Spring 2019

A Grammar Based Approach to Distributed Systems Fault Diagnosis Using Log Files

Stephen Hanka
Southern Methodist University, shanka.ieee@outlook.com

A GRAMMAR BASED APPROACH TO DISTRIBUTED SYSTEMS FAULT DIAGNOSIS USING LOG FILES

Approved by:

Dr. Frank Coyle
Senior Lecturer, CSE Dept.

Dr. Suku Nair
Professor, CSE Dept.

Dr Theodore Manikas
Clinical Professor, CSE Dept.

Dr Liguo Huang
Associate Professor, CSE Dept.

Dr. Eric Larsen
Assistant Professor, CSE Dept.

A GRAMMAR BASED APPROACH TO DISTRIBUTED SYSTEMS FAULT DIAGNOSIS USING LOG FILES

A Praxis Presented to the Graduate Faculty of

Bobby B. Lyle School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Doctor of Engineering

with a

Major in Software Engineering

by

Stephen Hanka

M.S., Software Engineering, Southern Methodist University

B.S. Business Data Systems, California State University

May 18, 2019

ACKNOWLEDGMENTS

This work could not have been accomplished without the extreme patience of my wife and without the gentle prodding and good humor of my advisor, Dr. Frank Coyle, and that of the faculty here at the Lyle School of Engineering at SMU.

In addition, I would like to thank my employer, Alcatel-Lucent Enterprise, for financial support while completing my degree.

Hanka, Stephen

M.S., Software Engineering, Southern Methodist University, Dallas, 2004
B.S., Business Data Systems, California State University, Dominguez Hills, 1980

A Grammar Based Approach to Distributed Systems Fault Diagnosis Using Log Files

Advisor: Dr. Frank Coyle

Doctor of Engineering conferred May 18, 2019

Praxis completed April 22, 2019

Diagnosing and correcting failures in complex, distributed systems is difficult. In a network of perhaps dozens of nodes, each of which is executing dozens of interacting applications, sometimes from different suppliers or vendors, finding the source of a system failure is a confusing, tedious piece of detective work. The person assigned this task must trace the failing command, event, or operation through the network components and find a deviation from the correct, desired interaction sequence. After a deviation is identified, the failing applications must be found, and the fault or faults traced to the incorrect source code.

Often the primary source for tracing failures is the set of event log files generated by the applications on each node. The event logs from several platforms and from multiple virtual machines on those platforms must be filtered, merged, correlated, and examined by a human expert. The expert must locate the point of failure within the logs and then deduce which interaction or component failed, then re-assign the problem to the persons responsible for the failing component sets. Those individuals must then, in turn, use the original logs filtered and merged using different criteria to find the failing code modules, analyze the cause of the failure, and correct the code or even the architecture of the failing components.

Reducing the human effort involved in diagnosing these test failures through automated analysis of data in the logs is the goal of this project. In this paper we propose generating grammars from test successful log sequences, then using the grammars to detect points of deviation in logs from the failed tests.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

THE PROBLEM DIAGNOSIS QUAGMIRE

1.1 Introduction to the problem – Quagmire and Frustration

It is Monday morning and a programmer arrives at her office and opens her email. Several messages are in the middle of the list assigning her problem reports. On the bug tracker web pages the problems are all marked critical or major and they state "Host Twillig down after reboot on seventh iteration of reload test", or "Virtual chassis split after failover", or some other cryptic problem summary that does nothing to explain what really happened, where to start looking for the problem, or even when the problem occurred. The message might be to the effect that "Test setup 12 failed the weekend regression test on Saturday morning. The test setup is still in the failed state, and you can access it at IP address 192.168.40.35. Please investigate. Problem report 263492 has been opened to track this". But test setup twelve is a mesh of eight packet switches each of which is running 100 different control applications, and the test failed with some unknown error on the seventh iteration of the test suite execution. Each switch has a dozen processors and as many application specific integrated circuits (ASICS). Each is its own set of application tasks to implement the routing and switching protocols and program the ASICs to route the network packets through the network.

Alternatively, the system under test could be a telephone switching network or a packet switched data network. Tracing a telephone call through an SS7 telephone circuit switched network involves dozens of systems to route and link the call through the switching systems, more systems to provide the billing information, and each system might have a dozen embedded processors including network line interfaces, digital signal processors, and switch controllers. Each switch in the chain might be manufactured by a different vendor with a unique processor,

with software written in a different programming language, executing a different operating system. And these same switches may support thousands of simultaneous and distinct voice conversations, each of which must be routed, tracked, terminated, and secured. The test setup reset two days earlier and all the internal state of the system at the time of the failure has gone into the proverbial bit bucket.

The programmer must somehow isolate the failure, and locate the programming error, design flaws, or even the requirements errors that precipitated the failure by retracing the system interactions backward from the anomaly detected by the test script to the input that started what might be a cascade of malfunctions or errors. There must be some way of looking back in time, of tracing the system through its steps from the time the test halted, maybe days or hours earlier, up to actual point of breakdown.

Our hypothetical programmer must often employ formidable detective skills to determine when a sequence of failures began, and which software component went awry. She must the identify the root cause of the problem and either correct it herself or assign the problem report to the programmer responsible for the failing component, or even assign it to the person who wrote the failing test script. If she assigns it to someone else, that person may go through the same type of investigation just described to locate the module or task failure that caused the test script to halt.

1.2 Event Logs –Threads through a Maze

Developing networked systems and delivering them is an enormously complex task. Locating failures in newly developed software while it is in the system the test and delivery process, when all the components have been integrated, requires tracing the execution flow through, and the data exchanges between the components. And although the use of source level

debuggers and other program verification tools may be the best choice during the software unit testing phase, these are less effective during system testing with the network under traffic loads.

The event logs, the internal state display and program trace statements inserted into program code by programmers since the early days of programming, become a primary source of information for problem diagnosis. Software developers have always put print or log statements into their code to provide run-time system state change and debugging traces through the application programs.

These log entries or events provide a trail through the program code showing which program modules executed and the input data triggering that execution. The log entries are often put in as a form of self-defense by the programmer against the inevitable finger pointing that arises during system verification, for they demonstrate that some function did indeed execute and may list the input and output parameters of that function.

For those of us who have worked troubleshooting embedded systems, notably telecommunication systems such as telephone switching systems and network packet switches, the use of event logs in problem diagnosis is assumed. Identifying the source and location of problems is well-nigh impossible without these event log trails starting at some input stimuli, winding through the internal component interactions, and finally the output signals and data.

Complex, distributed systems can create voluminous event log files to record the occurrence of significant state change information at run time. These systems might link several hardware platforms each with dozens of executable components exchanging information between peers both within and between platforms. As shown in Figure 1 each platform or component may have

its own set of event log files, recording interactions and progress as specific operations or requests are completed.

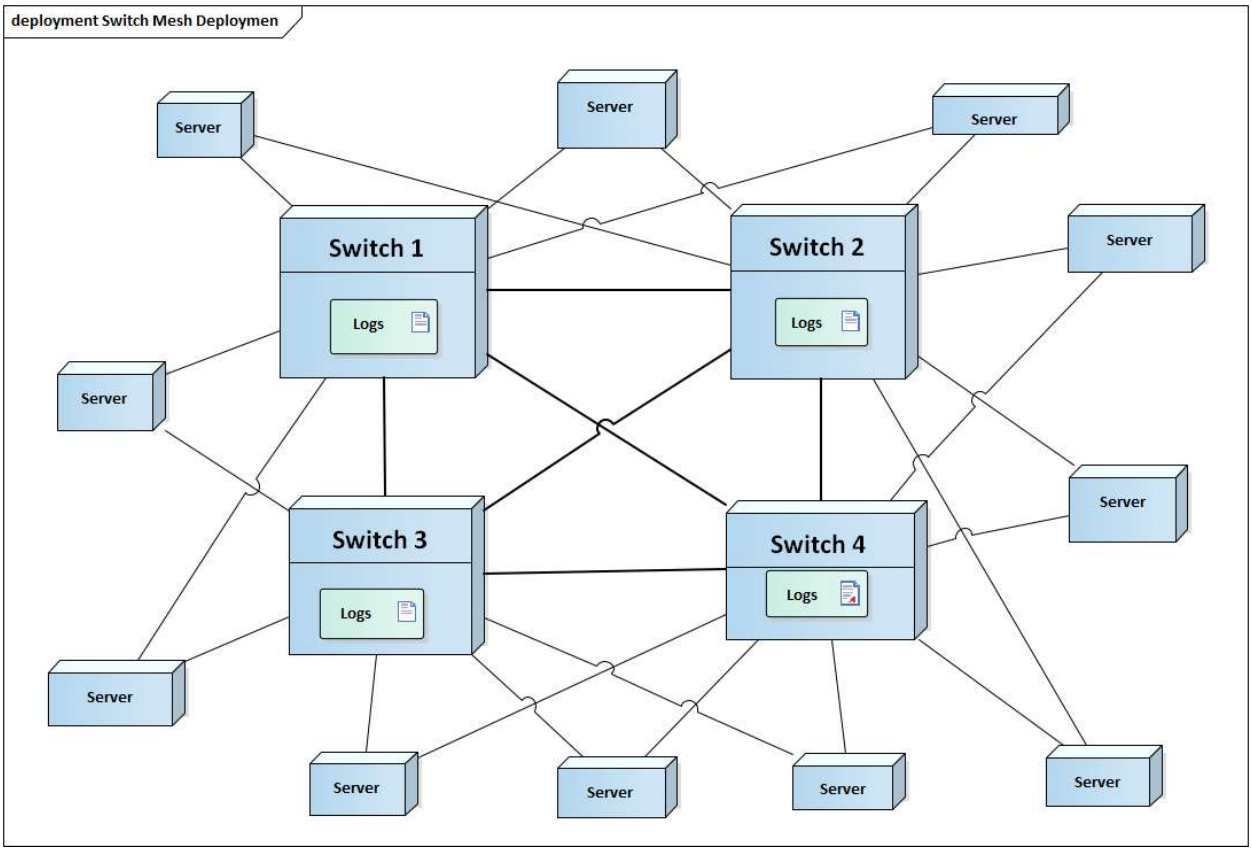

Figure 1 - Event Log Files in a Mesh of Packet Switches

When a test failure occurs, finding the sequence of interactions within a system test case that led to the malfunction is challenging. The event logs from several platforms and from multiple virtual machines on those platforms must be filtered, merged, correlated, and examined by a human expert. That expert must locate the point of deviation from the norm within the logs and deduce which interaction or component behaved unexpectedly. The expert must then assign the problem to the persons responsible for the failing components. Those individuals must then, in turn, use the original logs filtered and merged using different criteria to find the failing code

modules, analyze the cause of the failure, and correct the code or even the architecture of the system.

Reducing the human effort involved in diagnosing these test failures through automated analysis of the data in the logs is the goal of this project. Other researchers have proposed and implemented a variety of methods for log file analysis, including source code analysis, pattern analysis, grammar inference, and modeling the system through state machine generation.

In this paper we describe a method for automatically generating a grammar to characterize successful log sequences. We can then feed a log file from a failing system to a parser generated from the grammar and automatically detect deviations from the successful cases, a deviation being a missing log message, an unexpected log message, or a message that is out of sequence.

1.3 Paper Organization

Section 2 is a review of related work.

In Section 3 we describe the problem we are trying to solve in detail, propose a work flow for processing a log file into a grammar, and describe the work flow steps in detail.

In Section 4 we describe tests and measurements of the work flow, generating grammars from existing event logs for some specific system tests. By injecting errors into the system test logs we determine if the generated grammars for those tests can locate the failing test steps.

Section 5 contains our conclusions and a description for future work.

Chapter 2

PREVIOUS WORK

2.1 Background

Writing software is hard. Computer systems may be the most complex artifacts ever made by man and the bulk of that complexity is found in the software [1]. Constructing a large software system is time consuming, labor intensive, requires highly trained personnel, is prone to error, and is, therefore, expensive.

And though software is now ubiquitous, with program code hiding in almost every one of the even moderately complex devices we use to manage and navigate our civilization in the twenty-first century, it is still a bit of a foggy concept. Software is ordered lists of instructions to machines written in an ever-increasing variety of formalisms we call programming languages. Software components are stacked, linked, and communicate with each other and with us through voltage changes on often microscopic wires. At its most basic level software is insubstantial fluff, formalisms translated into electric charges flowing through enormous arrays of transistors and capacitors concealed in tiny integrated circuit chips. It has been described rather poetically as a hidden, delicate, dynamic dance [2] of instructions and data within the machine. Understanding the transformations, exchanges, and manipulations taking place within the machines that are now part of almost every human enterprise has been and still is intellectually daunting.

When software is under construction, the developers of the program code must verify that it is functioning correctly. They must monitor the program state changes within the ICs, verify that the variables within the software are changing as specified for a given input. If the program state is incorrect, the developers must locate the errant instructions and correct them. This is often

very confusing and difficult. There is a large software tools industry just for this initial debugging effort but using such tools on an integrated and running system impacts performance, resource usage and requires tapping into the system in a controlled environment. Including the debugging symbol table in a system, for example, increases the memory usage and program image size of an executing system. One of the executables created for this project, for example, is 50% smaller when compiled and linked with full optimization and no symbols than it is with the table. Additionally, accessing the system with a debugger might be very difficult, especially on an embedded system with limited outside connectivity.

Before these debugging tools came into existence, ever since software became an integral controlling component in embedded and networked components, programmers have been using console logs for debugging and locating programming errors, as documented by Gill in 1951 [3]. Gill describes one of the most common debugging tools, "trace debugging", akin to C-language "printf" statements, where programmers insert print commands to output run-time state data on a system's console device. These simple print calls have evolved into widely varying and complex logging systems and libraries, but the output format and content are still largely determined by the programmer's needs at the time the software was originally written and tested.

Today's systems might be composed of hundreds of software components developed by hundreds of programmers scattered across continents and countries, even when the components are created within one corporate organization. This diversity of software sources produces a variety of log styles and content even when a common event log format is used. And because capturing and recording these traces consumes system and network resources, the debug logging is often reduced or even disabled in delivered systems. The result can be and often is a confusing mass of text files giving a less-than-complete trace of hardware events and software responses to

those events. There is valuable information in these log files and getting that information out of the files is an ongoing research topic. [4] [5] [6].

Embedded systems pose another problem in that there may be little persistent storage space on the device for log data in flash memory or static RAM. There may be little network bandwidth for saving log data on an external system. The act of recording the log data may consume scarce computational resources that should be allocated to the device's main function. Log data may therefore be terse and sparse.

Reading and interpreting the logs is another problem. When presented with a large text file where each line may come from a different source, how does one filter the significant signal from the noise? This is also an area of research and the subject of this praxis. Can this be automated? Can we use methods familiar to most software developers in doing so? Can we automatically generate something like a programming language grammar and create a recognizer for valid log event sequences that will reject the failing sequences where they begin to diverge from the acceptable sequences?

2.2 Language Based Approaches

These questions are not new. Defining a programming language for log processing is an idea that goes back at least two decades. In 1998 Andrews [7] [4] described a framework, method, and language for formally specifying log file analyzer programs for use during system testing. The language, LFAL, provided a way for formally specifying test oracles for processing log files from test executions. Before test evaluation, the user must write an LFAL program, a formal test output description describing the expected log output from the test. Andrews argued that testing

using log file analysis was a useful method for software verification, providing an intermediate form between ad-hoc testing and formal verification methods[1].

Andrews and Zhang [8] describe using a formal log analyzer in conjunction with random testing, showing by experiment that this technique is competitive with other formal and informal unit testing methods. These techniques require manual specification of custom log file analyzers, defining a set of finite state machines modeling the communicating processes executing during a given test. Manual specification implies a substantial effort by the system developer in creating and then maintaining these models for a set of interacting components during system testing.

Also using the programming language theme, Barringer and others [9] [10] [11] used run-time traces to generate test cases. They proposed and implemented the formal specification languages RuleR and LogScope for verifying spacecraft software through analysis of telemetry data received from systems under test. With LogScope the use of the already present telemetry data, very closely related to event log messages, reduces the storage and instrumentation requirements in the software by taking advantage of data already transmitted by the systems. This technique provides a method for low-impact formal verification. The formal specification languages they describe are defined by a type of grammar which precisely defines the acceptable and unacceptable telemetry data expected from a system in response to a transmitted command or set of commands. The languages allow formal specification of statements like "after command *c* accept response messages *x, y,* and *z* in any order, but fail if messages *u* or *v* are

[1] The LFAL implementation produced Prolog code for processing the test log files, which was quite slow. Later work by Aulenbacher [3] produced an LFAL implementation generating C++ code, which ran more than eight times faster than the original Prolog generation.

seen." The technique requires manual coding of the LogScope script by the development engineer on a test-by-test basis, which is acceptable in unit testing but becomes onerous when testing large networked systems. It is also difficult for a development engineer not directly involved in acceptance test development to write such acceptance test oracles, which may be required in organizations maintaining separate engineering and testing groups or departments. In addition, the unit test suites and acceptance test suites may be in separate code bases maintained by geographically separated groups. We should note that LogScope techniques are similar to those employed by commercial software testing tools which verify command results against user interface output on Web pages or mobile device user interfaces such as Worksoft Certify [12], Ranorex Studio [13], or CA Technologies Application Test [14]. All of these require manual coding of test scripts and result interpretation[2].

Avoiding manually coded rule systems and patterns, Stearley describes the ironically named Sisyphus toolkit[3] for relieving the "never-ending curse" of tedious log analysis [15] [16]. Using a modification of Teiresias, an algorithm developed by IBM for string pattern matching in bioinformatics [17], Sisyphus searches for words in blocks of log message text to find patterns using Vaarandi's Simple Logfile Clustering Tool (SLCT) [18] and then displays the messages with the correlated patterns in *LogView* [19], an event log visualization tool. Sisyphus eliminates

[2] (Anecdotal: The author's employer has a team writing Python test scripts for engineering acceptance tests, and another team writing a different set of scripts for system testing. The development engineers are expected to write Python scripts for unit testing, and another group writes the integration tests executed before the final quality assurance group executes another set of tests which must pass a specific quality level before product release. None of these tests use the event logs generated by the system, all depend upon either user interface output or external test equipment measurements)

[3] In Greek mythology king Sisyphus was cursed by the gods for his deceitful wickedness, condemned to pushing a boulder up a hill for all eternity. When he approached the top of the hill, the boulder would roll back down, and he had to begin anew. His name is now associated with pointless and futile labor.

the manual coding of rules identifying expected or problematic sequences of messages, but the patterns it reveals are not suitable for finding the sources of failures exposed during testing.

2.3 Grammar Based Approaches

The idea of grammar inference, discovering a grammar from the output of existing software artifacts, has been explored in other contexts for decades. More than 50 years ago, Gold [20] investigated whether it is possible to synthesize a formal language from large textual documents, a language that accepts only strings of words found in the original documents. He found that identifying a regular language from a text containing all possible strings in a language was impossible. But others have made similar attempts with regular languages, programming languages and domain specific languages. Ocina and Garcia [21] demonstrated that regular languages can be identified from large text documents in polynomial time provided that positive (accepted) strings and negative (rejected) strings are identified as input. Colin Higuera [22] described the research trends in grammatical inference, again indicating that finding context free grammars from sample programming language code is a hard problem. Undaunted by the difficulty of these large problem domains, Javek, Crepinsek and others, [23] [24], explore context free grammar discovery for domain specific languages (DSL) using genetic algorithms, and demonstrate the suitability of the approach. They propose this approach for enhancement and maintenance of legacy systems.

Also following the programming language theme, Memon [25] describes a generic method using grammar inference for log message categorization and anomaly detection. Memon's goal was to find unexpected or unknown log entries in an event log, one of the goals of this project. Memon's method processes logs from successful tests or transactions generating a grammar describing those log entries. The grammar can be used to process another log and find

previously unseen log messages, often a harbinger of runtime problems or failures. Memon did not detect acceptable *sequences* of messages. In short, because there is no temporal sequence or ordering of the messages, there is no finite state machine or process description in Memon's generated grammar, only a list of expected and syntactically correct log messages. It identifies the acceptable event log messages in the grammar, but not syntactically legal sequences of messages. The grammar cannot find missing events or acceptable log messages that are received out of order, both of which may indicate an anomaly.

2.4 Text Mining Approaches

Non-language-oriented event log mining methods for problem diagnosis in large systems are also an ongoing research topic. Xu, Huang and others [26] [27] combine event log mining, *source code analysis* (emphasis added), and machine learning to detect operational problems in Hadoop systems and Google servers. The source code analysis provides insight into the software structure, which allows discovery of software operational features, and possible problems. They produce a decision tree mapping the problems to the pertinent log messages.

Maruyama and Matsuoka [28] use function call traces, comparing traces between successful and failed test cases. The call traces record function entry and exit on running systems, saving call parameters and results. On any real-time system tracing every function call can adversely affect performance, up to 7% according to the authors. This figure seems optimistic, but even this impact can affect the correctness of real-time systems, which must often provide a computed result within fixed time limits. Since event log creation is embedded in the application software, logging always impacts performance to some degree. The logging system must be designed with performance requirements in mind.

Fu, Lou, Wang, and Li [29] describe methods for finding anomalies in unstructured text logs taken from a generic distributed system. They use text mining techniques and string distance algorithms to classify and convert event log entries into a log key form, which is basically the format string in a C language printf statement. They then build finite state machines for individual event sources from the logs, using an algorithm described by Mariani and Pezz'e [30]. Through performance analysis on the FSMs, they attempt to detect possible system faults from the logs. This is very close to our goal, except that we want a generated grammar that will process a log file and find syntactically incorrect sequences.

Vaarandi and others [31] [18] present several event log mining algorithms, primarily focused on network security applications, but applicable to other log analysis purposes as well. SLCT and LogHound are log clustering algorithms that were employed in the Sandia Labs Sisyphus log analysis toolset, (see Stearley [16] above). Attempting to improve on SLCT they presented LogCluster [32] with improved wildcard matching, data clustering, and line pattern mining from textual event logs. Their goal was to mine patterns and find system faults in very large event logs. On log files containing tens of millions of events, the algorithm found more than 100 patterns in less than two hours of run time. This is useful in the security arena, where attacks on the system generate suspicious access patterns identifiable in the system logs.

Jiang, Munawar, Reidemeister, and Ward [33] describe a method for defining a set of invariants, long-term stable correlations, collected from logs over a long period of time using artificial neural networks. Outliers from these invariants are used to identify errors. They were trying to detect failures, but in our case the fact of the failure is known, and we are trying to find where the log deviates from a successful example to locate a failing component from among many candidates on many machines.

Makanju and others [34] describe a refined pattern matching algorithm (IPLoM) for use with very large log files that cannot fit entirely into a computer's memory, improving upon SLCT and LogHound. This algorithm splits the log file into partitions in multiple steps, first partitioning by line length in words, then by word position in each line, then by word pairs. Finally, a line pattern is defined for each partition. SLCT, LogHound, LogCluster, and IPLoM are intended to find patterns, clusters and groups of entries within a log, but do not create a temporal sequence of the clusters.

In his 2012 PhD dissertation [35] Reidemeister goes further in describing methods for fault diagnosis in generic enterprise networks and supporting decision methods for recovery actions. He describes methods for modeling event logs taken from some of the references above, methods for using historic log entries of failures to locate new faults, and then provides methods for decision selection and system recovery actions.

On a related topic, in his 2016 textbook van der Aalst [36] describes the use of event logs for business process discovery, which he dubs process mining. He converts the event logs into a Petri Net which describes the business process producing the log. The Petri Nets can be used to find process bottlenecks and inefficiencies. He then uses the Petri Nets to discover deviations from the established processes in subsequent logs, finding where the processes are likely to be violated. He employs the A* Algorithm (and other methods) for converting a set of event logs into a Petri Net. These algorithms are of interest in this project because many executions of the process can be fed into the algorithms, generating a Petri Net that will accept many slightly differing versions of a process. In a real-time system, we run into this type of issue because system timing may vary between executions of a specific test case, resulting in a change in the temporal arrangement of log entries, all of which are “legal” for proper system operation.

Hamooni, Debnath, Xu, Zhang, Jian, and Mueen [37] present LogMine, a tool for finding patterns in large heterogeneous event logs. As part of their process they tokenize every log message using white-space separation. They then process the tokens to detect a set of pre-defined types including dates, timestamps, IP addresses, and numbers. They replace the value of each token with a name describing the it. Dates are replaced with 'date', time stamps are replaced with 'time', IP addresses with 'IP' and so forth. Messages with a common originator and identical content after the replacements are assumed to be identical. The user selects the patterns for the pre-defined types. This is like the method employed by Memon for textual substitutions and form recognition.

2.5 Other Approaches

Another approach for employing event logs to locate system faults is to record patterns preceding known failures and search for those patterns in a running system's log output. Gurumdimma, Jhumka, Liakata, Chua, and Brown [38] describe such a method for detecting patterns in failure logs. They scan large sets of log files looking for patterns that precede a known failure and use the patterns for failure prediction by scanning the logs on a running system. They also preprocess and tokenize the log files to remove redundant data, group similar messages and give them a common identifier, removing identical and redundant events. This is the inverse of our goal, which is to find where a deviation from the norm begins, not to find known deviations. Note that on an embedded system in a test lab, resources for storing the logs may be scarce and large data sets containing known failures may not be available.

Pecchia and Russo [39] describe an experiment injecting software faults into systems. They conclude that system characteristics such as software architecture, placement of the logging instructions, and support from the execution environment significantly increase the accuracy of

logs at runtime. This seems obvious to anyone who has tried to trace the root causes of system failures from large log files. If the system architecture does not support accurate log collection, or if the logging instructions are not meaningfully placed the log files might be incomplete or flooded with unhelpful information. From the author's experience, a log flooded with useless information is probably the more common case, and filtering such information is a necessary step before meaningful log analysis can begin, or the analysis method must scrub such data as part of its processing. Very large log files introduce scaling problems such as memory exhaustion, and log filtering is a necessary part of log file analysis.

System failure doesn't necessarily imply a crash. In the security domain, a failure is an intrusion and a successful intrusion often leaves few indications of its occurrence. In their amusingly titled "Fear and Logging in the Internet of Things" Wang and colleagues [40] describe methods for intrusion detection in home automation systems, using carefully instrumented code and formatted log statements. Their "ProvThings" provides centralized auditing of log events and activities. They provide a method for detecting and tracing intrusions by exposing exploited weaknesses.

On very large networks, the log files become too large for human derivation of an operational model from the events. Aiming to improve developer understanding of complex systems and aid in debugging, Ivan Beschastnikh [6] presents three log analysis tools that generate finite state machine (FSM, DFSM) models of systems by parsing and analyzing the system event logs. His method requires only a set of regular expressions for parsing the logs. His toolset provides methods for converting the parsed logs into state machines, inferring models of the underlying operational system. These methods break the system down into communicating finite state

machines exchanging data using FIFOs. The models produced enhance human understanding of these systems.

2.6 Commercial and Open Source Tools

Processing large amounts of stored supercomputer log data searching for patterns that might indicate a significant alert or alarm is a research topic and the focus of several commercial and open-source log management tools. Oliner and others [41] introduce the NodeInfo tool, which tokenizes unstructured log data, and then uses techniques derived from Shannon's information entropy [42] to determine which tokens are significant and which are merely noise. By ranking log messages from each network node by information content, the tool finds significant messages from each node and passes them to the system administrators for investigation.

The open source and commercial log management products provide aggregation and analysis services. For the most part they focus on real-time anomaly detection for network monitoring security (e.g. intrusion detection) and node failure. They allow the user to add custom log parsers that are specific to the data formats produced by the specific applications and services running on the network. All of them run as a log server application on a network node, accepting log data from the other nodes via UDP sockets, filtering and then saving the data on the server's file system.

ELK is one such open-source tool [43]. Originally composed of three open-source projects, Elasticsearch, Logstash, and Kibana (hence the acronym), ELK provides log search, log storage management, and log visualization. It now includes other services, but the main purposes are security, monitoring, and reporting.

Marketed as an “enterprise log management tool”, Graylog is another open-source log service [44]. It also provides for log storage, search, and log viewing. Accepting UDP or TCP connections other network nodes, it is also purposed toward security, monitoring, and reporting.

SPLUNK [45]is a very popular commercial tool for log management and analysis and provides similar sets of services as ELK and Graylog.

Targeting the network administrator as their user instead of the developer, none of these tools are geared toward product debugging or product troubleshooting. They require considerable configuration effort and may monopolize at least one network node while doing their work. By gathering logs from multiple network nodes on a single server they can also increase network traffic in the user’s local or wide area network as the log data are forwarded, which may be undesirable.

2.7 Contributions

Tabulating the above in Table 1, we see the contributions of the various authors and the value added by this project.

Table 1 - Contributions

Author	Grammar Based	DFA	Automated DFA Generation	Failure Detection	Sequence Error Detection	Employs off the shelf components
Hanka (this paper)	Y	Y	Y	Y	Y	Y
Fu and others [29]	N	Y	Y	Y	Y	N
Andrews & Zhang [8]	N	Y	N	Y	N	N
Memon [25]	Y	N	N	Y	N	N
Barringer, Groce [11]	N	Y	N	Y	Y	N
Johnston [5]	N	Y	N	Y	Y	N
Stearley [16]	N	N	N	Y	N	N
Vaarandi [18]	N	N	N	Y	N	N
Van der Aalst [36]	N	N	Y	Y	N	N
Beschastnikh [6]	N	Y	Y	N	N	N
Gurumdimma [38]	N	N	N	Y	N	N

From all of the above we can see that only Fu's group automatically generated a DFA for recognizing event sequencing errors or differences in an event log. The others either require manual specification of a recognizer, or find clusters of messages that might precede a previously known failure, not previously unseen messages or temporal errors in the sequence.

Fu's group produces results closest to what we desire, using intense text mining methods. But we want something fast and simple for use during system testing, which will quickly create recognizers for specific test case logs. The tool must also be quickly adaptable to an individual user's logs.

Chapter 3

METHOD

3.1 Why Grammars?

Our goal, again, is to use the event log output from a set of successful test executions to find where a known failing test execution begins to go awry. We are building a tool for generating a program that accepts, recognizes, the successful test case logs and rejects the log from the failing tests. Our tool is fast, easily adaptable to an individual user's environment, uses an off-the-shelf parser generator, and quickly generates a recognizer for individual test cases.

When one thinks of recognizers one is immediately drawn to the concept of regular languages and their equivalent finite state recognizers. We can represent a single event log as a long, but non-repeating regular expression; a simple list of events. Even though timing and scheduling variance may reorder the event sequence from one test to another, if we have multiple, but similar event logs from a set of successful tests, we can create a union of regular expressions to create a recognizer for all of them.

If each unique event in a log is mapped to a unique symbol, a log file can be represented as a simple sequence of symbols, which can be represented as a simple regular expression without loops and repetitions. Since our tests are of finite length, the associated expressions are also of finite length, and we therefore need not concern ourselves with reducing repeated sequences of symbols into shorter expressions with looping constructs. We can use available language processing software to create and then recognize unions of these long regular expressions, rejecting a log that doesn't conform to these combined expressions.

To accomplish this we need a way to parse the events in a log into their constituent parts, place those parts in a set of lists or a database, then use that saved content to generate a recognizer for the regular language represented by the sequences of events in the logs. The common language processing software employs context-free grammars as input. We are, therefore, led to using a grammar-based solution to process the successful event logs into a finite automaton and a grammar for representing that finite automaton when processing the failing log or logs.

Why use a grammar instead of just writing a large set of regular expressions? Because there are commonly available tools for computer language processing accepting a grammar as input, but not for regular expressions. The language tools combine lexical analysis, parsing, and processing into a single input file, making the method easy to use. The generated output grammar, when fed to the same tools, automatically creates the recognizer for the log file with little extra work required. Much more work is involved with regular expressions, and they are much harder for the average human to read.

In addition, the more inclusive log file format we have chosen, RFC 5424, has a set of fields allowing user extension of the log file contents, structured data, which are more difficult to represent and parse with simple regular expressions. A grammar represents these fields more naturally and easily.

And finally an open-source Syslog ANTLR grammar is available for us to employ and modify, making use of pre-existing work [46].

3.2 Logs, Grammars, and Finite Automata

An event log is a sequence of text messages written by a group of programs into a common file. Each event in the log can be described by a formal grammar or even a regular expression. If we assign each unique event a symbol, the sequence of events in the log forms a string of symbols. For a finite test case, the log is of finite length, the string of associated symbols is finite and can be described as a simple list: a regular expression with the symbols as its alphabet, but without repetitions or loops.

We want a finite state recognizer that will accept such a list from a set of successful test execution logs and reject lists from a test failure with improper or unexpected event sequences. The salient issue with such a recognizer is that the sequence of log messages generated from multiple executions of a given test can be very long in similar but slightly varying order. Hardware devices or multiple servers might take longer to respond to a given set of stimuli, and the temporal ordering of the responses may vary from one test to the next causing a change in the recorded order of events. We must generate a recognizer that accepts these variations from the successful tests.

But to even get to the recognizer itself, we must first provide a method for uniquely identifying the events in the log and assigning them unique symbols that the recognizer can process as an alphabet. For this step, we turn to a grammar.

In 1956, Noam Chomsky described a formal notation for systems of rewriting rules that generate combinations of words forming acceptable sentences or sequences in a specific language [47]. In these formal grammars Chomsky first focused on natural spoken languages but his research branched into applications in computer programming languages [48], which was expanded by Backus [49]. Since then the concept of a “language” has diverged to include

anything representable as a constrained sequence of symbols such as jazz music chord progressions [50], and DNA sequences in molecular biology [51].

When we say a grammar generates all possible sequences of words in a language, we imply that the grammar can be employed to recognize those same sequences, an insight provided by Backus. We employ a grammar that generates generic RFC5424 log events to create another grammar that generates a specific set of RFC5424 log events in a specific ordering. We also take advantage of the representation of a sequence as a regular expression [52], which can be transformed into a minimal state DFA [53].

A grammar is a set of production rules that define transformations of strings, sequences of symbols, in a formal language. Formally, a grammar, *G*, generates a set of strings and is composed of

N: A set of non-terminal symbols that are not included in the strings formed by *G*

$\sum$: A set of terminal symbols, sometimes called an alphabet and disjoint from *N*

P: A set of production rules mapping one set of strings to another. The rules are of the form;

$$(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$$

Where * denotes the Kleene star operator specifying zero or more repetitions of an expression, and ∪ represents a set union.

S: A distinct starting symbol that is an element of the set *N*.

The production rule definition *P* states that any, possibly empty, string of terminal and non-terminal symbols prefixing at least one non-terminal symbol and followed by any string of terminal or non-terminal symbols maps to another string of terminal or non-terminal symbols.

A grammar G is therefore usually represented as a tuple:

$G = (N, \sum, P, S)$

Quoting Appel [54], "a language is a set of strings, each string a finite sequence of symbols taken from a finite alphabet". For event log processing we can restrict ourselves to *context-free* grammars where each production in the grammar describing the language is of the form:

$$symbol \rightarrow symbol\ symbol\ symbol \ldots symbol$$

There is always a single non-terminal symbol on the left-hand side of each production. The terminal symbols or tokens come from the alphabet $\sum$, the non-terminal symbols in N appear on the left side of a production in P but are not in the alphabet or in the strings produced by the grammar, and no token in $\sum$ appears on the left side of a production. We must relax that rule a bit when defining ANTLR grammars because the tokens must be defined someplace as some character or sequence of characters for lexical analysis.

This brings up the question of what is in the set of symbols we are trying to recognize with our grammar, which we cover in Section 3.5. But before we get to that, we must understand finite state machines and how they relate to language processing.

3.3 The Finite State Machine as a System Model

Several of the researchers mentioned above use event logs to synthesize finite automata of a process, system, or software component. We have already mentioned Fu and others [29] creating log keys and then finite state machines for performance modeling. Roder's group uses event logs from semiconductor manufacturing equipment to synthesize finite state machines (FSMs) for operational modeling [55]. Cook and Wolfe describe methods for engineering process discovery from process logs, modeling the processes as deterministic finite automata or

DFAs [56], another term for finite state machine. We want to synthesize a recognizer, a DFA or FSM, for event sequences filtered from logs recorded during successful test executions.

To avoid confusion, note that DFA, FSM, and finite state recognizer all refer to the same software construct. We do not include Harel state charts [57] or their UML variations since these constructs, with their sub-states and decision branching cannot be represented by a regular expression.

A graphical representation a very simple finite state machine is shown in Figure 2.

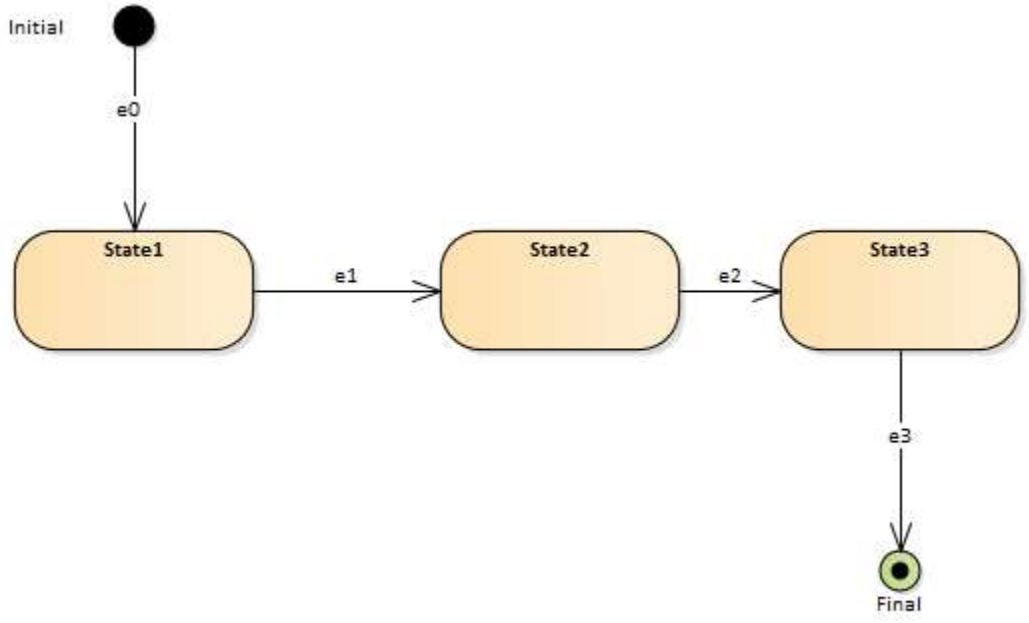

Figure 2 - Simple Finite State Machine

This machine accepts a sequence of symbols, (e0, e1, e2, e3) and then stops. It *recognizes* (e0, e1, e2, e3). Although it is simplistic, this is the type of linear machine we derive from a single test log, with e0 representing log event 1, e1 representing log event 2 and so forth.

If we have multiple, differing logs from multiple successful executions of that test we might have something like the machine shown in Figure 3. This machine recognizes two similar sequences (e0, e1, e2, e3) or (e0, e2, e1, e3). If we have two test event sequences that start and end on the same event, (e0 and e3), but some of the events arrive in a different order, we could use a recognizer like this.

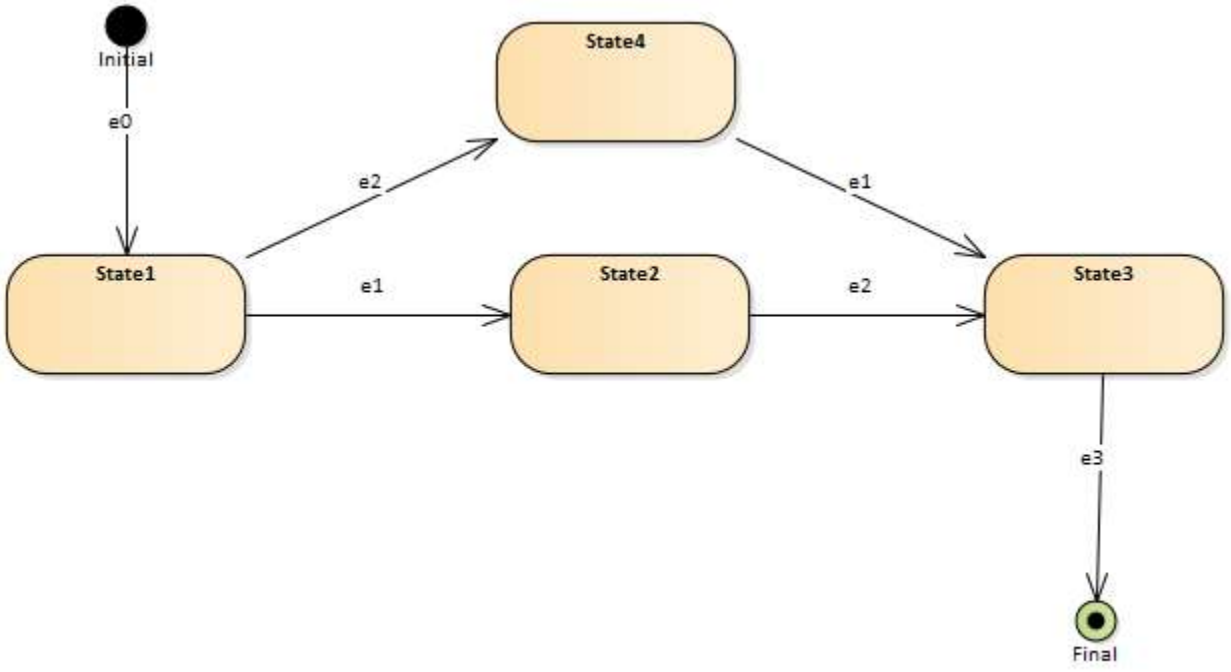

Figure 3 - Simple FSM Union

In the system test logs, we are given only event logs from successful test sequences and we are interested in detecting where an unsuccessful test execution goes awry. We cannot, therefore, use negative results to create our recognizer. But we are not interested in using grammar inference to generate a complete description of all acceptable event sequences for a given test case. We want to construct a recognizer that identifies the events and event sequences

most likely in a specific test scenario and then use that recognizer to find deviation points in logs from failing test cases. The failures are first detected by automated test scripts, and we want to detect where the system, and therefore the log, begins to exhibit aberrant behavior before the failure occurs.

A finite state recognizer can be expressed as an equivalent regular expression. It is possible to create a regular expression for an event log from a successful test case execution and convert that expression into a DFA. For multiple successful test executions, we can create a union of the regular expressions from successive test cases, convert the union into a DFA, and then minimize it [58] [59].

At the risk of being repetitive, we want an automatic method for generating the DFAs from the sequences of events we find in the event logs, but we also want to take advantage of existing tools. Lex and YACC [60], for example, allow specification of sequences of complex expressions with an input grammar and will generate a recognizer for the sequences. But their parsing method is LL(1), limiting the types of expressions they will accept and we would have to minimize the expressions in the grammar before feeding them to the tools.

Although there are algorithms for converting a DFA back into a regular expression such as state removal and Brzozowski's algebraic method [61] [62] [63], the output of these methods is repetitive and contains redundancies. Generating a minimal regular expression from a minimized finite state machine is known to be NP-hard and PSPACE hard [64] [65]. This is an issue if we want to use LEX and YACC type parser generators to create our recognizer because of the limitations of LL(1) parsers.

ANTLR allows us to sidestep this problem with its LL(*) capabilities, which allow an indefinite look-ahead in the input. We can generate several very similar regular expressions and represent them in a single rule as a disjunction. When processing such a rule, ANTLR builds a minimized DFA for the expression. When processing the failure log as input, ANTLR will backtrack on failure and to find a matching expression in the set of disjunctions or fail when all possibilities have been exhausted.

3.4 Choice of Log Format

The symbols we pass to our generated DFA are the events from the logs, not just the UTF-8 characters written into the log file. Each individual event becomes a symbol in our alphabet. We therefore require a fixed format for the messages so that we can map them to symbols. Because we must parse and classify the events before symbol mapping, this brings us to the question of event format.

Even though event logs are now acknowledged as a necessary component of information systems in general, there is no generally employed standard for the messages. Event log requirements are seldom specified in the overall requirements of a system. The event logs are often added after the fact, most often when a component is being unit tested because the developer needs runtime debug information, or when the system itself is being integrated. The log output is added by the programmers as needed for error diagnosis and correction.

This lack of a standard has spawned a plethora of different log file formats. Indeed, the Linux log file navigator program, lnav, supports more than thirty [66]. But there is no standard that most organizations follow except when specified by a system requirement for security or evidence auditing guidelines [67]. The IETF defines at least two, RFC5424 [68] for the syslog format and RFC 3164 [69] for the old BSD Syslog Format replaced by RFC5424. Then there

are Apache Commons Logging [70], the Microsoft Common Log File System [71], the W3C Log Format [72], and the W3C Extended Log Format [73] There are also XML and HTML formats such as Microsoft's EVTX [74] and Java Log4j's HTML option [75], but we are not considering them here because of their larger space and time requirements. In embedded systems where bandwidth and storage are limited, the use of storage space and CPU bandwidth for logs must be considered [76]. The use of the extra space and bandwidth employed for XML-style tags and field labeling may be prohibitive.

We have chosen RFC5424 format for use within our tool because it is a standard, contains virtually all the data fields in the other forms, is compact compared to the XML and HTML forms, its structured data fields make it extensible by the user, and because there are tools and parsers available for processing it.

We can now proceed with a discussion of grammars for processing these logs and grammars for accepting them.

3.5 Events to Tokens and Grammars

Loosely speaking, an event log is a text file recorded by a set of application programs running in some execution environment. The file is usually encoded in UTF-8 [77] or USASCII [78], where each line in the file represents something significant to a software developer writing the application. Sometimes the event, the line of text, is required by a network or security standard. An event might record a hardware state change, an application program state change, or an application failure or error. As mentioned earlier, the event is often only of significance to the software developer writing the application.

The log files are composed of a sequence of lines separated by new-line characters or sequences, with each line describing some event. Regardless of the forms, each event in the event log contains a time stamp, a set of source identifiers, a priority or level, and an arbitrary description of the event defined by the developer.

All the log formats define events with these fields, which specify when an event happened, which application on a specific server recorded the event, its priority, and a description of what happened. These are the when, who, and what elements of a log entry. For example, a simple event log entry or event might resemble Figure 4, taken from the log of a network packet switch:

```
2017-09-30T10:00:13.988 x3 swlogd vcmCmm chas_sup INFO
CMM:vcmCMM_cs_handle_app_ready@5019: vfc ready (chassis 0, slot 65, appid 143, pid 3544)
```

Figure 4 - Event log example, loosely based on RFC 3161

The figure shows the timestamp (2017-09-30…), the set of source identifiers, the level (INFO), and the description (CMM:vcmCMM...). This example is based on the form specified by RFC 3164, the BSD syslog format [69]. Another example is shown in Figure 5.

```
<16>1 2018-05-22T11:32:30.458Z MEDORA-DUT1 intfCmm - - [@aleLog327655 logger="swlogd"
subApp="Mgr"] cmmEsmUpdateInterfaceDynamicInfo: 2/3/14: adminStatus=1, autoNego=1(0,0,0,0),
splitterMode=0, linkStatus=2,linkChangeTime=15270
```

Figure 5 - Example RFC5424 Event

The RFC5424 syslog format is more complex than the older RFC 3164 specification, but it also contains more information and is extensible by the user. A breakdown of the message in Figure 5 is shown in Figure 6.

```
<16>1                                : Priority and version

2018-05-22T11:32:30.458Z             : Timestamp

MEDORA-DUT1                          : Host name

intfCmm                              : Application name

-                                    : Process ID (unused here, replaced with "-"
-                                    : Message ID (unused here, replaced with "-:

[@aleLog327655 logger="swlogd" subApp="Mgr"] : Structured Data (System Specific)

And finally the message itself followed by a new line character:

cmmEsmUpdateInterfaceDynamicInfo: 2/3/14: adminStatus=1, autoNego=1(0,0,0,0), splitterMode=0,
linkStatus=2,linkChangeTime=15270 <NL>
```

Figure 6 - RFC5424 Event Log Field Descriptions

We can see from these examples that an event is a tuple consisting of a timestamp, a priority, something identifying the network host where the event originated, an identifier for the originating application on that host, a process ID, a message ID, some user or application specific structured data, and finally the log message text describing what happened.

The RFC5424 document formally describes an event, a "Syslog Message", in Augmented Backus Naur Form [79]. We show a small sample in Figure 7:

```
SYSLOG-MSG = HEADER SP STRUCTURED-DATA [SP MSG]
HEADER = PRI VERSION SP TIMESTAMP SP HOSTNAME
      SP APP-NAME SP PROCID SP MSGID
```

Figure 7 - RFC5424 Syslog Message ABNF Sample

Conveniently, by its use of ABNF, RFC5424 provides a path for constructing a context-free grammar that will generate the events. And even more conveniently for us, Otto Fowler [46] defined such a grammar for the ANTLR tool, which we employed to generate a parser for these events. We modified Fowler's grammar and the ANTLR generated parser code to develop part of a tool chain that creates a finite state recognizer for event logs for successful test cases.

Since we are describing the events in the form of a grammar, an event tuple can be defined as a production as shown in Figure 8.

$$e \leftarrow priVer\ sp\ timestamp\ sp\ host\ sp\ app\ sp\ pid\ sp\ mid\ sp\ sdata\ sp\ msg\ nl$$

Where

$priVer$	: the priority and version of the event
sp	: a space character used to separate the fields in the event
$timestamp$	:time the event occurred
$host$	: name or IP address of the network host where the event occurred
app	: application program identifier for the application program that recorded the event
pid	: process ID of the recording application
mid	: message ID for the log message
$sdata$	: structured data associated with the event
msg	: text describing the event
nl	: a new-line character marking the end of the event

Figure 8 - Event Tuple Grammar Rule

Since an event log is simply a sequence of one or more events, we have a simple rule for the log itself, as shown in Figure 9.

$$L \leftarrow e^{+}$$

Figure 9 - Event Log Grammar Rule

We have further restrictions however. Timestamps in a log must be ordered, for example, where the timestamp for a given entry in a log is always less than or equal to its predecessor's time stamp, a requirement for sorting and merging.

All the fields in an event are text strings, and there are a limited number of strings for all the fields because there are a limited number of hosts, applications, and processes in the network and because the log file is of finite length. The timestamp is not dependent upon the source that

recorded the event, but all the others are. There is a fixed, maximum number of priVer strings defined in the RFC5424 standard. And since there are a limited number of applications generating output in a log, there are a limited number of event descriptions in a given log. These limitations imply that a relatively small database can store the strings for each event field from the entire log.

It must also be noted that some event data from the same event may naturally vary from execution to execution, excluding the timestamp which must change by its definition. These variable data may include local measurements, such as air temperature, or process IDs, IP addresses, and so forth. We must, therefore, provide some method of identifying these values and replacing them and ignoring them when the logs are processed. This method is described in Section 3.9 below.

And since we are garnering these logs from system testing, we are also assuming that repeated executions of the same test are being run on the same device or system under test, with an identical configuration. For a given set of test logs, the hardware configuration and network configuration does not vary between tests. This restriction can be softened by the addition of additional variable substitution rules, also described in Section 3.9.

To tie this back to Chomsky's grammar tuples, we can define our two grammars with some common elements. In our first grammar which processes the event logs from our successful test executions into a second grammar, we can write

$$G_a = (N_a, \sum_a, P_a, S_a)$$

Where

N_a : The set of non-terminals defined in Fowler's ANTLR grammar [46].

$\sum_a$: The set of UTF-8 characters allowed by RFC5424

P_a : The set of productions defined in Fowler's ANTLR grammar.

S_a : The starting event in the successful event log file being processed.

This grammar will process any well-formed RFC5424 event log file.

For the generated grammar, we are much more restrictive in that we must accept only previously processed sequences of events accepted by G_a above. In addition, the generated grammar rules must accept events with the variable fields replaced by an expression. The set of non-terminals, N_a, must be replaced by another set N_b, whose members map to sets of string constants previously encountered in the successful test logs.

For our generated grammar we have:

$$G_b = (N_b, \sum_a, P_b, S_a)$$

Where

N_b : A set of non-terminals defining a set of expressions mapped to a set of constants taken from the successful test event logs.

$\sum_a$: The set of UTF-8 characters defined in RFC 5424 (as above)

P_b : The set of productions taking the non-terminals in $N_b$to the set of constant strings in the logs.

S_b : The starting event in the successful event log file being processed, translated into a non-terminal in N_b.

For example, if our successful event log has a single event as shown in Figure 10 we get a set of productions like this for P_b.

$e0 \leftarrow p0\ sp\ ts\ sp\ h0\ sp\ app0\ sp\ dash\ sp\ dash\ sp\ sd0\ sp\ msg0\ nl$

$p0 \leftarrow$ "<16>1"

$h0 \leftarrow$ "MEDORA-DUT1"

$app0 \leftarrow$ "intfCmm"

$msg0 \leftarrow$ "cmmEsmUpdateInterfaceDynamicInfo: 2/3/14: adminStatus = 1, autoNego = 1(0, 0, 0, 0), splitterMode = 0, linkStatus = 2, linkChangeTime = 15270"

$sp \leftarrow$ " "

$dash \leftarrow$ "-"

$sd0 \leftarrow$ "[@aleLog327655 logger = "swlogd" subApp = "Mgr"]"

$ts \leftarrow$ fulldate CAP-T fulltime

Figure 10 - Sample Generated Grammar Productions

For our set of non-terminals, N_b, we get

$$N_b = \{e0, p0, h0, app0, msg0, sd0, sp, dash, ts, fulldate, fulltime\}$$

Figure 11 - Generated Grammar Non-terminals

Our starting symbol is the single event in the log file, $S_b = e0$.

The terminal symbols, the actual log file text, associated with each of the non-terminal symbols can be stored in ordered lists after each successful test execution log file is processed. These lists can serve as part of the input for processing the log from the next test execution allowing us to give a common set of identifiers for the common text strings over the successful tests. Since the events from successive test executions are generated by a common set of

applications processing similar input, the number of events is limited, and the number of text strings associated with each event is also limited.

As part of our process, we must include a user-implemented step of converting the log into a format acceptable by the tool. Because RFC 5424 is user-extensible through its structured data fields, almost any user's native form can be converted with no data loss.

3.6 Process

Our tool-chain must process our training logs, the successful test execution logs, into something recognizable by our ANTLR grammar, convert the logs into another grammar, and generate a recognizer for those logs and only those logs. This generated recognizer processes the logs from a failing test case and halts when the failing log deviates from the expected sequence of log messages as defined in the training logs.

For each successful test execution, the process involves:

a) Converting the logs into a common format as specified in RFC 5424. The user must provide a translator for this. We provide a C++ language example.
b) Merging and sorting the logs from different hosts. This is easily done with a simple python script, which we provide.
c) Creating the regular expressions for the variable fields. The user must provide this, but it isn't very much effort. We provide a standard set of expressions that the user can modify.
d) Removing the variable fields and replacing them with regular expressions. This is done automatically as part of the parsing and grammar generation with the user supplied file.

e) Trimming the first test execution log, (manual step),

f) Trimming the subsequent test execution logs. This is an automatic step included with the translating grammar processing

g) Generating the final grammar. This is done automatically.

h) Trimming the failure case log using the provided test log parser. This is done automatically also in the generated grammar code.

i) Processing the failure case log.

There are only a few steps that require coding or manual action by the user, which are providing a program for translating the logs into RFC5424 format, creating the variable replacement regular expression file, and trimming the log from the first successful test execution. These are steps a, c, and e above. The other steps just involve executing the provided tool chain with the correct command line arguments for each of the other steps.

The following sections describe the use of ANTLR and a grammar for processing an event log, and then the process itself.

3.7 Event Log Contents

As each successful test execution log is processed, we add to a small database of string definitions defining all the events in the log files. Memon [25] enumerated the unique combinations these strings as "forms" and our grammar definitions perform the same functions.

3.8 Event Sequencing

What Memon left out was the temporal relationships between the entries in the logs, leaving the sequences of the events in the logs undefined. His method could detect previously

unseen events, but could not detect missing events, unexpected multiple occurrences of events, or events that were out of sequence from previously processed sequences.

To accomplish this sequencing, we add another rule to the grammar defining the sequences of events seen in the successful test execution logs. If we add a second event to our hypothetical event log, we will have two events as shown in Figure 12, noting that only the

```
<16>1 2018-05-22T11:32:30.458Z MEDORA-DUT1 intfCmm - - [@aleLog327655 logger="swlogd"
subApp="Mgr"] cmmEsmUpdateInterfaceDynamicInfo: 2/3/14: adminStatus=1, autoNego=1(0,0,0,0),
splitterMode=0, linkStatus=2,linkChangeTime=15270

<16>1 2018-05-22T11:32:30.459Z MEDORA-DUT1 intfCmm - - [@aleLog327655 logger="swlogd"
subApp="Mgr"] cmmEsmUpdateInterfaceDynamicInfo: 2/3/15: adminStatus=1, autoNego=1(0,0,0,0),
splitterMode=0, linkStatus=2,linkChangeTime=15270
```

Figure 12 - Simple Event Log with two events

message text is different between the two events.

After a single test execution with such a log, our list of non-terminals will include something like Figure 13, where $e1$ and $msg1$ represent the differences between the two events:

$$N_b = \{e0, p0, h0, app0, sd0, msg0, sp, dash, ts, fulldate, fulltime, \boldsymbol{e1}, \boldsymbol{msg1}\}$$

Figure 13 - Non-terminal List for Two Events

We must add a new rule to the set of productions to show the event sequence as shown in Figure 14 - Grammar production main rule with two events:

$$mainRule \leftarrow e0\ e1$$

Figure 14 - Grammar production main rule with two events

As each successful event log is processed, new non-terminals may be added if new events are encountered, and the main rule must be augmented to include those new events, and to

produce a new event sequence $mainRule$ which is the union of the previous sequences as shown in Figure 15:

$$mainRule \leftarrow (e0\ e1\ e2\ e3 \ldots en) \mid \quad (e0\ e1\ e3 \ldots en) \mid (e0\ e3 \ldots en\ )$$

Figure 15 - Augmented Sequence Production From Multiple Test Executions

Note that this takes advantage of ANTLR's LL(*) lookahead capability. We can have unions of very similar expressions. When the ANTLR parser encounters a conflict, it will backtrack and try the other expressions in the union until a match is found, which may not work with parser generators such as YACC or Bison.

3.9 Variable Replacement

Because of natural variations in the ambient environment and in a network configuration, there are some variations between otherwise identical events in a log during successive test executions. For example, in an embedded system, a fan speed controller might measure the air temperature and log the measurement in the event log as it makes fan speed decisions and adjustments. IP addresses might vary from execution to execution as addresses are allocated by a local DHCP [80] server on a LAN. A process ID (PID) for a given task might vary from execution to execution. Variations in these fields will prevent the grammar from correctly identifying identical events between test executions, and these variable text strings must be replaced by either a constant string or by some expression that the grammar will interpret as equal.

At the same time, we want the generated grammar to identify events in an unmodified failing execution test log. The generated grammar must, therefore, include expressions that allow translation of the variable fields in the failing log as it is processed.

There are several approaches to this problem. For example, Fu's group [29] used a rough scan of the logs to find the variable strings in the log keys and eliminate them from the processed log strings. Memon [25] generated a report for user examination and allows the user to categorize the changed events for inclusion in his grammar.

The solution we employ parses the event message text into individual words, uses concise regular expressions to identify the strings being replaced, and replaces the strings with an expression that can be processed by ANTLR. ANTLR cannot process complex regular expressions and attempting to introduce them creates ambiguities and maintenance issues in the grammar. But we can use very complex and concise regular expressions to preprocess each event message using the C++ std::regex library and convert these into more limited expressions that ANTLR will process. Since the simpler expressions replace a stronger regular expression within a strictly defined constant string, the chance of an erroneous identification in the failing log file is reduced.

These substitutions are test and application dependent and must be provided by the user. We provide and employ a user-modifiable input text file containing a set of regular expressions that we want to identify along with the corresponding ANTLR expression to be inserted and the definition of that expression. The file format consists of 3-line groups with the form in Figure 16:

```
repl: <regular expression>
replWith: <nonTerminalName>
antlrExp: <nonTerminalString>
```

Figure 16 - Replacement File Format

The <regular expression> is a regular expression in the format recognized by most of the regular expression libraries, in particular the C++ std::regex library. The <nonTerminalName>

is the string that will replace the identified regular expression, and the <nonTerminalString> is the expression that will appear in the output grammar. For example, an IP address can be identified and replaced by the expressions in Figure 17.

```
repl: (25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-
      9]?[0-9])\.(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.(25[0-5]|2[0-4][0-
      9]|1[0-9][0-9]|[1-9]?[0-9])
replWith: ipAddr
antlrExp: number PERIOD number PERIOD number PERIOD number
```

Figure 17 - IP Address Replacement

The complex regular expression labeled “repl” will precisely match an IP version 4 address in standard dot form with limits on the octet ranges. The string labeled “replWith” replaces the matched string with the non-terminal “ipAddr”, and the string labeled “antlrExp” is output in the generated grammar as a non-terminal definition of the form “ipAddr: number PERIOD number PERIOD number PERIOD number”. The grammar contains a non-terminal named “number” identifying a decimal or hexadecimal number, and a terminal named “PERIOD” identifying a period (.).

To complete the example, if an event contains the message text “accepting connection from 192.168.40.4”, it is replaced with “accepting connection from “ ipAddr. The output grammar has rules as shown in Figure 18:

```
msg52: `accepting connection from ` ipAddr;
ipAddr: number PERIOD number PERIOD number PERIOD number
```

Figure 18 - Grammar Rule Variable Substitution

The word delimiters are white space characters, equals symbols (“=”), parenthesis, and the percent symbol.

3.10 Log Trimming

As shown in Figure 19, an event log doesn't start and stop with the beginning and end of a test case. The system under test was operating and generating events before the test case execution started and continued to operate after the test case finished, and the log therefore contains extraneous events forming a junk prefix and junk suffix of events before and after the test case events. We must trim these entries from the event logs of each successful test case before generating the grammar. For the failing test case we must trim the only the prefix since the failing log has an unknown terminating identifier and we cannot tell where it might end.

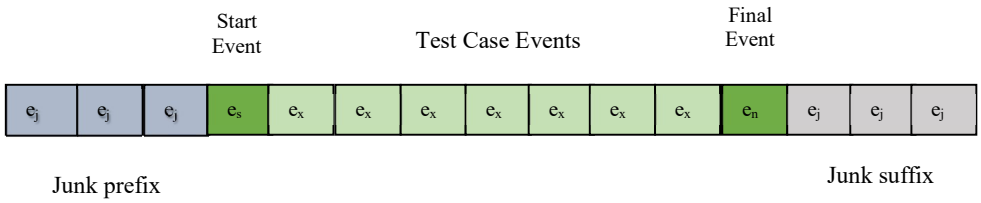

Figure 19 - Event Log Junk Prefix and Suffix

If we assume that the user has trimmed the first log file such that the first event is the test starting event and the last event in the log is the final event in the test case, we can map the event identifier sequence in the first log file onto a long string of wide characters. If we convert the event identifier sequences in the second, third, and nth log files into strings of wide characters, we can use the Levenshtein distance [81] (edit distance) to find the substring in the n^{th} log with the closest match to the first log's wide character string and use that to trim the junk prefixes and suffixes from the later test executions. We can use this same technique to trim the prefix from the failure case log file before using the generated grammar to process it.

If the first, user-trimmed successful event log, L_0 is a sequence of events, each identified by a symbol:

$$L_0 = e_0\ e_1\ e_2\ \ldots\ e_{n-1}\ e_n$$

We can map the events $e_0, e_1, e_2\ \ldots\ e_n$ to a sequence of wide characters, 32-bit numbers, $w_0, w_1, w_2\ \ldots, w_n$ simply by using the index on each event as the wide character.

For the second successful log file, we will have:

$$L_1 = P_1\ e_0\ Q_1\ e_n\ S_1$$

Where

P_1 is a junk event prefix of events $P_1 = e_a\ e_b\ \ldots\ e_x$,

e_0 is the event that starts the test,

Q_1 is the log event sequence from the second test execution starting one symbol after the test starting event and one event before the ending event, a string of events that is similar to the sequence $e_1, e_2\ \ldots\ e_{n-1}$

e_n is the event that ends the test,

S_1 is the junk event suffix appended to the end of the test $S_1 = e_{aa}\ e_{bb}\ \ldots\ e_{xx}$,

We must note that the event string Q_1 is some event string that is like the string $e_1\ e_2\ \ldots\ e_{n-1}$ from the first test but is not equal to it, since the same test is being executed again with some change in timing and system response causing a variation. If the second test event sequence is identical to the first, we can discard it.

If we map the sequence of events from the first log file, $L_0 = e_0\ e_1\ e_2\ \ldots\ e_{n-1}\ e_n$, to a string of wide characters $W_0 = w_0\ w_1\ w_2\ \ldots\ w_{n-1}\ w_n$, and we map the sequence of events from the second log file, $L_1 = P_1\ e_0\ Q_1\ e_n\ S_1$, to another string of wide characters, $W_1 =$

$W_{P1}\ w_0\ W_{Q1} w_n\ W_{S1}$, then if we find the set U_1 of all substrings of W_1 that start with end and with w_0 and end with w_n, and choose the substring $u_{min} \in U_1$ with the smallest edit distance with W_0, we will have the test event sequence that most closely matches the first test event sequence as formalized here.

$$U_1 = \{u_x : u_x \, is \, substring \, of \, W_1 \, such \, that \, t = length(\ u_x\ ), u_x[0] = w_0\ , u_x[t] = w_n\}$$

$$u_{min} \in U_1 \, such \, that \, \forall u \in U_1, u \neq u_{min} : ed(W_0, u) > ed(W_0, u_{min})$$

If we know the starting offset of u_{min} within W_1 then we know the offset of the start of the second test sequence within L_1 and we can trim the junk prefix and junk suffix from L_1.

3.11 Implementation

The process flow for our implementation consists of log file preparation, log trimming, and grammar generation. Log preparation is composed of several steps, and the user must provide for two of them. Similarly, the user must trim the first test case log, but subsequent test case logs are trimmed automatically. The grammar generation is performed automatically.

The log preparation involves translating the event log files from hosts in the system under test into RFC 5424 format, merging the event log files into a single file, then filtering the resulting file to remove data from applications that are not relevant to the user's analysis. The user must provide a program for translating the log for which we provide examples. We provide a short Python program for merging the files. The user must then filter out extraneous events from the resulting merged file, which is easily done using the grep utility or one like it.

All the other steps are automatic and are provided as part of the tool.

The user may also provide the variable replacement regular expression file, although we provide a file that handles floating point numbers, IP addresses, MAC addresses, dates, time stamps and 32-bit integers.

3.12 Log Preparation

The log preparation data flow is shown in Figure 20. The steps are log file translation, log file merge, log file filtering, and then log file trimming.

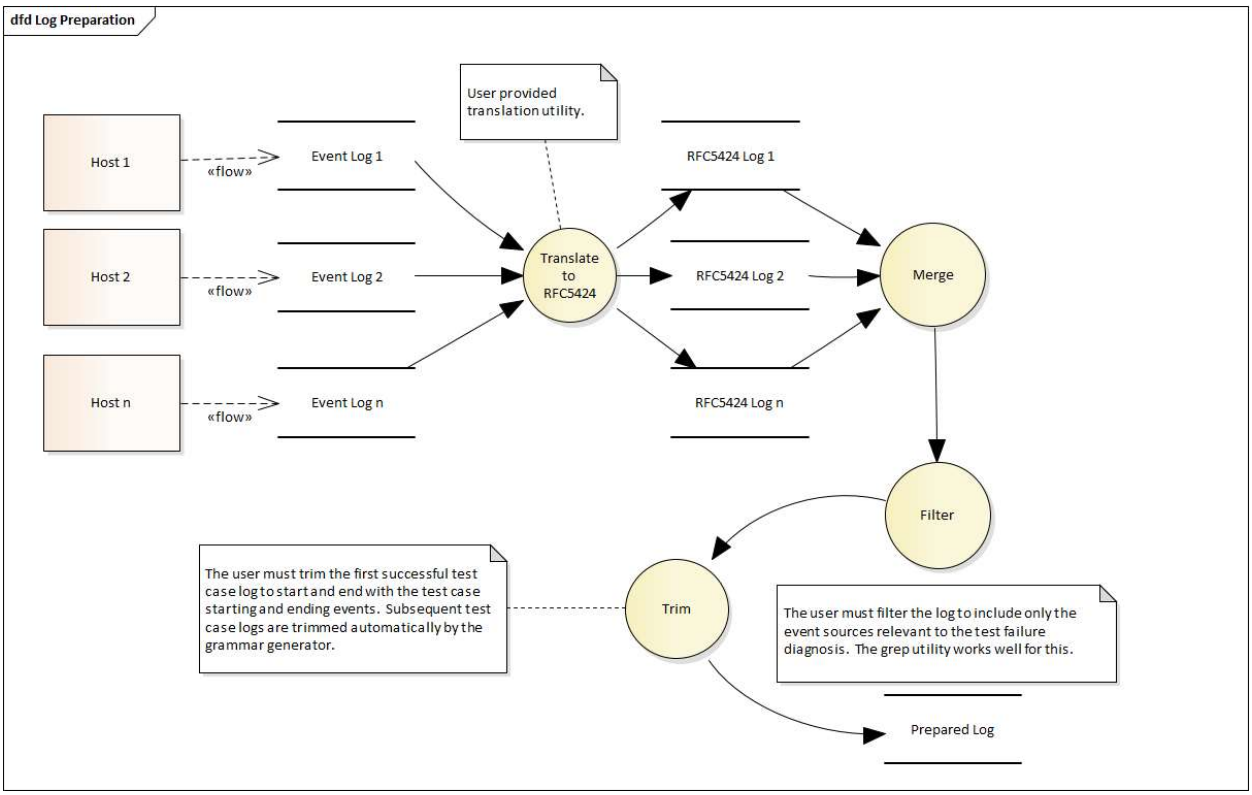

Figure 20 - Log Preparation Data Flow

These steps are largely site specific. The user must provide the translation program from hist native format into RFC5424, although this program should be reusable from one test case to the next, assuming that common log file formats used between test cases and programs writing to the log. The user must also provide the arguments for grep to filter the input file sources to the

set of interest, and the user must trim the first filtered log. We provide a table-driven Python program that will automatically generate BASH scripts to perform the steps above.

We describe these steps in the next sections.

3.13 Log File Translation

The user must provide a program for converting native log files into RFC 5424 form. The commercial and open source log servers, e.g. SPLUNK and Graylog, require a user provided parser for their native event logs. We decided to require user provision of a simple translator program which converts native logs into RFC 5424 form, which our tool will then process.

A simple translator program is much simpler than a parser or grammar. For example, for our experiments, the translator program converting an old BSD syslog derivative format to RFC 5424 form was less than 200 lines of C++ code, much of which is housekeeping such as file I/O and header file inclusion.

Since raw event logs from different hosts must be uniquely identifiable for incorporation into the output grammar file and those hosts might be running the same programs, our example translation program allows input of a user parameter, which is output in the RFC5424 structured data on each event. If the events on different hosts generated by the same application are identical, a host identifier can be inserted into the structured data to differentiate logs from different sources before the log is processed as shown in the highlighted text in Figure 21.

```
<16>1 2018-05-22T11:32:30.458Z MEDORA-DUT1 intfCmm - - [@aleLog327655 host="chassis1_CMMA"
  logger="swlogd" subApp="Mgr"] cmmEsmUpdateInterfaceDynamicInfo: 2/3/14: adminStatus=1,
  autoNego=1(0,0,0,0), splitterMode=0, linkStatus=2,linkChangeTime=15270
```

Figure 21 - User data inserted during log translation

3.14 Log File Merging

After translation to the RFC5424, a common, provided merge program can be used to combine the event log files from multiple hosts into a single large event log for incorporation into the grammar. This is a very short Python program using the heapq library functions to merge the logs using the timestamp in the log files. It is only 79 lines in length including white space and comments.

Because the Python library reads the entire file into memory, this small merge program may have the scalability issues like those described in section 4.3 below.

3.15 Log File Filtering

When a test execution fails, the failure is assigned to a specific developer for analysis. Only that developer knows which applications are pertinent to the failure, and we leave it to that developer to remove the events from "uninteresting" applications.

This can be done with a simple grep command, selecting only those events containing application names that are of interest to the inquiring developer.

There are other utilities available, like the Linux log file navigator, *lnav* [66], that can be of assistance in this in finding the strings for use in the grep filtration command line and can even perform the filtration itself. But *lnav* requires SQL select statements, and grep is easier to use than SQL, at least for those not regularly using SQL. And, of course, to execute an SQL command like "select * from syslog_file where log_procname = 'ChassisSupervisor'" one must already know to search for the string "ChassisSupervisor", so use of a log file navigator like *lnav* may be counterproductive. It is, therefore, expected that the user knows which application events to choose to begin the analysis.

Figure 22 is an example of a grep command line for filtering four applications from a merged log in a Linux environment.

```
grep -i -e chassissuper -e mip_gateway -e bfdcmm -e ses log1.log > log1.filtered.txt
```

Figure 22 - Grep filtering command example

3.16 Log File Trimming

As mentioned in section 3.10 above, the user must trim the merged log file from the first successful test execution such that the first and last events in the file are the first and last events of the test case. This must also be done by the user. For the second and following successful test executions, the log files are trimmed automatically by the tool.

3.17 Log Preparation Script File

Of course, all these steps except the trimming can be put into a script so that the user need not key them in repeatedly. But since the filtering and translation must be provided by the user, a general purpose script is not possible unless we force the user to use a fixed set of programming and executable file naming conventions.

We chose not to enforce such restrictions, but we do provide an example Python language program that inputs a configuration file and outputs a BASH shell script that will execute the log file preparation steps using the parameters in the configuration file. This program also outputs a BASH shell script that executes the grammar generation and failure file processing described in following sections.

3.18 Grammar Generation

The grammar generation involves processing the log files from each successful test case, which builds up a set of intermediate files containing the unique event data and generates an ANTLR grammar file. This grammar file is then fed as input to ANTLR, which generates a set of C++ source files for the successful event log parser. These files are, in turn, compiled and linked.

The failing test case event log file can then be input by the generated parser and its internal DFA will reject the failing event log, stopping on the first event that is out of sequence.

The grammar generation itself process is shown in Figure 23 and in Figure 25.

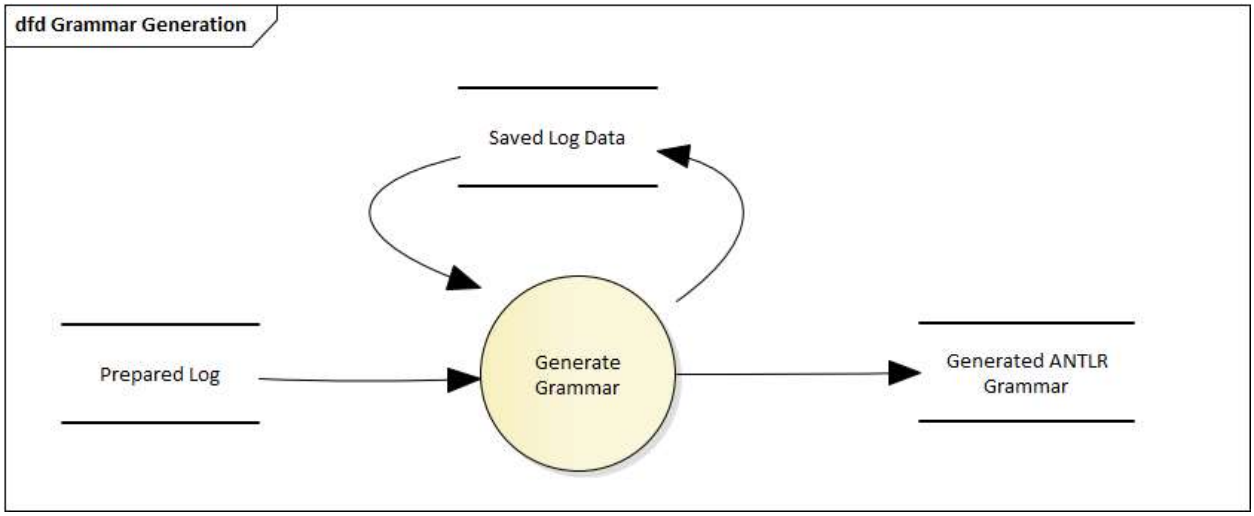

Figure 23 - Grammar Generation Data Flow

As each successful event log file is processed, a set of intermediate data files is constructed as shown in Figure 24. These data files contain, in sequence, the unique RFC 5424 fields from all the events after the variable substitution. As each subsequent successful test case

log is processed, the events in the subsequent logs are matched against the previously discovered events.

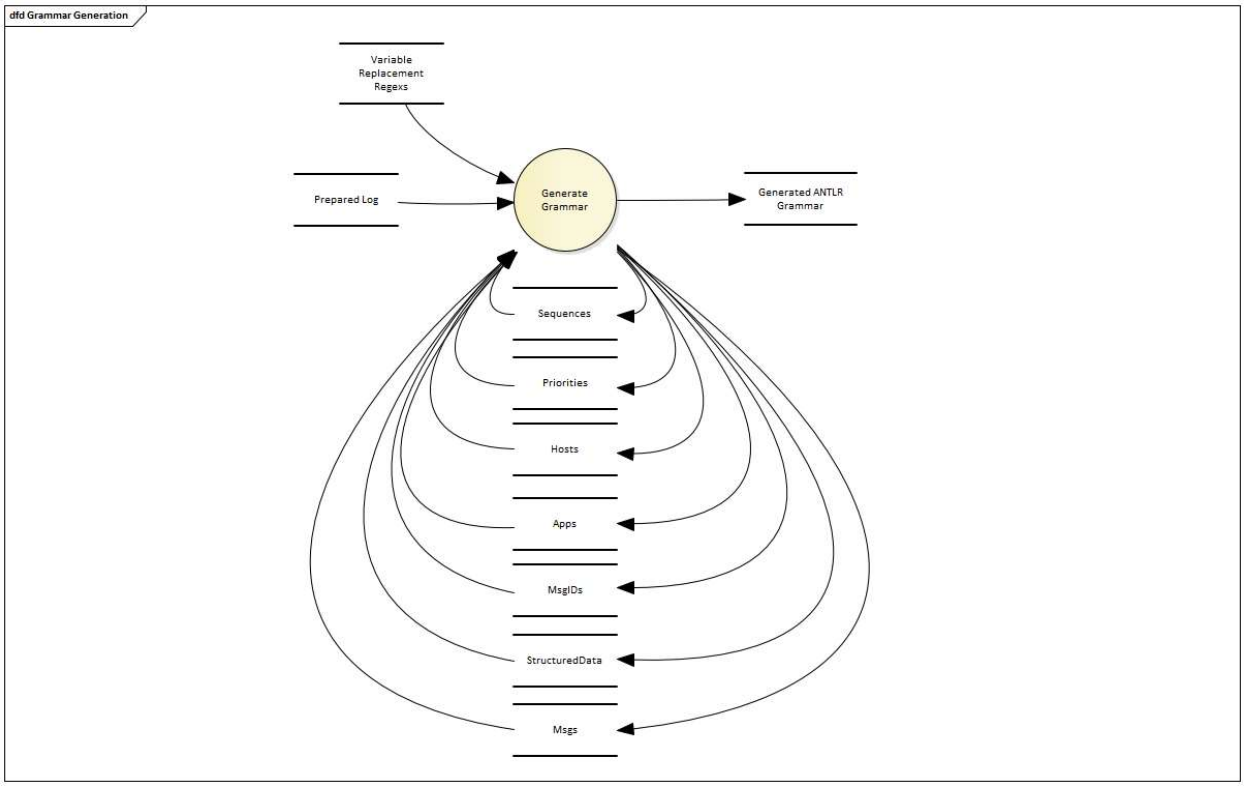

Figure 24 - Grammar Generation

The individual log file record processing during grammar generation is illustrated in Figure 25. When the grammar generation starts, the data from previous successful test logs are read. Then as each event from the test is read and parsed into fields, the variable data are replaced by regular expressions, and the event is either matched against a previously seen event or added as a new event to the data lists.

At the end of the event log file, the event sequence is trimmed to remove the junk prefixes and junk suffixes and an ANTLR grammar file is generated with a new set of event

sequences and event definitions. The main rule in the generated grammar is the union of the sequences from the previously processed logs and the latest event log processed.

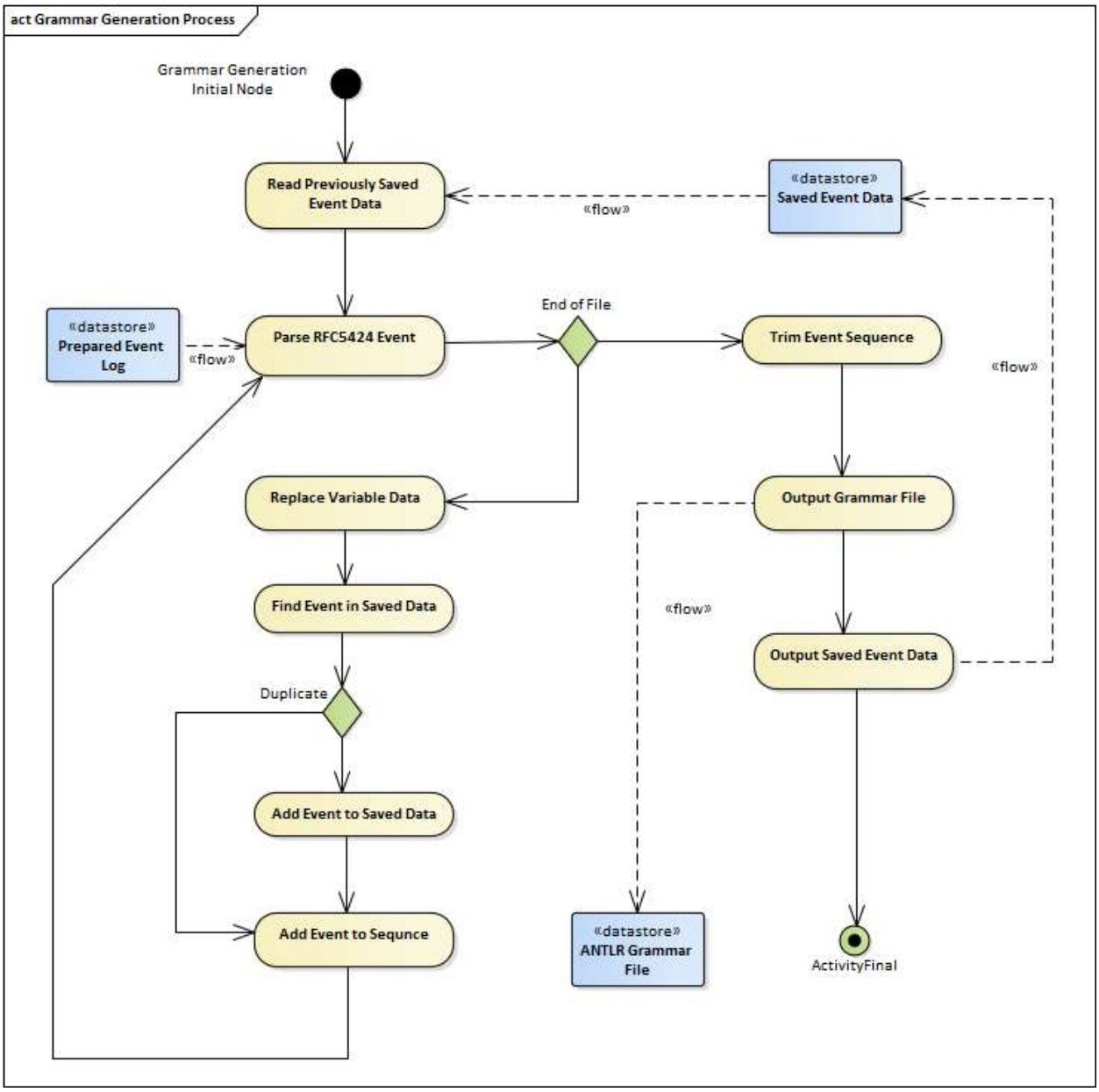

Figure 25 - Grammar Generation Process

When the log from the failing test case is processed, it too must be trimmed to remove its junk prefix and junk suffix. We employ our RFC5424 grammar to process the failing test log

into events and trim it before passing it to the generated grammar. The trimming process is shown in Figure 26.

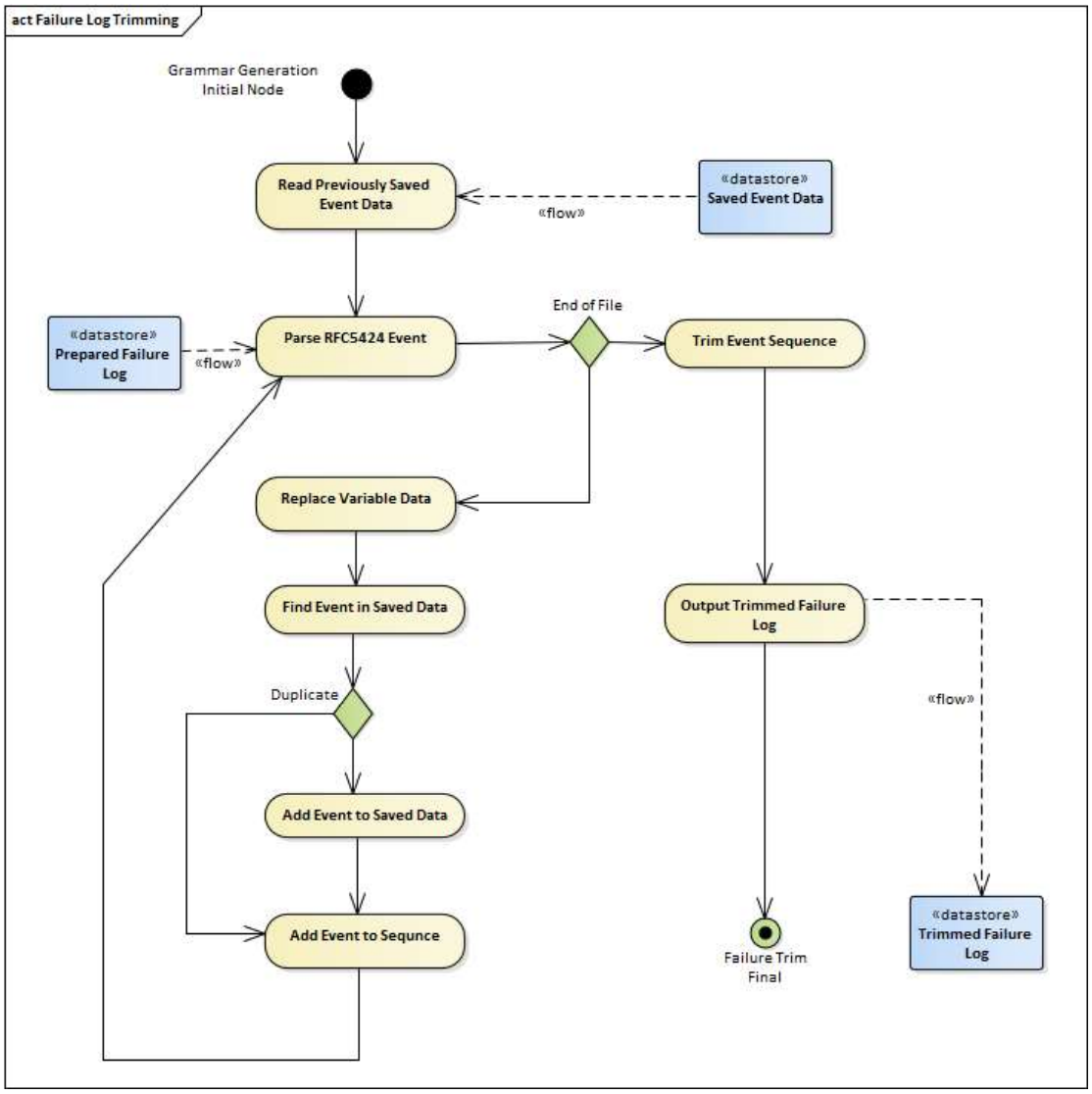

Figure 26 - Failure Log Trimming

3.19 ANTLR Grammar Class Structure

The ANTLR runtime library and the ANTLR generated C++ code provide the basic parsing functions and the base classes for grammar-specific processing.

The class structure is shown in Figure 27.

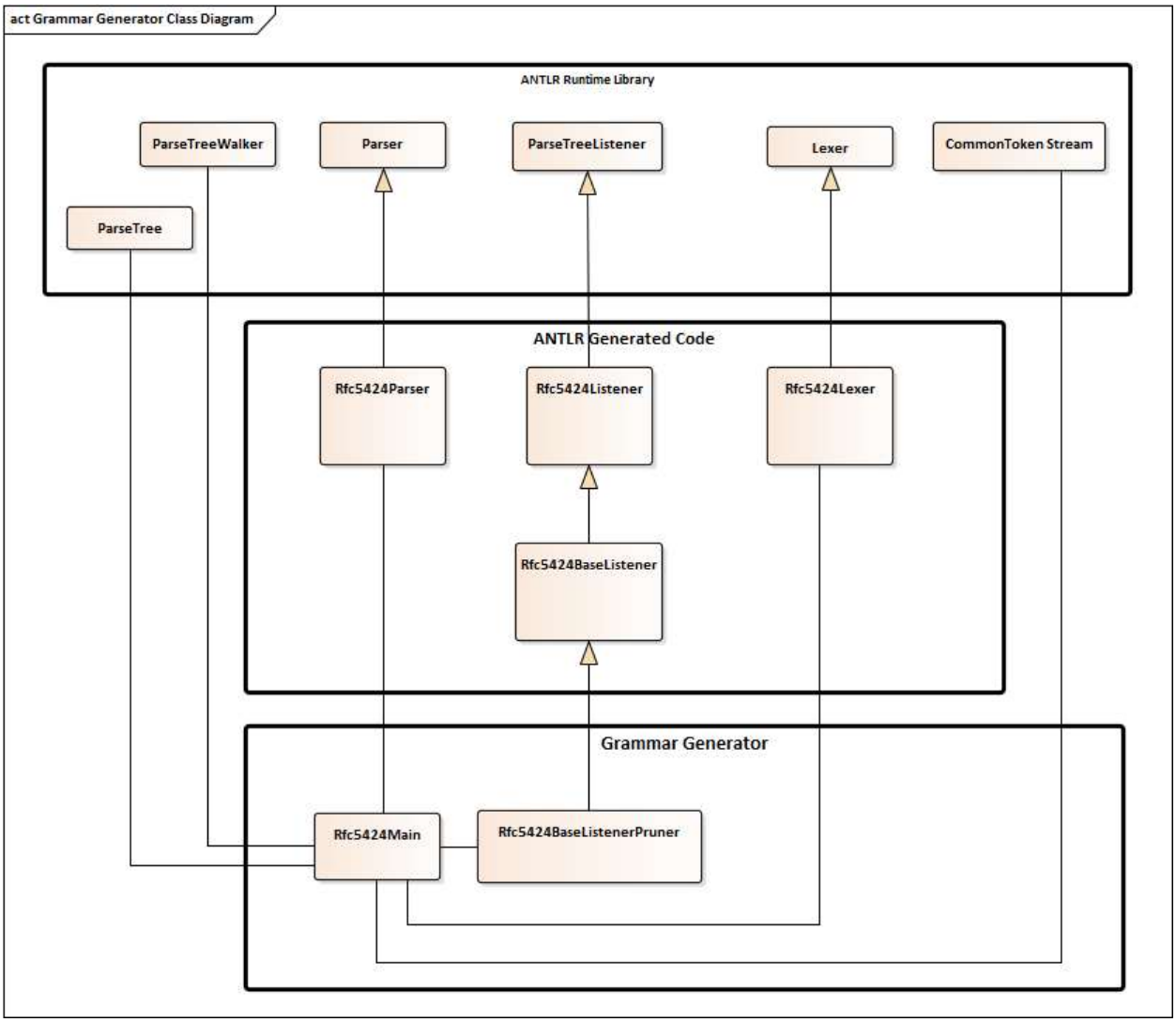

Figure 27 - Grammar Generator and ANTLR Runtime Components

The ANTLR runtime library contains the base classes for the parse tree, the parse tree walker, the parser, the listener, the lexical analyzer, and the token stream. The listener contains virtual functions that are executed whenever a rule in the grammar is detected. It is "listening" for the grammar rules as they are processed by the parser. When an ANTLR grammar like our

RFC5424 grammar is processed ANTLR generates grammar-specific subclasses for the parser, listener and lexical analyzer.

We provide a main module and a subclass of the listener. The main module processes command line arguments and creates the parser and parse tree walker objects. It also creates our sub-classed listener object, then calls the tree walker to start the parsing.

The sub-classed listener, RFC5424BaseListenerPruner, is the grammar generator. As a sub-class of the ANTLR generated RFC5424BaseListener class, it contains virtual functions that are called as each rule in the RFC5424 grammar is entered and exited. These virtual functions create and manage all the intermediate files described above, identifying the events, and recording the event sequences in each of the successful test log files.

When the grammar main rule exits, the exitMainRule virtual function uses the stored data to trim the event sequence, creates the output grammar file and rewrites the intermediate files from its stored data. The intermediate files are then available for processing the next successful test event log.

As mentioned previously, the edit distance algorithm is a modification of [82] to input wide-character strings and to compile and link with C++ 14. The ANTLR grammar is a modification of [46].

3.20 Generated Grammar

As described above, each unique event in the event log is identified as a concatenation of a set of strings, each string corresponding to a field in the RFC5424 event record except for the timestamp, which is used only for sorting the logs in the merge step. Each unique string set is

assigned an identifier that reflects the position in the log where that event was first detected (e.g. e0, e1, e2, etc).

The generated grammar contains a top-level rule, mainRule, which is a union of sequences of these identifiers. This main rule defines the DFA, the finite state recognizer for the successful test event logs, which ANTLR minimizes when the generated grammar is processed and transformed into the LL(*) recognizer for the grammar.

A section from a generated grammar is shown in Figure 28.

```
grammar xxx;

mainRule:
(
e0
e1
e2
…
);
e0: pri0 SP timestamp SP host0 SP appId0 SP procId0 SP msgId0 SP structuredData0 msg0 NL ;
e1: pri0 SP timestamp SP host0 SP appId0 SP procId0 SP msgId0 SP structuredData0 msg1 NL ;
```

Figure 28 - Generated Grammar Segment

When ANTLR processes the generated grammar, it outputs another set of C++ files, which are used to process the failing event logs directly. But the failing log file must also be trimmed to get rid of its junk prefix. Since the failing log terminates abnormally, it will not contain the terminating event, and we cannot remove its junk suffix. We can, however, trim the prefix to the correct starting event with the closest edit distance match to the first successful test case for the rest of the main rule sequence.

As mentioned above we must trim the failing test case log and we provide a trimming function within the RFC5424 listener subclass that will parse the failing log into a sequence and

use the main rule sequence to find a sequence that starts with the starting event and has a minimal edit distance between the rest of the log file and the generated grammar's sequence. The trimming function writes a new event log that starts with the correct starting event instead of writing out a modified grammar.

The processing of the failing event log is shown in Figure 29.

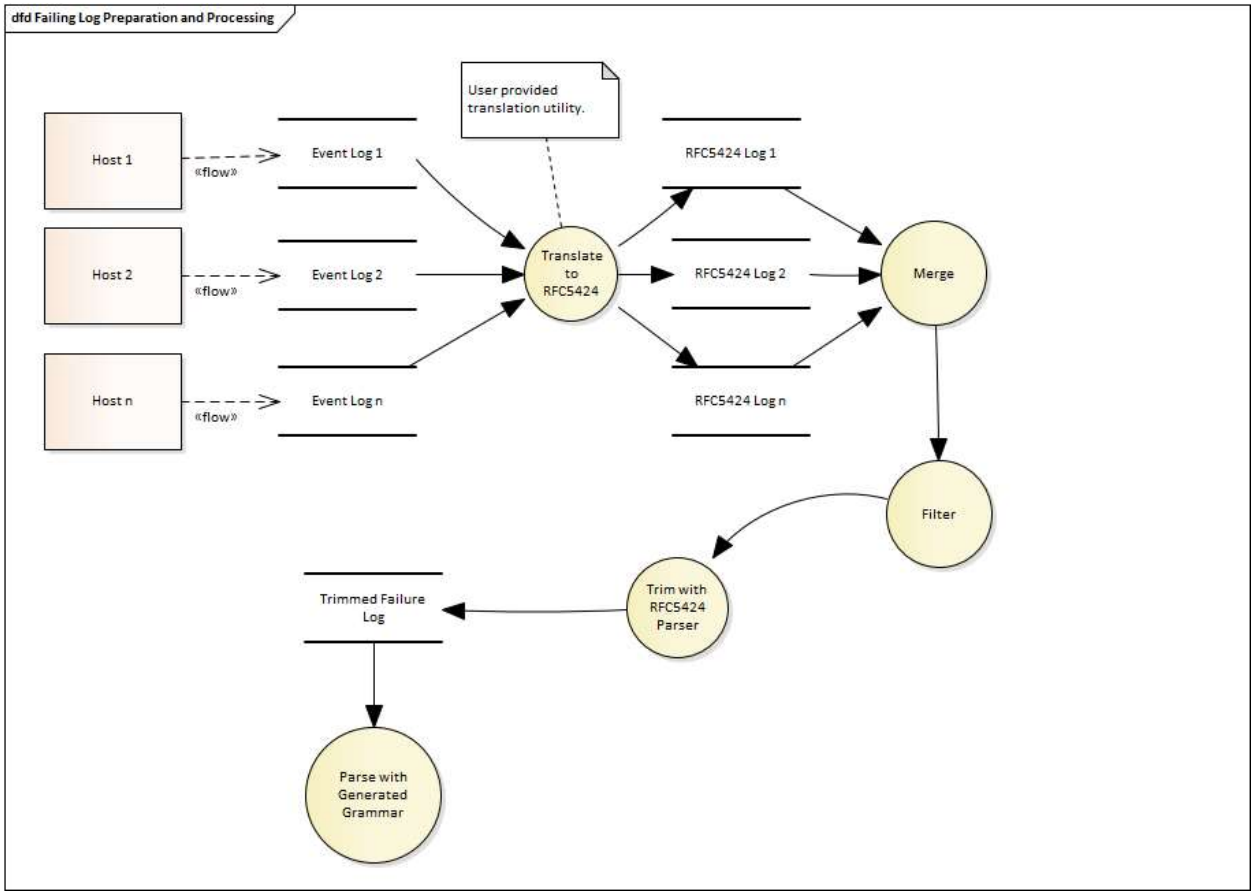

Figure 29 - Failing Log Preparation and Processing

The difference from the successful log preparation and processing is that instead of moving on to generate a grammar, we output a trimmed failure log that the previously generated grammar can process.

3.21 Error Handling

Once a deviation is detected, it must be reported in a useful manner. Useful in this case is locating the point in the error log file where the deviation occurs and displaying what was expected vs. what was found.

The ANTLR Java and C++ default error handler functions are overridden in our generated log processor program. The function arguments provide the list of rules processed before the error was detected, as well as the line number and character offset within the line where the parsing halted. This information and the trimming event line numbers are all we need to provide our user with the event log context to begin problem diagnosis.

Since we know that the error log matches at least one of the successful test logs up to the point of failure, we can display the lines from that offset adjusted by the trimming line numbers from the successful test logs to give the user an idea of what the error test case should have recorded.

When the grammar halts, we execute the emacs text editor a command to position to the line in question. We open other emacs windows showing the lines at the same offset in the training log. See section 3.22 on the user interface below.

3.22 User Interface

The user interface is provided through a Python language programs and a set of text editors. The Python language program accepts a list of successful test log files, the failing log file, and a grammar name. It then executes the rest of the process in the proper sequence. It passes the successful logs to the main program for generating the failure log

ANTLR grammar file, compiles the generated grammar into C++ files, compiles and links the generated C++ files into an executable program, and then executes the program with the failure log as input. When the program halts on the failure log, the Python program parses the output and opens text editor windows for the input files at the line number where the generated C++ program stopped.

```
./processTestLogs.py -g april23 -i "apr23CS.log apr23CS-1.log" -f apr23CSBad.log
grammar name is  april23
file names are  apr23CS.log apr23CS-1.log
file list is  ['apr23CS.log', 'apr23CS-1.log']
command is  ./Rfc5424Parse -i apr23CS.log -o april23.g4 -n
input file is apr23CS.log
output file is april23.g4
Removing temporary files
command is  ./Rfc5424Parse -i apr23CS-1.log -o april23.g4 --trimInput
java -jar /usr/local/lib/antlr-4.7-complete.jar -Dlanguage=Cpp april23.g4
sed 's/xxx/april23/g' templateMain.cpp > april23Main.cpp
g++ -std=c++0x -g -c april23Main.cpp -I. -I /mnt/c/PraxisResearch/antlr/antlr4-cpp-runtime-
4.7.1-source/runtime/src
g++ -std=c++0x -g -c april23Listener.cpp -I. -I /mnt/c/PraxisResearch/antlr/antlr4-cpp-
runtime-4.7.1-source/runtime/src
g++ -std=c++0x -g -c april23BaseListener.cpp -I. -I /mnt/c/PraxisResearch/antlr/antlr4-cpp-
runtime-4.7.1-source/runtime/src
g++ -std=c++0x -g -c april23Lexer.cpp -I. -I /mnt/c/PraxisResearch/antlr/antlr4-cpp-runtime-
4.7.1-source/runtime/src
g++ -std=c++0x -g -c april23Parser.cpp -I. -I /mnt/c/PraxisResearch/antlr/antlr4-cpp-
runtime-4.7.1-source/runtime/src
g++ -o april23Parse -g april23Main.o april23Listener.o april23BaseListener.o april23Lexer.o
april23Parser.o /mnt/c/PraxisResearch/antlr/antlr4-cpp-runtime-4.7.1-source/dist/libantlr4-
runtime.a
Process error command is  ./april23Parse -i apr23CSBad.log.trimmed --stopOnError
Error scan  ['input file is apr23CSBad.log.trimmed\n', 'output file is
default_outfile.txt\n', 'Line: 940 : 112\n', "Error: Can't choose between alternatives\n"]
```

Figure 30 - Example Log Processing Execution

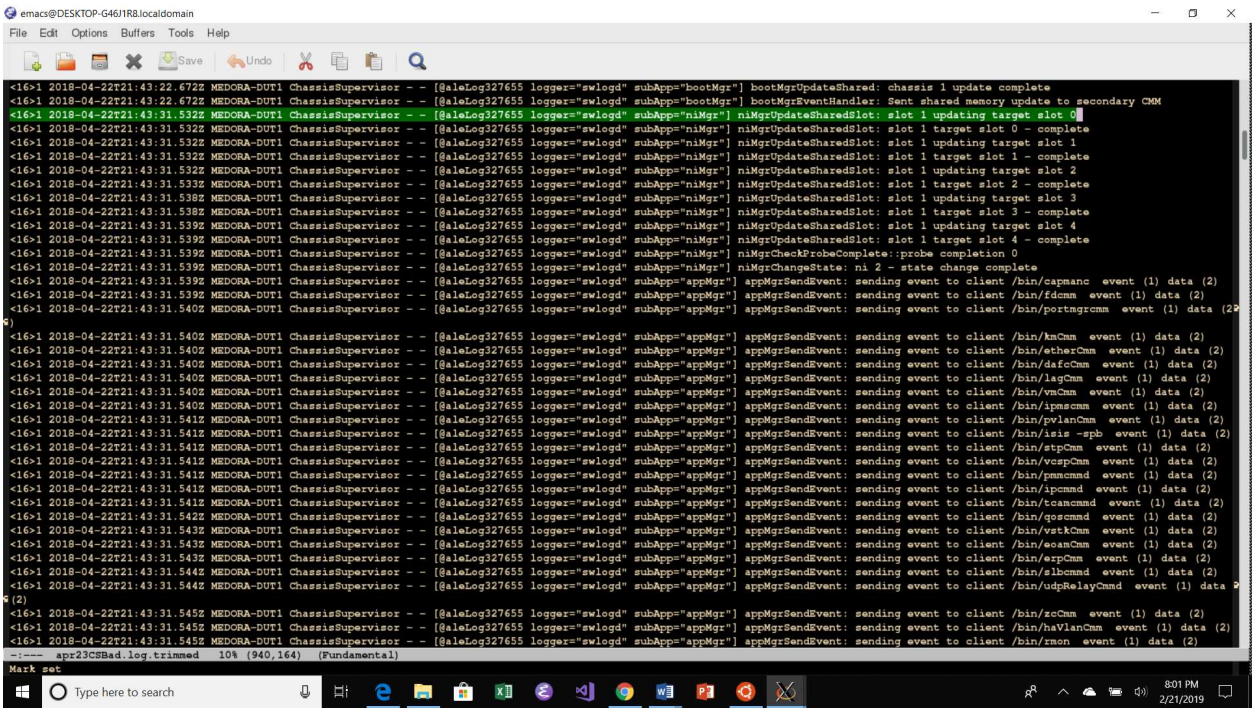

Figure 31 - Text Editor Window Positioned to Log Failure Point

Chapter 4

TESTING

4.1 Testing Methods

The tool was tested with event log files taken from linked network packet switches running integration tests and system tests in a software test lab for a commercial product. Failure cases were generated by injecting errors into the logs from successful tests either by removing events, rearranging events, and by adding previously unseen events. In addition, failure cases reported in the lab by the test organization were employed when such logs were available.

To test the log trimming existing prefix and suffixes from the gathered test cases were left in place, with the first log in the test sequence being manually trimmed as described above.

A total of 8 test cases were executed and the execution times for each processing step are shown in Table 2.

Table 2 - Test Characteristics and Execution Times

Test #	Number of Test Executions	Events per Test Execution (average)	Generated Grammar Size (Lines)	Log Processing Time (Seconds)	Compile Time (seconds)	Total Time
1	3	4406	16932	20.96	232.45	261.97
2	4	1444	7447	12.51	78.87	95.16
3	5	1766	11003	20.284	107.63	128.58
4	4	1590	8769	47.52	129.49	179.09
5	2	4556	12981	23.02	280.98	312.34
6	4	8290	17592	1495.784 (25 Minutes)	57.61	1,556.22 (25.93 Minutes)
7	2	10435	23617	49.46	216.13	271.99
8	2	240	1412	5.65	35.59	42.19

All tests were executed on a Dell Inspiron 5570 laptop computer with an Intel Core i7-855U CPU running at 1.8 GHz and 16 GB of installed RAM. The tests were executed in a Ubuntu Linux 16.04 virtual machine running under 64-bit Microsoft Windows 10 Pro.

4.2 Performance

This method is intended to assist in locating of deviations from normal sequences in logs from a failing test when logs from one or more successful executions of the same test are available. We use the logs from the successful tests to create a finite recognizer for the successful test cases. We then use that recognizer to find where the failing test case deviates from the successful test execution. Since it is intended to aid the developer in real-time, the tool must generate output quickly from the collected event logs.

As shown in Table 2, the grammar generation and test log processing took less than six minutes in most cases, which is acceptable. The exceptions to this were in test cases where the log contained many hundred instances of the starting event, causing hundreds of executions of the log trimming algorithm. Since this algorithm is of complexity $O(n^2)$ over the size of the log, the log trimming time dominated the processing time. But even so, it still took less than 30 minutes, which is not extreme.

4.2.1 Log Preparation Time

The log preparation is composed of log translation, log filtering, and log merging, and trimming the log for the first successful test case. We provide software for all of these steps except for the initial log trimming. The user is expected to provide a program for log translation into RFC 5424 form, but this program need only be created once per organization, since the logs within an organization will most likely share a common form.

Since most of our test cases were created artificially by injecting differences and then errors into successful test case logs taken from an industrial packet switch, the log preparation time for creating those test logs is skewed. We were, however, able to collect and separate multiple successful execution logs from two problem reports in a commercial test environment.

The log preparation for both of these cases took in excess of thirty minutes each, which is acceptable but still somewhat burdensome.

4.2.2 Log Trimming Performance

We employ Levenshtein's Edit Distance algorithm [81] to find the wide-character substring of a successful test case log event sequence giving the closest match to the wide-character sequence in the manually trimmed successful test case log. We do this by assigning a symbol to each unique event in the two logs and compare those symbols instead of the event text strings themselves.

For example, if a log contains 1,000 lines with 200 of those being unique, then we have a sequence that is 1,000 symbols in length with 200 unique symbols in the sequence. We compute the edit distances using these symbol strings instead of the individual characters in the lines in the event log.

This algorithm runs in $O(mn)$ time and in $O(m+n)$ space, where m is the length of the first string and n is the length of the second string [83]. We reduce the number of executions of the algorithm by only considering strings that start and stop with the same symbol as first and last symbols in the manually trimmed test case. We search from the start of the log for the first symbol and work backwards from the end of the log to find the last symbol.

If the first and last symbols are uncommon in both logs, which one expects in a test case with some starting command and some ending result, then the number of executions is relatively small. If the first and last symbols are common then the number of executions increases, increasing the time required to trim the log.

We tried this with logs with various characteristics and recorded results one would expect as shown in Table 3.

Table 3 - Log Trimming Time

Test Number	Number of Events	Number of start symbol repetitions	Total Log File Processing Time	Trim Execution Time	Log Processing Time
1	258	1	42.19 Seconds	< 1second	5.65 seconds
2	10435	3	272 Seconds	4.03 seconds	49.46 seconds
3	8290	719	1,556.22 (25.93 Minutes)	1476.31 seconds (24.6 minutes)	22.3 Seconds

The total log processing time includes the generated grammar C++ compile and link time, which on larger files is appreciable. Even for shorter log files, the C++ compilation time was larger than the log file processing itself. The log processing time includes the trimming time.

Test number 3 above demonstrates that proper choice of starting events is important. If the test case log contains multiple instances of the starting event, the trimming time goes up substantially because of the number of executions of the edit distance algorithm.

Each execution of the edit distance algorithm on a 10,000 event file takes approximately 1.6 seconds to execute on a 1.8 GHz Dell Inspiron Laptop with 16 GB of RAM. In test case 4, there were 719 instances of the start symbol in the failing test case log, and the trimming took more than twenty minutes.

4.3 Scaling issues using ANTLR

Scaling problems with even small logs showed themselves early during this project. The first problem was with the generated Java language code. ANTLR's default output language is Java, and ANTLR creates one Java class for each rule in the grammar, wrapping these within a larger containing class. This causes the generated Java source files to balloon in size, not to mention creating hundreds of ".class" files from a single source file during compilation. The Java virtual machine limits an individual Java member function to 65,534 bytes [84][4], and the ANTLR generated Java code would not compile for even relatively small log files.

Because of the Java language limit we moved to generating C++ code instead of Java. ANTLR version 4 will generate C++ code, and C++ does not have this method size limitation. The GNU C++ compiler, g++, version 5.04 will compile generated parser files for grammars generated from 10,000 line log files. But we run into difficulties elsewhere. With a file greater than 30,000 events in length, the C++ compiler gets segmentation faults when processing the ANTLR generated parser files because ANTLR generates strings of constants larger than the g++ compiler's internal limits.

There are scaling issues within ANTLR itself. With files of 40,000 events the Java runtime library throws an array element out of range exception during the C++ code generation. If there are too many unique log messages we see similar Java runtime failures. And with very large log files there are memory exhaustion issues on a 16 GB laptop running Ubuntu 16.04 as the parse tree grows larger than the memory size. With an average event length of 150 bytes,

[4] From the Java Virtual Machine specification"The *fact that end_pc is* exclusive *is a historical mistake in the design of the Java Virtual Machine: if the Java Virtual Machine code for a method is exactly 65535 bytes long and ends with an instruction that is 1 byte long, then that instruction cannot be protected by an exception handler. A compiler writer can work around this bug by limiting the maximum size of the generated Java Virtual Machine code for any method, instance initialization method, or static initializer (the size of any code array) to 65534 bytes."*

ANTLR runs out of memory with about 200,000 events. ANTLR reads the entire input file into the system memory and there will, therefore, always be memory exhaustion issues for files that approach the system memory size. Terence Parr and the ANTLR authors provide unbuffered streams in their Java runtime to reduce ameliorate this problem, but the functions did not make it into the ANTLR C++ runtime library as of this writing (see [85] pages 246-248).

There is also an issue with the ANTLR DFA data structures. The ANTLR runtime employs a C++ standard template library bit set (std::bitset) within each DFA node to mark transitions on a given event. When the run-time library is compiled, the number of bits in a bitset is included as a fixed constant. If the number of distinct events in a log exceeds the number of bits in the bit set, then log file processing will fail with a run time exception. The ANTRL C++ runtime source specifies the bitset size as 1024 bits.

Depending upon the user's environment, this number may be exceeded in moderately large logs, depending, among other parameters, how many applications outputs are included in the filtered log and how the variable fields in each event are replaced as the events are processed. This was the case in one of the test cases in Table 4, with more than 10,000 events in the logs, and more than 1024 distinct events. To avoid this limitation we recompiled the ANTLR runtime with a larger constant in its bitset definition, increasing the constant from 1024 to 4096.

For our system test cases, a 30,000-line combined log file maximum is adequate since the logs are filtered down to events generated by a subset of the applications executing during the test. Table 4 illustrates event counts for individual applications included in a small set of error reports or test cases on a multi-chassis network packet switch. The tests all failed after more than one successful iteration, and the event counts are averages.

Table 4 - Test Case Application Event Counts

Problem Report #	CS	VCM	INTFCMM	VLAN MGR	Port Mgr	SVC CMM	STP CMM	IPV6	ISIS VC
1	2226	755	1737	62	93	182	133	105	101
2	2195	655	1721	462	131	129	946	11	103
3	3509	1303	2116	100	186	293	240	56	153
4[5]	360	125	287	32	26	101	94	86	13

In short, these methods work with log files containing less than 30,000 events provided that the number of unique log messages per test is less than 2,000. If we must process test sequences with more than 30,000 events for a single application subset, we must avoid generating a very large C++ lexer and parser file. See chapter 5 below for alternative methods which can handle larger event log files.

4.4 False Positive and False Negative Indications

Because the generated grammar is not, and indeed cannot be complete the problem of false positive and false negative detection must be addressed.

A false positive is produced when the log file from the failing test execution event sequence diverges from the successful test execution sequences in multiple places before failing outright, where one of the branches is from a previously unseen but normal timing variance. If the first divergence is from such a normal divergence that doesn't indicate an error, it must be ignored and the second divergence employed for determining the actual failure. The generated DFA will detect at least two errors and the user must know to examine both of them.

A false negative occurs when the DFA ignores a divergence that is an actual error. This might occur if a successful execution masks the error in the failing execution. When a test case

[5] This was a regression test consisting of a sequence of 75 tests executed consecutively. The data shown reflect an average over the 75 tests.

halts with an error, we know that the test script program detected the error and the test didn't complete successfully. But the false negative in our tool indicates there is no divergence in the log from the successful test executions. The DFA was traversed from its starting state to its final state without an error, and therefore the event sequence exactly matched one of the successful test execution logs.

This may happen if the user interface or system output read by the test script is incorrect, but the flow of events in the log follows one of the successful test cases. This is actually fairly likely in the case of a simple computational programming error, and the use of our method to locate the source of the error may not be warranted. Only the developer examining the error report can determine if this is the case.

To verify the false positive cases we executed 50 random line insertions and deletions on a log file with 1080 events and executed the generated parser for that log file test case. For single line insertions and deletions, there were no false positives, which is what was expected. For multiple line insertions and deletions, the generated parser halted on the first insertion.

Chapter 5

CONCLUSIONS AND FUTURE WORK

We have demonstrated that the methods we employed for locating faults in event logs are effective within certain limits. That is, given a set of event logs from successful test executions in a specific test environment, we can use those logs to create a recognizer for those successful test executions that will halt and display the area in the logs where the failure occurred. We have demonstrated processing the logs into that recognizer and processing a failing log can be done in a few minutes for filtered log files on the order of 10,000 events.

But we have sidestepped a few issues. Test coverage is one, and scalability another. And another issue is acceptance within a software development organization, a process and corporate political issue.

Test coverage, the question of whether the DFA derived from successful test logs is robust enough to ensure that the generated DFA halts on a real deviation rather than on some legitimate event is difficult. As part of our problem definition we know that the log from the failing test contains a failure. We also know when the failure is detected by the generated DFA while processing the failure log that there has been some behavioral deviation from the successful cases. We do not know whether the first point of deviation detected by the DFA is the start of an actual failure, or if the real failure occurs somewhere later in the event log. Because test budgets are limited, resources are limited, and the number of test executions necessary to get complete coverage is unknown, increasing the certainty in this area is expensive. We also know from other work that deriving a complete grammar for a large text only from successful test cases is impossible.

The scaling problem described in section 4.3 presents an opportunity for improvements. For filtered files less than about 200,00 events we can still allow ANTLR to create the in-memory parse tree, but we must generate a simpler grammar and employ an external DFA to determine where the failing test log goes awry.

For the external DFA, there are several open-source implementations available. An elegant Java implementation by Anders Moeller [86] at Aarhus University, "bric.dk", is available and will process Unicode characters in its alphabet. Unicode is a 16-bit character encoding, allowing up to 64,000 symbols, or 64,000 different log messages in our grammar. The bric.dk DFA can be called from our C++ listener classes using the Java Native Interface (JNI) [87], or we could switch our generated listener code to Java and call it directly.

Alternatively, a partial C language implementation of the bric.dk finite automaton, libfa, is available as part of David Lutterkort's Augeas package [88]. But this finite automaton employs only 8-bit UTF-8 characters, which are not sufficient for our purposes. It appears to be modifiable to use C-language wide character types (wchar_t) and therefore useful as a recognizer. Linked with the generated C++ parser and lexer, the libfa DFA can be used to process the very large expressions that recognize an event log file.

Implementing either of these involves a modified data flow:

1. Translate and filter the successful test log as above.
2. Use the Rfc5424 grammar with ANTLR to process the successful test log files, creating the internal databases identifying each log event as above.
3. Functions for processing the individual records in the file are very similar to the implementation above, except that the exitMainRule function (see above) does not

generate a grammar. It simply writes out the internal database for all the events processed including the sequences of log messages encountered in the successful test cases.

4. The log file trimmer will work as in the pure ANTLR grammar above.
5. The failure log processing employs a fixed ANTLR RFC5424 grammar with listener functions that
    a. Read the generated internal files identifying the previously discovered events and sequences.
    b. Construct a DFA for the discovered sequences by creating and minimizing a union of the sequences from the successful test cases.
    c. Identify each input event as it is read from the failure log and passes it to the DFA functions where it is either accepted or rejected.
    d. If the input event is rejected by the DFA it identifies the point of failure.

The state of the DFA contains a set of expected transitions which are reported to the user allowing output such as "unexpected event <event text> - expected <event list>". This method allows reuse of almost all of the code we have written for the pure ANTLR implementation.

The variable list replacement is the same as in the first method, except that the variables must be replaced by regular expressions instead of ANTLR grammar rules For example, for the smaller event log files, we might have an RFC 5424 msg rule:

```
msg403: 'This message embeds a MAC address (' macAddr ') in the middle';
macAddr: hexByte COLON hexByte COLON hexByte COLON hexByte COLON hexByte COLON hexByte;

hexByte : hex_0_0 | hex_0_1 | hex_0_2 | hex_0_3 | hex_0_4 | hex_0_5 | hex_0_6 | hex_0_7 |
hex_0_8 | hex_0_9 | hex_0_a | hex_0_A | hex_0_b | hex_0_B | hex_0_c | hex_0_C | hex_0_d |
hex_0_D | hex_0_e | hex_0_E | hex_0_f | hex_0_F |" …

hex_0_0: ZERO ZERO
```

Figure 32 - Example RFC 5424 variable replacement rule

Using this method, we eliminate the macAddr symbol and replace it with the regular expression recognizing MAC addresses:

```
msg403: 'This message embeds a MAC address (' ([0-9A-Fa-f]{2}[:-]){5}([0-9A-Fa-f]{2}) ') in
the middle'.
```

Figure 33 - Variable replacement with regular expressions

We must also scan the literal parts of each message and escape the regular expression delimiters before they are stored. And for efficiency we must pre-compile the regular expressions as they are read. The original grammar regular expressions were named substrings, but the regex library won't work with those, requiring an in-line expression. Therefor the generated output must contain in-line regular expressions instead of the named strings – the regex variable output will not use the variable names.

There are still scaling problems with this method but they will only show up with much larger log files. If we read them into memory, we must assume that the number of event substrings limited to fit into the memory, which is probably the case since there are a limited number of applications on the switches generating them.

Although this method is a bit more work, it avoids generating the very large ANTLR internal DFA for all the successful test logs, which triggers the C++ compiler segmentation fault. Additionally, it avoids generating the huge ANTLR grammar which causes the Java runtime array overruns. It also prevents the generation of the very large Java methods, which exceed the virtual machine 64 KB limit, if we want to go back to generating Java code instead of C++.

Note that the bric.dk external DFA and Augeas, the bric.dk C implementation, have the bitset size issue we encountered with the ANTLR C++ library.

Also note that employing an external DFA allows reuse of most of the ANTLR grammar parser code we implemented for the ANTRL grammar solution.

Another issue concerns the ability and willingness of an organization to adjust its processes to allow use of these methods. The tool set we presented requires that a test organization save the logs after successful test executions as well as after test failures. This requires modifying the automated test scripts or manual test procedures to gather the event logs from the systems under test after each test execution and store them. This also requires computational time for file transfers, machine resources for data storage, and human resources on the part of the test personnel. If those writing the test scripts and executing them are not trained software developers, modifying the scripts and testing the changes can be onerous. And since software test budgets are constrained at least as much as software development budgets, allocating resources for failure diagnosis when no failure has yet occurred can be burdensome.

BIBLIOGRAPHY

[1] J. P. Bowen and V. Stavridou, "Safety Critical Systems and Formal Methods," in *Towards Verified Systems*, Elsevier Science Ltd., 1994, pp. 3-4.

[2] R. Bird, *Personal conversation,* Santa Ana, CA, 1982.

[3] S. Gill, "The Diagnosis of Mistakes in Programmes on the EDSAC," *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences,* vol. 206, no. 1, pp. 538-554, 22 May 1951.

[4] J. H. Andrews, "Theory and Practice of Log File Analysis," University of Western Ontario, London, Ontario, Canada, 1998.

[5] I. L. Aulenbacher, "Generating Log File Analyzers," University of Western Ontario, London, Ontario, Canada, 2012.

[6] I. Beschastnikh, *Modeling Systems from Logs of their Behavior,* Seattle, Washington: University of Washington, 2013.

[7] J. H. Andrews, "Testing Using Log File Analysis: Tools, Methods, and Issues," in *13th IEEE Conference on Automated Software Engineering*, Honolulu, Hawaii, USA, 1998.

[8] J. H. Andrews and Y. Zhang, "General Test Result Checking with Log File Analysis," *IEEE Transactions on Software Engineering,* vol. 29, no. 7, July 2003.

[9] H. Barringer, K. Havelund, D. Rydeheard and A. Groce, "Rule Systems For Runtime Verification: A Short Tutorial," in *Proceedings of the 9th International Workshop on Runtime Verification*, 2009.

[10] H. Barringer, K. Havelund, M. Smith and A. Groce, "Formal Analysis of Log Files," *Journal of Aerospace Computing, Information, and Communication,* November 2010.

[11] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser and R. Washington, "Combining test case generation and runtime verification," *Theoretical Computer Science,* vol. 336, no. 3-6, pp. 209-204, May 2005.

[12] Worksoft Corporation, "Worksoft Certify," Worksoft Corporation, 2018. [Online]. Available: https://www.worksoft.com/products/worksoft-certify. [Accessed 10 December 2018].

[13] Ranorex, Gmbh, "Test Automation for All," Ranorex, Gmbh, 2018. [Online]. Available: https://www.ranorex.com/. [Accessed 10 December 2018].

[14] Broadcom, Inc., "CA Application Test," CA Technologies, 2018. [Online]. Available: https://www.ca.com/us/products/ca-application-test.html. [Accessed 10 December 2018].

[15] J. Stearley, "Towards Informatic Analysis of Syslogs," in *IEEE International Conference on Cluster Computing*, San Diego, CA USA, 2004.

[16] J. Stearley and G. Laguna, "Sisyphus - An Event Log Analysis Toolset," U.S. Department Of Energy, 1 September 2004. [Online]. Available: https://www.osti.gov/biblio/1230768-sisyphus-event-log-analysis-toolset. [Accessed 12 11 2018].

[17] I. Rigoutsos and A. Floratos, "Combinatorial pattern discovery in biological sequences: The Teiresias Algorithm," *Bioinformatics,* vol. 14, no. 1, 1998.

[18] R. Vaarandi, "Mining Event Logs with SLCT and LogHound," in *Proceedings of the 2008 IEEE/IFIP Network Operations and Management Symposium*, Salvador, Brazil, 2008.

[19] A. Makanju, A. N. Zincir-Heywood, S. Brooks and E. Milios, "LogView: Visualizing Event Log Clusters," in *Sixth Annual Conference on Privacy, Security and Trust, PST 2008*, Fredericton, New Brunswick, Canada, 2008.

[20] E. M. Gold, "Language Identification in the Limit," *Information and Control,* pp. 447-474, 1967.

[21] J. Oncina and P. Garcia, "Identifying Regular Languages in Polynomial Time," *Pattern Recongition and Image Analysis,* pp. 49-61, 1992.

[22] C. de la Higuera, "Current Trends in Grammatical Inference," in *Advances in Pattern Recognition, Joint IAPR International Workshops*, 2000.

[23] F. Javed, M. Crepinsek, B. R. Bryant, M. Mernik and A. Sprague, "Context Free Grammar Induction Using Genetic Algorithms," in *ACM SE*, Huntsville, Alabama, 2004.

[24] M. Crepinsek, M. Mernik, B. R. Bryant, F. Javed and A. Sprague, "Inferring Context Free Grammars for Domain Specific Languages," *Electronic Notes in Theoretical Computer Science 141,* pp. 99-116, 2005.

[25] A. U. Memon, *Log File Categorization and Anomaly Analysis Using Grammar Inference,* Ontario: Queen's University, 2008.

[26] W. Xu, L. Huang, A. Fox, D. Patterson and M. Jordan, "Experience Mining Google's Production Console Logs," in *Proceedings of the 2010 Workshop On Managing Systems via Log Analysis and Machine Learning Techniques*, Berkley, CA, USA, 2010.

[27] W. Xu, L. Huang, A. Fox, D. Patterson and M. I. Jordan, "Detecting Large-Scale System Problems by Mining Console Logs," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, New York, NY, USA, 2009.

[28] N. Maruyama and S. Matsuoka, "Model-Based Fault Localization in Large-Scale Computing Systems," in *IEEE 2008 Symposium on Parallel and Distributed Processing*, Miami, Florida, 2008.

[29] Q. Fu, J.-G. Lou, Y. Wang and J. Li, "Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis," in *Ninth IEEE Conference on Data Mining,*, Miami Beach, Florida, 2009.

[30] L. Mariani and M. Pezzè, "Dynamic Detection of COTS Component Compatibility"," *IEEE Software,* vol. 5, pp. 76-85, 2007.

[31] R. Vaarandi, "A Data Clustering Algorithm for Mining Patterns From Event Logs," in *d3rd IEEE Workshop on IP Operations and Management*, Kansas City, MO, USA, 2003.

[32] R. Vaarandi and M. Pihelgas, "LogCluster – A Data Clustering and Pattern Mining Algorithm for Event Logs"," in *11th International Conference on Network and Service Management*, Barcelona, Spain, 2015.

[33] M. Jiang, M. A. Munawar, T. Reidemeister and P. A. Ward, *Detection and Diagnosis of Recurrent Faults in Software Systems by Invariant Analysis 11th IEEE Systems Engineering Symposium, IEEE, Nanjing, China, December 2008,* Nanjing: IEEE, 2008.

[34] A. Makanju, A. N. Zincir-Heywood and E. Milios, *A Lightweight Algorithm for Message Type Extraction in System Application Logs,* vol. 24, IEEE, 2012.

[35] T. Reidemeister, *Fault Diagnosis in Enterprise Software Systems Using Discrete Monitoring Data,* Waterloo, Ontario: University of Waterloo, 2012.

[36] W. van der Aalst, Process Mining: Data Science in Action, Second Edition, Berlin, Germany: Springer, 2016.

[37] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang and A. Mueen, "Hamooni, H., Debnath, B., Xu, J., Zhang, H., Jiang, G. and Mueen, A., “LogMine: Fast Pattern Recognition for Log Analytics”, ACM International Conference on Information and Knowledge Management 2016 (CIKM 2016)," in *ACM International Conference on Information and Knowledge Management 2016 (CIKM 2016),*, Indianapolis, IN, USA, 2016.

[38] N. Gurumdimma, A. Jhumka, M. Liakata, E. Chuah and J. Browne, "Towards Detecting Patterns in Failure Logs of Large-scale Distributed Systems," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, Hyderabad, India, 2015.

[39] A. Pecchia and S. Russo, "Detection of Software Failures Through Event Logs: An Experimental Study," in *IEEE 23rd International Symposium on Software Reliability Engineering*, Dallas, TX, USA, 2012.

[40] Q. Wang, W. U. Hassan, A. Bates and C. Gunter, "Fear and Logging in the Internet Of Things," in *Network and Distributed Systems Security Symposium*, San Diego, CA, 2018.

[41] A. Oliner and J. Stearly, "Alert Detection in System Logs," in *Proceedings of the IEEE International Conference on Data Mining*, 2008.

[42] C. E. Shannon, "A Mathematical Theory of Communication," *Bell Systems Technical Journal,* vol. 27, pp. 379–423, 623–656, July-October 1948.

[43] Elastic, Inc., "What is the ELK Stack," 6 Oct 2018. [Online]. Available: https://www.elastic.co/elk-stack.

[44] Graylog , "Graylog," 6 October 2018. [Online]. Available: https://www.graylog.org/.

[45] Splunk, Inc., "Splunk Next," 6 Oct 2018. [Online]. Available: https://www.splunk.com/.

[46] O. Fowler, "palindromicity/simple-syslog-5424," 2 Sep 2018. [Online]. Available: https://github.com/palindromicity/simple-syslog-5424.

[47] N. Chomsky, "Three models for the description of language," *IRE Transactions on Information Theory. 2 (3): 113–124,* vol. 2, no. 3, pp. 113-124, 1956.

[48] N. Chomsky, "On Certain Formal Properties of Grammars," *Information and Control,* vol. 2, pp. 137-167, 1959.

[49] J. W. Backus, "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," in *Proceedings of the International Conference on Information Processing. UNESCO. pp. 125–132.*, Paris, 1959.

[50] M. J. Steedman, "A Generative Grammar for Jazz Chord Sequences," *Music Perception: An Interdisciplinary Journal,,* vol. 2, no. 1, pp. 52-77, 1984.

[51] D. B. Searles, "The Computational Linguistics," in *Artificial Intelligence and Molecular Biology. L. Hunter (ed.), MIT Press, 1993, pp. 47-120.*, Boston, MA, MIT Press, 1993, pp. 47-120.

[52] S. C. Kleene, "Representation of Events in Nerve Nets and Finite Automata," in *Automata Studies*, Princetone University Press, 1951, pp. 3-42.

[53] M. Sipser, "Regular Languages," in *Introduction to the Theory of Computation*, Boston, MA: PWS Publishing, 1997, pp. 31-66.

[54] A. W. Appel and M. Ginsburg, Modern Compiler Implementation in C, Cambridge: Cambridge University Press, 1998, pp. 39-45.

[55] A. Röder, V. Vasyutynskyy, K. Kabitzsch, T. Zarbock and G. Luhn, "Log-based State Machine Construction for Analyzing Internal Logistics of Semiconductor Equipment," in *Proceedings of the International Conference on Modeling and Analysis of Semiconductor Manufacturing (MASM2005)*, Singapore, 2005.

[56] J. E. Cook and A. L. Wolf, "Discovering Models of Software Processes from Event-Based Data," *ACM Transactions on Software Engineering and Methodology,* vol. 7, no. 3, pp. 215-249, 1998.

[57] D. Harel, "State Charts: A Visual Formalism for Complex Systems," *Science of Computer Programming,* vol. 8, pp. 231-274, 1987.

[58] B. Watson and J. Daciuk, "An efficient incremental DFA minimization algorithm," *Natural Language Engineering,* vol. 9, no. 1, pp. 49-64, 2003.

[59] J. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," in *Proceedings of the International Symposium on the theory of machines and computations*, Haifa, 1971.

[60] J. Levine, Flex and Bison, O'Reilly Media, 2009.

[61] C. Neumann, "Converting Deterministic Finite Automata to Regular Expressions," 16 Mar 2005. [Online]. Available: https://www.researchgate.net/publication/249958848_Converting_Deterministic_Finite_Automata_to_Regular_Expressions. [Accessed 15 Sep 2018].

[62] "How to convert Finite Automata to Regular Expressions," 16 June 2012. [Online]. Available: https://cs.stackexchange.com/questions/2016/how-to-convert-finite-automata-to-regular-expressions/2017#2017.

[63] J. A. Brzozowski, "Derivatives of Regular Expressions," *Journal of the Association for Computing Machinery,* vol. 11, no. 4, October 1964.

[64] L. Pitt and M. K. Warmuth, "The Minimum Consistent DFA Problem Cannot Be Approximated with any Polynomial," *Journal of the Association for Computing Machinery,* vol. 40, no. 1, pp. 95-142, January 1993.

[65] A. R. Meyer and L. J. Stockmeyer, "The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space," in *13th Annual Symposium on Switching and Automata Theory (swat 1972)*, 1972.

[66] T. Stack, "LNAV Documentation," August 2018. [Online]. Available: https://lnav.readthedocs.io/en/latest/formats.html. [Accessed 2 December 2018].

[67] D. Brezinski and T. Killalea, "RFC3227 - Guidelines for Evidence Collection and Archiving," Internet Engineering Task Force, 2005.

[68] IETF Network Working Group, *RFC 5424 - The Syslog Protocol,* IETF, 2009.

[69] IETF Network Working Group, *RFC 3164 - The BSD syslog Protocol,* IETF, 2001.

[70] Apache Software Foundation, "Commons Logging," Apache Commons , 2012 2 2016. [Online]. Available: https://commons.apache.org/proper/commons-logging/. [Accessed 11 11 2018].

[71] Microsoft Corporation, "Introduction to the Common Log File System," Microsoft Corporation, 15 6 2017. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-the-common-log-file-system. [Accessed 11 11 2018].

[72] W3C Consortium, "The Common Logfile Format," W3C Consortium, [Online]. Available: https://www.w3.org/Daemon/User/Config/Logging.html#common-logfile-format. [Accessed 11 11 2018].

[73] W3C Consortium, "Extended Log File Format," W3C Consortium, 23 03 1996. [Online]. Available: https://www.w3.org/TR/WD-logfile.html. [Accessed 11 11 2018].

[74] B. Charter, "Logging Technogy and Techniques," SANS Technology Institute, 13 11 2008. [Online]. Available: https://www.sans.org/reading-room/whitepapers/logging/paper/32949. [Accessed 11 11 2018].

[75] Apache Software Foundation, "Welcome to Apache Log4j 2!," Apache Software Foundation, 2018. [Online]. Available: https://logging.apache.org/log4j/2.x/manual/index.html. [Accessed 11 11 2018].

[76] J. Kraft, A. Wall and H. Kienle, "Trace Recording for Embedded Systems; Lessons Learned from Five Industrial Projects," in *International Conference on Runtime Verification*, Malta, 2010.

[77] IETF, "RFC3629 - UTF-8, a transformation format of ISO 10646," November 2003. [Online]. Available: https://tools.ietf.org/html/rfc3629. [Accessed 23 February 2019].

[78] American Standards Association, "American Standard Code for Information Interchange, ASA X3.4-1963". American Standards Association (ASA). 1963-06-17," American Standards Association, Washington, D.C., 1963.

[79] IETF, "RFC5234 - Augmented BNF for Syntax Specifications: ABNF," Internet Engineering Task Force, 2008.

[80] Internet Engineering Task Force, "RFC 2131 Dynamic Host Configuration Protocol," March 1997. [Online]. Available: https://tools.ietf.org/pdf/rfc2131.pdf. [Accessed 23 November 2018].

[81] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady,* vol. 10, no. 8, pp. 707-710, February 1966.

[82] S. Hjelmqvist, "Fast, memory efficient Levenshtein algorithm," 26 March 2012. [Online]. Available: https://www.codeproject.com/Articles/13525/Fast-memory-efficient-Levenshtein-algorithm.

[83] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM,* vol. 18, no. 6, pp. 341-343, 1975.

[84] Oracle America, Inc., "The Java Virtual Machine Specification - Chapter 4. The class File Format," Oracle America, Inc., 2013. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.7.3. [Accessed 12 December 2018].

[85] T. Parr, The Definitive ANTLR4 Reference, The Pragmatic Programmers, LLC, 2012.

[86] A. Moeller, "dk.brics.automaton," Aarhus University, 4 July 2017. [Online]. Available: http://www.brics.dk/automaton/. [Accessed 12 January 2019].

[87] Oracle Corp., "Java Native Interface Specification," 2017. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html. [Accessed 14 January 2019].

[88] D. Lutterkort, "Augeas," 24 August 2018. [Online]. Available: http://augeas.net/. [Accessed 12 January 2019].

[89] G. Navarro, "A guided tour to approximate string matching" (PDF). ACM Computing Surveys. 33 (1): 31–88. doi:10.1145/375360.375365," *ACM Computing Surveys,* p. 31–88, 2001.

[90] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM,* vol. 18, no. 6, pp. 341-343, 1975.

[91] F. Hertzum, "Iosifovich," 18 06 2018. [Online]. Available: https://bitbucket.org/clearer/iosifovich/overview.

[92] M. Cumming, "Suffix Tree, Ukkonen, C++," 19 August 2016. [Online]. Available: https://www.murrayc.com/permalink/2016/08/19/suffix-tree-ukkonen-c/.

[93] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica,* vol. 14, no. 3, pp. 249-260, 1995.

[94] J. E. Hopcroft, R. Motwani and J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, 2nd ed., Addison-Wesley, 2001.

[95] T. Parr, Language Implementation Patterns: Create Your Own Domain Specific and General Programming Languages, Raleigh, North Carolina; Dallas, Texas, USA: The Pragmatic Bookshelf, 2010.

[96] T. Polcar and B. Melichar, "The Longest Common Subsequence Problem - A Finite Automata Approach," in *Implementation and Application of Automata - Lecture Notes In Computer Science 2759*, Santa Barbara, CA, USA, 2003.

[97] W. Jiang, C. Hu, S. Pasupathy, A. Kanevsky, Z. Li and Y. Zhou, "Understanding customer problem troubleshooting from storage system logs," in *FAST '09 Proccedings of the 7th conference on File and storage technologies*, San Francisco, 2009.

[98] L. Chinghway, N. Singh and S. Yajnik, "A Log Mining Approach to Failure Analysis of Enterprise Telephony Systems," in *International Conference on Dependable Systems & Networks*, Anchorage, Alaska, 2008.

[99] G. S. Ingersoll, T. S. Morton and A. L. Farris, Taming Text - How to find, organize, and manipulate it, Shelter Island, New York: Manning Publications Co., 2013, pp. 89-91.

[100] K. Fisher and T. Parr, "LL(*): the foundation of the ANTLR parser generator," in *PLDI '11 Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Jose, California, USA.

[101] S. C. Johnson, "Yacc: Yet Another Compiler-Compiler," *AT&T Bell Laboratories Technical Reports. ,* 1975.

[102] M. E. Lesk and E. Schmidt, "Lex − A Lexical Analyzer Generator, , Volume 2B. bell-labs.com," in *UNIX TIME-SHARING SYSTEM:UNIX PROGRAMMER'S MANUAL, Seventh Edition*, vol. 2B, Bell Labs, 1975.

[103] T. Parr and K. S. Fisher, "LL(*): The Foundation of the ANTLR Parser Generator," in *PLDI - Programming Language Design and Implementation*, San Jose, CA, 2011.

[104] Apache Software Foundation, "Apache Log4j 2 User's Guide," Apache Software Foundation, 2018.