Generating C++ Log File Analyzers

ILSE LEAL–AULENBACHER
Instituto de Investigaciones Eléctricas
Gestión Integral de Procesos
Reforma 113, 62490 Cuernavaca
MEXICO
ilse.leal@iie.org.mx

JAMES H. ANDREWS
Western University
Department of Computer Science
Middlesex College N6A 5B7, Ontario
CANADA
andrews@csd.uwo.ca

Abstract: Software testing is a crucial part of the software development process, because it helps developers ensure that software works correctly and according to stakeholders' requirements and specifications. Faulty or problematic software can cause huge financial losses. Therefore, automation of testing tasks can have a positive impact on software development, by reducing costs and minimizing human error. To evaluate test results, testers need to examine the output of the software under test (SUT) to determine if it performed as expected. Test oracles can be used to automatize the evaluation of test results. However, it is not always easy to directly capture the inputs and outputs of a program. It is already a common practice for developers to instrument their code so that important events get recorded in a log file. Therefore, a good alternative is to use test oracles capable of analyzing log files. Test oracles that analyze log files are known as log file analyzers. The work presented in this paper builds upon previous research by Dr. James H. Andrews, who proposed a framework in which log file analyzers are generated automatically based on the SUT's expected behavior. These log file analyzers are used as test oracles to determine if the log file reveals faults in the software. In this paper, we describe how we extended Andrews' log file analysis framework in order to incorporate new capabilities that make our log file analyzers more flexible and powerful. One of our main motivations was to improve the performance of our log file analyzers. The original log file analyzers were based on the Prolog language. Our new log file analyzers are based on the C++ language, which allowed us to take advantage of the object–oriented paradigm to add new features to our analyzers. We discuss those features and the experiments we performed in order to evaluate the performance of our analyzers. Our results show that our C++ analyzers are faster than their Prolog counterparts. We believe that log file analyzers have a lot of potential in the area of software testing by making it easier for testers to automatize the evaluation of test results.

Key–Words: Software Testing, Test Oracles, Log File Analysis

1 Introduction

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. A *test oracle* refers to a mechanism capable of determining if the software under test (SUT) output is correct or incorrect. Test oracles are not easy to implement in part because it is not always easy to directly capture a program's inputs and outputs. Therefore, a good alternative is to analyze log files produced by the SUT.

Most programs already produce log files because it is a common practice for developers to instrument their code so that important events, warnings or error messages get saved into a text file.

The work presented in this paper builds upon the Log File Analysis (LFA) framework proposed by Andrews [1], in which log file analyzers are automatically generated from the specification of a program's expected behavior. The original LFA framework consists of the following elements: *a)* a language known as LFAL (Log File Analysis Language) which captures the expected program behavior as a set of state machines, *b)* a log file analyzer generator, which takes a LFAL specification and generates Prolog code and *c)* an executable log file analyzer, which is obtained after compiling the generated Prolog code. The log file analyzer is a test oracle that determines whether a log file reveals a fault in the SUT.

This paper describes how we extended the original LFA framework in order to incorporate new features to log file analyzers to make them more flexible. In Section 2, we provide a brief overview of related work regarding test oracles. We then describe the original LFA framework in Section 3.

Another important motivation was to improve the general performance of our log file analyzers. For that reason, we designed and developed a new log file analyzer generator that produces C++ code instead of Prolog code. That allowed us to extend the original

LFAL language in order to introduce new features. We describe the design of our new C++ log file analyzer generator in Section 4. In addition, we describe the process by which a LFAL 2.0 specification is processed to produce C++ code.

Upon compiling the C++ code produced by our log file analyzer generator, we obtain an executable log file analyzer. In Section 5, we present an example that illustrates how an analyzer is used to determine if the SUT is working in compliance with its expected behavior.

Section 6 qualitatively compares our implementation with the original version by giving an overview of the new features that make our new analyzers more flexible and powerful.

In Section 7 we present the experiments we carried out in order to compare the performance of the original LFAL analyzers and our new implementation. The analyzer we generated for our experiments also serves to illustrate some of the new features described in Section 6.

Finally, we present our conclusion and describe our plans for future work.

2 Related work

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 [2]. 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 [2].

The whole point of testing is to reveal system failure or provide assurance 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. However, much of the research on software testing has focused on the development and analysis of input data [2]. In fact, research literature on test oracles is a relatively small part of the research literature on software testing [3]. Therefore, researchers that work with test oracles have tried to raise awareness on the importance oracles in the software testing process.

2.1 Deriving oracles

Being the evaluation of test results an important part of the software testing process, the question of why test oracles are not always used comes to mind. The explanation is that oracles are not particularly easy to derive.

For instance, Peters and Parnas [4] recognize that the documentation used to generate an oracle can be almost as complicated as the software under test.

Richardson et al. [5] 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 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. [6] 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 Deriving oracles automatically

Peters and Parnas [4], 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.

3 The Log File Analysis framework

Dr. James H. Andrews, proposed a framework in which the expected program behavior is expressed as a set of state machines. In this section, we discuss the framework proposed by Andrews [1] [7] [8] [9] 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). Throughout this paper, we refer to Andrews' log file analysis framework as LFA.

In LFA, a *log file* is defined as a sequence of log lines. 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.

3.1 The Log File Analyzer Language

Throughout section 2.1, we gave an overview of different ways to derive oracles. Oracles need to capture the expected program behavior in some way. In order to determine whether a program is behaving correctly or incorrectly, we first need to define the *expected program behavior*. Andrews introduced the Log File Analysis Language (LFAL), which is used to specify the expected program behavior in the form of state machines.

3.2 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 [7] 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.

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.

This framework has been applied to several lab–built pieces of software such as an elevator controller and a heater monitor [7] and to two pieces of commercial software [9].

4 Our implementation

While LFA 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.

To address this issue, we developed a new implementation of LFA, which instead of producing Prolog–based log file analyzers, produces C++ log file analyzers. Henceforth, we will refer to our new implementation of the LFA framework as LFA2.

Our implementation involved two main aspects: the first was to extend the original LFAL language in order to take advantage of the object–oriented paradigm and to be able to incorporate new features, such as support for regular expressions. We refer to our new, extended version of the LFAL language as LFAL 2.0.

The second aspect involved designing and developing a new translator, which translates a LFAL 2.0 program into C++ code.

Our objective was to extend the original LFA framework in order to make log file analyzers more flexible and powerful. By producing C++ log file analyzers, we also expected to improve the performance of our analyzers.

4.1 The log file analyzer generation process

Figure 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 a LFAL 2.0 program. This code is scanned and parsed resulting in an abstract syntax tree (AST). The AST is analyzed by the code generator and the following files are generated: C++ code for machine classes , a Makefile and C++ code for the log file analyzer program. Using the Makefile and a C++ compiler, these files are compiled and linked with the base libraries. At the end of the process, we obtain an executable log file analyzer. In Section 5, we provide an example that illustrates how log file analyzers are used as test oracles.

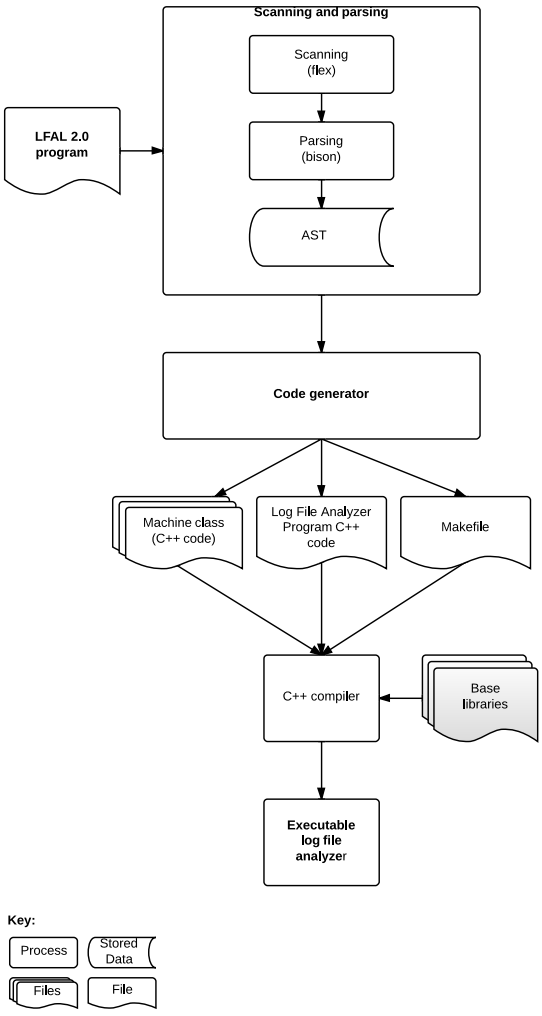

Figure 1: Log File Analyzer Generation Process

4.1.1 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 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. This library was developed to make LFAL 2.0 analyzers capable of reading log files from shared memory areas. This is particularly useful for real–time systems.

4.1.2 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, a 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 2 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 2 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`.

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.

4.1.3 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.

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.

5 A simple example

The LFAL 2.0 program shown in Figure 3 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*

```
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 }
```

Figure 2: A LFAL 2.0 program with two machine classes

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".

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 with our log file analyzer generator, we would obtain (after compilation) an executable log file 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.

```
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: A simple example of a LFAL 2.0 program

6 New features in LFAL 2.0

In this section, we present a brief overview of the new features introduced in LFAL 2.0. For a more detailed explanation of each new feature together with examples, see [10].

6.1 Support for regular expressions

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.

In LFAL 2.0, 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. Patterns are defined at the top. The reason for this is that several machine classes can notice a pattern.

Figure 4 shows an excerpt of an LFAL 2.0 program, which shows an 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`".

It is important to mention that the syntax `(?<T>)` is part of the standard PCRE syntax for named substrings. 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. In the original version of LFAL, analyzers did *not* have a way to specify a pattern for a log line. Therefore, if the

```
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;
...
```

Figure 4: A pattern definition

example shown in figure 4 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.

6.2 Dynamic and static analyzer machines

In log files, it is common to find groups of lines, which are related to a specific identifier such as a transaction ID number. LFAL 2.0 supports *dynamic machines*, which are used to analyze these kind of log files. 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. Our new implementation takes advantage of C++ object–oriented–paradigm to dynamically create or delete analyzer machines when certain log lines are noticed.

Dynamic analyzer machines are necessary to correctly analyze log files with more than one instance of a group of log lines.

For example, let us suppose we need to generate a log file analyzer that works as a memory leak checker. Let us suppose that the SUT outputs a log file that, among other information, contains lines that indicate memory allocations or deallocations. To correctly analyze such a log file, we would need the log file analyzer to be able to process"allocate" and "deallocate" lines in any possible sequence. Dynamic analyzer machines make this possible by creating or deleting analyzer machines upon noticing log lines reflecting memory allocation or deallocation, respectively. This analyzer machine could check that once reaching the end of the file, no instances of analyzer machines remain, thus indicating that for every allocation, there was a deallocation.

6.3 New types of actions in 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.

These new transition actions provide the user with a fine-grained control over analyzer machine transitions in LFAL 2.0. For example, in Section 7, we describe the log file analyzer we used for our performance experiments. Such analyzer features a special type of transition that executes C++ code.

6.4 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.

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 LFA2 program. In Section 7, we describe the log file analyzer we generated for our experiments, which incorporates a data declaration.

7 Performance tests

The objective of our experiment was to compare the performance of the original, Prolog-based log file analyzers with our LFAL 2.0 analyzers.

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 [11]. We designed our experiment in this way because a significant amount of the work on log files focuses on the analysis of logs generated by servers[12]. In addition, we were 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*.

This log file was obtained from the Sandia National Laboratories webpage [13] and it contains 4,747,963 messages in 709 megabytes.

7.1 Experiment design

Our experiment consisted in generating the BG/L analyzer in both the original version of LFAL and LFAL 2.0. 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 performance, we generated four versions of the BG/L analyzer. We will refer to them as `bgl01`, `bgl02`, `bgl03` and `bgl04`. The analyzer `bgl01` matches and counts lines with *one* pattern. `bgl02` looks for lines with *two* different patterns. Similarly, `bgl03` matches and counts lines with *three* different patterns. `bgl04` matches and counts *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 [14].

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`.

7.2 Performance of LFAL 1.0 analyzers

As the reader might remember from Section 6.1, 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 simple entries such as:

```
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. Sed is a stream editor used to perform basic text transformations on an input stream [15]. Our sed script matches the patterns and replaces them by its corresponding name.

Our sed script matches the patterns and replaces them by its corresponding name. For instance, a line such as:

`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. This is necessary because LFAL 1.0 analyzers print an error message when a log file line is not noticed by any analyzer machine. 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 7.3, LFAL 2.0 analyzers allow users to suppress error messages concerning unnoticed lines, without having to modify the log file. Figure 6 shows

```
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
```

Figure 5: 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.

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. 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. It is important to note that

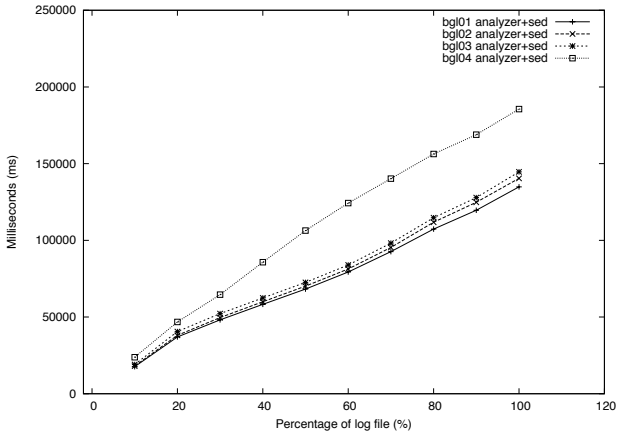

Figure 6: The CPU user time for bgl01, bgl02, bgl03 and bgl04.

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 7 shows the system time for the four BG/L analyzers. 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. Similarly to the user time graph in figure 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 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.

7.3 Performance of LFAL 2.0 analyzers

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 with-

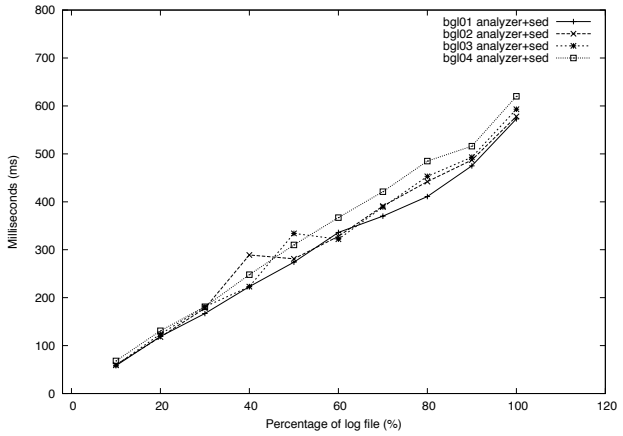

Figure 7: The CPU system time for bgl01, bgl02, bgl03 and bgl04.

out requiring the log file to be adapted or modified in any way.

Figure 8 shows the LFAL 2.0 version of the `bgl04` analyzer. This BG/L analyzer declares four

```
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  }
```

Figure 8: The bgl04 analyzer in LFAL 2.0. This analyzer matches and counts four patterns.

patterns in lines 1–4. This analyzer uses the data declaration feature in LFAL 2.0 (see Section 6.4) 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 an example of a special type of transition (see Section 6.3). 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 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 11 shows a graph with our measurements for the average user CPU time for `bgl01-04` analyzers. 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. Our results show that the time increases with both log file size and number of patterns.

Figure 9 shows the average system CPU time for the four LFAL 2.0 BG/L analyzers. 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. 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.

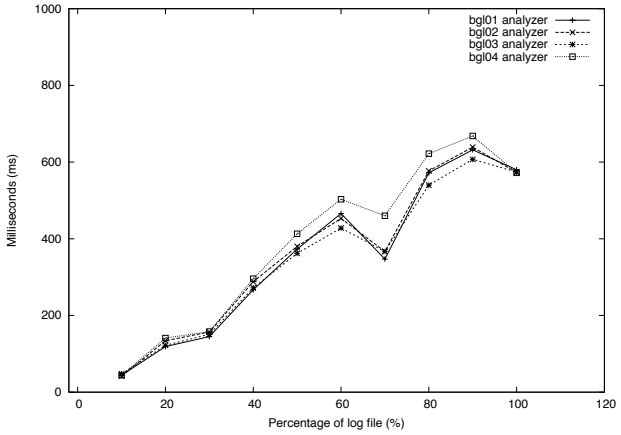

Figure 9: The CPU system time for bgl01, bgl02, bgl03 and bgl04.

7.4 Results

By comparing the user CPU time results for LFAL 1.0 in figure 6 and LFAL 2.0 in figure 11, 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 CPU time for LFAL 1.0 and the average user CPU time for LFAL 2.0. In Figure 10, the *x* axis represents the percentage of the log file that was analyzed and the *y* axis represents the `LFAL1:LFAL2` ratio. The graph 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 12 shows the ratio between the system time in LFAL 1.0 and LFAL 2.0. The *x* axis represents the percentage of the log file that was analyzed and the *y* axis represents the `LFAL1:LFAL2` ratio. 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 2.0 analyzers, LFAL 2.0 analyzers are still faster than LFAL 1.0 analyzers.

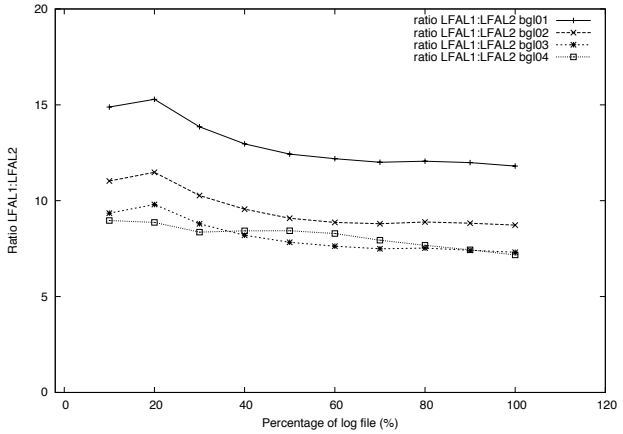

Figure 10: The CPU user time ratio between LFAL 1.0 and LFAL 2.0 for bgl01, bgl02, bgl03 and bgl04.

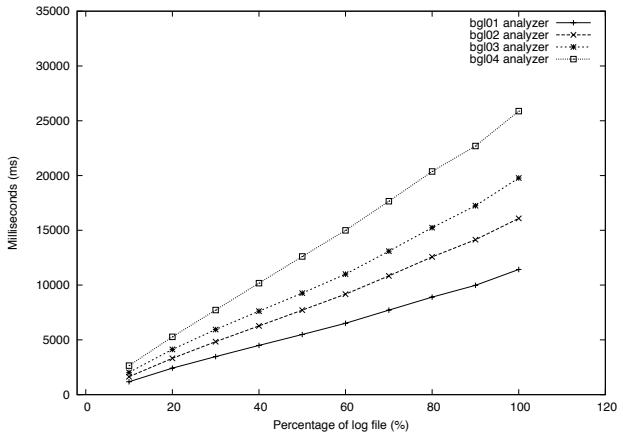

Figure 11: The CPU user time for bgl01, bgl02, bgl03 and bgl04.

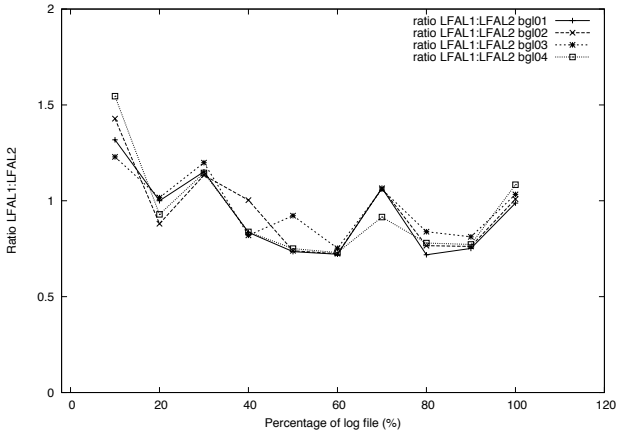

Figure 12: The CPU system time ratio between LFAL 1.0 and LFAL 2.0 for bgl01, bgl02, bgl03 and bgl04.

8 Conclusion

The main objective of the work presented in this paper 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++ 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. 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. 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.

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.

9 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 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.

Acknowledgements: In the case of the first author, this research was supported by 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).

References:

[1] 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.

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

[3] 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`.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] Leal Aulenbacher, Ilse, "Generating Log File Analyzers" (2012). University of Western Ontario – Electronic Thesis and Dissertation Repository. Paper 780. `http://ir.lib.uwo.ca/etd/780`

[11] 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.

[12] 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.

[13] Supercomputer event logs. `http://www.cs.sandia.gov/~jrstear/logs/`. [On–line. Accessed June 2012].

[14] Linux programmer's manual - time. Linux Man Pages. [Accessed July 2012].

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