Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

August 2012

Generating Log File Analyzers

Ilse Leal Aulenbacher
The University of Western Ontario

Supervisor
James H. Andrews
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science

© Ilse Leal Aulenbacher 2012

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Part of the Software Engineering Commons

Recommended Citation
Leal Aulenbacher, Ilse, "Generating Log File Analyzers" (2012). *Electronic Thesis and Dissertation Repository*. 780.
https://ir.lib.uwo.ca/etd/780

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact tadam@uwo.ca.

المنارة للاستشارات

www.manaraa.com

GENERATING LOG FILE ANALYZERS

(Spine Title: Generating Log File Analyzers)

(Thesis Format: Monograph)

by

Ilse Leal Aulenbacher

Graduate Program in Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science

School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario
July, 2012

© Ilse Leal Aulenbacher 2012

المنارة للاستشارات

www.manaraa.com

THE UNIVERSITY OF WESTERN ONTARIO
SCHOOL OF GRADUATE AND POSTDOCTORAL STUDIES

CERTIFICATE OF EXAMINATION

Supervisor	Examiners
James H. Andrews	Dr. Lucian Ilie
Supervisory Committee	Dr. Marc Moreno Maza
	Dr. Jagath Samarabandu

The thesis by
Ilse Leal Aulenbacher
entitled

GENERATING LOG FILE ANALYZERS

is accepted in partial fulfillment of the
requirements for the degree of
Master of Science

Date	Chair of Thesis Examining Board

المنارة للاستشارات
www.manaraa.com

Abstract

Software testing is a crucial part of the software development process, because it helps developers ensure that the software works correctly and according to stakeholders' requirements and specifications. Faulty or problematic software can cause huge financial losses. Automation of testing tasks can have a positive impact on software development, by reducing costs and minimizing human error. Software testing can be divided into three tasks: choosing test cases, running test cases on the software under test (SUT) and evaluating the test results. To evaluate test results, testers need to examine the output of the SUT to determine if it performed as expected. Programs often store some of their outputs in files known as log files. The task of evaluating test results can be automated by using a log file analyzer. The main goal of this thesis is to design an approach to generate log file analyzers based on a set of state machine specifications. Our analyzers are generated in C++ and are capable of reading log files from disk or shared memory areas. Regular expressions have been incorporated, so that analyzers can be adapted to different logging policies. We analyze the purpose and benefits of this framework and discuss differences with a previous implementation based on Prolog. In particular, we discuss the results of a series of experiments that we performed in order to compare the performance between Prolog–based analyzers and C++ analyzers. Our results show that C++ analyzers are between 8 and 15 times faster than Prolog–based analyzers.

Keywords: Software Testing, Test Oracles, Log File Analysis

المنارة للاستشارات

www.manaraa.com

Acknowledgments

I would like to thank Dr. Jamie Andrews for his support, guidance and patience. Professor, thank you for believing in me and for giving me the opportunity to learn so much from you. It has been an honor and a privilege to work with you.

I would like to acknowledge the financial support of the National Council of Science and Technology in Mexico (Consejo Nacional de Ciencia y Tecnología) and the Institute of Electrical Research in Mexico (Instituto de Investigaciones Eléctricas).

I would like to thank the Department of Computer Science and its wonderful faculty and staff for their support and teachings.

To my beloved husband, Andrés Ayala García, thank you for loving, inspiring and supporting me each step of the way.

There are not enough words to express my gratitude to my parents, who made a lot of sacrifices so that I could get a good education and fulfill my dreams.

I would like to honor my late grandparents Margarita de Leal, Juanito Leal, Ilse de Aulenbacher and Carlos Aulenbacher. Your example has inpired me all of my life. I love you all so much.

My dear Opa Carlos, you passed away this year while I was writing this thesis. I know you are now in heaven with our beloved Oma. Thank you for sharing so many things with me and for being my mentor and friend. You will always be in my heart. Mein lieber Opa, ich vermisse dich.

Finally, I would like to thank all of my family and friends in Mexico and Canada. Thank you for your help, kind words and for being there for me.

المنارة للاستشارات
www.manaraa.com

Table of Contents

Certificate of Examination **ii**

Table of Contents **v**

List of Figures **ix**

1 Introduction **1**

1.1 Test oracles 2

1.2 Log files and logging policies 3

1.2.1 Log files and shared memory areas 3

1.3 Log File Analyzers 4

1.4 Thesis focus 6

1.5 Thesis organization 6

2 Related Work **8**

2.1 Analysis of server logs 8

2.2 Test oracles 9

2.2.1 The oracle assumption 10

المنارة للاستشارات

www.manaraa.com

2.2.2 Deriving oracles 11
2.2.3 Deriving oracles automatically 12
2.3 Generating log file analyzers 13
2.3.1 Log files 13
2.3.2 Expected program behavior 14
2.3.3 Log File Analyzers 15
2.3.4 Log File Analysis Language 16
2.3.5 Log file analyzer generation 17
2.4 Other related work 18

3 The Log File Analyzer Generator 19

3.1 The LFAL 2.0 specification 20
3.1.1 The elements of an LFAL 2.0 program 20
3.1.2 New features in the LFAL 2 language 22
3.2 Scanning and Parsing LFAL 2.0 23
3.2.1 Lexical Analysis 23
3.2.2 Syntax Analysis 23
3.3 The log file analyzer generator 24
3.3.1 Base libraries 24
3.3.2 Machine Classes 25
3.3.3 The Log File Analyzer Program 26
3.4 Analyzing log files 27

المنارة للاستشارات
www.manaraa.com

4 New Features in Analyzers **37**

4.1 C++ Analyzers 37

4.2 Pattern definitions 38

4.3 Dynamic and static analyzer machines 41

4.3.1 Static analyzer machines 41

4.3.2 Dynamic analyzer machines 44

4.4 New types of actions in transitions 46

4.4.1 “Error” transitions 46

4.4.2 “Ignore” and “stay” transitions 47

4.4.3 “Doing” transitions 50

4.5 Data declarations 50

4.6 Shared memory integration 52

5 Evaluating the Performance of Log File Analyzers **55**

5.1 Performance experiments for LFAL 1.0 and LFAL 2.0 56

5.1.1 Experiment design 57

5.1.2 The LFAL 1.0 BG/L Analyzer 59

5.1.3 The LFAL 2.0 BG/L Analyzer 63

5.1.4 Results 66

5.2 Performance experiments for shared memory analyzers 68

5.2.1 Reading from disk 70

5.2.2 Reading from memory 71

المنارة للاستشارات
www.manaraa.com

5.2.3 Results 72
5.3 Logging overhead in disk versus memory 73
5.3.1 Results 74
5.4 An example temporal filter analyzer 76

6 Conclusion **81**

6.1 Conclusion 81
6.2 Future work 84

Vita **89**

المنارة للاستشارات
www.manaraa.com

List of Figures

2.1 A simple log file by Andrews [15] 14

2.2 Analyzer machines for the elevator controller by Andrews [15] 16

2.3 An example of a LFAL specification by Andrews [15] 17

3.1 Log File Analyzer Generation Process 32

3.2 A simple example of a LFAL 2.0 program 33

3.3 The original LFAL specification, by Andrews [15] 33

3.4 The LFAL 2.0 specification, part 1 34

3.5 The LFAL 2.0 specification, part2 35

3.6 An LFAL 2.0 program with two machine classes 36

4.1 A pattern definition 39

4.2 An example of a static machine 42

4.3 An example log file 43

4.4 An example of a dynamic machine 44

4.5 Data declarations 51

للاستشارات المنارة

www.manaraa.com

4.6 A log file in a shared memory area (SmaLog). In this example, a log file is being written by two processes. At the same time, a log file analyzer is accessing the log file in shared memory. 53

5.1 The four log patterns used in our experiments. The name of the patterns (bold underlined) are followed by a corresponding regular expression (bold). Below each pattern, a sample log entry from the BG/L log file. 59

5.2 The bgl01 analyzer in LFAL 1.0. This analyzer matches and counts one pattern. 60

5.3 The bgl02 analyzer in LFAL 1.0. This analyzer matches and counts two patterns. 60

5.4 The bgl03 analyzer in LFAL 1.0. This analyzer matches and counts three patterns. 61

5.5 The bgl04 analyzer in LFAL 1.0. This analyzer matches and counts four patterns. 62

5.6 The CPU user time for bgl01, bgl02, bgl03 and bgl04. The x axis represents the percentage of the log file that was analyzed and the y axis represents the number of milliseconds of user CPU time it took on average to run the analyzer. 63

5.7 The CPU system time for bgl01, bgl02, bgl03 and bgl04. The x axis represents the percentage of the log file that was analyzed and the y axis represents the number of milliseconds of system CPU time it took on average to run the analyzer. 64

5.8 The bgl04 analyzer in LFAL 2.0. This analyzer matches and counts four patterns. 65

5.9 The CPU user time for bgl01, bgl02, bgl03 and bgl04. The x axis represents the percentage of the log file that was analyzed and the y axis represents the number of milliseconds of user CPU time it took on average to run the analyzer. 66

المنارة للاستشارات
www.manaraa.com

5.10 The CPU system time for bgl01, bgl02, bgl03 and bgl04. The x axis represents the percentage of the log file that was analyzed and the y axis represents the number of milliseconds of system CPU time it took on average to run the analyzer. 67

5.11 The CPU user time ratio between LFAL 1.0 and LFAL 2.0 for bgl01, bgl02, bgl03 and bgl04. The x axis represents the percentage of the log file that was analyzed and the y axis represents the `LFAL1:LFAL2` ratio. 68

5.12 The CPU system time ratio between LFAL 1.0 and LFAL 2.0 for bgl01, bgl02, bgl03 and bgl04. The x axis represents the percentage of the log file that was analyzed and the y axis represents the `LFAL1:LFAL2` ratio. 69

5.13 A sample program that illustrates how to create a log file in a shared memory area using LFAL 2.0's base libraries. 78

5.14 A table that summarizes the results of our experiment to compare the performance of analyzers that read the BG/L log file from disk or shared memory 79

5.15 A table that summarizes the results of our experiment to measure the time to write the BG/L log file in both shared memory and disk. . . . 79

5.16 A log file analyzer that filters system log file. 80

المنارة للاستشارات

www.manaraa.com

Chapter 1

Introduction

Although we might not be aware of it all the time, software plays an important role in our lives. From our appliances at home, to the planes we take, software is involved. Software testing is a crucial part of the software engineering process, because it helps testers to find and correct faults before releasing a program to the public.

A very important part of the testing process is evaluating test results. This is achieved through the use of *test oracles*. A test oracle captures the inputs and outputs of the software under test (SUT) and determines whether the SUT execution was correct or incorrect. Test oracles are difficult to implement in part because of the complexity involved in capturing the inputs and outputs of a program. An alternative is to analyze log files. Oracles that analyze log files are known as *log file analyzers*. It is already a common practice for developers to instrument their code so that relevant events are logged to a file. The resulting log files are used for debugging purposes. We are interested in generating oracles capable of analyzing log files, in order to determine if they reveal faults in the SUT.

Although oracles are an integral part of the testing process, they are often difficult

المنارة للاستشارات
www.manaraa.com

to derive. For that reason, attempting to write an ad-hoc analyzer for each program we develop would not be practical and would require a lot of effort. The main focus of this thesis is to design a framework in which log file analyzers are automatically generated from the specification of a program's expected behavior.

1.1 Test oracles

The word *oracle*, comes from the Latin noun *oraculum* and from the verb *orare* which means "to speak". In the past, this word was used to refer to a priest or priestess acting as a medium through whom advice or prophecy was sought from the gods in classical antiquity. In its modern use, *oracle* refers to a person or thing regarded as an infallible authority or guide on something [12]. In the context of software testing, an *oracle* refers to a mechanism capable of determining if the software under test (SUT) output is correct or incorrect.

Software testing can be divided into three tasks: choosing test cases, running test cases on the SUT and evaluating the test results [15]. The task of evaluating test results is often taken for granted and considered to be straightforward. In fact, this assumption is known as the *oracle assumption* [34]. Test oracles are indeed a very important part of software testing and should not be overlooked or underestimated. As we will see in the next chapter, many oracle designs capture a program's inputs and outputs directly. Even though that is the classic approach to writing oracles, it can be difficult to implement and can greatly increase the difficulty of writing a test oracle. For that reason, our thesis focuses on oracles that analyze log files produced by programs. Henceforth, we will refer to these kind of oracles as *log file analyzers*.

المنارة للاستشارات
www.manaraa.com

1.2 Log files and logging policies

Logs have been used to record important information, even before computers were first invented. For example, logs have been kept for ships. To this day, such logs are used to recreate the events that happened during past wars or to discover immigration patterns [6]. With the advent of computers and with the increasing complexity of software, developers found themselves in the need to record important events for their programs. In practice, developers often use log files to recreate the events that happened during the execution of a program, allowing them to debug their programs. That is, they instrument their code so that important events, warnings or error messages get saved into a text file. The information that can be obtained from analyzing a log file depends on the what types of events the developer decides to log. We refer to this as a *logging policy*.

1.2.1 Log files and shared memory areas

Log files are also very useful to debug real-time programs. Since real-time programs are time-sensitive, it is often infeasible to use standard debugging tools. In my own professional experience, log files are very useful to debug real-time systems. While participating in the development of a real-time monitoring system [16], we noticed that log files were our main point of reference for troubleshooting or debugging processes. However, logging events into a disk file can affect the performance of real-time systems. Therefore, we opted to log events to a shared-memory area [19]. *Shared Memory* is the fastest form of inter-process communication (IPC) available [30]. It allows several processes to share a memory area and exchange data efficiently.

المنارة للاستشارات
www.manaraa.com

Storing a log file in a shared memory area has various benefits. For instance, if a system is comprised of several processes, each process could log events into a single shared memory area. In that way, we would be able to analyze a time-ordered log with the interactions of the different processes, instead of having to merge several logs manually. In addition, reading and writing logs to memory helps minimize the overhead of logging. It is faster to read and write from memory than from a disk.

Taking into consideration the above-mentioned experience, in this thesis we will address the problem of logging overhead by adding shared memory integration to our log file analyzers. This new capability will make log file analyzers not only capable of reading logs from files, but also from shared memory areas. We will also describe a simple-to-use library we developed so that programs can easily create log files in a shared memory area.

1.3 Log File Analyzers

Having described the role of test oracles in software testing and the uses of log files, we will discuss log file analyzers in more detail. As we mentioned before, log file analyzers can be defined as oracles that process log files produced by programs.

In order to apply log file analysis, we need to make some basic assumptions regarding the software environment. These assumptions are known as the *log file analysis assumptions* and were introduced by Andrews [15]:

1. The SUT writes a record of events to a log file.
2. There exists a well–defined, agreed-upon logging policy that defines what pre-

المنارة للاستشارات
www.manaraa.com

cisely the SUT should write to the log file and under what conditions.

3. It is possible to write a log file analyzer program that takes a log file as input and either accepts the log file or rejects it with an informative error message.

These assumptions are important because they dictate the conditions under which we can generate log file analyzers for the SUT. The logging policy is particularly important, because the SUT must log events of interest so that the log file analyzer can correctly assess the log file.

A log file analyzer processes the messages contained in the log file and determines whether the SUT that generated such log file executed correctly or incorrectly. A log file is considered to reflect a correct program execution, if the log messages it contains are consistent with the *expected program behavior*.

For example, suppose we were looking at the log file produced by the software controlling an ATM. Let us assume that such software has a logging policy that records a message for each relevant transaction. Let us also suppose that we examine the log file and find a line indicating that a user has entered his PIN incorrectly, followed by a line indicating that such user was granted access to "his" account. Of course, this is not part of the ATM's expected behavior. Therefore, such a log file would be considered incorrect. An ATM should not give access to unauthorized users. In fact, such a log file would reveal that the ATM software is not working as expected. To be able to specify the expected behavior of any program, we need to have a language that allows us to express how a program should work. We express a program's expected behavior in a language called *Log File Analysis Language* (LFAL), which was first introduced by Andrews [13].

المنارة للاستشارات
www.manaraa.com

1.4 Thesis focus

The main focus of this thesis is to generate log file analyzers from a set of specifications. Our research builds upon previous work by Dr. James H. Andrews, who proposed a framework in which the expected program behavior is expressed as a set of state machines in LFAL (*Log File Analysis Language*). The LFAL program is translated into Prolog code. Upon compilation, a log file analyzer is obtained [13]. One of our objectives is to extend LFAL so that we can incorporate new features that make analyzers more flexible. We will discuss the purpose and benefits of these features. Another objective is to generate analyzers based on the C++ language instead of Prolog. This will allow us to take advantage of the power of C++ and support the new features of the extended LFAL. We also evaluate the performance of our C++ analyzers and compare it against Prolog-based analyzers.

The results of our experiments show that log file analyzers based on C++ are between 8 and 15 times faster than Prolog–based analyzers. These experiments, along with the example C++ analyzers that we present in this thesis, illustrate the benefits of making log file analyzers more flexible. For example, support for regular expressions makes C++ analyzers capable of processing log files that contain complex log message patterns.

1.5 Thesis organization

Chapter 1 contains an introduction to our topic by explaining some basic concepts and the purpose of our research. In chapter 2, we give an overview of related work on test oracles and log file analyzers. We also describe some important concepts. In chapter

المنارة للاستشارات
www.manaraa.com

3, we describe the design of our log file analyzer generator. We begin by explaining the extensions made to LFAL and presenting the new language specification. We then describe the basic libraries, which contain functionality needed by all analyzers. These libraries include a state machine class and shared-memory management routines. We also describe the design of our log file analyzer generator, which translates a LFAL specification into C++ code. We conclude the chapter explaining how log file analyzers are used. We explain the benefits of generating log file analyzers in C++ instead of Prolog in chapter 4. We also describe the new features introduced to the LFAL language in more detail. In chapter 5, we describe the experiments we conducted to evaluate the performance of C++ log file analyzers. In chapter 6, we present our conclusions.

المنارة للاستشارات
www.manaraa.com

Chapter 2

Related Work

In this chapter we give an overview of work related to log file analyzer generation. We begin by describing related work on analysis of server logs. We then give an overview of work done in the area of test oracles. We also discuss the area that is most related to our research: generation of log file analyzers.

2.1 Analysis of server logs

A *log* can be defined as a record of the events occurring within the systems and networks of an organization. Logs are composed of *log entries*. Each log entry contains information related to a specific event that has occurred within a system or network [22]. The field of log file analysis is as ample as the variety of uses for logs. Some uses include debugging and troubleshooting systems. Other uses include recording user actions and investigating malicious activity. A significant amount of the work on log files focuses on the analysis of logs generated by servers [33].

المنارة للاستشارات
www.manaraa.com

Servers generate large log files that contain messages, which help system administrators to monitor a system and identify failures. Oliner and Stearley [26], studied logs generated by five supercomputers in order to understand their behavior. The authors recognize that even though system logs are the first place system administrators go to find the cause of a problem, there is a pressing need for better tools to process and understand log files. This task is often complicated by the lack of *operational context* in logs, which capture the system's expected behavior. To facilitate the analysis of large logs, they developed a filtering algorithm that reduces a set of alerts to a single initial alert per failure. Oliner et al. describe a tool known as *Sisyphus*, which uses an unsupervised data mining algorithm for anomaly detection [27]. This tool helps system administrators discover relevant information in large log files, even if they do not know what they are looking for. Log entries are sorted so that entries considered most "interesting" or "abnormal" are given more importance.

Web traffic analysis is another important application of log file analysis. Server log parsers analyze raw traffic data from server logs. These data can be visualized and manipulated in Web browser windows [23]. An example is AWStats, which generates Web, streaming, ftp or mail server statistics, graphically [2]. Other examples of Web log analyzers include Analog [1], Webalizer [11] and Sawmill Analytics [8].

2.2 Test oracles

In the previous section, we discussed what seems to be the most evolving and developed area of log file analysis: analysis of server logs [33]. This thesis, however, focuses on log file analysis applied to the analysis of software. As we mentioned in chapter 1, our thesis builds upon previous work by Andrews [13]. Andrews applies log

المنارة للاستشارات
www.manaraa.com

file analysis to software testing, in particular, to the evaluation of test results. Test oracles are used for this task, because they evaluate whether the output of a program is correct or incorrect. In Andrews' framework, a log file (produced by a program execution) is analyzed to determine if it reveals a failure in the software under test. In fact, a log file analyzer is a type of oracle. Therefore, it is important to discuss previous work in the area of test oracles.

2.2.1 The oracle assumption

As mentioned in chapter 1, software testing can be divided into three tasks: choosing test cases, running test cases on the SUT and evaluating test results [15]. Weyuker, defined an *oracle* as a mechanism that checks for the correctness of a program execution. Weyuker also coined the term *oracle assumption*, which refers to the belief that the tester is routinely able to determine the program correctness on the test data [34]. Because of the oracle assumption, the task of evaluating test results is often considered "straightforward". Therefore, it is common practice to have the tester examine the results of a program execution by hand. The problem with this approach, is that is *assumed* that the tester will know the correct answer.

Many papers on oracles refer to the importance of test oracles. Testing without an oracle can cause loss of time, due to *tester misconception*. That could cause the tester to "fix" a program that was already correct. Conversely, the tester might believe that the program is correct, thereby releasing a program with errors [34].

In some cases, the task of evaluating test results is relegated to having the tester examine a program's output manually. However, such practice is neither reliable nor cost-effective. The whole point of testing is to reveal system failure or provide as-

المنارة للاستشارات
www.manaraa.com

surance of system correctness. If we do not have a reliable way to evaluate whether a test case was successful or not, we cannot confidently ascertain the correctness of a program. In Richardson et al.'s words: "If the testing process does not determine whether the system behaves correctly, there is nothing to be gained by performing the tests" [29]. These last words might seem too definitive and many testers might argue that all stages of software are valuable and important. However, it is important to recognize that much of the research on software testing has focused on the development and analysis of input data [34]. In fact, research literature on test oracles is a relatively small part of the research literature on software testing [17]. Therefore, researchers that work with test oracles have tried to raise awareness on the importance oracles in the software testing process.

2.2.2 Deriving oracles

In the previous section, we discussed the importance of test oracles in the software testing process. The reader might be wondering why it is that test oracles are not always used. The explanation is that oracles are not particularly easy to derive. For instance, Peters and Parnas [28] recognize that the documentation used to generate an oracle can be almost as complicated as the software under test.

Richardson et al. [29] derive oracles from a program's specifications. This requires a mapping from the name space of the test data, to the name space of the oracle information. The oracle information represents the expected behavior of a program. The authors express this expected behavior as a set of *assertions*. An assertion is a logical expression specifying a program state that must exist, or a set of conditions that program variables must satisfy at a particular point in program execution. A

المنارة للاستشارات
www.manaraa.com

monitor program is used to verify the assertions. Any unsatisfied assertion identifies an inconsistency between the expected program behavior and the specification-based oracle.

Memon et al. [25] developed a technique to develop an automated Graphic User Interface (GUI) test oracle. The GUI is modeled through operators that represent GUI actions in terms of their preconditions and effects. The test oracle automatically derives the expected states (the expected program behavior). An execution monitor obtains the current state of the GUI. The oracle compares the two states and determines if the GUI is performing as expected.

2.2.3 Deriving oracles automatically

Peters and Parnas [28], describe an interesting approach, capable of *automatically* generating test oracles from tabular documentation. This work is closer to our research objective, because its focus is not limited to describing a method for deriving oracles. Rather, the main objective is to achieve the automatic generation of test oracles from program documentation.

The authors argue that if program documentation is mathematical, it is possible to derive an oracle from it. Therefore, the expected program behavior is captured through relational documentation, which is written using tabular expressions. In contrast with assertions, the documentation is separate from the code, rather than embedded in it. This facilitates analysis and review separate from the implementation. A Test Oracle Generator (TOG) generates wrappers that call the functions to be tested. The test case is executed by calling the wrappers instead of the real functions. Finally, the wrapper evaluates the output to determine if it is correct.

المنارة للاستشارات
www.manaraa.com

2.3 Generating log file analyzers

In this section, we will discuss the framework proposed by Andrews [13] [15] [14] [35] in which log file analysis is applied to software testing. In fact, Andrews' log file analyzers are test oracles that determine if a log file reveals a fault in the software under test (SUT). This section provides much of the concepts and background needed to understand this thesis. Henceforth, we will refer to Andrews' approach as Log File Analysis (LFA).

2.3.1 Log files

A *log file* is defined as a sequence of log lines. In LFA, a *log file line* is defined as a sequence of keywords, strings and numbers beginning with a lowercase alphanumeric character. Log lines begin with a keyword, separated by blanks and terminated by a new-line. A *keyword* is a sequence of alphanumeric characters and underscores beginning with a lowercase letter. Figure 2.1 shows an example of a log file [15] from a hypothetical elevator controller program. We can observe that each line begins with a keyword, for example `go_up`, `go_down`, etc.

To illustrate the concepts behind LFA, we will use the elevator controller program and the log file in figure 2.1 as a running example. Let us first understand how the log file in this example is interpreted:

- A user, who is in the third floor, calls the elevator (line `call 3`).
- The elevator goes up and reaches floors two and three (lines `reach 2` and `reach 3`).

المنارة للاستشارات
www.manaraa.com

Figure 2.1 A simple log file by Andrews [15]

```
call 3
go_up
reach 2
reach 3
stop
door_open 3 103325
door_close 3 103340
go_down
reach 2
stop
door_open 2 103355
```

- The elevator stops (line `stop`) and the door opens on floor 3, at time `103325` (lines `stop` and `door_open 103325`).
- The elevator door closes on floor 3, at time `103340` (line `door_close 3 103340`).

The rest of the lines are interpreted in a similar way.

2.3.2 Expected program behavior

Throughout section 2.2, we gave an overview of different ways to derive oracles. Oracles need to capture the expected program behavior in some way. The first step would be to list a set of requirements for the elevator controller [15].

1. The door of the elevator must be closed if the elevator is moving.

المنارة للاستشارات

www.manaraa.com

2. The elevator must not be moving when the controller program terminates.

3. The door must never stay open more than 30 seconds.

Having enumerated the requirements to be checked, we can now examine the log file to see if it reveals any fault in the controller. Log files often contain several *threads of information*. For instance, if we wanted to check requirements 1, 2 and 3, we would need to pay attention to lines such as `door_open`, `go_up` and `stop`. Lines with keywords `reach` or `call` are not relevant for requirements 1, 2 or 3. In fact, such lines might be relevant to checking *other* requirements. Threads of information are often arbitrarily interleaved in log files.

2.3.3 Log File Analyzers

Taking into consideration the fact that log files often contain interleaved threads of information, a log file analyzer can be viewed as a set of simpler programs. Andrews [15] explains that each program "notices" a set of log lines that represent a thread of information, to check one or more closely related requirements. More formally, these programs are state machines running in parallel. Each state machine recognizes or "notices" only a subset of the lines in the file and makes transitions triggered by the log file line it notices.

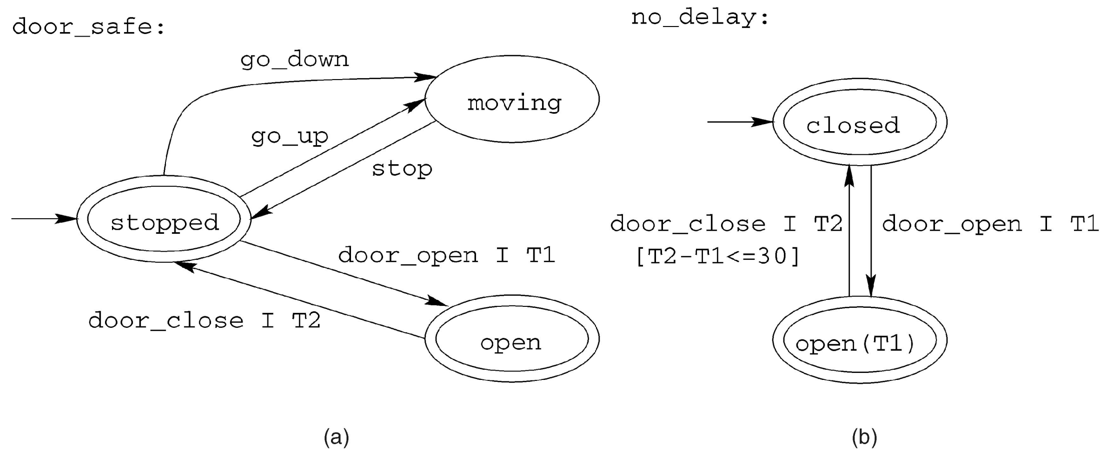
State machines in a log file analyzer report an error when one of these conditions occur:

- A line is not noticed
- A line is noticed, but the state machine cannot make a transition on it.

المنارة للاستشارات
www.manaraa.com

Figure 2.2 shows an example of two state machines that check for the elevator controller requirements outlined in section 2.3.1. The machine shown at the left checks requirements 1 and 2. The machine at the right checks for requirement 3.

Figure 2.2 Analyzer machines for the elevator controller by Andrews [15]



It is important to note that log file analyzers are composed of *infinite*-state machines. This is necessary because log files often contain data that must be remembered by an analyzer to judge a later line. For example, in figure 2.2 at the right, we can observe the `no_delay` machine, where the state `open` is parameterized. That allows the analyzer to "remember" the time at which the elevator door was opened and thus, determine if the door stayed open for more than 30 seconds.

2.3.4 Log File Analysis Language

Andrews introduced the Log File Analysis Language (LFAL), which is used to specify the expected program behavior. This expected behavior is represented in the form of state machines. Figure 2.3 shows the LFAL specification for the `no_delay` machine.

المنارة للاستشارات
www.manaraa.com

We can observe that the language is a straightforward representation of the `no_delay` machine that defines the initial state of the machine, the transitions along with their corresponding conditions and the final state.

Figure 2.3 An example of a LFAL specification by Andrews [15]

```
machine no_delay;
  initial_state closed;
  from closed, on door_open(I,T1), to open(T1);
  from open(T1), on door_close(I,T2),
    if (T2-T1 =< 30), to closed;
  final_state Any.
```

2.3.5 Log file analyzer generation

The implementation of LFA includes a translator, an auxiliary library and a compiler script. The translator translates an LFAL specification into Prolog code. The compiler script compiles the Prolog code and produces an executable that can be used to analyze log files.

Log file analysis has been applied to several lab-built pieces of software such as an elevator controller and a heater monitor [15] and to two pieces of commercial software [35]. While it proved to be useful, the use of the Prolog as the target programming language posed some difficulties that could be addressed by using a more familiar programming language such as C++ or Java.

المنارة للاستشارات
www.manaraa.com

2.4 Other related work

To conclude this chapter, in this section we present related work on log file analysis, applied to software engineering in general.

Chang and Ren [18] developed a Test Behavior Language (TBL), which uses parameterized patterns as logical predicates. Using TBL, software properties can be validated through execution traces. The authors define patterns through regular expressions. As we will see in the following chapters, we incorporate the use of regular expressions to our work to make log file analyzers more flexible and adaptable.

Tan et al. [31] introduced a framework called SALSA, which stands for "Analyzing Logs as State Machines". Its objective is to derive a program's control flow in terms of state machines. This information is derived from log files. It is very different from Andrews' approach in that its objective is to derive state machines from a log file, while Andrews uses state machines to capture the expected program behavior *a priori*. SALSA is intended to help developers understand what a program is doing by helping them visualize a program execution in terms of state machines. In contrast, Andrews' LFA is intended to be applied to software testing.

المنارة للاستشارات
www.manaraa.com

Chapter 3

The Log File Analyzer Generator

In this chapter, we describe the design of our Log File Analyzer Generator. We begin by discussing the LFAL 2.0 language in section 3.1. We explain how LFAL 2.0 programs are transformed into C++ code in section 3.3. Finally, we describe how log file analyzers are used as oracles to analyze log files and reveal faults in the software under test (SUT) in section 3.4.

Figure 3.1 shows a high-level view of the architecture of our Log File Analyzer Generation Process, by illustrating its different stages. The process starts with an LFAL 2.0 program. This code is scanned and parsed resulting in an abstract syntax tree (AST) (see section 3.2). The AST is analyzed by the code generator and the following files are generated: C++ code for machine classes (see section 3.3.2), a Makefile and C++ code for the log file analyzer program (see section 3.3.3). Using the Makefile and a C++ compiler, these files are compiled and linked with the base libraries (see section 3.3.1). At the end of the process, we obtain an executable log file analyzer (see section 3.4). We will discuss this process in more detail in the following sections.

المنارة للاستشارات
www.manaraa.com

3.1 The LFAL 2.0 specification

In section 1.3 in chapter 1, we mentioned that to analyze a log file, we need to have captured a program's expected behavior. We use the Log File Analyzer Language (LFAL), which was introduced by Andrews [13], for that purpose. One of the objectives of this thesis is to extend LFAL so that we can incorporate new features that take advantage of the C++ programming language and make log file analyzers more flexible. Henceforth, we will refer to our extended version of LFAL as LFAL 2.0.

In this section, we describe the LFAL 2.0 language specification. In section 3.1.1, we give an overview of the different elements of an LFAL 2.0 program. In section 3.1.2, we introduce the new elements of the LFAL 2.0 language and its specification.

3.1.1 The elements of an LFAL 2.0 program

LFAL 2.0 captures a program's expected behavior as a set of state machines. A state machine is composed of a set of unique states, an initial state and one or more final states. A machine changes from one state to another through transitions. Transitions are triggered by events [32].

An LFAL 2.0 program can be seen as a set of *analyzer machines*. Transitions in analyzer machines are triggered by log lines. An analyzer machine has the same basic elements as a generic state machine. Such elements are:

المنارة للاستشارات
www.manaraa.com

- **State definitions**, which define the states of an analyzer machine. States can have *state variables*. For example, a state can be declared as `state RECEIVED (int Time)`. The state would be `RECEIVED` and the state variable would be `Time`.

- **Initial state definition**, which specifies the state in which an analyzer machine is at the start.

- **Transitions**. These elements define the way analyzer machines change from one state to another. Transitions can include conditions and, if successful, cause the analyzer machine to change to another state. As we will see in the next section, transitions in LFAL 2.0 are very flexible and can involve other actions besides changing the machine state.

- **Final states**. An analyzer machine can have one or more final states. These indicate the state the analyzer machine must be in, before exiting.

Figure 3.2 shows an example of an LFAL 2.0 program with the definition of an analyzer machine called `elevatorState` (see line 5). This LFAL 2.0 program specifies the expected behavior of a very simple elevator controller program. We can observe that `elevatorState` has two state definitions: `on_service` and `normal` (lines 8 and 9). The initial state for `elevatorState` is `on_service` (line 12). Lines 15 and 16 contain the transitions, which change the state of the analyzer machine triggered by the `go_on_service` or `go_off_service` lines. The final state definition is shown on line 19. We will describe in more detail how the machine in this example works in section 3.4.

www.manaraa.com

3.1.2 New features in the LFAL 2 language

LFAL 2.0 retains the elements of the original LFAL, which include the basic analyzer machine elements outlined in section 3.1.1.

Figure 3.3 shows a summarized version of the original specification of LFAL [15] in Backus Normal Form (BNF). Figures 3.4 and 3.5 show the BNF specification of the LFAL 2.0 language.

LFAL 2.0 extends LFAL by incorporating the following new elements:

- **Pattern definitions**. Used to incorporate regular expressions into analyzers (see rule `<pat-def>` in figure 3.5). An example of a pattern definition in LFAL 2.0 is shown in figure 3.2, lines 2 and 3.
- **Data declaration**. This element can be used to incorporate C++ objects into analyzers (see rule `<data-decl>` in figure 3.5).
- **State definitions**. LFAL 2.0 supports C++ data types. Therefore, in LFAL 2.0, states need to be declared together with their state variables names and types (see rule `<state-def>` in figure 3.5). An example of the syntax is shown in figure 3.2, lines 8 and 9.
- **Transitions**. The original version of LFAL allows the user to define a series of transitions for the analyzer. This remains true for LFAL 2.0. However, by introducing new kinds of actions, transitions in LFAL 2.0 are much more flexible. For example, a transition can be defined to execute C++ code, create or delete an analyzer machine object, show a specific error message, etc. (see rules `<trans-def>` and `<action>` in figure 3.5). Examples of transitions in an

المنارة للاستشارات
www.manaraa.com

LFAL 2.0 program are shown in figure 3.2, lines 15 and 16.

We will explain the purpose and benefits of these elements in more detail in chapter 4.

3.2 Scanning and Parsing LFAL 2.0

3.2.1 Lexical Analysis

The analysis of the syntax of a program can be divided in two parts: *lexical analysis* (also called scanning or lexing) and *parsing*. This section focuses on the lexical analysis of LFAL 2.0.

Scanning divides the input into meaningful pieces, known as *tokens*. Scanners work by looking for patterns of characters in the input. These patterns are known as *regular expressions* [24]. In terms of lexical analysis, LFAL 2.0 is defined by a set of reserved words. We implemented our lexical analyzer using *Flex*. Flex is an open-source tool for generating programs that perform pattern-matching on text [4]. Our scanner works in conjunction with a parser. When a token is recognized, our scanner passes it to the parser so it can be processed.

3.2.2 Syntax Analysis

Syntax analysis is also known as parsing. A *parser* is a program that reads a stream of tokens and determines their relationship [24]. The relationship between tokens can

المنارة للاستشارات
www.manaraa.com

be represented as a set of *rules* like the ones shown in figures 3.4 and 3.5. These rules specify how tokens in LFAL 2.0 can be combined to define a log file analyzer. We implemented our parser using *Bison*. Bison is an open-source parser generator [3]. The scanner generated by Flex processes the input (a LFAL 2.0 program), detects tokens and passes them to the Bison parser. Our Bison parser builds an *Abstract Syntax Tree* (AST), which is a representation of the syntactic structure of input written in a programming language [24].

3.3 The log file analyzer generator

In the previous section, we explained how LFAL 2.0 is parsed to obtain an AST. This AST contains the LFAL 2.0 program structure. It is used by the Log File Analyzer Generator to produce C++ code. The generated C++ code can be divided in two parts: the code for machine classes and the code for the log file analyzer main program. These two parts, together with several base libraries, are compiled and linked to produce an executable log file analyzer. In this section we describe how machine classes, libraries and the log file analyzer work together to provide the necessary functionality for a log file analyzer.

3.3.1 Base libraries

We have developed a set of libraries that encapsulate the basic functionality needed by all machine classes. In this way, we minimize the amount of generated code, thus avoiding the generation of repetitive code. Therefore, the C++ code for these classes is not generated for every log file analyzer. Rather, log file analyzers use these libraries

المنارة للاستشارات
www.manaraa.com

through mechanisms such as inheritance or aggregation.

Our base libraries contain the following functionality:

- **State Machine.** Contains the basic functionality needed for the analyzer machines to work. It includes routines that manage the machine's transition table and states.
- **Log File Parser.** Contains the data structures and methods that analyzer machines need to recognize certain lines (regular expressions) in a log file.
- **Shared Memory Management.** Contains data structures and methods to create, open, and delete shared memory areas. In addition, this library contains methods to create and manage log files in a shared memory area.

3.3.2 Machine Classes

LFAL 2.0 programs can have one or more analyzer machines. Each analyzer machine can check for one or more requirements. That is, an LFAL 2.0 program can have one or more types of analyzer machines, each noticing a different set of lines and requirements. We refer to them as *machine classes.* Figure 3.6 shows an example of an LFAL program with two machine classes: `doors` (shown in lines 10–29) and `mem` (lines 31–46). We will use figure 3.6 as a running example to illustrate the concepts presented here.

The log file analyzer generator produces C++ code for each machine class. That is, for this example, it would generate a class for the analyzer machine `doors` and for the analyzer machine `mem`.

www.manaraa.com

In our implementation, each machine class has a transition table in the form of an array of structures. Each transition record contains the source and target states and a pointer to a *transition method*. Transition methods contain C++ code that defines the *action* to execute. The most common action is to change the state of the machine. However, depending on the contents of the LFAL 2.0 program, the transition method might contain other C++ code such as conditions or actions specified by the user. In the next chapter, we will describe the advanced features of LFAL 2.0 in more detail.

3.3.3 The Log File Analyzer Program

Once C++ code is generated for the machine classes, the code generator has the necessary elements to generate C++ code for the main log file analyzer program. This program instantiates the machine classes and processes the log file.

In general, the log file analyzer program performs the following actions:

1. Open log file, standard input or shared memory log file, according to the option specified at the command line.
2. Instantiate parser object and configure. This sets up the parser to match log file lines.
3. Initialize machine classes. For each machine class:
   (a) Create and initialize transition table
   (b) Instantiate machine class and add to vector of analyzer machines
   (c) Set the analyzer machine's initial state

المنارة للاستشارات
www.manaraa.com

4. The log file analyzer program enters an infinite loop that ends until the entire log file is processed. Inside the loop, the analyzer performs the following actions:
   (a) Read log file line
   (b) Determine whether the read line is noticed by a machine
   (c) For each machine in the vector of analyzer machines:
      i. If the line is noticed by the machine, call its transition method.
      ii. Display error message if applicable. For example, if the machine noticed the line but was unable to make a transition, an error message will be displayed.
5. At end of file, report whether the log file was accepted or rejected.

3.4 Analyzing log files

Once the log file analyzer generation is complete, the executable log file analyzer can be used to determine if a log file reveals errors in a program.

In this section, we present two examples that illustrate how our generated log file analyzer processes log files.

Our first example is the LFAL 2.0 program shown in figure 3.2. This LFAL 2.0 program specifies a log file analyzer consisting of one machine class: `elevatorState`. An elevator is usually known to be "on service" when it is being controlled *exclusively* by an operator who inserts a key inside the keyhole in its control panel. This state is meant to indicate that the elevator is undergoing some kind of maintenance. Once the elevator is ready for normal operation, it is known to be "off service".

المنارة للاستشارات
www.manaraa.com

The elevator system is expected to start up on service and eventually be taken off service in order to operate normally. After some time, the elevator might be put on service again to perform repair or maintenance tasks and then taken off service again. This is expected to happen several times. Evidently, a log file produced by the elevator controller software should reflect that the elevator was on service before the controller program exited.

The `elevatorState` analyzer machine checks that the following requirements are met:

1. The elevator state alternates from being "on service" to being in normal operation mode ("off service") and vice versa

2. The elevator system starts up and shuts down while "on service"

Lines 2 and 3 define two patterns: `go_on_service` and `go_off_service`. These patterns define two regular expressions. For instance, the line `''elevator put on service''` would be matched by the pattern `go_on_service` on line 2. Lines 9 and 10 show the two possible states for this machine: `on_service` and `normal`. Line 12 sets `on_service` as the initial state of the machine. The analyzer machine's transitions are defined in lines 15 and 16. When the machine recognizes (or "notices") a line such as `''elevator taken off service''`, the machine changes to the state `''normal''`. Similarly, if the analyzer machine notices a line of the type `''elevator put on service''`, the analyzer machine will transition to the `on_service` state. Line 19 defines the final state. Therefore, when the end of the log file is reached, the machine should be in the state `on_service`.

If we translated our example LFAL 2.0 program shown in figure 3.2 with our log file analyzer generator, we would obtain (after compilation) an executable log file

المنارة للاستشارات
www.manaraa.com

analyzer. Let us suppose we gave the resulting log file analyzer the following log file as an input:

```
1 elevator taken off service
2 elevator put on service
```

In this case, the log file analyzer would *accept* this log file. The line ``elevator taken off service'' would cause the analyzer machine elevatorState to transition to the state normal. Then, the line ``elevator put on service'' would cause the machine to transition to the state on_service. Since we have reached the end of the log file, the analyzer reports that the log file is *correct*.

Let us look at an example of a log file that would cause this example log file analyzer to report an error:

```
1 elevator taken off service
2 elevator put on service
3 elevator taken off service
```

The process for the first two lines would be identical as in the previous example. However, the third line ``elevator taken off service'' causes the elevatorState analyzer machine to transition to the normal state. Therefore, when the log file analyzer reaches the end of the file, the analyzer machine would *not* be on the required state on_service. Therefore, the log file above would be *rejected*, because the analyzer detected that the elevator was in normal operation mode when the elevator controller program ended.

Let us analyze a slightly more complicated example. Figure 3.6 shows an LFAL 2.0 program of a log file analyzer with two machine classes: elevatorState (the same as

المنارة للاستشارات
www.manaraa.com

in the previous example which checks that the elevator starts and ends "on service") and doors. In this case, let us suppose that we are trying to determine if an elevator controller program is working correctly by analyzing two aspects of its operation: its status (off/on service) and the doors. To define the expected program behavior, we write the LFAL 2.0 code shown in figure 3.6. Let us suppose we want to check for the following requirements:

1. The elevator starts and ends in the expected state (on_service).
2. The states alternate as expected.
3. The elevator door must never be open while the elevator is moving.
4. The elevator door must not stay open more than 30 seconds.

To check for these requirements, we define two machine classes: elevatorState, which checks for requirements 1–2, and doors, which checks for requirements 3–4.

We have mentioned that log file analyzers can process different threads of information. This example will give an example of that. Let us suppose we have the following log file:

```
1 elevator taken off service
2 door open at 1
3 door close at 5
4 go up
5 door open at 8
6 elevator put on service
```

المنارة للاستشارات
www.manaraa.com

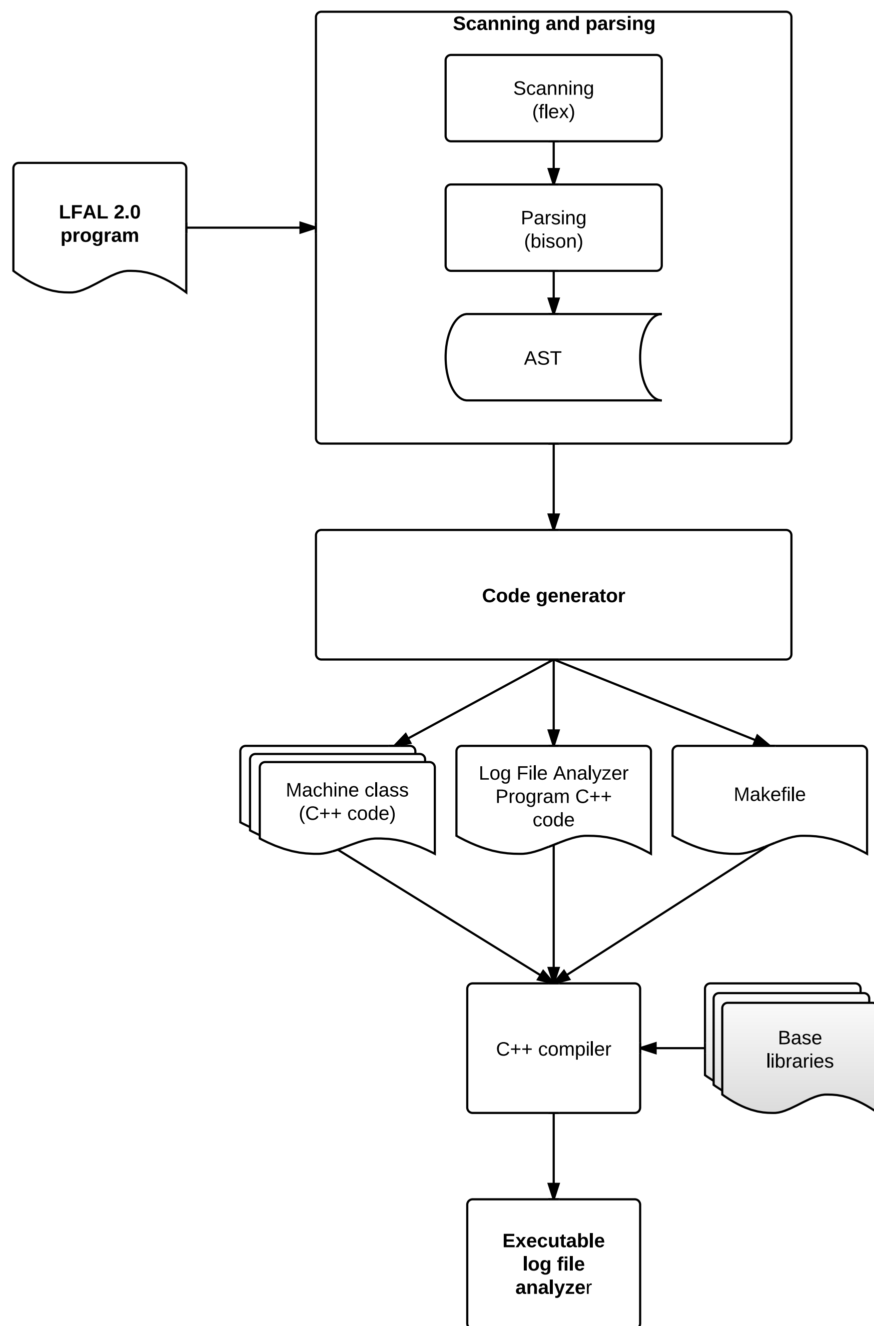
It is important to note that the machine doors has slightly more complicated pattern definitions. In figure 3.6, lines 4 and 5, we can observe two examples of patterns with variables. For instance, the pattern ''door_open (int VAR1)'' in line 4, defines a regular expression where the variable VAR1 saves a timestamp as an integer (int). Therefore, the line ''door open at 1'' would be matched by the ''door_open (int VAR1)'' pattern. The value 1 would be stored in the variable VAR1. We will discuss patterns in more detail in chapter 4, section 4.2.

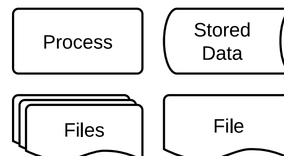
The elevatorState machine will notice line 1 (elevator taken off service). The doors machine won't. Similarly, elevatorState won't notice line 2 (door open at 1) but doors will. We can observe that each analyzer machine processes two different threads of information in the same log file. The log file analyzer will *reject* this log file, because after reading line 4 of the log file, the doors machine will be in the state moving. After reading line 5, doors will try to make a transition. However, by looking again at the code in figure 3.6, we notice that there is no transition from the state moving to the state open(int T1). The elevatorState notices line 6 and transitions to the state on_service. This log file would therefore be *rejected* because it reveals a fault in the elevator controller program (i.e. the doors analyzer machine detected that the elevator's door was opened while it was moving).

In this chapter we gave an overview of our log file analyzer generator. We discussed the LFAL 2.0 language and provided some basic examples. In the next chapter, we will focus on the new features of LFAL 2.0 (see section 3.1.2) in more detail.

المنارة للاستشارات
www.manaraa.com

Figure 3.1 Log File Analyzer Generation Process





www.manaraa.com

Figure 3.2 A simple example of a LFAL 2.0 program

```
1   //Pattern definitions
2   pattern go_on_service "elevator put on service"
3   pattern go_off_service "elevator taken off service"
4
5   machine elevatorState = {
6
7     //State definitions
8     state on_service;
9     state normal;
10
11    //Initial state
12    initial state on_service;
13
14    //Transition definitions
15    from on_service, on go_off_service, to normal;
16    from normal, on go_on_service, to on_service;
17
18    //Final state
19    final state on_service;
20   }
```

Figure 3.3 The original LFAL specification, by Andrews [15]

```
<analyzer>      ::= {<machine-desc> | <program-text>}*

<machine-desc>  ::= machine <term> {; <decl>}*

<decl>          ::= initial_state <term>;
                |   <trans-clause> [, <trans_clause>]*;
                |   final_state <term>;

<trans-clause>  ::= from <term>
                |   on <term>
                |   to  <term>
                |   if <condition>
                |   where <condition>

<term>          ::= <keyword> | <string> | <number> | <var>
                |   <keyword>"("<term> [, <term>]*")"
```

www.manaraa.com

Figure 3.4 The LFAL 2.0 specification, part 1

```
<term>        ::= <term1>
              | <term1> <arithop> <term1>

<term1>       ::= <varname> | <int-literal> | <float-literal>
              | <char-literal> | <string-literal>
              | <varname> "." <funcname> "(" [<term> ["," <term>]*] ")"
              | "(" <term> ")"

<arithop>     ::= "+" | "-" | "/" | "%" | "*"

<condition>   ::= <cond1>
              | <cond1> "&&" <cond1>
              | <cond1> "||" <cond1>
              | "!" <cond1>

<cond1>       ::= <term> <relop> <term>
              | <varname> "." <funcname> "(" [<term> ["," <term>]*] ")"
              | "(" <condition> ")"

<relop>       ::= "==" | "!=" | "<" | ">" | "<=" | ">=" | "matches"
```

المنارة للاستشارات
www.manaraa.com

Figure 3.5 The LFAL 2.0 specification, part2

```
<analyzer>          :: = [<pat-def>]* [<machine-desc>]*

<pat-def>           :: = "pattern" <abst-format> <regexp>

<abst-format>       ::= <keyword> ["(" <param-def> ["," <param-def>]* ")"]

<param-def>         ::= <type> <varname>

<type>              ::= "int" | "char" | "float" | "string" | "time"

<machine_desc>      ::= "machine" <keyword> "=" "{" <machine-body> "}"

<machine-body>      ::= [<state-def>]* [<data-decl>] [istate-decl>]
                        [<trans-def>]* [<fstate-decl>]*

<state-def>         ::= "state" <abst-format> ";"

<data_decl>         ::= "data" <typename> <varname> ";"

<istate-decl>       ::= "initial" "state" <state-expr> ";"

<fstate-decl>       ::= "final" "state" <state-matcher> ";"

<trans-def>         ::= ["from" <state-matcher> ","] "on" <pat-matcher> ","
                        ["if" "(" <condition> ")" ","] <action> ";"

<state-matcher>     ::= <keyword> ["(" <varname> ["," <varname>]* ")"]

<pat-matcher>       ::= <keyword> ["(" <varname> ["," <varname>]* ")"]

<action>            ::= "doing" "{" <source-code> "}" "," <action>
                        | "create" "new" "in" <state-expr>
                        | "to" <state-expr>
                        | "error" [<string-literal>]
                        | "ignore" [<string-literal>]
                        | "stay" [<string-literal>]
                        | "delete"

<state-expr>        ::= <keyword> ["(" <term> ["," <term>]* ")"]
```

المنارة للاستشارات

www.manaraa.com

Figure 3.6 An LFAL 2.0 program with two machine classes

```
1  //Pattern definitions
2  pattern go_on_service "elevator put on service"
3  pattern go_off_service "elevator taken off service"
4  pattern door_open (int VAR1) "door open at (?<VAR1>)"
5  pattern door_close (int VAR2) "door close at (?<VAR2>)"
6  pattern go_up "go up"
7  pattern go_down "go down"
8  pattern stop "stop elevator"
9
10 machine elevatorState = {
11
12   //State definitions
13   state on_service;
14   state normal;
15
16   //Initial state
17   initial state on_service
18
19   //Transition definitions
20   from on_service, on go_off_service, to normal;
21   from normal, on go_on_service, to on_service;
22
23   //Final state
24   final state on_service;
25 }
26
27 machine doors = {
28
29   //State definitions
30   state stopped;
31   state moving;
32   state open(int T1);
33
34   //Initial state
35   initial state stopped;
36
37   //Transition definitions
38   from stopped, on go_up, to moving;
39   from stopped, on go_down, to moving;
40   from moving, on stop, to stopped;
41   from stopped, on door_open(T1), to open(T1);
42   from open(T1), on door_close(T2), if ( T2 - T1 <= 30 ), to stopped;
43
44   //Final state
45   final state stopped;
46 }
```

المنارة للاستشارات
www.manaraa.com

Chapter 4

New Features in Analyzers

In chapter 3, section 3.1.2 we gave an overview of how LFAL 2.0 was extended to support new features.

In this chapter, we describe the new features of LFAL 2.0 analyzers in more detail and we explain the benefits that they provide. In addition, we provide examples that illustrate how these new features are used in log file analyzers.

4.1 C++ Analyzers

The most notable difference between the original version of LFAL and LFAL 2.0 is that log file analyzers are generated in the C++ programming language instead of Prolog. We decided to use C++ because we expect LFAL 2.0 analyzers to have better performance with respect to their Prolog counterparts. In addition, C++ is a more commonly–used and more widely available language.

للاستشارات

www.manaraa.com

C++ allows log file analyzers to take advantage of the object–oriented paradigm. As we mentioned in chapter 3, section 3.3.2, the generated log file analyzer code is organized in classes, which encapsulate the functionality of each machine class. In that way, machine classes can be instantiated, allowing the analyzer to create or delete new analyzer machines as needed. As we will see, this is very important for some of the new features of LFAL 2.0 analyzers.

It is important to mention that most of the features described in this chapter need a fast, object-oriented and modern programming language such as C++.

4.2 Pattern definitions

One of the main challenges in log file analysis is that log file formats differ significantly from system to system. In addition, log lines often are a complex combination of words, numbers and other types of data. For that reason, it is important for log file analyzers to be flexible regarding the format of the log file lines they can process. In that way, log file analyzers can be adapted to existing systems and their corresponding logging policies. To achieve this, we have incorporated regular expressions into LFAL 2.0 log file analyzers.

A *regular expression* is a specific kind of text pattern that can be used to recognize or "match" strings of text [21]. There are many different regular expression implementations. Perl–compatible regular expressions are among the most popular implementations [20]. One of the most popular Perl–compatible implementations is called PCRE (Perl–Compatible Regular Expressions). We decided to use this implementation because it is provided as an open-source library and is used in many

المنارة للاستشارات
www.manaraa.com

programming languages and applications. In fact, PCRE is built into PHP and is used in high–profile open source projects such as Apache, KDE, Postfix, Nmap, etc. PCRE is also used in commercial software, such as Apple's Safari browser [7].

In LFAL 2.0, we define *patterns* which specify the format of log lines, so that they can be matched by analyzer machines. According to LFAL 2.0 syntax (see rule `<pat-def>` in figure 3.5), a pattern is defined by specifying three elements: a pattern name, a list of variables enclosed in parentheses and a string with a regular expression. In an LFAL 2.0 program, patterns are defined at the top, before the definition of the machine classes. The reason for this is that several machine classes can notice a pattern. Figure 4.1 shows an excerpt of an LFAL 2.0 program, which shows an

Figure 4.1 A pattern definition

```
1  pattern temp(float T) "The temperature is (?<T>)"
2
3  machine example {
4  state normal;
5  state highTemp;
6
7  from normal, on temp(T), if(T > 50), to highTemp;
...
```

example of a pattern definition on line 1. The name of the pattern `temp` is in bold. In this example, the pattern specifies that a float value is to be captured by declaring the variable T (underlined in line 1). The regular expression in this example is `‘‘The temperature is (?<T>)’’`. In this case the `T` indicates the position of the variable `float T` in the pattern. This pattern would match lines such as `‘‘The temperature is 9.4’’` or `‘‘The temperature is 46.3’’`.

As we can observe in line 7, figure 4.1, a pattern is noticed by a machine if it appears

المنارة للاستشارات
www.manaraa.com

in one of its transitions. Variables in log file lines, are used by analyzer machines in a variety of ways. In this example, the variable `T` is used to evaluate if the temperature value in a log line is greater than 50. If the condition is met, the machine transitions to the state `highTemp`.

It is important to mention that the syntax `(?<T>)` is part of the standard PCRE syntax for named substrings. The parentheses indicate that a substring is to be captured. In PCRE named substrings are enclosed between `?<` and `>`. We emphasize this fact because we designed LFAL 2.0 patterns so that developers with experience in PCRE or Perl–style regular expressions, can use them without having to learn new syntax.

The incorporation of regular expressions to LFAL 2.0 allows log file analyzers to process log lines with a much more complex format. For example, consider figure 4.1. We have explained that the user can define flexible patterns, such as in the case of the pattern `temp(int T)`. In the original version of LFAL, analyzers did not have a way to specify a pattern for a log line. Therefore, if the example shown in figure 4.1 were written in the original version of LFAL, the analyzer could only look for lines such as `‘‘temp 9.4’’` or `‘‘temp 46.3’’`. That makes it difficult for analyzers to adapt to existing log files. In fact, in chapter 5, where we compare LFAL against LFAL 2.0 analyzers, we describe how we had to modify an existing log file so that it could be analyzed with a Prolog-based analyzer.

المنارة للاستشارات
www.manaraa.com

4.3 Dynamic and static analyzer machines

In this section, we introduce one of the most important new features of LFAL 2.0. The examples we have seen so far involve what we call *static machines*. Static machines have two main features: They need an initial state and only one instance of them exists in a log file analyzer. These kind of machines are used to analyze aspects that involve the whole log file.

In log files, it is common to find groups of lines, which are related to a specific identifier such as a transaction ID number. *Dynamic machines* are used to analyze this kind of log file. Dynamic analyzer machines are created "on the fly" by a special type of transition that creates a new analyzer machine when a certain log line is noticed. Dynamic analyzers do not need initial states because the transition that triggers the creation of the machine specifies its initial state.

To clarify the difference between static and dynamic machines, we will contrast two examples of log file analyzers: one using a static machine and one using a dynamic machine. The objective of both analyzers is to check for memory leaks. However, as we will see, this kind of analysis can only be performed effectively using dynamic machines.

4.3.1 Static analyzer machines

In figure 4.2 we present an example of an LFAL 2.0 program with a static machine that checks for memory leaks. Line 7 contains the initial state definition. This machine intends to check that for every memory allocation, there is a corresponding deallocation. Line 16 shows the transition definition where, upon noticing a line

المنارة للاستشارات
www.manaraa.com

Figure 4.2 An example of a static machine

```
1  //Pattern definitions
2
3  pattern malloc (int Ptr) "malloc returned (?<Ptr>)"
4  pattern free (int Ptr2) "free called on (?<Ptr2>)"
5
6  machine staticMem = {
7
8    //State definitions
9    state unalloc;
10   state alloc(int Ptr);
11
12   //Initial state
13   initial state unalloc;
14
15   //Transition definitions
16   from unalloc, on malloc(Ptr), to alloc(Ptr);
17   from alloc(Ptr), on free(Ptr2), if(Ptr==Ptr2), to unalloc;
18
19   //Final state unalloc
20   final state unalloc;
21 }
```

with the pattern `malloc(int Ptr)`, the machine transitions to the state `alloc(Ptr)`. In line 17, we can observe that when the machine notices a line with the pattern `free(int Ptr2)`, the machine transitions to the state `unalloc` *if* the address in the log line corresponds to the address saved in the state variable `alloc(Ptr)`. For example, let us suppose we have the following log file:

```
1 malloc returned 100

2 free called on 100

3 malloc returned 200

4 free called on 200
```

The log file analyzer produced by the LFAL 2.0 program shown in figure 4.2 would process the log file above like this: The static machine `mem` would notice the line

المنارة للاستشارات

www.manaraa.com

"`malloc returned 100`". That would cause the machine to transition to the state `alloc(100)`. Then, the line "`free called on 100`" would cause the machine to transition to the state `unalloc`. It is important to note that the condition that precedes the transition (`Ptr==Ptr2`), is true. That is, the number in the log line "`free called on 100`" (`Ptr2`) is equal to the value saved in the state variable `malloc(Ptr)`, which is `100`. Lines 3 and 4 in the example log file above are processed in a similar way. When the end of the log file is reached, the analyzer reports that the log file is *correct*, because the machine *mem* is in its correct final state `unalloc`.

The previous example is limited because this analyzer will only check for memory leaks correctly *if* there is ***at most*** one block of memory allocated at a time. Thus, such an analyzer would be unable to check *any* sequence of malloc and free calls to detect memory leaks successfully. For example, consider the log file shown in figure 4.3. In this case, the log file would report an error in line 4 of the log file. Upon

Figure 4.3 An example log file

```
1  malloc returned 100
2  free called on 100
3  malloc returned 200
4  malloc returned 300
5  free called on 200
6  free called on 300
```

processing line 4, the static machine `mem` would be in state `alloc(200)`. Because static machines have only *one* instance in a log file analyzer, the machine `mem` would report an error indicating that it was unable to perform a valid transition with the line `malloc returned 300`. That happens because although the machine notices the line, it cannot transition from `alloc(200)` to `alloc(300)`.

المنارة للاستشارات
www.manaraa.com

4.3.2 Dynamic analyzer machines

From the previous example we can see that *dynamic* analyzer machines are necessary to correctly analyze log files with *more than one instance* of a group of log lines. For a memory leak checker, we need the log file analyzer to be able to process "malloc" and "free" lines in any possible sequence.

Figure 4.4 An example of a dynamic machine

```
1  //Pattern definitions
2  pattern malloc(int Address) "malloc returned (?<Address>)"
3  pattern free(int Address) "free called on (?<Address>)"
4
5  machine memDynamic = {
6
7    //State definitions
8    state allocated(int A);
9    state unallocated(int A);
10
11   //'Create' transition
12   on malloc(A), create new in allocated(A);
13
14   //'Delete' transition
15   from allocated(A), on free(B), if (A==B), delete;
16
17   //'Ignore' transition
18   from allocated(A), on free(B), if (A!=B), ignore;
19
20   //'Error' transition
21   from allocated(A), on malloc(B), if (A==B),
22       error "Error: Malloc had already been called for that address";
23
24   //'Ignore' transition
25   from allocated(A), on malloc(B), if (A!=B), ignore;
26
27   final state unallocated(A);
28  }
```

***Create* transitions.** Dynamic machines contain a special type of transition which creates a new analyzer machine upon noticing a specific log line. We refer to this

المنارة للاستشارات
www.manaraa.com

type of transition as a *create transition*. Figure 4.4 shows an example of a dynamic machine. In this example, we observe the definition of a log file analyzer comprised of one dynamic machine class called `memDynamic`. This analyzer machine does not have a initial state definition. Line 12 shows the *create transition*. This transition specifies that upon noticing the pattern `malloc(int Address)`, the log file analyzer will instantiate a new `memDynamic` machine. In fact, the log file analyzer starts with *zero* machines. As the log file analyzer encounters lines with the pattern `malloc(int Address)`, it creates new instances of the `memDynamic` machine.

***Delete* transitions.** Dynamic machines can also contain transitions that delete an analyzer machine upon noticing a certain log line. We refer to these type of transitions as *delete transitions*. Line 15 in figure 4.4 shows an example of a delete transition. When a line with the pattern `free(int Address)` is noticed, the analyzer machine instance is deleted. In this case, the `delete` statement is preceded by a condition. The condition checks whether the address specified in the log file is the same as the address contained in the state variable `allocated(int A)`.

Let us suppose that we are analyzing the log file shown in figure 4.3. Lines 1, 3 and 5 in the log file would trigger the creation of instances of the `memDynamic` machine shown in figure 4.4. Lines Lines 3, 5 and 6 in the log file will trigger the deletion of an instance of `memDynamic`. The basic logic of this log file analyzer is to create an instance of a `memDynamic` machine for each malloc call. If the analyzer finds the corresponding free for a malloc, it deletes the related `memDynamic` instance. It is important to note, however, that our description of the `memDynamic` analyzer machine is incomplete without describing other types of transition actions that are important for the correct functioning of dynamic machines. In the next section, we will describe these special types of transitions. We will also complement our explanation of how

المنارة للاستشارات
www.manaraa.com

the example log file in figure 4.3 would be processed.

4.4 New types of actions in transitions

As we have mentioned previously, a log file analyzer is comprised of one or more analyzer machines. An analyzer machine is a state machine that performs transitions, which are triggered by log file lines. Transitions usually involve a change in the state of the analyzer machine. LFAL 2.0 introduces new types of actions that allow transitions that besides changing the state of the machine can perform other actions such as executing C++ code or printing user-defined messages. These new transition actions provide the user with a fine-grained control over analyzer machine transitions in LFAL 2.0.

In this section we describe these new types of transition actions. We will continue to use the analyzer in figure 4.4 as a running example to illustrate how some of these new transition actions work.

4.4.1 "Error" transitions

In chapter 2 section 2.3.3, we mentioned the conditions that cause log file analyzers in the original version of LFAL to report errors: When a line is not noticed by a machine and when a line is noticed but the machine cannot make a transition on it.

LFAL 2.0 introduces a new type of transition action called *error*. We refer to transitions that contain error actions as *error transitions*. These kind of transitions are used to differentiate the case when the analyzer machine does perform a valid transi-

المنارة للاستشارات
www.manaraa.com

tion, but the user wants to mark such a transition as an error. Error transitions are defined by the keyword `error` and a user-defined error message.

Figure 4.4 shows an example of an error transition (see lines 21–22). This error transition provides the `memDynamic` machine with the capability of detecting two or more "malloc" log lines with the same address. Consider the following log file:

```
1 malloc called on 100
2 malloc called on 100
```

From our previous discussion in section 4.3.2, the line `malloc called on 100` would trigger the creation of a new instance of the `memDynamic` machine in the `alloc(100)` state. Let us call this instance `m1`. Line 2, would also trigger the creation of a new instance, also in the `alloc(100)` state. Let us call that instance `m2`. Instance `m1` would perform an error transition because it would find that the address in its alloc state `alloc(100)` is equal to the address found in line 2 of the above log file example. Therefore, `m1` would perform the error transition (lines 21–22 in figure 4.4), that would print the following error message: "`Malloc had already been called for that address`".

4.4.2 "Ignore" and "stay" transitions

In occasions, it is necessary to distinguish transitions that are valid, but need to be ignored. This is especially true with log file analyzers that might contain several instances of an analyzer machine, like in the case of dynamic machines. That allows dynamic machines to ignore any lines that would be noticed by other machines of the same type. We refer to this type of transition as *ignore transitions*.

المنارة للاستشارات
www.manaraa.com

We can observe that our running example, figure 4.4 contains two ignore transitions in lines 18 and 25:

```
18 from allocated (A), on free(B), if (A != B), ignore;
25 from allocated (A), on malloc(B), if (A != B), ignore;
```

These ignore transitions avoid conflicts between instances of the `memDynamic` analyzer machines. For instance, let us suppose we have two instances of `memDynamic`: `m1` with the state `alloc(100)` and `m2` with the state `alloc(200)`. If the log file analyzer were to process the line "`malloc returned 300`", both `m1` and `m2` would ignore such a line. `m1` would perform the ignore transition on shown above (line 25). The machine would compare the address in the log line "`malloc returned 300`" (`B`) against the address stored in the state variable `allocated(100)` (`A`). Since the values are different, the line "`malloc returned 300`" would be ignored by `m1`. `m2` would also ignore such a line.

Stay transitions are used to indicate that a transition will not cause the state of the machine to change. Like error transitions, stay transitions display a user–defined message.

Our running example revisited. Now that we have explained how *ignore* and *error* transitions work, we can conclude our discussion of how the example log file analyzer in figure 4.4 would analyze a log file like the one shown in figure 4.3.

1. The log file analyzer starts with zero analyzer machine instances.
2. Line 1 (`malloc returned 100`) in the log file triggers the creation of a new `memDynamic` analyzer machine. Let us call this instance `m1`. This instance is created in the state `allocated(100)` (see create transition in line 12, figure 4.4).

المنارة للاستشارات
www.manaraa.com

At this point, the analyzer has one `memDynamic` analyzer machine instance.

3. Line 2 (`free called on 100`) in the log file triggers the deletion of instance `m1`.

4. Line 3 (`malloc returned 200`) triggers the creation of a new instance of `memDynamic`. Let us call this instance `m2`. Similarly to point 2 in this list, this instance would be created with the state `allocated(200)`.

5. Line 4 (`malloc returned 300`) is ignored by `m2` (see ignore transition in line 24, figure 4.4). This line also triggers the creation of the `m3` instance with the state `allocated(300)`. At this point, the log file analyzer has two analyzer `memDynamic` machine instances.

6. Line 5 (`free called on 200`) is ignored by `m3` (see ignore transition in line 18, figure 4.4). This line also triggers the deletion of the `m2` analyzer machine instance.

7. Line 6 (`free called on 300`) triggers the deletion of the `m3` analyzer machine instance.

8. The log file analyzer ends with zero analyzer machine instances. The log file is accepted because every "malloc" call has a corresponding "free" call for the same address.

It is important to mention that in this example, the final state declaration on line 27 is not used because `memDynamic` machines are deleted when they are no longer needed.

The previous example shows how the new capabilities of LFAL 2.0 can be used to create dynamic log file analyzers that, combined with the new types of transition

المنارة للاستشارات
www.manaraa.com

actions, provide the user with more control over how analyzer machine transitions behave.

4.4.3 "Doing" transitions

Another powerful new type of transition action in LFAL 2.0 analyzers is the capability of executing user–defined C++ code. This new feature gives users the flexibility to customize or extend log file analyzers. We refer to this kind of transitions as "*doing*" transitions.

Figure 4.5 *a)* in line 12 shows an example of a transition with a doing action. We can observe that the user can include C++ statements such as references to objects or method calls. This new feature is strongly related to data declarations, which is described in the next section.

4.5 Data declarations

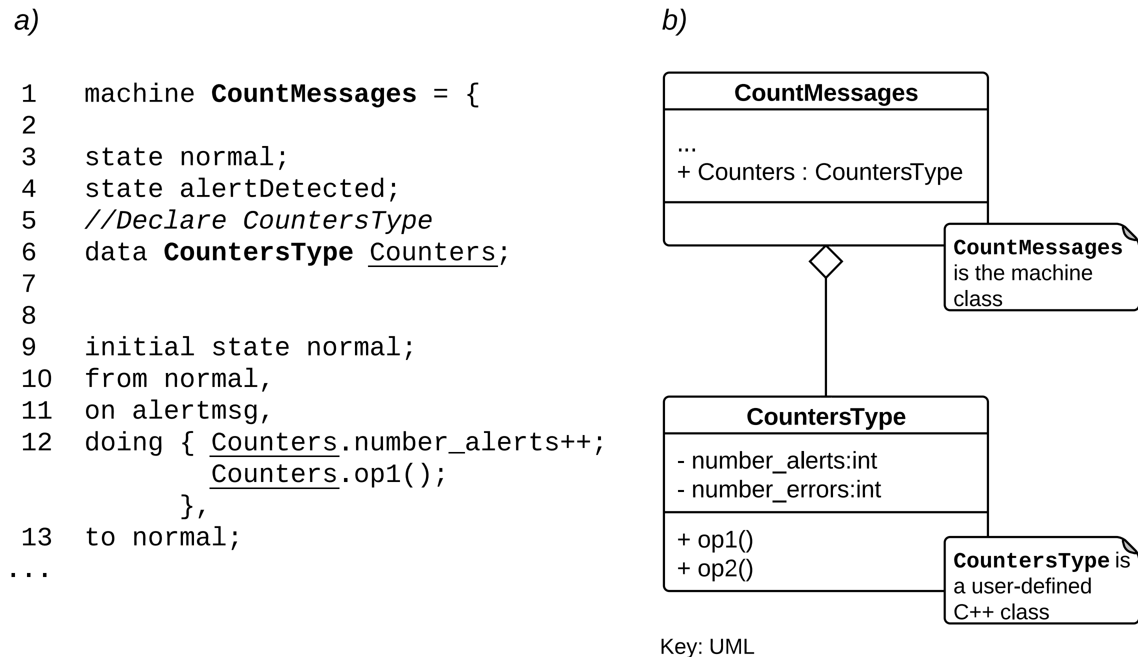
Data declarations allow users to declare external C++ objects and use them in a log file analyzers. Normally, log file analyzers work with variables extracted from log files or state variables. However, in some cases, users might need a mechanism that allows them to use their own C++ objects inside a log file analyzer. This motivated us to incorporate data declarations to LFAL 2.0.

To illustrate the use of data declarations, let us suppose we are interested in analyzing a log file while computing some stats, such as the number of times a line appears in the log. To do that, we would need to define a variable that could "remember" how

المنارة للاستشارات
www.manaraa.com

many times a transition on a given line has occurred. Figure 4.5 a) shows part of

Figure 4.5 Data declarations



an LFAL 2.0 program with a machine class called `CountMessages`. Line 6 shows the data declaration. In this example, we declare the `Counters` object, which is of type `CountersType`. Lines 10–13 show a transition where the Counters object is used to increment a counter and call a method.

Figure 4.5 b) shows what happens when a LFAL 2.0 program has a data declaration. During the code generation process, `Counters` becomes a member of the machine class `CountMessages`. That allows `Counters` to be referenced inside the `CountMessages` analyzer machine. The code generator adjusts the `Makefile` so that the class in the data declaration is linked with the log file analyzer code.

With data declarations, users can embed extra functionality to log file analyzers. The

المنارة للاستشارات
www.manaraa.com

example shown above gives an idea of how data declarations can be used. Users can use this feature to execute complex operations during a transition. The operations they can perform are only limited by the content of the class they decide to specify as a data declaration in an LFAL 2.0 program.

4.6 Shared memory integration

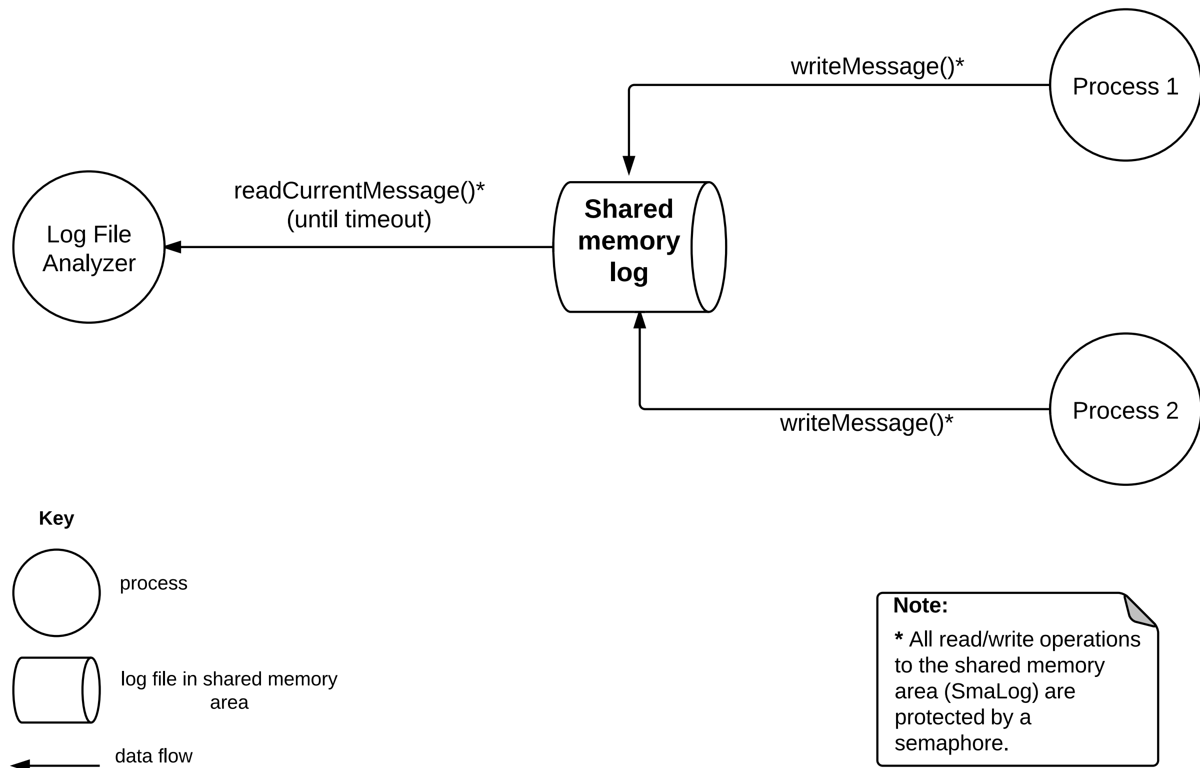
Log file analyzers usually read log files stored in disk. LFAL 2.0 analyzers have the capability of reading log files stored in a shared memory area. In chapter 1, we mentioned that shared memory areas are the fastest method of inter–process communication [30]. We have developed a library that can be easily integrated to a program, providing the capability of creating a log file in a shared memory area. We refer to this library as the `SmaLog` library. Shared memory areas allow multiple processes to read from and write to them at virtually the "same time".

Figure 4.6 shows an example of how a log file is generated by two processes. Each process writes to the shared memory log by using the `writeMessage()` method found in the `SmaLog` library. It is important to note that the shared memory log is protected by a *semaphore*. A semaphore is an IPC method that is used to synchronize processes. In this case, the semaphore is used to protect the integrity of the log file, by preventing two or more processes from writing to or reading from the log file at the same time. While the two processes in this example are writing to the log file, a log file analyzer can read and process the log lines. The log file analyzer will continue reading lines from the shared memory area until no messages are received after a timeout.

It is important to note that shared files in memory are *circular*. Depending on the size

www.manaraa.com

Figure 4.6 A log file in a shared memory area (SmaLog). In this example, a log file is being written by two processes. At the same time, a log file analyzer is accessing the log file in shared memory.



of the shared memory area, after a certain amount of time, the memory area will be full and the oldest information in the log file will be replaced by new information. This characteristic is what prevents shared memory log files from writing to undesirable addresses in memory, which can cause segmentation faults. Our shared memory log files are designed so that processes can be writing to them for days without causing any problem in a system.

One of the main benefits of reading log files from memory is performance. Andrews recognizes that one of the limitations of logging is overhead [15]. By logging messages

www.manaraa.com

to memory instead of a disk file, we expect to tackle this problem by minimizing the time required for an application to log a message. This feature provides analyzers with the possibility of being integrated into multi–process, real–time systems.

The capability of reading log files is integrated into every generated log file analyzer. To instruct a log file analyzer to read from a shared memory area, the user only needs to invoke the analyzer with the `-M` option in the command line, followed by a shared memory ID. In Unix/Linux systems, a user can see the a list of shared memory areas, semaphores and other IPC mechanisms by issuing the command `ipcs`.

In the next chapter, we will present some experiments we conducted to evaluate the performance of our log file analyzers. One of the experiments involves measuring the difference in performance between an analyzer that reads a large log file from a shared memory area versus an analyzer that reads the same log file from disk.

المنارة للاستشارات
www.manaraa.com

Chapter 5

Evaluating the Performance of Log File Analyzers

In this chapter we describe a series of experiments that we performed to evaluate the performance of our log file analyzers. One of the objectives of this thesis is to compare our LFAL 2.0 log file analyzers with the original, Prolog–based analyzers. In chapter 4, we described how C++ log file analyzers differ from their Prolog counterparts in qualitative terms, by explaining the new features introduced in LFAL 2.0 analyzers. In this chapter, we compare log file analyzers in quantitative terms by measuring their performance. We also illustrate the flexibility of our analyzers by providing examples of how that they can be used to prioritize or filter system logs.

المنارة للاستشارات
www.manaraa.com

5.1 Performance experiments for LFAL 1.0 and LFAL 2.0

The objective of our experiment is to compare the performance of the original, Prolog-based log file analyzers with our LFAL 2.0 analyzers. In particular, we are interested in finding out how fast our log file analyzers can process large log files. To achieve our objective, we decided to generate a log file analyzer that finds specific types of messages in a log file and counts their occurrences. We refer to this log file analyzer as the *BG/L analyzer*.

To perform our experiments, we decided to use a system log from Blue Gene/L (BG/L), which is one of the five world's most powerful supercomputers [26]. This log file was obtained from the Sandia National Laboratories webpage [10] and it contains 4,747,963 messages in 709 megabytes.

Besides evaluating the performance of our analyzers, we also wanted to investigate how complex it would be to prioritize or filter log files based on work by Kent and Souppaya in [22] and Oliner and Stearley [26]. With the new features introduced in LFAL 2.0, our log file analyzers are more flexible. Thus, besides acting as test oracles, our log file analyzers can be used to analyze or filter system logs.

The BG/L analyzer described in this section identifies messages based on regular expressions that correspond to specific log entry types. This is one of the prioritization criteria mentioned by Kent and Souppaya [22].

This log file analyzer could help system administrators to find how many "fatal" messages are found in a log file. For example, the BG/L log file contains lines such

المنارة للاستشارات
www.manaraa.com

as:

```
KERNSOCK 1136390405 2006.01.04 R31-M0-NC-I:J18-U11
2006-01-04-08.00.05.204230 R31-M0-NC-I:J18-U11 RAS KERNEL FATAL
idoproxy communication failure:  socket closed
```

By matching lines with the words `RAS KERNEL FATAL`, a system administrator could get a quick idea of how many entries marked as "fatal" are present in the log file.

It is important to note that the BG/L analyzer only counts different entry types in the log file. In order to be useful as a filtering/prioritizing tool, it would have to output the matched lines to the standard output or a file. Our BG/L analyzer does not output the matched lines because we are only interested in measuring the time it takes the analyzer to process a large log file. However, in section 5.4 we will present an example of a log file analyzer that acts as a temporal filter.

5.1.1 Experiment design

Our experiment consists in generating the BG/L analyzer in both the original version of LFAL and LFAL 2.0. For simplicity, in this chapter we will refer to the original version of LFAL as LFAL 1.0.

Our experiments are designed to evaluate two factors:

1. **The number of patterns** in a log file analyzer
2. The **size** of the log file

To evaluate how the **number of patterns** defined in a log file analyzer affects its

المنارة للاستشارات
www.manaraa.com

performance, we generated four versions of the BG/L analyzer. We will refer to them as `bgl01`, `bgl02`, `bgl03` and `bgl04`. The analyzer `bgl01` will match and count lines with *one* pattern. `bgl02` will look for lines with *two* different patterns. Similarly, `bgl03` will match and count lines with *three* different patterns. `bgl04` will match and count *four* different patterns.

To observe how the **size** of a log file affects the performance of our log file analyzers, we divided the BG/L log file into ten parts. In that way, we can observe the performance of `bgl01-bgl04` with 10%, 20%, etc. up to 100% of the log file.

We measured the performance of both LFAL 1.0 and LFAL 2.0 analyzers in *CPU time*, using the `/usr/bin/time` Linux command. CPU time is the time a process spends executing processor instructions. CPU time can be divided into *User CPU time* and *System CPU time*. User CPU time represents the time spent in executing a program's instructions. System CPU time is the time spent in system calls [5].

We ran each version of the BG/L analyzer (`bgl01-04`) ten times on *each* of the ten parts of the log file and measured user and system CPU time. After ten runs, we computed the average user and system CPU times. For example, `bgl01` was run ten times on 10% of the BG/L log. `bgl01` was then run ten times on 20% of the BG/L log file. This operation was repeated until 100% of the log file was reached. This process is repeated for `bgl02-04`.

Figure 5.1 shows the four patterns used in our experiments. These patterns are presented in LFAL 2.0 syntax (see chapter 4 section 4.2) where a pattern name is followed by a regular expression. The patterns identify different types of log messages that indicate different events in the BG/L system.

www.manaraa.com

Figure 5.1 The four log patterns used in our experiments. The name of the patterns (bold underlined) are followed by a corresponding regular expression (bold). Below each pattern, a sample log entry from the BG/L log file.

```
pattern cores "RAS KERNEL INFO generating core.[0-9]+"
Example:
- 1117839086 2005.06.03 R24-M1-N6-C:J06-U11 2005-06-03-15.51.26.713752
  R24-M1-N6-C:J06-U11 RAS KERNEL INFO generating core.238

pattern cache_parity "RAS KERNEL INFO instruction cache parity error corrected"
Example:
- 06-03-15.50.17.648619 R02-M1-N0-C:J12-U11 RAS KERNEL INFO instruction cache
  parity error corrected

pattern sym "RAS KERNEL INFO CE sym [0-9]"
Example:
- 1117839710 2005.06.03 R16-M1-N2-C:J17-U01 2005-06-03-16.01.50.945374
  R16-M1-N2-C:J17-U01 RAS KERNEL INFO CE sym 0, at 0x0b8580c0, mask 0x10

pattern fatal "RAS KERNEL FATAL"
Example:
- 1117839710 KERNSOCK 1136390405 2006.01.04 R00-M0-NC-I:J18-U11
  2006-01-04-08.00.05.167045 R00-M0-NC-I:J18-U11 RAS KERNEL FATAL idoproxy
  communication failure: socket closed
```

5.1.2 The LFAL 1.0 BG/L Analyzer

As mentioned above, we generated four versions of the BG/L analyzer in LFAL 1.0. Figure 5.2 shows the LFAL 1.0 program to generate the `bgl01` analyzer, which matches and counts lines of the type "`fatal`". Figure 5.3 shows `bgl02`, which matches and counts two types of log entries. Figures 5.4 and 5.5 show `bgl03` and `bgl04` respectively.

As the reader might remember from chapter 4, section 4.2, LFAL 1.0 does not support patterns. Therefore, we have no way to generate an LFAL 1.0 analyzer that could process the original BG/L log file. In fact, LFAL 1.0 BG/L analyzers will expect to process a log file with entries such as:

المنارة للاستشارات
www.manaraa.com

Figure 5.2 The bgl01 analyzer in LFAL 1.0. This analyzer matches and counts one pattern.

```
machine bgl01;
 initial_state normal(0, 0, 0, 0);
 from normal(A, B, C, D), on fatal, to normal(A, B, C, D1),
   where (D1 is D + 1);
 final_state Any.
```

Figure 5.3 The bgl02 analyzer in LFAL 1.0. This analyzer matches and counts two patterns.

```
machine bgl02;
 initial_state normal(0, 0, 0, 0);
 from normal(A, B, C, D), on sym, to normal(A, B, C1, D),
   where (C1 is C + 1);
 from normal(A, B, C, D), on fatal, to normal(A, B, C, D1),
   where (D1 is D + 1);
 final_state Any.
```

```
cores

cache_parity

sym

fatal
```

Thus, in order to be able to analyze the BG/L log file with LFAL 1.0, we had to "simplify" its log entries. To emulate the work accomplished by LFAL 2.0 patterns, we used a `sed` script to match the patterns shown in figure 5.1. Sed is a stream editor used to perform basic text transformations on an input stream [9]. Our sed script matches the patterns and replaces them by its corresponding name. For instance, a line such as:

المنارة للاستشارات

www.manaraa.com

Figure 5.4 The bgl03 analyzer in LFAL 1.0. This analyzer matches and counts three patterns.

```
machine bgl03;
 initial_state normal(0, 0, 0, 0);
 from normal(A, B, C, D), on cache_parity, to normal(A, B1, C, D),
   where (B1 is B + 1);
 from normal(A, B, C, D), on sym, to normal(A, B, C1, D),
   where (C1 is C + 1);
 from normal(A, B, C, D), on fatal, to normal(A, B, C, D1),
   where (D1 is D + 1);
 final_state Any.
```

```
- 1117839086 2005.06.03 R24-M1-N6-C:J06-U11 2005-06-03-15.51.26.713752
R24-M1-N6-C:J06-U11 RAS KERNEL INFO generating core.238
```

which corresponds to the regex `"RAS KERNEL INFO generating core.[0-9]+"`, would be changed into:

```
cores
```

In addition, we eliminated any "*unnoticed*" log lines—that is, any lines that would not be matched by any of the patterns in figure 5.1. This is necessary because LFAL 1.0 analyzers print an error message when a log file line is not noticed by any analyzer machine. For instance, `bgl01` in figure 5.2 would report an error if it found any line different than `cores`. This is generally useful. However, for the purposes of our experiment, this feature would affect our results. The reason is that thousands of error messages would be printed to the standard output because the BG/L log file has many different types of messages.

Since we are *not* interested in measuring the time it takes log analyzers to print error messages, we suppressed the offending log entries. As we will see in section 5.1.3,

المنارة للاستشارات
www.manaraa.com

Figure 5.5 The bgl04 analyzer in LFAL 1.0. This analyzer matches and counts four patterns.

```
machine bgl04;
 initial_state normal(0, 0, 0, 0);
 from normal(A, B, C, D), on cores, to normal(A1, B, C, D),
   where (A1 is A + 1);
 from normal(A, B, C, D), on cache_parity, to normal(A, B1, C, D),
   where (B1 is B + 1);
 from normal(A, B, C, D), on sym, to normal(A, B, C1, D),
   where (C1 is C + 1);
 from normal(A, B, C, D), on fatal, to normal(A, B, C, D1),
   where (D1 is D + 1);
 final_state Any.
```

LFAL 2.0 analyzers allow users to suppress error messages concerning unnoticed lines, without having to modify the log file.

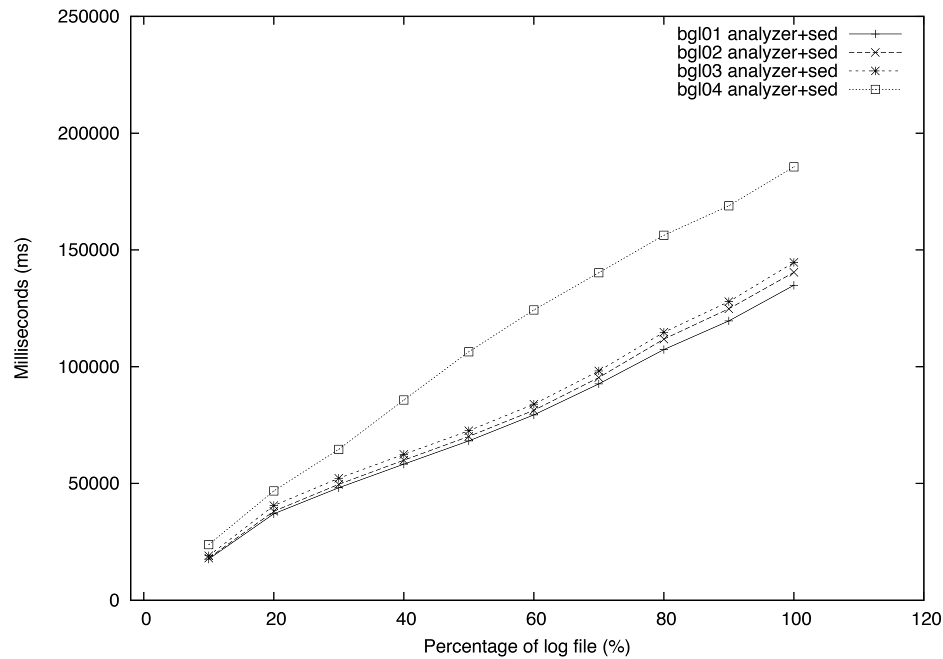
Figure 5.6 shows a graph with our measurements for the average user CPU time for `bgl01-04` analyzers. Each of the points on the graph represent one of the 40 observations in our experiment—that is, the average of ten runs for each of the analyzers and each of the percentages.

It is important to note that the times shown in this graph include the time it took to process the BG/L log file with a `sed` script. We can observe that, in general, time increases with the percentage (size) of the log file processed. Similarly, the analyzer that matches the greatest number of patterns (`bgl04`) takes the longest time to process the log file.

Figure 5.7 shows the system time for the four BG/L analyzers. Similarly to the user time graph in figure 5.6, time increases with the size of the log file. However, system CPU time seems to be very similar for all four BG/L analyzers (`bgl01-04`) in 10% of

المنارة للاستشارات
www.manaraa.com

Figure 5.6 The CPU user time for bgl01, bgl02, bgl03 and bgl04. The x axis represents the percentage of the log file that was analyzed and the y axis represents the number of milliseconds of user CPU time it took on average to run the analyzer.



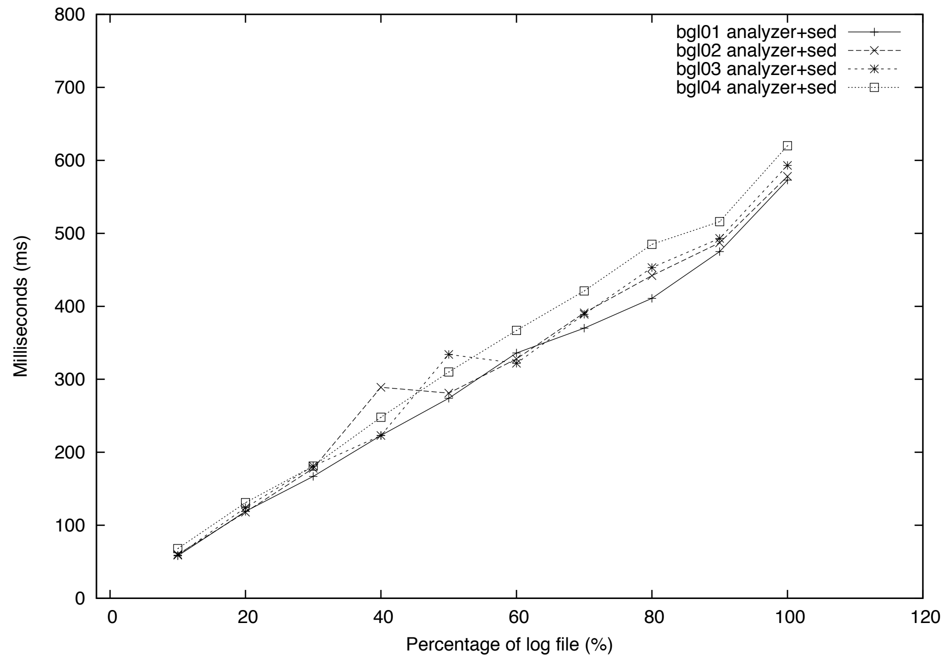
the log file up to 40%, while in the rest of the percentages (50%–100%) system time is not necessarily higher for analyzers with more patterns.

5.1.3 The LFAL 2.0 BG/L Analyzer

The procedure for our experiments is the same as the one explained in the previous section. The only difference is that the BG/L log file was processed directly by our LFAL 2.0 analyzer. Another important difference is that LFAL 2.0 analyzers can omit error messages caused by "*unnoticed*" lines by specifying the option `-u` in the command line. Therefore, LFAL 2.0 analyzers were able to analyze the BG/L log file

المنارة للاستشارات
www.manaraa.com

Figure 5.7 The CPU system time for bgl01, bgl02, bgl03 and bgl04. The x axis represents the percentage of the log file that was analyzed and the y axis represents the number of milliseconds of system CPU time it took on average to run the analyzer.



without requiring the log file to be adapted or modified in any way.

Figure 5.8 shows the LFAL 2.0 version of the `bgl04` analyzer. This BG/L analyzer declares four patterns in lines 1–4. This analyzer uses a data declaration (see chapter 4, section 4.5) to count the number of times each type of log entry is matched. In the transitions (lines 13–16), we can observe that the members of the `Counters` object are incremented when a corresponding entry is noticed. Line 20 shows a special type of transition. LFAL 2.0 allows users to specify transitions that are to be executed *only* at the beginning or the end of a log file. This special type of transitions are declared by using the predefined `begin__` and `end__` patterns. In this example, the transition on line 20 is executed only when the end of the log file is reached, just

المنارة للاستشارات
www.manaraa.com

Figure 5.8 The bgl04 analyzer in LFAL 2.0. This analyzer matches and counts four patterns.

```
1  pattern cores "RAS KERNEL INFO generating core.[0-9]+"
2  pattern cache_parity "RAS KERNEL INFO instruction cache parity error
                      corrected"
3  pattern sym "RAS KERNEL INFO CE sym [0-9]"
4  pattern fatal "RAS KERNEL FATAL"
5
6  machine bgl04 = {
7
8  state normal;
9  data CountersType Counters;
10
11  initial state normal;
12
13  from normal, on cores, doing { Counters.coremsgs++; }, to normal;
14  from normal, on cache_parity, doing { Counters.cachepar++; }, to normal;
15  from normal, on sym, doing {Counters.sym++; }, to normal;
16  from normal, on fatal, doing {Counters.fatal++;}, to normal;
17
18  //This transition is only executed when the end of a log file is reached
19  from normal, on end__,
20  doing { cout << "-> Core messages: " <<
            Counters.coremsgs << endl; cout << "Cache messages: " <<
            Counters.cachepar << endl; cout << "Sym messages: " <<
            Counters.sym << endl; cout << "Fatal messages: " <<
            Counters.fatal << endl;  },
21  to normal;
22
23  final state any;
24  }
```

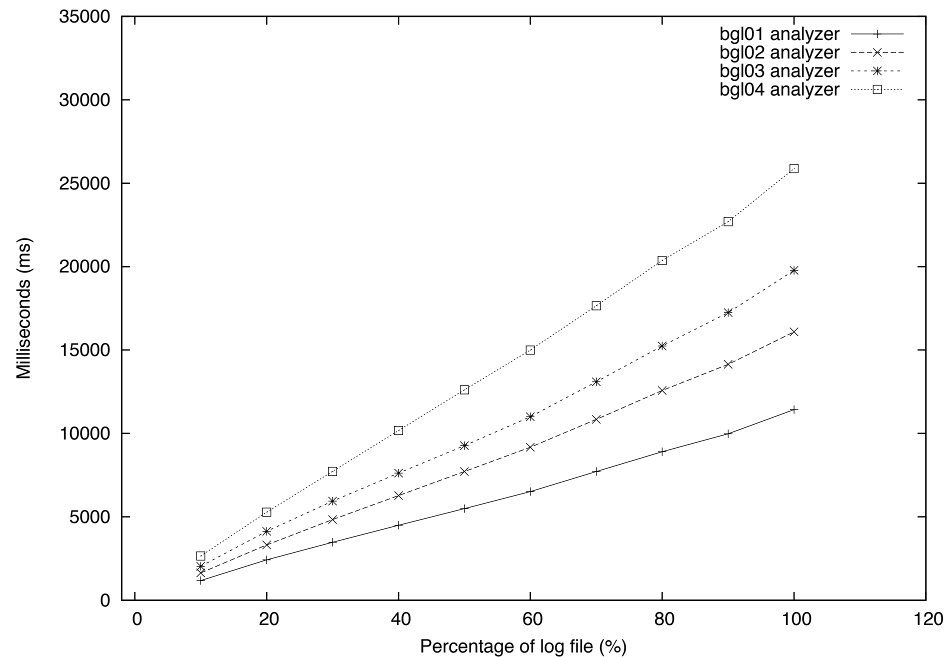
before the log file analyzer program exits. This is useful because it allows the total count of each of the log entry types to be printed.

Figure 5.9 shows a graph with our measurements for the average user CPU time for `bgl01-04` analyzers. The results show that the time increases with both log file size and number of patterns.

Figure 5.10 shows the average system CPU time for the four LFAL 2.0 BG/L analyz-

المنارة للاستشارات
www.manaraa.com

Figure 5.9 The CPU user time for bgl01, bgl02, bgl03 and bgl04. The x axis represents the percentage of the log file that was analyzed and the y axis represents the number of milliseconds of user CPU time it took on average to run the analyzer.



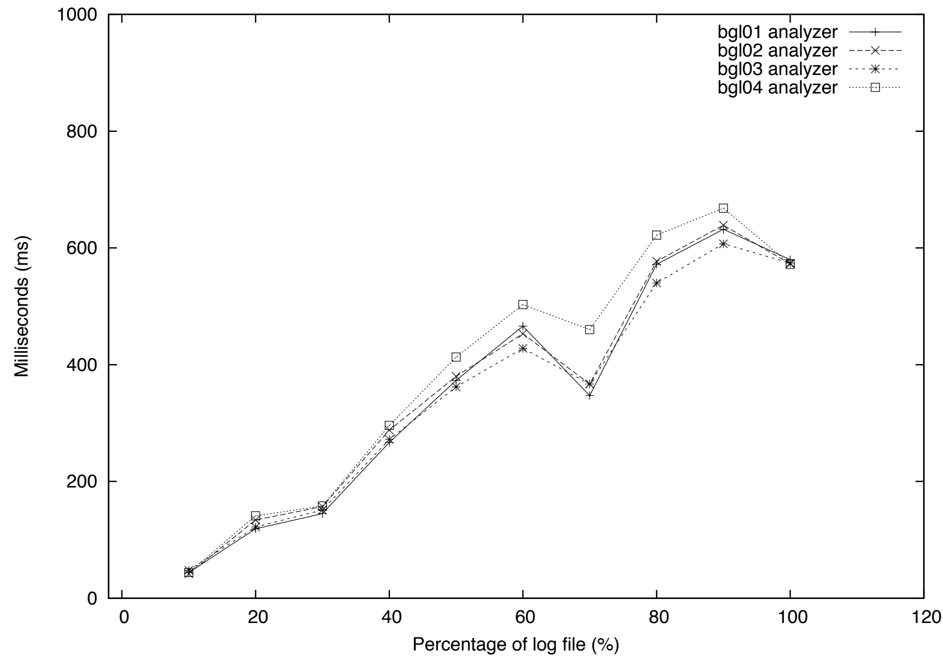
ers. In general, system time for the four analyzers is similar and increases with the size of the log file. This does not happen in 70% and 100% of the log file, where the system CPU time decreases.

5.1.4 Results

By comparing the user CPU time results for LFAL 1.0 in figure 5.6 and LFAL 2.0 in figure 5.9, we can conclude that LFAL 2.0 analyzers are indeed faster than their LFAL 1.0 counterparts. To visualize how much faster LFAL 2.0 analyzers are with respect to LFAL 1.0 analyzers, we generated a graph with the ratio between the average user

www.manaraa.com

Figure 5.10 The CPU system time for bgl01, bgl02, bgl03 and bgl04. The x axis represents the percentage of the log file that was analyzed and the y axis represents the number of milliseconds of system CPU time it took on average to run the analyzer.

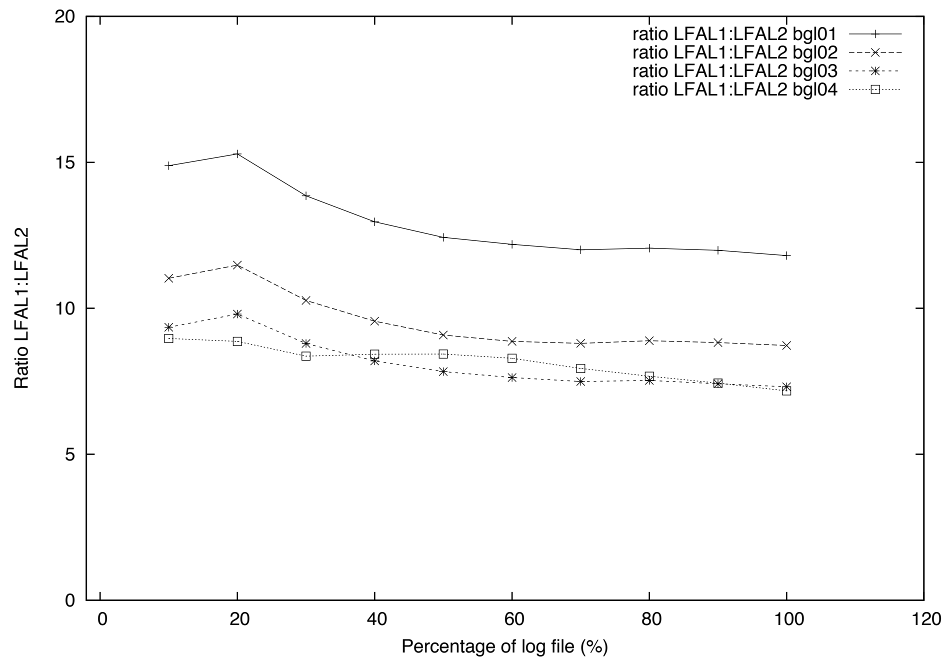


CPU time for LFAL 1.0 and the average user CPU time for LFAL 2.0. Figure 5.11 shows that LFAL 2.0 analyzers are between 8 and 15 times faster depending on the number of patterns. In fact, the ratio seems to decrease as the number of patterns increases. For example, LFAL 2.0 is 15 times faster than LFAL 1.0 for the `bgl01` analyzer and between 8 and 9 times for the `bgl04` analyzer.

Figure 5.12 shows the ratio between the system time in LFAL 1.0 and LFAL 2.0. In this case, LFAL 2.0 takes slightly more system time than LFAL 1.0. The ratio starts at 1.5 but decreases to values between 0.7 and 0.9. This might be an effect of system calls in C++. However, it is important to note that system CPU time is only a small portion of the total time spent by a process. Because user CPU time is less for LFAL

www.manaraa.com

Figure 5.11 The CPU user time ratio between LFAL 1.0 and LFAL 2.0 for bgl01, bgl02, bgl03 and bgl04. The x axis represents the percentage of the log file that was analyzed and the y axis represents the `LFAL1:LFAL2` ratio.



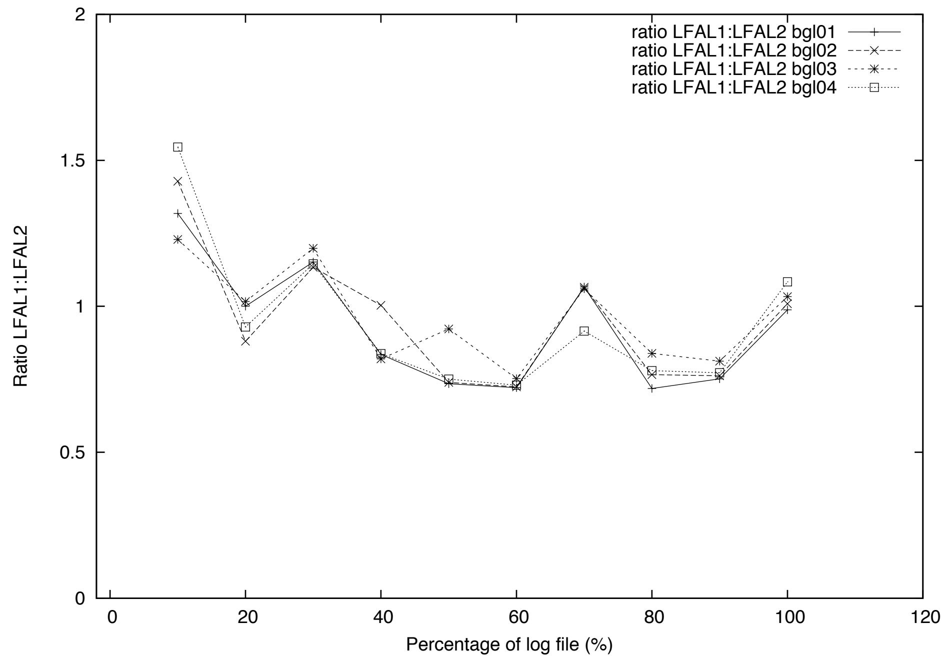
2.0 analyzers, LFAL 2.0 analyzers are still faster than LFAL 1.0 analyzers.

5.2 Performance experiments for shared memory analyzers

In chapter 4, section 4.6 we introduced shared memory integration in LFAL 2.0 analyzers. We decided to develop this feature in order to enable our analyzers to process output from real–time or multi–process applications. Besides making all of our analyzers capable of reading log files from shared memory areas, we developed a library

المنارة للاستشارات
www.manaraa.com

Figure 5.12 The CPU system time ratio between LFAL 1.0 and LFAL 2.0 for bgl01, bgl02, bgl03 and bgl04. The x axis represents the percentage of the log file that was analyzed and the y axis represents the `LFAL1:LFAL2` ratio.



that allows developers to easily log messages to shared memory areas.

In this section, we describe the experiment we performed to measure the difference in performance between a log file analyzer that reads the BG/L log file from a disk file and one that reads the same log file from memory. For our experiment, we chose the `bgl04` analyzer, which matches and counts four different log entry patterns (see figure 5.8). We also provide more details as to how our log file analyzers are generated and invoked from the command line.

In section 5.2.1 we explain how we performed our experiment with the BG/L log file on a disk file. In section 5.2.2 we explain the procedure we followed to load the log file on a shared memory file in order to illustrate the ease of use of our shared memory

المنارة للاستشارات
www.manaraa.com

library. We also demonstrate that our log file analyzers are used in the same way, regardless of whether the log file is in memory or disk. Finally, in section 5.2.3 we compare the performance between an analyzer processing a file in disk, versus the same analyzer processing a file in shared memory.

5.2.1 Reading from disk

To process the BG/L log file from disk, we translated the `bgl04.lfal` LFAL 2.0 program into a C++ program. To do so, we invoked our code generator (see section 3.3 in chapter 3) in the following way:

```
lfalc bgl04.lfal
```

This produces the necessary C++ files and a corresponding Makefile to generate an executable log file analyzer.

Since the `bgl04` analyzer includes a data declaration, the files `CountersType.cpp` and `CountersType.h` should be provided by the user. As the reader might recall from section 4.5 in chapter 4, data declarations allow users to integrate their own data types into LFAL 2.0 analyzers. In order to successfully compile the generated C++ code with the data declaration (`CountersType`), the user needs to compile it in advance so that the file `CountersType.o` is present and therefore linked with the log file analyzer.

The user can compile the C++ generated code by issuing the `make` command. This produces the `bgl04` executable log file analyzer.

To measure the performance of the `bgl04` analyzer, we ran our analyzer *ten times.*

www.manaraa.com

We invoked the analyzer through the following command:

```
/usr/bin/time -f ''%U\t%S'' ./bgl04 -u BGL.log
```

The utility `/usr/bin/time` measures the system and user CPU time that `bgl04` takes to process the file. The option `-u` omits error messages for "unnoticed" lines. After ten runs, we computed the average user and system CPU time.

We also ran an extra set of tests by placing the BG/L log file in a Network File System (NFS) which is part of our department's research network. This was done in order to evaluate the difference between processing a local file and a file in a network file system.

5.2.2 Reading from memory

In order to read a log file in a shared memory area, the user does not need to build the `bgl04` log file analyzer in any special way. Therefore, the experiment to read the BG/L log file from memory used the same `bgl04` executable file described in section 5.2.1.

In order to read the BG/L log from a shared memory area, we first need to load it. This task is greatly simplified by the routines available in our Shared Memory Management base libraries (see chapter 3, section 3.3.1).

Figure 5.13 shows how the BG/L log file was loaded. In line 12, the `log` object is created, specifying a log file that will store 4,747,970 lines. Line 14 creates the log file in memory. Lines 14–23 deal with opening the file `BGL.log`. In line 32, the method `log.logMsg()` is called repeatedly to copy each line of `BGL.log` in disk to the shared

www.manaraa.com

memory area log.

Normally, a process would write log messages directly to shared memory rather than write them to a disk file and then copy the disk file to shared memory. However, we are performing the experiment like this so that we can compare the performance as directly as possible.

As we can observe, creating a log file in shared memory only requires two lines of code. The developer can then use the `logMsg()` method to write log messages to the shared memory area.

To process the BG/L log file in memory, we invoke the analyzer in the following way:

```
/usr/bin/time -f ''%U\t%S'' ./bgl04 -u -M 1234
```

We can observe that `bgl04` is invoked in a very similar way as in section 5.2.1. However, instead of specifying a log file, we specify a numeric *memory identifier* after the `-M` option. In most Linux/UNIX machines, the list of shared memory areas with their corresponding identifiers can be obtained by issuing the `ipcs` command.

5.2.3 Results

After running the `bgl04` analyzer on the BG/L log in both disk and memory, we obtained the results shown in figure 5.14.

System time. As we can observe in figure 5.14, the system time for the analyzer in a NFS disk was the longest. The system time to process the BG/L log file in **shared memory** was **14.14% lower** with respect to the NFS file. However, the system

المنارة للاستشارات
www.manaraa.com

time to process the BG/L log file in a **local file** was **26.42% lower** with respect to the NFS file. From these results, we can see that processing a file locally causes the processor to spend less time in system calls compared to reading the file from a shared memory area or a file in a NFS system.

User time. In figure 5.14 we can observe that `bgl04` takes the **longest** when processing the BG/L log file in a **network file system**. By reading the BG/L log from a **local file**, the time **reduces by 2.7%**. However, when reading the BG/L log file from a **shared memory area**, the time **reduces by 8.56%**.

From the results above, we observe that the time to process the BG/L file from a **shared memory area** is **6.03% less** with respect to processing the same log file from a **local file on disk**. This difference is not as large as we expected. A possible explanation is that this might be an effect of disk caching. The operating system might be optimizing the reading process by caching parts of the file in memory.

As we can see, processing a log file in shared memory is faster than doing so in a file on disk. The difference is more prominent between reading from a file in a NFS file system and reading from a shared memory area. This leads us to conclude that in time–critical distributed systems, it is much better to use a shared memory area to concentrate and share data among processes. The fact that LFAL 2.0 supports shared memory areas makes it easier to integrate it to these kind of systems.

5.3 Logging overhead in disk versus memory

In chapter 4, section 4.6, we mentioned our intention to tackle the problem of *logging overhead* by writing messages to shared memory instead of doing so to disk. In section

المنارة للاستشارات
www.manaraa.com

5.2 we described the experiments we performed in order to evaluate the difference between analyzing a log file in memory and analyzing the same file in a shared memory area. We were able to show that, in general, processing a log file in a shared memory is faster. However, the problem of logging overhead has more to do with the overhead that is introduced by developers when they log messages in their programs.

In this section we present an experiment we performed in order to measure the difference between writing a large log file to disk, versus writing the same log file to a shared memory area. We used the BG/L log file for these experiments. Our experiment consisted of two parts:

- To measure the **time it takes to write the BG/L log file to a shared memory area**, we ran the program in figure 5.13 *ten times* and averaged the results. For simplicity, we refer to this program as `writeMemory.exe`.
- To measure the **time it takes to write the BG/L log to disk**, we wrote a program that reads the BG/L log and writes each log entry to a new file. We ran the program *ten times* and averaged the results. We refer to this program as `writeDisk.exe`.

5.3.1 Results

The table in figure 5.15 summarizes the results obtained in our experiment. It is important to mention that besides including the CPU user and system times, we have included the clock time. This is because the `/usr/bin/time` utility does not seem to measure disk I/O operations. As we will see, the user and system CPU time

المنارة للاستشارات
www.manaraa.com

measurements for `writeDisk.exe` do not reflect the time the process takes to write the BG/L log file to disk.

We can observe that the **average clock time** for `writeDisk.exe` is 27,262.40 ms. In contrast, `writeMemory.exe` takes 15,880.30 ms. Thus, the time to **write the BG/L log file in memory is 41.47% lower** compared to the time to write this log file to disk `writeDisk.exe`.

In terms of **average user time**, we can observe that the time for `writeMemory.exe` is 6,927.40 ms. Conversely, the time for `writeDisk.exe` is only 1,766.10 ms. Regarding **average system time**, `writeMemory.exe` took 8,920.50 ms, while `writeDisk.exe` took 2,935.80. That is, in general, the system time is higher when the file is written to memory with respect to disk.

It is important to note that the sum of system and user time for `writeDisk.exe` (4,701.90 ms) is much lower than the actual clock time (27,262.40 ms). We believe that the reason for this is that the `/usr/bin/time` utility is only taking into consideration the time the process takes to execute CPU instructions. The I/O operations on the disk are not part of the measurements reported by `/usr/bin/time`.

From these results, we conclude that storing log files in shared memory reduces the overhead of logging. Developers can opt to use shared memory in order to be able to log messages at a lower cost in terms of time.

المنارة للاستشارات
www.manaraa.com

5.4 An example temporal filter analyzer

In chapter 2, we mentioned that an important part of the work on log file analysis focuses on analysis of system logs (see section 2.1). In section 5.1, we mentioned we are interested in evaluating if our log file analyzers can perform tasks in the area of log entry prioritization. For that reason, in this section we present an example of a log file analyzer that acts as temporal filter. Oliner and Stearley describe a temporal filter for system log files that allows administrators to filter certain log entries that occur within a certain number of seconds of each other [26]. For example, if a message appears every second for an hour, the filter will only keep the first message. This allows system administrators to avoid repetitive messages. Figure 5.16 shows a log file analyzer that matches a certain pattern and filters the matched lines that are within 5 seconds of each other.

Line 1 contains the pattern `cache_parity`. We can observe that the regular expression contains two variables: `TS` and `S`. The variable `TS` captures the timestamp in the log entry while `S` captures the rest of the log entry.

Line 8 contains a data declaration for an object named `Counters` of type `CountersType`. This object is used to store the timestamp of the last matched line. The transition in lines 12–15 echoes the first log entry that matches the `cache_parity` pattern and stores its timestamp in `Counters.ts_last`.

The transition in lines 17–21 compares the timestamp of the last log entry that matched the pattern `cache_parity` with the threshold, which in this case is 5. If the difference between the timestamp of the last line that matched the pattern and the current line is more than 5, the line is printed.

www.manaraa.com

We ran this analyzer ten times on the BG/L log file and were able to successfully obtain 7,828 lines that match the `cache_parity` pattern and that are more than 5 seconds within each other. The analyzer took an average of 17,937 ms of clock time to process and filter the BG/L log file. The corresponding system and user times were 8,177 ms and 6,011 ms.

This example shows that LFAL 2.0 analyzers are flexible and can be used not only as test oracles, but also to perform filtering tasks on system logs. Patterns and data declarations make filtering possible by enabling analyzers to match and manipulate complex text patterns.

المنارة للاستشارات
www.manaraa.com

Figure 5.13 A sample program that illustrates how to create a log file in a shared memory area using LFAL 2.0's base libraries.

```
 1 #include "SmaLog.h"
 2 #include <iostream>
 3 using namespace std;
 4
 5 int main() {
 6
 7   string msg;
 8   ifstream bgl;
 9
10   //Create a shared-memory log instance
11   //specifying a log file with 4747970 lines.
12   SmaLog log("bgl_mem", 4747970);
13   //Create shared-memory log
14   log.createLog();
15
16   //Open file
17   bgl.open("/home/ilse/BGL.log");
18
19   bool file_opened = bgl.is_open();
20   if(!file_opened) {
21     cout << "Could not open BGL.log" << endl;
22     return -1;
23   }
24
25   while (1) {
26     if (bgl.good())
27       getline(bgl, msg);
28     else
29       break;
30
31     //Copy message to shared memory area
32     log.logMsg((char *)msg.c_str());
33   }
34   cout << "DONE." << endl;
35   return 0;
36 }
```

المنارة للاستشارات
www.manaraa.com

Figure 5.14 A table that summarizes the results of our experiment to compare the performance of analyzers that read the BG/L log file from disk or shared memory

	BG/L log in NFS (ms)	BG/L log in local file (ms)	BG/L in shared memory (ms)
System time (ms)	757	27,196	650
User time (ms)	27,948	557	25,555

Figure 5.15 A table that summarizes the results of our experiment to measure the time to write the BG/L log file in both shared memory and disk.

	writeMemory.exe	writeDisk.exe
Average clock time (ms)	15,880.30	27,262.40
Average user time (ms)	6,927.40	1,766.10
Average system time (ms)	8,920.50	2,935.80

المنارة للاستشارات
www.manaraa.com

Figure 5.16 A log file analyzer that filters system log file.

```
1  pattern cache_parity(int TS, string S)  "- (?<TS>) (?<S>.*instruction cache
                                            parity error corrected)"
2
3  machine Filter = {
4
5  state normal;
6  state first;
7
8  data CountersType Counters;
9
10  initial state first;
11
12  //Echo first matched line
13  from first, on cache_parity(TS, S),
14     doing { cout << "- " << TS << " " << S << endl;
               Counters.ts_last = TS; },
15     to normal;
16
17  //Capture timestamp from log entry
18  from normal, on cache_parity(TS, S),
19    //If the threshold condition is met, echo the line
20    doing { if ( (TS - Counters.ts_last)  > 5)
              cout << "- " << TS << " " << S << endl;
              Counters.ts_last = TS; },
21    to normal;
22
23  final state any;
24
25  }
```

المنارة للاستشارات
www.manaraa.com

Chapter 6

Conclusion

6.1 Conclusion

In this thesis, we have described our work regarding the automatic generation of log file analyzers from the specification of a program's expected behavior. Our work builds upon previous work by Andrews, who introduced a language known as Log File Analysis Language (LFAL) that expresses a program's expected behavior in terms of state machines. Andrews also developed and tested a log file analyzer generator that translates an LFAL program into Prolog code.

The main objective of this thesis was to design and develop a log file analyzer generator that generates analyzers based on the C++ language instead of Prolog. This was motivated by several reasons. One of them was that C++ is a modern, fast and well-known programming language. Another important motivation was that we expected to improve the general performance of log file analyzers by using the C++

www.manaraa.com

programming language instead of Prolog. In addition, we wanted to take advantage of a C++ implementation in order to incorporate new features that could make our log file analyzers more flexible and powerful.

We extended the original LFAL language and incorporated new elements such as support for PCRE regular expressions, different kinds of transition actions and the possibility to extend the functionality of log file analyzers by incorporating user–defined data members or embedding C++ code in transitions. We presented several examples that show how these new features can be used to provide the user with more control over the behavior in log file analyzers.

For instance, in the Prolog implementation, a log file analyzer reports an error if a line is not noticed or if a line is noticed but none of the analyzer machines are able to perform a valid transition. LFAL 2.0 analyzers allow users to define certain types of transitions that, although valid, still print a user–defined error message. These new features provide the user the ability to customize log file analyzers to a greater extent.

In addition, we successfully designed and implemented a log file analyzer generator that translates an LFAL 2.0 program into C++ code.

To evaluate whether the use of C++ improved the performance of our log file analyzers, we performed a series of experiments that compare the performance between Prolog–based analyzers and C++ analyzers. Our results show that, depending on the number of patterns defined in an analyzer, C++ analyzers can be between 8 and 15 times faster compared to their Prolog counterparts.

Our experiments also revealed the benefits and flexibility of the C++ implementation.

المنارة للاستشارات
www.manaraa.com

For instance, the task of analyzing a system file with the Prolog implementation proved to be laborious. This was because the original log file needed to be modified so that it could be analyzed. In contrast, we were able to show that LFAL 2.0's support for PCRE regular expressions allow our log file analyzers to match complex log file entries. This represents a major advantage, since LFAL 2.0 analyzers do not require developers to change the way they log events in their programs. In fact, LFAL 2.0 analyzers adapt to the logs. Logs do not have to be adapted to LFAL 2.0. This is relevant because one of the main challenges in log file analysis is precisely the fact that log formats differ greatly from system to system.

Another aspect of our work consisted in making our log file analyzers capable of reading log files from shared memory areas. We performed a series of experiments to determine how much faster it is to process a log file in disk compared to in a shared memory area. Our results show that the time to analyze a log file is less if it is located in a shared memory area. In fact, the log file analyzer in our experiment took approximately 6% less user CPU time when the log file was located in a shared memory area, compared to the time the same analyzer took to process the same log file in a local disk. While this reduction in time does not seem to be very large, it is important to consider that storing log files in shared memory areas has qualitative advantages as well. This feature allows our log file analyzers to easily integrate with multi–process, real-time systems. In addition, this opens the possibility for our analyzers to process log files "on–the–fly", as they are generated.

Besides integration support for shared–memory log files, we developed a easy–to–use library which developers can use to create log files in shared memory areas. In fact, the original motivation for developing this library was to tackle the logging overhead problem. In order to analyze the behavior of a program, our log file analyzers require

المنارة للاستشارات

www.manaraa.com

that developers log events of interest in their programs. This might not represent a problem for small programs. However, the overhead of logging messages to disk can become a problem in real–time systems. We performed a series of experiments to evaluate how much faster it is to write a log file to disk compared to writing the same file to shared memory using our library. Our results showed that writing log messages in memory takes 41% less time, compared to the time it takes to write the same file to disk.

We also showed that our log file analyzers are flexible enough to be used in the area of system logs prioritization and filtering.

We consider that log file analyzers have a lot of potential in the area of software testing. As we mentioned earlier, the task of evaluating test results is often performed manually and thus it can be time–consuming and unreliable. Log file analyzers are test oracles that determine if a log file produced by a program reveals faults in it. Developers often log events in their programs for debugging purposes and analyze them precisely to determine if a program behaved as expected. However, log files are often long or intricate and thus, difficult to analyze manually. Log file analyzers provide a way to analyze log files automatically and reliably, helping testers to automatize the evaluation of test results.

6.2 Future work

We believe that it is important to continue maintaining our LFAL 2.0 framework by treating it as an open–source project, so that it can be evaluated further.

We consider that it would be very interesting to develop several case studies where

المنارة للاستشارات
www.manaraa.com

we apply log file analysis to a large, multi–process system. This would allow us to evaluate how easy it is to specify the expected behavior of a whole system using LFAL 2.0. In addition, it would allow us to observe the performance and effectiveness of a more complex log file analyzer.

المنارة للاستشارات

www.manaraa.com

Bibliography

[1] Analog official web site. http://www.analog.cx/. [Online. Accessed May 2012].

[2] Awstats official web site. http://awstats.sourceforge.net/. [Online. Accessed January 2012].

[3] Bison - GNU parser generator homepage. http://http://www.gnu.org/software/bison/. [Online. Accessed January 2012].

[4] Flex (the fast lexical analyzer) homepage. http://flex.sourceforge.net/. [Online. Accessed January 2012].

[5] Linux programmer's manual – time. Linux Man Pages. [Accessed July 2012].

[6] National maritime museum. http://bit.ly/RoyalNavyLogs. [Online. Accessed May 2012].

[7] Pcre - perl compatible regular expressions homepage. http://www.pcre.org/. [Online. Accessed January 2012].

[8] Sawmill analytics. http://www.sawmill.co.uk/. [Online. Accessed May 2012].

[9] sed linux man page. Linux Man Pages. [Accessed July 2012].

[10] Supercomputer event logs. http://www.cs.sandia.gov/~jrstear/logs/. [Online. Accessed June 2012].

[11] Webalizer official web site. http://www.webalizer.org/. [Online. Accessed May 2012].

[12] *New Oxford American Dictionary*. Oxford University Press, Inc., third edition, 2010.

[13] J.H. Andrews. Testing using log file analysis: tools, methods, and issues. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, pages 157 –166, 1998.

www.manaraa.com

[14] J.H. Andrews. Deriving state-based test oracles for conformance testing. In *Proceedings of the Second International Workshop on Dynamic Analysis (WODA 2004)*, pages 9–16, 2004.

[15] J.H. Andrews and Y. Zhang. General test result checking with log file analysis. *Software Engineering, IEEE Transactions on*, 29(7):634–648, 2003.

[16] Ilse Leal Aulenbacher, José María Suárez Jurado, and Efrén R. Coronel Flores. A real-time data acquisition system for the Laguna Verde nuclear power plant. *W. Trans. on Comp.*, 9(7):778–787, July 2010.

[17] Luciano Baresi and Michal Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001. `http://www.cs.uoregon.edu/~michal/pubs/oracles.html`.

[18] F. Chang and J. Ren. Validating system properties exhibited in execution traces. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 517–520. ACM, 2007.

[19] E.R. Coronel Flores. Desarrollo de una herramienta de monitoreo y depuración de errores de software en tiempo real para el nuevo sistema de adquisición de datos del SIIP de la Central Nucleoeléctrica Laguna Verde. *IX Congreso Internacional sobre Innovación y Desarrollo Tecnológico (CIINDET)*, (Article ID 409), November 2011.

[20] J. Friedl. *Mastering regular expressions.* O'Reilly Media, Inc., 2006.

[21] J. Goyvaerts and S. Levithan. *Regular expressions cookbook.* O'Reilly Media, 2009.

[22] K. Kent and M. Souppaya. Guide to computer security log management. Technical Report NIST Special Publication 800–92, National Institute of Standards and Technology, Gaithersburg, USA, September 2006.

[23] Michael Khoo, Joe Pagano, Anne L. Washington, Mimi Recker, Bart Palmer, and Robert A. Donahue. Using web metrics to analyze digital libraries. In *Proceedings of the 8th ACM/IEEE-CS joint conference on Digital libraries*, JCDL '08, pages 375–384, New York, NY, USA, 2008. ACM.

[24] J. Levine. *Flex & bison.* O'Reilly Media, Inc., 2009.

[25] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Automated test oracles for GUIs. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications*, SIGSOFT '00/FSE-8, pages 30–39, New York, NY, USA, 2000. ACM.

المنارة للاستشارات
www.manaraa.com

[26] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 575 –584, June 2007.

[27] A.J. Oliner, A. Aiken, and J. Stearley. Alert detection in system logs. In *Data Mining, 2008. ICDM '08. Eighth IEEE International Conference on*, pages 959 –964, December 2008.

[28] D.K. Peters and D.L. Parnas. Using test oracles generated from program documentation. *Software Engineering, IEEE Transactions on*, 24(3):161 –173, March 1998.

[29] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th international conference on Software engineering*, ICSE '92, pages 105–118, New York, NY, USA, 1992. ACM.

[30] W.R. Stevens. *UNIX Network Programming: Interprocess Communications*, volume 2. Prentice Hall, second edition, 1999.

[31] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Salsa: analyzing logs as state machines. In *Proceedings of the First USENIX conference on Analysis of system logs*, pages 6–6. USENIX Association, 2008.

[32] Dave Thomas and Andy Hunt. State machines. *IEEE Software*, 19(6):10–12, 2002.

[33] J. Valdman. Log file analysis. Technical report, Department of Computer Science and Engineering, University of West Bohemia in Pilsen, Czech Republic, 2001. Tech. Rep. DCSE/TR-2001-04.

[34] E.J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[35] D.J. Yantzi and J.H. Andrews. Industrial evaluation of a log file analysis methodology. In *Dynamic Analysis, 2007. WODA '07. Fifth International Workshop on*, page 4 (paper index); 7 pages total, New York, NY, USA, May 2007. ACM.

المنارة للاستشارات

www.manaraa.com

Vita

Name	Ilse Leal Aulenbacher
Place of Birth	Distrito Federal, México
Year of Birth	1981
Post-secondary Education and Degrees	The University of Western Ontario London, Ontario, Canada 2010–2011 M. Sc. of Computer Science
	Universidad del Sol Cuernavaca, México 1999–2003 B.Sc. in Computer Systems
Honours and Awards:	Mexican National Council of Science and Technology (CONACYT) Graduate Research Scholarship, 2011–2012
	Mexican Institute of Electrical Research (IIE) Graduate Research Scholarship, 2011–2012
Related work experience:	Researcher, developrment of real–time systems Mexican Institute of Electrical Research 2004–2010
Publications:	Ilse Leal Aulenbacher, José María Suárez Jurado, and Efrén R. Coronel Flores. A real-time data acquisition system for the Laguna Verde nuclear power plant. *W. Trans. on Comp.*, 9(7):778–787, July 2010.
	Ilse Leal Aulenbacher and José María Suárez Jurado. A data acquisition system for the Laguna Verde nuclear power plant. In *Proceedings of the 8th WSEAS international conference on Data Networks, Communications, Computers,* (DNCOCO'09), pages 142–146, tevens Point, Wisconsin, USA, 2009. World Scientific and Engineering Academy and Society (WSEAS).

المنارة للاستشارات
www.manaraa.com