Broad-Spectrum Studies of Log File Analysis

James H. Andrews, Yingjun Zhang
Dept. of Computer Science
Univ. of W estern Ontario
London, Ontario
CANAD A N6A 5B7
(519)661-2111 x86856
andrews,yjzhang@csd.uwo.ca

ABSTRA CT
This paper reports on research into applying the technique of log file analysis for checking test results to a broad range of testing and other tasks. The studies undertaken included applying log file analysis to both unit- and system-level testing and to requirements of both safety-critical and non-critical systems, and the use of log file analysis in combination with other testing methods. The paper also reports on the technique of using log file analyzers to simulate the software under test, both in order to validate the analyzers and to clarify requirements. It also discusses practical issues to do with the completeness of the approach, and includes comparisons to other recently-published approaches to log file analysis.

Keywords
Testing, specification, safety verification, lightweight formal methods, test oracles

1 INTR ODUCTION

Testing is an important and costly component of any software development project, and the efficient, reliable validation of test results is an important component of the testing task. In previous papers [3, 4, 2], we described the technique of *log file analysis* (LFA) for test result evaluation. In log file analysis, we formally analyze software event logs in order to evaluate whether the software under test has behaved as expected. W e undertook various studies to see how broadly applicable LFA was to a variety of different testing tasks, how well LFA could be integrated with other testing techniques, and whether LFA artifacts could be used in other development tasks.

This paper reports on these studies. Our studies indicate that LFA is applicable to both unit- and system-level testing, that it scales up well to more complex requirements, that it can be used in combination with other testing methods, and that our proposed state-machine-based approach to LFA results in log file analyzers which can be used for such things as simulation of the SUT.

ICSE 2000, Limerick, Ireland

Log file analysis is not a comprehensive approach to testing, but rather a general framework for reliably automating one part of the testing task (test result evaluation). The effective use of the log file analysis framework is based on four main hypotheses:

1. The software under test (SUT) writes a record of events to a distinguished output called the *log file*. These events are typically such things as inputs and outputs, messages sent and received, parameters and return values of key functions, and changes of state of key variables.
2. W e are following a well-defined, agreed-upon *logging policy* which states precisely what the SUT should write to the log file, and under what precise conditions. This may be enforced by automatic instrumentation or simply standard code review and inspection procedures.
3. We have written, or can write, a *log file analyzer* program which takes a log file as input and either accepts the log file or rejects it with an informative error message.
4. Given hypotheses 1-3, we are confident that the analyzer will accept a log file if and only if the run which produced it did not reveal any of the failures which we wanted to check for in the SUT.

Hypothesis (1) is already true of much commercial software development today; "test logs" or "debug logs" are a common method of facilitating testing and debugging. Hypotheses (2)-(4) can be seen as modifying the SUT so that it produces a formal artifact which can be analyzed formally. The connection between the formal analysis and the informal requirements lies in the logging policy. The use of LFA in testing is therefore a "semi-formal" or "lightweight formal" method.

There are two studies by independent researchers that we are aware of which also use the log file analysis hypotheses. Feather, in his study of lightweight formal methods for spacecraft control software [8], used a database query tool to analyze the log file produced by test runs on the software. The lines in the log files in his study contained sensor data and actuator commands. Qiao and Zhang, in their study of testing communication safety properties of distributed systems [18], built a customized tool which checks log files produced by the systems under test. The lines in the log files there recorded communication events.

Our previous paper [2] details the approach we have taken to log file analysis, which views an analyzer as a collection of state machines with precise behaviour. Such an approach is more general than tools customized to particular application domains. In addition, because it specifies well-defined formal artifacts, it allows analyzers to be amenable to formal validation techniques such as simulation.

Log file analysis is connected to research on automatically deriving test oracles from precise specifications [17, 16], and to research on event-based debugging [9, 19, 13, 14], which emphasizes issues of classification and presentation of events to users. In LFA, we are instead essentially interested in directly writing an oracle, independent of the platform of the SUT, which evaluates the event sequences for correctness.

The remainder of this paper is structured as follows. Section 2 briefly describes our state machine approach to log file analysis. Section 3 summarizes the studies we undertook. Section 4 describes how we used LFA in two unit testing studies, and Section 5 how we used LFA in system testing studies. Section 6 discusses the use of our state machine analyzers to simulate the SUT and, in turn, to validate the anlyzers. Section 7 discusses the completeness of the state machine approach, and Section 8 draws some conclusions.

2 THE STATE MACHINE APPROACH TO LOG FILE ANALYSIS

Here, we describe the approach to LFA which we have taken in previous papers, such as [2].

Foundations

The state machine approach is a particular instance of the LFA framework. In this approach, we view an analyzer as a collection of state machines (each with a possibly infinite number of states) which make transitions based on log file lines and report errors if no transitions are possible.

More precisely, given a countable set L of potential log file lines, an *analyzer machine* consists of: a countable set S of states; a distinguished *initial state* $i \in S$ and countable set $F \subseteq S$ of *final states*; a countable set $N \subseteq L$ of lines which the machine *notices*; and a computable *transition relation* $\Delta \subseteq S \times N \times S$, showing how a log file line moves the machine from one state to another. An analyzer machine is thus like a standard state machine, except that it notices some inputs and ignores others. This extension is motivated by the fact that a machine in a log file analyzer will typically be tracking only one "thread" of messages on the log file, which will be interspersed with other threads reporting on other aspects of the SUT.

For an analyzer machine to evolve from one state s to another state s' on a given input line l, we must have $\langle s, l, s' \rangle \in \Delta$. A given analyzer machine accepts a log file (sequence of lines) if it can evolve from its initial state to one of its final states on the sub-sequence of lines in the log file which it notices. In particular, an analyzer machine rejects a log file if some prefix of the log file causes it to evolve to a state in which it notices the next line, but cannot evolve on it.

A *log file analyzer* is a countable set of analyzer machines. An analyzer accepts a log file if all its constituent machines accept it.

As a simple example, consider a program which maintains a data structure representing a set of integers and queries the set. The program writes `add` n to the log file whenever it adds n to the set, `find` n whenever it successfully finds n in the set, and `delete` n whenever it deletes n from the set. Our requirements are that the program adds n only when n is not yet in the set, and that it deletes or successfully finds n only when n is in the set.

A state machine log file analyzer for such requirements can be formulated in two different ways. In the first way, the analyzer consists of a single state machine with an infinite number of states, each state representing one possible finite subset of the integers; the Δ relation for the machine moves it from state s to state $s \cup \{n\}$ whenever `add` n is encountered and $n \notin s$, from s to $s - \{n\}$ whenever `delete` n is encountered and $n \in s$, and from s back to s whenever `find` n is encountered and $n \in s$. In all other cases – for instance, when `add` n is encountered and $n \in s$, or when `find` n is encountered and $n \notin s$ – the analyzer reports an error as a consequence of the definition of acceptance or rejection of a log file.

The other way of specifying an analyzer for the program is to consider it to be an infinite collection of machines $watch_n$, each watching for a particular integer n, and each consisting of only two states – *inset* and *notinset*. The Δ relation of each machine $watch_n$ moves it from *notinset* to *inset* when `add` n is encountered, from *inset* to *notinset* when `delete` n is encountered, and from *inset* to *inset* when `find` n is encountered. This formu-

lation has the advantage of reducing each constituent machine to two states with no conditions on the transitions. It is completely equivalent to the previous formulation, in the sense that it accepts or rejects log files in exactly the same circumstances.

What we have described above is the basic framework; for more details of the full framework, including communication between analyzer machines, see [3].

Log File Analysis Language
In order to express state-based log file analyzers, we defined a simple language called LFAL. In LFAL, we write an analyzer by writing definitions for each of the machines composing it. LFAL allows us to express infinite numbers of machines, and infinite numbers of states in a given machine, by allowing us to parameterize machine names and state names.

As an example, consider the second analyzer mentioned in the previous section. This could be formulated in LFAL as follows:

```
machine watch(N);
  initial_state notinset;
  from notinset, on add(N), to inset;
  from inset, on delete(N), to notinset;
  from inset, on find(N), to inset;
  final_state Any.
```

In LFAL, upper-case identifiers are variables and lower-case identifiers are constants or function symbols, as in Prolog. Machine names, states and messages are Prolog (first order) terms. Transitions can have conditions on them, which are Prolog queries. A machine is assumed to notice all and only the forms of log file lines which appear in the "on" clauses of its transitions. We have written an LFAL compiler which translates an LFAL analyzer into Prolog and then compiles it. We use GNU Prolog, a public-domain Prolog which produces fast stand-alone executables. (See [2] for more details on LFAL.)

When using LFAL analyzers, we must make an assumption about the format of the input. We assume that every line starts with a keyword (a lower-case identifier), and continues with a sequence of whitespace-separated keywords, numeric constants and double-quoted character strings. This assumption allows us to map log file lines easily to LFAL constructs; the line `add 5`, for instance, corresponds to the first-order term `add(5)`. In combination with the general log file analysis hypotheses, this provides us with an easy answer to the dilemma of mapping program inputs and outputs from the namespace of the program to the that of the specification, a problem which has bedevilled attempts to generate oracles from formal specifications in the past [17, 6, 16].

3 SUMMARY OF STUDIES

Tables 1 and 2 summarize the various studies we have done and the two studies by other researchers mentioned in the Introduction. The following is a somewhat fuller description of each study.

- Study A used LFA in unit testing of four implementations of a generic C module (a data structure for storing data with associated keys).
- Study B used LFA to test a simulator for the ACSE (Association Control Service Element) protocol, one of the elements of the OSI protocol stack.
- Study C used LFA to test a 760-line Java program which translated text from a convenient line-based, point-form format to structured LaTeX input.
- In study D, we wrote a log file analyzer to test implementations of the well-known steam-boiler control system described by Abrial et al. [1].
- In study E, we wrote a log file analyzer for checking user interface requirements of a hypothetical GUI-based editor program.
- Study F used LFA to test the basic requirements of the Unix utility `diff`, which reports on differences between two files.
- Study g is the study by Feather [8] of analysis of log files produced by a spacecraft control module.
- Study h is the study by Qiao and Zhang [18] of the use of a customized log file analyzer to test communication safety properties of distributed systems.

All of our studies used the state-machine approach; those of Feather (g) and Qiao and Zhang (h) used the log file analysis approaches mentioned earlier.

The rationale for our choice of studies was as follows. We wanted to study whether LFA was applicable on both a unit and system testing level. It seemed clear, from earlier studies on unit testing by other researchers [6, 10, 21], that unit testing of well-defined classes can be done with straightforward formal specifications. We therefore concentrated on unit testing studies that showed we could do similar work with LFA (studies A and B), and on system testing studies (C, D, E and F).

Industry development groups developing safety-critical software are often the earliest to experiment with new technology, since the cost of exploring new technology may be less than that of producing potentially hazardous software. Hence it is important to study new development techniques in the context of safety-critical

Study Name	Description of SUT
A	C dictionary implementations
B	ACSE protocol simulator
C	Document formatting system
D	Steam boiler specification problem
E	GUI-based editor
F	GNU `diff`
g	Spacecraft control (Feather)
h	Distributed system testing (Qiao, Zhang)

Table 1: Descriptions of log file analysis studies. Upper-case letters denote our studies, lower-case letters the studies of other researchers.

Property	A	B	C	D	E	F	g	h
LFA framework:								
•LFAL	•	•	•	•	•	•		
•Database search							•	
•Customized								•
Level of testing:								
•Unit	•	•						
•System			•	•	•	•	•	•
Safety-criticalness:								
•Not critical			•		•			
•Neutral	•	•				•		•
•Critical				•			•	
Realization:								
•Actual code	•	•	•			•	•	•
•Analyzer only				•	•			
Other methods:								
•Random testing	•	•	•			•		
•Code coverage	•	•				•		
•Simulation	•	•		•	•			

Table 2: Comparison of properties of log file analysis studies.

Metric	A	B	C	D	E	F
SUT code (LOC)	1014	582	761	na	na	13891
Coverage (%)	100	100	na	na	na	22
Analyzer (LOC)	15	106	57	485	95	88

Table 3: Comparison of metrics of our log file analysis studies. LOC is raw lines of code; coverage is line coverage as measured by the GNU coverage tool `gcov`.

systems, but also to study its applicability outside the safety-critical domain to evaluate whether the technology can be transferred in the future to non-critical applications.

We therefore undertook two studies on clearly non-critical software (C and E), and one study on a safety-critical specification problem (D). Our other studies concerned software which may be incorporated into safety-critical software, depending on the context (A and B). We note that Feather's study (g) also involved a safety-critical application.

Since we were primarily interested in the breadth of application of state-machine LFA and in whether it scaled up well, we were primarily interested in writing analyzers to do with different project domains and with varying degrees of specification complexity. Nevertheless, we used actual code whenever possible. Table 3 summarizes the metrics associated with the SUT and LFAL code. Studies A, B and F involved C code; study C involved a Java program with nine object classes. The two cases in which we did not use actual code were for the editor specification (E), which was undertaken mainly to study LFAL-based modelling and simulation of a non-safety-critical application with a reactive interface, and the steam boiler control problem (D), which is a well-known formal specification problem with many available alternative specifications.

Finally, since test result evaluation is only a small part of the software development process, we wanted to see whether and how log file analyzers could be used in combination with other testing techniques and/or in other development tasks. We therefore used LFA in combination with testing from random inputs on four studies (A, B, C and F) and code coverage on three studies (A, B and F; we had no access to Java code coverage tools). We also added a feature to the LFAL framework to allow us to run LFAL analyzers in "simulate" mode, in which the user could type in log file lines representing inputs and the analyzer would respond with a correct sequence of log file lines representing non-inputs. This allowed us to treat the LFAL analyzer as a specification of high-level behaviour of the SUT, in order to validate the analyzer and simulate the SUT. We applied this simulation to studies A, B, D and E.

4 LFA IN UNIT TESTING

We used LFAL analyzers to test two units: implementations of a container module, and a simulator of the ACSE (Association Control Service Element) of the OSI protocol stack.

C Dictionary Implementations

We tested four C implementations of a "dictionary" module somewhat more complex than that described in the Introduction, using LFA to check test results. The

dictionary module specification contained functions to add, delete and find keys with associated data in the dictionary. The four implementations used a linear list, a binary tree, a B-tree and an AVL tree as the underlying data structures. This study was of similar complexity to those discussed by Hoffman and Strooper [10].

We tested each implementation using the same driver and log file analyzer, using techniques of random testing [7] and fault injection [20]. (We chose these techniques because they were powerful testing tools for which their creators cited checking of test results as an important outstanding issue.) The driver repeatedly selected random functions to call and gave them random 4-bit integer arguments, logging the parameters and results of each function call. We analyzed the resulting log files to check for errors in the implementations. We found that the log file analysis helped most after we had worked out any problems which caused the software to crash (e.g. Unix segmentation faults), but that we did find errors in the code as a result of the log file analysis.

We then measured how many runs of the same number of function calls were typically necessary to achieve 100% block coverage (as measured by the GNU project coverage tool `gcov`), and how long runs had to be to detect faults injected into the implementations. As expected, the more complex implementations (e.g. the AVL tree implementation) took longer to reach 100% coverage and longer to detect injected faults than the less complex implementations (e.g. the linear list implementation).

Our preliminary results support the hypothesis that this form of random testing with log file analysis with the goal of achieving 100% block coverage is an effective test method for these modules. A full tabulation of the results is expected in the second author's MSc thesis. We are extending the results to larger and more complex modules by testing the container classes of the C++ Standard Template Library (STL) [12].

Three main points emerge from this study. First, due to the log file analysis hypotheses, the oracle (here, the log file analyzer) is a separate process with a separate address space. It is therefore guaranteed not to interfere with the operation of the SUT (unless, of course, hard real-time constraints prevent logging altogether). This is in contrast to software testing practices in which driver, SUT and oracle are compiled together. Second, the log file analysis hypotheses and the LFAL formulation of the analyzer allow us to give a simple but precise specification of the SUT which can be applied directly, using only the LFAL compiler. Third, the LFAL analyzer is independent of the language in which the SUT is written, and even the platform on which it is run. It can therefore potentially be viewed as a specification; for instance, an analyzer for a utility class, together with a simple logging policy which class drivers must adhere to, could be taken as a standard to which to hold implementations of the class.

The use of random inputs in combination with LFA allows the data structure or object under test to be potentially in any allowable state. This is in contrast to approaches which restrict the allowable state space in order to simplify the oracle [10]. Random testing has well-known deficiencies, including difficulties in covering all code and requirements due to the random nature of the walk through the state space of the program. Our study suggests that if we use random testing in conjunction with LFA and code coverage tools, and we design our driver, logging policy and log file analyzer carefully, we can achieve higher assurance than with random testing alone.

ACSE Protocol Simulator

The ACSE, or Association Control Service Element [11], is the element of the ISO protocol stack which sets up associations between agents, and handles protocol version conflicts and conflicts in the releasing of the associations. We had previously developed a formal specification of the ACSE for validating a set of protocols which used it [5], and we wished to see whether it could be translated easily into LFAL and used in testing.

We wrote a simulator program which read input lines denoting messages received by the ACSE. The simulator echoed its inputs on standard output as "messages received", and also wrote the "messages sent in response" on standard output. We took this output as the log file. Again using random testing, we wrote a driver to generate random messages to be piped to the simulator (which ignored out-of-sequence messages), and analyzed the log file using an LFAL analyzer. This analyzer was based on one developed earlier [3], modified to allow the simulator to continue once the ACSE had arrived back at the "idle" state.

We found that in typical runs of the software, about 15% of the messages generated by the input generator corresponded to in-sequence messages; thus a run of 10,000 random inputs generated a log file of about 3,000 lines (the echoed inputs and the system's responses). We found that 100 such runs were usually sufficient to achieve maximal coverage of the simulator code, including the handling of all messages (whether in-sequence or out-of-sequence) from the two least accessible states of the protocol machine. The log file analysis never found any faults in the simulator, an earlier version of which had previously been thoroughly tested by students in the context of a software testing course.

This study supports the claim that general state machine log file analysis can be used to do the kinds of

protocol conformance testing which have long been done in the protocol testing community [21].

5 LFA IN SYSTEM TESTING

Document Formatting System

To study the applicability of LFA to system-level requirements, we used it to test a document processing system called `poi`, developed by the first author, which translates text from a simple and convenient point form into structured input for the LaTeX text processing package. The two main requirements we wanted to check were that LaTeX `begin` and `end` directives were emitted in a matched fashion, and that every line of text read in was output in some way on the LaTeX output.

The main point that emerged from this study was that it is surprisingly difficult, even for such relatively simple requirements, to develop a consistent logging policy and log file analyzer. We made the mistake of writing the analyzer and adding the logging instrumentation with only a vague definition of the logging policy in mind. The first result was false negatives, as the analyzer complained that blank lines were input but never output (in fact, they were correctly ignored). After "correcting" the analyzer to accept the ignored blank lines, we found that the system stopped processing some files without reaching the end of the input, and the analyzer did not complain about it (false positives). This turned out to be due to the system interpreting some input conditions as being equivalent to end of file, and the analyzer now being too lax in accepting non-echoed input text. (It is interesting that we had predicted the dangers of false negatives and positives in [2], and were now experiencing them first-hand.)

It was only after we stepped back and formulated a well-defined logging policy taking into account all eventualities, and based our analyzer on that, that we finally felt we had truly satisfied all the log file analysis hypotheses. We then tested the system using inputs made up of random lines selected from a test file, finding and correcting one fault in the system.

This study illustrated the point that LFA is applicable to testing tasks at the system level as well as the unit level. The code logging text read in and the code logging text written out were in separate classes with no common ancestor in the class hierarchy. This would make it difficult or impossible to use object-oriented frameworks such as ASTOOT [6] in this testing. It was only through running the integrated system and analyzing the resulting log file that the various kinds of errors were revealed. Such a situation is common in system testing, which tests to what extent operations performed by distant modules have been coordinated correctly.

The Steam Boiler Specification Problem

In order to study whether LFA was general enough to apply to complex specifications of software for which reliability testing is important, we expressed in LFAL the requirements for a complex safety-critical system. The steam-boiler control specification problem [1] was the subject of an international study in which researchers were asked to express the specification in their preferred specification formalisms. Many of the researchers went further, showing how the tools they developed could be used to prove properties of the specified system or show how implementations met the specification.

We worked from the pretence that log file analysis hypotheses (1) and (2) had been satisfied (i.e. that we had a working system which logged significant data), and proceeded to write a log file analyzer for the (imaginary) system. In the original specification problem, all data was passed back and forth to the system in well-defined message formats, on a five-second cycle. We assumed a logging policy in which each message sent and received was logged, and in which the end of each transmission to and from the system was noted by a line in the log file.

The resulting LFAL analyzer was 294 SLOC (lines of LFAL code without comments or whitespace), comprising 100 transitions in 10 machine specifications. Not surprisingly, the analyzer came out looking like a formal specification of the system. We did not attempt to model the prediction of current water levels based on available data and maximal and minimal throughput of faulty equipment, although we did ensure that the analyzer checked for proper responses to any messages resulting from these predictions. We validated the analyzer by running it on various use cases based on the specification, and by simulating the system (see next section).

The study supported the claim that LFAL could be used for more complex specifications. We believe that LFAL's success in this respect was due in part to its compositionality. Different aspects of the problem, such as the correct behaviour of the basic control modes and the correct behaviour of the pumps, were specified with separate analyzer machines analyzing different threads of messages. Their combined behaviour produced a more complex analysis which would take longer to express in a more procedural specification.

GUI-based Editor

The next specification task we undertook was to write an LFAL analyzer corresponding to specifications for a non-safety-critical application program with a graphical user interface (a text editor). We worked from the hypothesis that the editor logged input and output events at a relatively high level (for instance, writing the line "`menu file save`" when "Save" was selected from the File menu). Our requirements included the requirement

```
machine no_other_inputs_when_popup;
  % state records which popup is open
  initial_state none;
  % When no popup, all input events allowed
  from none, on menu(_,_), to none;
  from none, on typing(_), to none;
  from none, on open_popup(Name), to open(Name);
  % When popup, no events allowed except
  %   response from this popup.
  from open(Name),
    on popup_response(Name, Response), to none;
  from open(Name),
    on popup_response(Name, Response, Arg),
    to none;
  final_state none.

machine all_popups_get_closed;
  initial_state none;
  from none, on open_popup(Name), to open(Name);
  from open(Name), on close_popup(Name), to none;
  final_state none.
```

Figure 1: Part of an LFAL specification of an editor.

that the "save" popup be displayed if and only if the user selected "Close" or "Quit" when the file had been modified since the last save, that no input events are allowed when a popup is active except the buttons on the popup, and that all popups eventually get closed. Part of the analyzer is shown in Figure 1. (Lines beginning with % are LFAL comments.)

This study illustrated that even in situations in which it would be very difficult to map the raw inputs and outputs of the SUT to constructs in a specification, it is still possible to write an oracle for the SUT given the log file analysis hypotheses. In this case, the raw inputs and outputs of the SUT may be primitive keyboard events and video controller commands, but we assume that only a high-level, tractable view of the I/O is logged. In this way, we sidestep the problems of mapping SUT inputs and outputs to specification constructs, which arise when we attempt to generate an oracle from a formal specification of a non-logging system [17, 6, 16].

GNU `diff`

Finally, we tested the basic requirements of the Unix utility `diff`, which reports on differences between two files, using random testing and LFA. We used the GNU project version of the utility. This study showed that, for some programs with relatively simple output, we can use LFAL without instrumenting the source code if we have some other way of writing the log file.

We constructed random test cases with a generator program which wrote two files of 0-79 lines, each line consisting of a letter possibly preceded and followed by whitespace. The generator output the details of the test case, which we appended to a log file. We then ran `diff` on the two generated files, and piped the output through an editing script which converted it to the log file format described in Section 2. After a sequence of these test cases, we gave the log file to the analyzer. As expected, no errors were found in GNU `diff`. We then altered valid log files slightly to see whether the analyzer would catch the "fault" (it always did).

We tested only the standard-format output. We did not test the many options which selected for other formats of output, and we did not work with very large files, which are processed specially by GNU `diff`. Hence, we achieved rather low coverage of the code. Nevertheless, we feel that the study was realistic; in general, we may want to test only certain features of programs with log file analysis, and not others.

6 ANALYZER-BASED SIMULATION

An LFAL analyzer is a formal object which analyzes formal objects, and therefore it is reasonable to ask whether we can use formal analysis techniques to validate it. Here, we show that it is possible to simulate the SUT given an LFAL analyzer for it and some additional information, and to use the results of the simulation in validating the analyzer.

Simulation Mechanism

We adapted the compiler for LFAL to expand the capabilities of an LFAL analyzer. The user can run the compiled analyzer with an option to put it into "simulation mode". In simulation mode, the analyzer prompts the user for log file lines corresponding to inputs, and displays log file lines not corresponding to inputs. The user (or LFAL programmer) must list, in an auxiliary source file, which lines "correspond to inputs" and which do not. The lines which correspond to inputs should be those which report on such things as input events, lines read in, and messages received.

As an example, consider the ACSE log file analyzer described in Section 4. We listed prototypes for lines reporting on messages received (the input lines) and lines reporting on messages sent (the non-input lines) in a prototypes file, and compiled the analyzer normally. Now, when we run the analyzer in simulation mode, it prompts us for lines corresponding to messages which the ACSE can receive and responds with messages which the ACSE can send. Since an ACSE responds with exactly one message to all in-sequence messages, the analyzer precisely simulates a correct SUT.

The algorithm we used for simulating the SUT based on a general log file analyzer is shown in Figure 2. In general, at any point there may be many different ac-

Repeat:

1. If all machines are in their final states, and the user wishes to terminate the simulation, then terminate.
2. Otherwise, if there is a non-input log file line on which some machine in the analyzer can evolve to a new state (see Section 2):
   2.1. If there is more than one such line, allow the user to select one such line l; otherwise, let l be the only such line.
   2.2. Display l and set the state of the analyzer to the new state after l has been processed.
3. Otherwise, prompt the user for the next input line.
   3.1. If the user has entered an end-of-file, then terminate.
   3.2. If the user has entered a line on which the system cannot evolve, then print an error message.
   3.3. Otherwise, the user has entered a line l on which the system can evolve. Set the state of the analyzer to the new state after l has been processed.

Figure 2: The algorithm used for simulating the SUT based on a log file analyzer.

ceptable log file lines, some input and some non-input. If both input and non-input lines are acceptable at some point, we choose a non-input line. If there is more than one non-input line which will cause the system to evolve, we display the choices to the user and allow them to make an immediate choice of which the system should perform. This might be the case if the analyzer defines acceptable behaviour but does not constrain the system enough to generate a unique output sequence for every input. Work is ongoing to provide users with more control over the precise behaviour of the simulation mode, in case they want to make different choices of how the simulator should act.

Experiences

We found the simulation mode a valuable tool for validating analyzers. During simulation, we could see what inputs and non-inputs were really accepted by the system, and tune the analyzer accordingly, reducing the risk of false negatives and positives.

Consider the editor analyzer of Section 5, part of which is given in Figure 1. Analysis of various test case log files suggested that the analyzer was working acceptably. This was misleading. The bottommost machine in Figure 1 requires that every popup which is opened becomes closed, and the topmost machine implicitly requires that a line indicating that a popup is opened is followed by a line describing the user's response to the popup. However, the analyzer does not require that the response to the popup come before the popup is closed.

```
machine all_popups_get_closed;
  initial_state none;
  from none, on open_popup(Name), to open(Name);
  from open(Name), on popup_response(Name, X),
    to exp_close(Name);
  from exp_close(Name), on close_popup(Name),
    to none;
  final_state none.
```

Figure 3: A repaired version of one of the machines from the editor analyzer.

This problem was revealed when we ran the analyzer in simulation mode and found that an `open_popup` line displayed by the simulation was followed immediately by a corresponding `close_popup` line. Such an analyzer would incorrectly accept a log file produced by a system which closed a popup prematurely, allowing the user no way of making the required response. The fix for the problem was to write a transition for the second machine from `open(Name)` to a new state `exp_close(Name)` on the appropriate `popup_response` line (see Figure 3). We had similar experiences in validating the analyzer for the steam boiler problem (Section 5).

Our experience indicates that simulation is an important technique for validating log file analyzers, as it is for validating formal specifications. Such simulation is not something that one would expect to be able to do with ad hoc log file analyzers written in procedural languages, with domain-specific analyzers such as Qiao and Zhang's for distributed systems [18], or with the database queries which Feather used for log file analysis [8].

7 COMPLETENESS OF STATE MACHINE APPROACH

One of the common concerns raised about the state machine approach to LFA is the fear that it might not be general enough; that is, that it can apply only to simple tasks or tasks involving only state-based software. Here we argue that, in a formal sense, it can be used to test any program against maximally precise specifications.

We can characterize any working program as a mathematical object consisting of:

- A countable set S of states (denoting the possible memory configurations of the program);

- A distinguished initial state $i \in S$ and countable set $F \subseteq S$ of final states;
- Two sets I and O of input and output symbols; and
- A computable transition relation $\Delta \subseteq S \times I \times S \times O^*$, where O^* is the set of finite sequences of outputs, such that $\langle s, i, s', \Gamma \rangle \in \Delta$ if the program evolves from state s on input i to state s', outputting sequence Γ.

Such an object can also act as a (maximally precise) specification of the desired behaviour of a program. Given such a specification, we can write an LFAL analyzer which confirms that a program meets the specification by defining the Δ relation as a Prolog predicate `delta`, and writing an analyzer as follows:

```
machine check;
  initial_state initial;
  from S1, on input(I),
    where delta(S1, I, S2, []), to S2;
  from S1, on input(I),
    where delta(S1, I, S2, [O|Os]),
    to expecting([O|Os], S2);
  from expecting([O], S), on output(O), to S;
  from expecting([O1,O2|Os], S), on output(O1),
    to expecting([O2|Os], S);
  final_state final(X).
```

Here, `initial` stands for the initial state, and `final(X)` stands for any one of the final states. Such an analyzer expects lines on the log file listing each input and output in sequence, and on each input makes a transition into a state where it is expecting to see a log file record of each of the required outputs. Because the Δ relation of the specification is computable and Prolog is Turing-complete, we can always write a correct, complete `delta` predicate.

More practically, we can ask whether an approach based on countable state machines generally results in analyzers which are compact, easy to write and useful. Leveson, Cha and Shimeall [15] state that even in complex, safety-critical systems, it is reasonable to assume "that there are relatively few [hazardous] states... and that these can be determined." In our experience with the steam boiler problem using our state machine-based LFAL, these hazardous states could be described relatively easily; however, more experience is needed to say whether safety properties are usually amenable to this kind of analysis.

Analyzers such as the one we have described above for the general problem would be very complex in Prolog. However, we found that none of the analyzers we wrote used any Prolog more advanced than that taught in a typical programming language survey course; the steam boiler problem, for instance, used only three auxiliary predicate definitions, each consisting solely of Prolog facts.

8 CONCLUSIONS AND FUTURE WORK

Our studies support the conclusion that log file analysis (LFA) for test results evaluation can be used in both unit- and system-level testing, on safety-critical and non-critical specifications, and for testing tasks done in different ways by previous researchers. They also support the claims that LFA integrates well with different approaches to other aspects of testing, such as random test case generation, and that it does not require major changes in software development methodology.

Our studies also indicate that state machine-based LFA, as implemented with the language LFAL, is a powerful, general technique for LFA. The formal, well-defined nature of LFAL analyzers allows us to use them in other ways, such as to simulate the SUT and thus validate the analyzers themselves.

In the future, we wish to apply LFA to significant real-world systems; we believe that embedded systems might serve as a good testbed for our studies. We also wish to use LFAL analyzers for other tasks. It should be possible to define a notion of coverage of an analyzer's transitions by a test suite; given this and prototypes for input and non-input log file lines, it should then be possible to *generate* test cases based on the analyzer. Finally, recognizing that LFAL is not necessarily the final word on log file analysis languages, we would like to study its ease of use and to formulate alternatives to it.

9 ACKNOWLEDGEMENTS

Our thanks to the anonymous referees for their suggestions. This research is supported by Natural Sciences and Research Council of Canada individual grant 203247-98.

REFERENCES

[1] J.-R. Abrial. Steam-boiler control specification problem. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *LNCS*. Springer, October 1996.

[2] J. H. Andrews. Testing using log file analysis: Tools, methods and issues. In *Proceedings of the 1998 International Conference on Automated Software Engineering (ASE'98)*, Honolulu, Hawaii, October 1998.

[3] J. H. Andrews. Theory and practice of log file analysis. Technical Report 524, Department of Com-

puter Science, University of Western Ontario, May 1998.

[4] J. H. Andrews. Testing using log file analysis: An extended example. In *Electronic Commerce Technology Trends: Challenges and Opportunities*. Midrange Computing, 1999. Proceedings of the First International Workshop on Technological Challenges of Electronic Commerce, Markham, Ont., Canada, September 1998.

[5] J. H. Andrews, N. Day, and J. Joyce. Using a formal description technique to model aspects of a global air traffic telecommunications network. In *Formal Description Techniques and Protocol Specification, Testing and Verification: FORTE X / PSTV XVII*, pages 417–432, Osaka, Japan, November 1997. Chapman and Hall.

[6] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, April 1994.

[7] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, July 1984.

[8] M. S. Feather. Rapid application of lightweight formal methods for consistency analyses. *IEEE Transactions on Software Engineering*, 24(11), November 1998.

[9] D. Heimbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[10] D. Hoffman and P. Strooper. Classbench: A framework for automated class testing. *Software Practice and Experience*, 27(5):573–597, May 1997.

[11] ISO (International Organization for Standardization). *ACSE Protocol, ITU-T Rec. X.227 – ISO/IEC 8650-1*. International Organization for Standardization, 2nd edition, 1994.

[12] M. Jazayeri. Component programming – a fresh look at software components. In *Proceedings of the 5th European Software Engineering Conference*, number 989 in LNCS, pages 457–478, Sitges, Spain, September 1995. Springer.

[13] J. Kundu and J. E. Cuny. A scalable, visual interface for debugging with event-based behavioral abstraction. In *Proceedings of New Frontiers on Massively Parallel Processing*, pages 472–479, February 1995.

[14] T. Kunz, J. Black, D. Taylor, and T. Basten. Poet: Target-system-independent visualizations of complex distributed-application executions. *The Computer Journal*, 40(8):499–512, September 1997.

[15] N. G. Leveson, S. S. Cha, and T. J. Shimeall. Safety verification of Ada programs using software fault trees. *IEEE Software*, July 1991.

[16] T. O. O'Malley, D. J. Richardson, and L. K. Dillon. Efficient specification-based oracles for critical systems. In *Proceedings of the California Software Symposium*, 1996.

[17] D. K. Peters and D. L. Parnas. Using test oracles generated from program documentation. In *Proceedings of the International Symposium on Software Testing and Analysis*, 1984.

[18] S. Qiao and H. Zhang. An automatic logfile analyzer for parallel programs. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Vol. III*, pages 1371–1376, Las Vegas, Nevada, USA, 1999.

[19] K.-C. Tai, R. H. Carver, and E. E. Obaid. Debugging concurrent Ada programs by deterministic execution. *IEEE Trans. Softw. Engineering*, 17(1):45–63, January 1991.

[20] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting how badly "good" software can behave. *IEEE Software*, 14(4):73–83, July/August 1997.

[21] G. von Bochmann, R. Dssouli, and J. R. Zhao. Trace analysis for conformance and arbitration testing. *IEEE Transactions on Software Engineering*, 15(11), November 1989.