link-statement> | <packet-statement> |
    <port-open-statement> |
              <ping-statement> | <traceroute-statement> | <vlan-statement>
                  | <dns-statement> |
              <iperf-statement> | <file-fetch-statement>

#Host and Network Defines
<definition> ::= "define" <definition-type> <nonwhitespace> "as"
    <nonwhitespace>
<definition-type> :: = "host" | "network"

#DHCP Statements
<dhcp-statement> ::= <dhcp-dns-statement> | <dhcp-addr-range-statement> |
    <dhcp-server-statement>
<dhcp-dns-statement> ::= "dhcp" "dns" "server" <should-expr> "be" <IP-addr>
<dhcp-server-statement> ::= "dhcp" "server" <should-expr> "be" <IP-addr>
<dhcp-addr-range-statement> ::= "dhcp" "network" <should-expr> "be"
    <IP-addr>

#Link Statements
<link-statement> ::= <link-speed> | <link-duplex>
<link-speed> ::= "link speed" <should-expr> "be" <speed>
<link-speed> ::= "link duplex" <should-expr> "be" <duplex>
<duplex> ::= "full" | "half"
<speed> ::= "10Mb" | "100Mb" | "1000Mb" | "1Gb"

<port-open-statement> ::= <reachable> "on" <transport> "port" <port-number>

<ping-statement> ::= <reachable> "by ping"

<traceroute-statement> ::= "traceroute to" <nonwhitespace> "should
    traverse [" <routers> "]"
<routers> ::= <routers><nonwhitespace> | ""

<vlan-statement> ::= "access vlan" <should-expr> "be" <integer>

<dns-statement> ::= "domain name" <nonwhitespace> <should-expr> "resolve"
    <resolve-target> "using server" <nonwhitespace>
```

```
<resolve-target> ::= "" | "to" <nonwhitespace>

<file-fetch-statement> ::= <fetch-protocol> "server at" <nonwhitespace>
   <should-expr> "serve" <nonwhitespace> <fetch-port>
<fetch-port> ::= "" | "on port" <port-number>
<fetch-protocol> ::= "http" | "tftp" | "ftp"
<on-port> ::= "on" <transport> port <port-number>
<transport> ::= "tcp" | "udp"
<port-number> ::= 0 - 65535


#iperf statement
<iperf-statement> ::= "iperf" <iperf-direction> <nonwhitespace> "should be
   at" <iperf-comparison> <nonwhitespace>
<iperf-direction> ::= "upload to" | "download from"
<iperf-comparison ::= "most" | "least"


#Packet Statements
<packet-statement> ::= <packet-from-statement> | <packet-port-statement> |
   <packet-type-statement> | <frame-type-statement>
<packet-from-statement> ::= "packets from" <packet-from-type>
   <nonwhitespace> <should-expr> "be seen"
<packet-from-type> ::= "host" | "network"
<packet-port-statement> ::= "packets with" <protocol> <packet-direction>
   "port" <port-number> <should-expr> "be seen"
<packet-type-statement> ::= "packets with type" <packet-from-type>
   <nonwhitespace> <should-expr> "be seen"
<frame-type-statement> ::= "frames with ethertype" <nonwhitespace>
   <should-expr> "be seen"

#expressions
<should-expr> ::= "should" | "should not"
<reachable> ::= <nonwhitespace> <should-expr> "be reachable"

#primitives
<integer> ::= 0|1|2|3|4|5|6|8|9|0|<integer><integer>
```

APPENDIX B. NETCDL Grammar - SDF

```
%%%
%%% SDF grammar for NETCDL language
%%%
%%%  Grammar: NETCDL
%%%  Version: 1
%%%
%%% Description:
%%%  Network Certification Description Language
%%%  http://netcdl.org


definition

module Main
exports
sorts Identifier Document
lexical syntax

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Reusable Elements, keywords, and Formatting
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 [\ \t]+ -> WS
 [\ \t\n] -> LAYOUT
 "#" ~[\n]* [\n] -> Comment
 Comment -> LAYOUT

 [A-Za-z0-9\.\_\-]+ -> Identifier
 [0-9]+ -> Number
 "0x" [0-9a-fA-f]+ -> HexType

 "on" -> On
 "port" -> Port
 "on port" -> OnPort
 "serve" -> Serve
 "server at" -> ServerAt
 "as" -> As
 "should" -> Should
 "not" -> Not
 "be" -> Be
 "reachable" -> Reachable
 "define" -> Define
 "host" -> Host
 "network" -> Network
 "TCP" -> TCP
 "UDP" -> UDP
 "source" -> Source
 "destination" -> Destination
```

```
  "http" ->FileFetchProtocol
  "ftp" -> FileFetchProtocol
  "tftp" -> FileFetchProtocol

  "access" -> VlanAccess
  "vlan" -> Vlan

  "server" -> Server
  "gateway" -> Gateway

  "dhcp" -> DHCP
  "dns" -> DNS

  "least" -> Least
  "most" -> Most

  "domain name" -> DomainName
  "using server" -> UsingServer
  "to" -> To

  "upload to" -> UploadTo
  "download from" -> Download From
  "should be at" -> ShouldAlwaysBeAt

  "iperf" -> Iperf
  Number+ "Kbps" -> IperfBitrate

  [0-9]+ [MGK] "b" -> LinkSpeed

  "frames" -> Frames
  "packets" -> Packets
  "with" -> With
  "from" -> From
  "ethertype" -> Ethertype
  "type" -> PacketType
  "seen" -> Seen

  "by ping" -> ByPing

 "traceroute to" -> TracerouteTo
 "should traverse" -> ShouldTraverse
 "[" -> OpenSquareBracket
 "]" -> CloseSquareBracket


context-free restrictions
%%% Makes layout optional
 LAYOUT? -/- [\ \t\n]

context-free syntax
```

```
 Should Not? -> ShouldExpr
 ShouldExpr Be -> ShouldBeExpr

 Identifier ShouldBeExpr Reachable -> ShouldBeReachable

 TCP | UDP -> TransportProtocol
 Source | Destination -> SourceOrDestination

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Define Statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


 Define Network Identifier As Identifier -> NetworkDefine
 Define Host Identifier As Identifier -> HostDefine

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% DHCP Statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 DHCP DNS ShouldBeExpr Identifier -> DHCPStatement
 DHCP Server ShouldBeExpr Identifier -> DHCPStatement
 DHCP Gateway ShouldBeExpr Identifier -> DHCPStatement
 DHCP Network ShouldBeExpr Identifier -> DHCPStatement


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Port Open Statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 ShouldBeReachable On TransportProtocol Port Number -> PortOpenStatement

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Link Statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

FullDuplex | HalfDuplex -> LinkDuplex
Link Speed ShouldBeExpr LinkSpeed-> LinkStatement
Link Duplex ShouldBeExpr LinkDuplex -> LinkStatement

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Packet Statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Host | Network -> PacketFromType

Packets From PacketFromType Identifier ShouldBeExpr Seen -> PacketStatement
Packets With TransportProtocol SourceOrDestination Port Number
   ShouldBeExpr Seen -> PacketStatement
Packets With PacketType HexType ShouldBeExpr Seen -> PacketStatement
```

```
Frames With Ethertype HexType ShouldBeExpr Seen -> PacketStatement

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Ping Statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ShouldBeReachable ByPing -> PingStatement

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Traceroute Statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

TracerouteTo Identifier ShouldTraverse OpenSquareBracket (Identifier WS)+
    CloseSquareBracket -> TracerouteStatement

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Vlan Statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 VlanAccess -> VlanType
 VlanType Vlan ShouldExpr Be Number -> VlanStatement


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% DNS Statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DomainName ShouldExpr Resolve (To Identifier)? UsingSserver Identifier ->
    DNSStatement

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Iperf Statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Least | Most -> LeastOrMost
UploadTo | DownloadFrom -> IperfDirection
Iperf IperfDirection Identifier ShouldAlwaysBeAt LeastOrMost IperfBitrate
    -> IperfStatement

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File Fetching Statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 FileFetchProtocol ServerAt ShouldExpr Serve Identifier (OnPort Number)?
     -> FileFetchStatement


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Composing Statements Into Final Document
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
HostDefine -> Statement
NetworkDefine -> Statement
DHCPStatement -> Statement
PortOpenStatement -> Statement
LinkStatement -> Statement
PacketStatement -> Statement
PingStatement -> Statement
TracerouteStatement -> Statement
VlanStatement -> Statement
DNSStatement -> Statement
IperfStatement -> Statement
FileFetchStatement -> Statement

Statement+ -> Document
```

APPENDIX C. NETCDL Grammar - TextX

```
/*
    NETCDL Language Grammar
    Copyright 2016 Cody Hanson
*/

Document:
    statements+=Statement
;

NonWhiteSpace:
    /[^\s\n,\[\]]+/
;

Statement:
    Comment | NetworkDefineStatement | HostDefineStatement | LinkStatement
    | PacketStatement | PortOpenStatement | PingStatement |
        TraceRouteStatement | FileFetchStatement | DHCPStatement
    | VlanStatement | DNSStatement | IperfStatement
;

HostDefineStatement:
    'define host' name=NonWhiteSpace 'as' value=NonWhiteSpace
;

NetworkDefineStatement:
    'define network' name=NonWhiteSpace 'as' value=NonWhiteSpace
;

IperfStatement:
    'iperf' direction=/upload to|download from/ server=NonWhiteSpace
        'should be at' comparison=/least|most/ bitrate=NonWhiteSpace
;

DHCPStatement:
    'dhcp' type=/server|dns|network|gateway/ should=Should 'be'
        value=NonWhiteSpace
;

ReachabilityClause:
    host=NonWhiteSpace should=Should 'be reachable'
;

PortOpenStatement:
     reachable=ReachabilityClause 'on' protocol=/TCP|UDP/ 'port' port=INT
;

PingStatement:
     reachable=ReachabilityClause 'by ping'
;
```

```
DNSStatement:
    'domain name' domain=NonWhiteSpace should=Should 'resolve' ('to'
        resolve_to=NonWhiteSpace)? 'using server' server=NonWhiteSpace
;

FileFetchStatement:
    protocol=/http|ftp|tftp/ 'server at' target=NonWhiteSpace
        should=Should 'serve' filename=STRING ('on port' port=INT)?
;

TraceRouteStatement:
    'traceroute to' host=NonWhiteSpace 'should traverse' '['
        routers+=NonWhiteSpace ']'
;

Should:
    /should(\s+not)?/
;

LinkSpeed:
    /\d+Mb\/s/
;

VlanStatement:
    'access vlan' should=Should 'be' vlan=INT
;

LinkStatement:
    LinkSpeedStatement | LinkDuplexStatement
;

LinkSpeedStatement:
    'link speed' should=Should 'be' speed=LinkSpeed
;

LinkDuplexStatement:
    'link duplex' should=Should 'be' duplex=/half|full/
;

PacketFromStatement:
    'packets from' type=/host|network/ target=NonWhiteSpace should=Should
        'be seen'
;

PacketTypeStatement:
    'packets with type' value=NonWhiteSpace should=Should 'be seen'
;

FrameTypeStatement:
    'frames with ethertype' value=NonWhiteSpace should=Should 'be seen'
```

```
;

PacketPortStatement:
    'packets with' protocol=/TCP|UDP/ direction=/source|destination/
       'port' port=INT should=Should 'be seen'
;

PacketStatement:
    PacketFromStatement | PacketPortStatement | PacketTypeStatement |
       FrameTypeStatement
;

Comment:
    /^#.*/
;
```

APPENDIX D. NETCDL Certifier Implementation Standards

This document, while not a formal RFC, should serve as a guide for implementors of future NETCDL Certifier Software. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119[7].

A NETCDL certifier MUST recognize and parse a body of text that conforms to the official NETCDL grammar specification. If a parse error occurs, the certifier SHOULD inform the user the cause of the error, so that they may fix it.

During certification the certifier SHALL evaluate every assertion in the input document. The certifier MAY carry out the assertions in any order. The certifier SHOULD attempt to minimize the total assertion time, or the performance impact on the network, according to user preferences.

After certification activities have completed, the certifier SHALL provide the user with a report of certification, by some means. The report MUST show overall success, where greater than zero failures results in failed certification, and zero failures results in passed certification. The report SHOULD indicate to the user the cause of the failures. After certification activities have completed, the software SHOULD release any resources acquired, such as DHCP leases, slots on bandwidth limited servers, and other such limited resources.

Certifiers MAY provide a graphical user interface for the user. Certifiers MAY be loaded onto dedicated hardware platforms that carry out certification functions.

Certifier implementations MAY be proprietary in nature.

APPENDIX E. Software Version Notes

SdfMetz Grammar Analysis

Operating System Ubuntu Linux Warty Warthog 4.10

Kernel 2.6.8.1-3

gcc 3.3.4

SdfMetz v1.1

sdf2-bundle 2.0.1

ghc 6.2.2

xt-aterm 2.0.5

NETCDL Certifier Software

Python 2.7.6

docopt 0.6.2

requests 2.11.1

scapy 2.3.1

textx 1.4

tftpy 0.6.2

fabulous 0.3.0

netifaces 0.10.4

iptools 0.6.1

Sphinx 1.4.4

pytest 2.8.7

pytest-cov 2.2.1

pytest-mock 1.2.0

Operating System Tested with Linux Mint Rosa 17.3 Kernel 3.19.0-32-generic

APPENDIX F. Example NETCDL Document

```
# Certification Subject: Home Network
# Author: Cody Hanson <chanson@uwalumni.com>
# Date: 10/1/2016
# Version: 1.0

define host myRouter as 192.168.1.1
define network myNetwork as 192.168.1.0/24
define network DMZ as 10.0.0.0/24
define host ftpSite as speedtest.tele2.net

link speed should be 1000Mb/s
link duplex should be full

iperf download from ent.local should be at most 30Kbps
iperf upload to ent.local should be at least 30Kbps

access vlan should be 500

dhcp server should be myRouter
dhcp dns should be myRouter
dhcp network should be 192.168.1.0/24
dhcp gateway should be 192.168.1.1

myRouter should be reachable by ping

domain name google.com should resolve using server 8.8.8.8
domain name ent.local should resolve to 192.168.1.144 using server myRouter

myRouter should be reachable on TCP port 22
myRouter should not be reachable on TCP port 23
myRouter should not be reachable on UDP port 100

#File fetch assertions, default port used for protocol, if omitted
http server at myRouter should not serve "/path/to/file" on port 8080
http server at google.com should serve "/index.html"
http server at google.com should not serve "/index.html" on port 12345

tftp server at ent.local should serve "RouterUpdate.img"
tftp server at ent.local should not serve "RouterUpdate.img.missing"

ftp server at ftpSite should serve "1KB.zip"
ftp server at ftpSite should not serve "missingfile"

traceroute to 184.99.1.89 should traverse [192.168.1.1 10.0.0.1
    184.99.0.12]
traceroute to 10.0.0.1 should traverse [192.168.1.1]

packets from network DMZ should not be seen
```

```
packets from host 10.250.0.1 should not be seen
packets with type 0x18 should not be seen
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

frames with ethertype 0x18 should not be seen

# Document End - Thanks for reading!
```