A Framework for Log File Analysis

James H. Andrews

Dept. of Computer Science
University of Western Ontario
London, Ontario, Canada N6A 5B7

Abstract. Large software systems typically keep log files of events for use in debugging or regression testing. A formal framework is proposed for analyzing these log files to verify that the associated system has the desired behaviour. Taking into account some common properties of log files, a log file analyzer is defined as a set of possibly communicating state machines, which accept a log file if the file causes the machines to move through valid sequences of states. A prototype implementation consistent with the formal definition is described, and examples of its use are given. Suggestions are made as to how such log file analysis could be used in general software testing.

1 Introduction

Large-scale programs or software systems often keep what are called "log files" – continuous reports of events that have occurred during the running of the program. These log files are typically used for debugging, fault location in production software, regression testing, or administrative information. In the words of a software quality assurance specialist:

> Use of log files helps testing by making debugging easier. It allows you to follow the logic of the program, at a high level, without having to run it in debug mode. [Mor98]

However, analysis of these log files is often made difficult by their size and the complexity of their structure. In this paper, we describe a formal framework to assist in log file analysis, including a prototype system, and suggest how use of automatic log file analysis could help in the wider problem of software testing.

Typically, a program's use of log files (we will concentrate on individual programs for simplicity) has the following characteristics.

- The log file is an output file of ASCII text, distinct from the other outputs of the program.
- On startup of the program, the log file is empty or contains whatever was left by some previous run. The system continuously appends lines to the log file, never deleting or changing any previously-written text.
- Each line or group of lines in the file reports on some given event: for instance, an input or output, the parameters or results of a function call, the setting of variables, or the current values of variables.

`temp 21`	`temp 21`
`temp 20`	`malloc 2096`
`temp 19`	`temp 19`
`heater on`	`malloc 2088`
`temp 19`	`malloc 1016`
`temp 20`	`heater on`
`temp 21`	`temp 19`
`heater off`	`temp 21`
`temp 21`	`free 2088`
	`heater off`
	`temp 21`

Fig. 1. Left: a simple log file. Right: a more complex log file.

– The information reported on is the information that programmers feel will be useful in monitoring the system and/or locating faults.

The left-hand side of Figure 1 shows part of a log file from a hypothetical program, one of whose tasks is to control a heater in a room (compare [CG94]). The program gets input every five seconds from a digital thermometer, printing a message of the form `temp` n every time a room temperature of n degrees Celsius is reported. The program is supposed to switch on the heater whenever the temperature drops below 20, and switch it off as soon as the temperature rises to 20 or more again; it reports on these events with log file messages of the form `heater on` or `heater off`.

Though the format of the log file is simple, it is tedious and sometimes difficult to analyze such a log file for errors by human inspection alone. Note, for example, that the log file given does *not* conform to the informal specification given above; the heater is not turned off until at least five seconds after the temperature has returned to 20 or above. It would be useful to have some automated support for analyzing such log files and producing reports of whether the log file conforms to the specification, or in what way it does not. We can take ad hoc measures such as writing Perl scripts, but what we really need is a well-defined framework in which all such log file analysis can be done.

Simple log files with one "thread" of events can be validated easily by conventional grammars; for instance, a context-free grammar of "valid heater monitor log files" can be written. The same cannot be said for more complex log files. The right-hand side of Figure 1 is an example. Here, the system not only controls the temperature in the room, but also does tasks which involve the allocation and deallocation of memory. The system reports on each call to the C functions `malloc` and `free`, to aid in detecting problems like memory leaks. The result is two separate "threads" of log file reports which are arbitrarily interleaved.

It is possible to define a grammar for such log files, but in general the size of the grammar is exponential in the number of separate threads. We need log file analyzers which are conceived and expressed as a set of grammars, or state machines, running in parallel on the reports in the log file. This is what this paper explores.

1.1 The Organization of This Paper

In Section 2, we give formal definitions for the abstract notion of "report trace" and its correspondence to the concrete notion of log file. In Section 3, we define a log file analyzer as a set of parallel state machines with a well-defined semantics, giving examples. In Section 4, we describe a prototype implementation which allows us to define, compile and run log file analyzers on actual log files. In Section 5, we suggest how log file analysis could be applied to the general problem of software testing. Finally, Section 6 discusses related work and Section 7 gives conclusions.

2 Report Traces and Log Files

Definition 2.1 Given a set R of *report elements*, and a distinguished subset $K \subseteq R$ of *keywords*, we define a *report* as a finite sequence of report elements beginning with a keyword.

We assume that each report starts with a keyword because this is a common pattern in log files, and because it will later help us define a first order term corresponding to a report. We write $\mathcal{R}_R$ for the set of reports arising from R.

Definition 2.2 We define a *report trace* as a finite or infinite sequence of reports[1]. For a report trace ρ of finite length, we define $|\rho|$ as the length of the trace; for infinite ρ, we define $|\rho| = \omega$.

We must consider infinite report traces because of the possibility of a process which is intended to run continuously, such as an operating system. Generally, we write a (finite or infinite) sequence of which the first element is e and the rest is η as e, η.

Report traces are the mathematical notion; log files are their real-world manifestation. For clarity, we give a simplified definition of a log file corresponding to a given finite report trace. An infinite report trace would have to be split up across a sequence of log files.

Definition 2.3 A *portrayal function* is an injective function a from report elements to sequences of non-blank, printable ASCII characters, such that for a keyword k, $a(k)$ is a sequence of alphanumeric characters and underscores beginning with a lower case letter.

We extend a portrayal function a to a function from reports to printable ASCII strings by defining $a(e_1, \ldots, e_n)$ as $a(e_1) :: b :: \ldots :: b :: a(e_n)$, where :: is

[1] We choose the term "report trace" because the unadorned term "trace" has been used with a somewhat different meaning in the testing literature [WP94].

the string concatenation operator and b is the ASCII blank. We further extend a to a function from finite report traces to ASCII strings by defining $a(r_0, \ldots, r_n)$ as $a(r_0) :: m :: \ldots :: m :: a(r_n) :: m$, where m is a sequence of characters indicating the end of a line (for instance, ASCII code 13 for Unix systems).

For a given report trace t, we call $a(t)$ the *log file corresponding to* t.

3 Log File Analyzers

The considerations mentioned in the Introduction suggest a formalism for log file analyzers in which object-like state machines are specified, each analyzing some separate aspect of the report trace. A further advantage to using a state-based formalism over a grammar-based formalism is that state machines are understandable to a broad range of software developers without extensive new training.

In this section, we define log file analyzers and their semantics along these lines, and give examples. The definitions in this section use standard notation for describing state machines, e.g. from [Yu97].

3.1 Definitions

Definition 3.1 A *(log file) machine* for report element set R consists of:

1. An *identifying name* $Name$.
2. A countable set Q of *machine states*.
3. A distinguished *initial* state $i \in Q$.
4. A set $F \subseteq Q$ of *final* states.
5. A countable set $N \subseteq \mathcal{R}_R$ of reports which the machine *notices*.
6. A relation $\delta \subseteq Q \times N \times Q$; $\delta(s_1, r, s_2)$ is intended to represent the fact that if the machine is in state s_1 and receives report r, it can make a transition to state s_2.

Note that the set of states is not necessarily finite, and that the transition relation may give more than one transition for any given report.

A *log file analyzer* is defined as a countable set of machines. We will refer to the set of machine states of machine m in a given log file analyzer as Q_m, the δ relation of that machine as δ_m, and so on. Note that even an analyzer containing a countably infinite set of machines is implementable if only a finite number of machines are in non-initial states at any one time.

Informally, a log file analyzer accepts a given report trace if the reports in the trace cause the machines in the analyzer to move through transitions beginning from their initial states, and all ending at final states (if the report trace is finite). The following situations result in a report trace not being accepted:

- A report in the trace which is not noticed by any of the analyzer's machines.
- A report r in the trace such that some m notices r, but such that m cannot make a transition on r.

– A machine which is not in one of its final states after the end of the report trace.

An implementation of a log file analyzer should, of course, inform the user of these situations in its output.

3.2 Formal Semantics

More formally, given a log file analyzer M (i.e. a set M of log file machines), let us call a *state function* for M a function which maps each $m \in M$ to a state in Q_m. A state function captures the "current state" of all the machines in the analyzer at a particular point in the report trace. We use ϕ to stand for a sequence of state functions.

Definition 3.2 We say that a sequence of state functions for M is a *partial accepting sequence* for a report trace (i.e. a sequence of reports) only under the following conditions.

- The sequence of state functions consisting of the single function $curr$ is a partial accepting sequence for the empty report trace, if $curr(m) \in F_m$ for all $m \in M$.
- The sequence $(curr, curr', \phi)$ of state functions is a partial accepting sequence for the sequence (r, ρ) of reports if:
  - $r \in N_m$ for some $m \in M$;
  - For all $m \in M$, either $r \in N_m$ and $\delta_m(curr(m), r, curr'(m))$, or $r \notin N_m$ and $curr'(m) = curr(m)$; and
  - The sequence $(curr', \phi)$ is a partial accepting sequence for ρ.

We say that a non-empty sequence of state functions for M beginning with $curr$ is an *accepting sequence* for a report trace if it is a partial accepting sequence and $curr(m) = i_m$ for all $m \in M$.

We say that a report trace ρ is *non-conforming* with respect to an analyzer M if there is no sequence of state functions for M which is an accepting sequence for ρ. We claim that a report trace ρ is non-conforming in the situations described informally in the last section.

3.3 Examples

The top of Figure 2 shows a log file machine, `heatermonitor`, which accepts correct log files for the heater monitor specification from the Introduction. (It correctly does not accept the log file at the left of Figure 1, because it cannot make a transition from the `should_be_off` state on the log file.) The usual conventions for depicting state machines are used; final states are indicated by double circles, the initial state is indicated by a small arrow, and conditions on transitions appear in square brackets, as in the statecharts formalism [Har87].

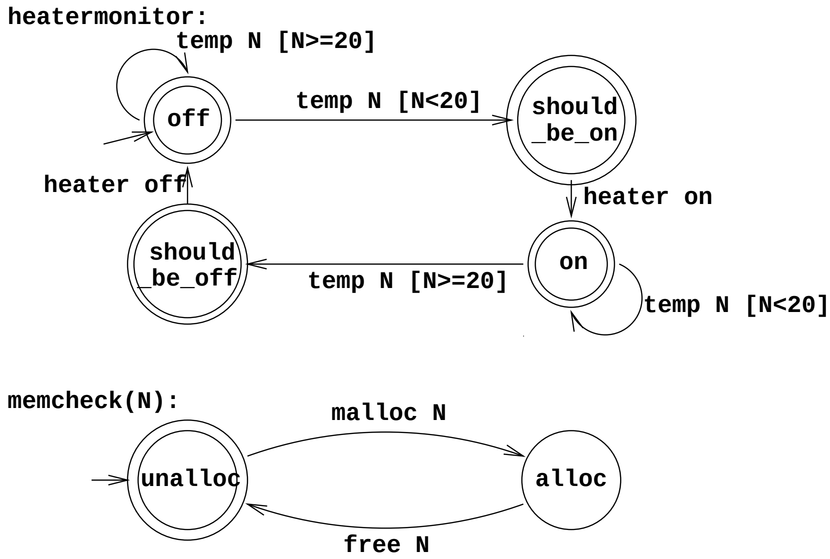

Fig. 2. Top: a log file machine for the heater monitor system. Bottom: a log file machine for memory leak checking.

The bottom of Figure 2 shows a schema for the countably infinite set of log file machines named `memcheck`(n), where n is any integer. Each such machine has only two states, `alloc` (meaning that the integer is a pointer to a block of memory which has been allocated) and `unalloc` (meaning that the corresponding block of memory is currently unallocated). Note that all the states in `heatermonitor` are final, but only the `unalloc` state in the `memcheck` machines is final. This is intended to represent the common requirement that a program free all allocated memory before terminating. The log file analyzer $\{\texttt{heatermonitor}\} \cup \{\texttt{memcheck}(n) | n \geq 0\}$ accepts correct log files for the heater monitor problem which also show that the system has freed all memory it has allocated.

We can extend the definition of log file analyzer to include communication amongst the machines of an analyzer. Space does not permit us to explore this issue, but the resulting formalism is similar to that of Harel's statecharts [Har87], where parallel machines communicate via broadcast communication.

4 Implementation of Log File Analyzers

To implement log file analyzers, we require a system which takes a specification written in a simple but powerful notation and compiles the specified oracle into executable code. We have implemented a prototype of such a system, which we describe here.

```
<analyzer>     ::= {<machine> | <prolog-clause>}*
<machine>      ::= machine <term>; <decl>* end.
<decl>         ::= initial_state <term>;
                 | <trans-clause> [, <trans-clause>]*;
                 | final_state <term>;
<trans-clause> ::= from <term>
                 | on <term>
                 | to <term>
                 | if <prolog-goal>
                 | where <prolog-goal>
```

Fig. 3. The syntax of the log file analyzer specification langauge.

In some ways, the desiderata of simplicity and power are in conflict. However, it seems that the Prolog programming language [O'K90] offers the best base on which to build such a language. It has a simple syntax, we can use Prolog's inherent unification capabilities to match patterns in reports simply and logically, and we can use Prolog's built-in predicates and write our own predicates to express the sometimes complex conditions on transitions. Modern Prolog compilers also allow us to generate efficient executable code.

The syntax of a prototype language is shown in Figure 3. `<term>`, `<prolog-goal>` and `<prolog-clause>` represent terms, goals and clauses of Prolog. The meanings of the constructs are straightforward except for the `<prolog-clause>` construct, which can be used to supply auxiliary clauses used in the definitions of machines, states and transitions. `where` and `if` clauses (which mean the same thing, but are more natural in different contexts) can be used to link definitions to general logical relations.

As an example, consider the log file machines from Figure 2. These are specified in the prototype language in Figure 4. In `heatermonitor`, note the use of the Prolog variable `N` to match the current temperature read from the log file, and the use of Prolog goals like `N >= 20` to represent transition conditions. In `memcheck`, note the parameterization of the machine name with the Prolog variable `Ptr`, which is then matched to the pointers read from the file. In general, we use a straightforward mapping of reports onto Prolog terms (e.g., the term `temp(20)` for the report `temp 20`) to match report patterns.

The prototype implementation consists of two predicates, `translate` and `analyze`. When we place an analyzer definition in a file whose name is *fname*`.ora` (`ora` standing for "oracle"), and call the Prolog goal `translate`(*fname*), the system produces the file *fname*`.pl` containing Prolog clauses which correspond to the definition. When we then load the `.pl` file and call the Prolog goal `translate`(*logfile*), where *logfile* is the name of a log file, we get a "conformance report" either indicating that the log file conforms to the loaded oracle, or describing why it does not.

```
machine heatermonitor;

  initial_state off;

  from off, on temp(N), if (N >= 20), to off;
  from off, on temp(N), if (N < 20),  to should_be_on;
  from should_be_on, on heater(on), to on;
  from on, on temp(N), if (N < 20),  to on;
  from on, on temp(N), if (N >= 20), to should_be_off;
  from should_be_off, on heater(off), to off;

  final_state Any.


machine memcheck(Ptr);

  initial_state unalloc;

  from unalloc, on malloc(Ptr), to alloc;
  from alloc, on free(Ptr), to unalloc;

  final_state unalloc.
```

Fig. 4. The definition of the `heatermonitor` and `memcheck` log file machines in the log file analyzer language.

5 Log Files and Testing

In testing a piece of software, we have two tasks: the selection of the test cases, and the writing of the code to evaluate the results (the "test oracle"). Log file analyzers cannot help with the first task. However, if the informal specification of the system under test states clearly what logging is to be done, then we can trust the log file as a reliable report of what is going on inside that system. An analyzer can therefore act as a test oracle for the system.

We believe that automated log file analysis would be especially useful in situations in which tailored test oracles are unavailable; for instance, in random testing, or for test cases which have been included to achieve some structural coverage criterion.

Log file analysis may also be used as an aid in regression testing. Traditional regression testing is not always appropriate for systems in which time-sensitive information is logged, or in which events can happen in different orders on different runs. This volatile information may be able to be extracted from log files for automatic analysis, leaving only a smaller, non-volatile subset of information to be subjected to regression testing.

6 Related Work

A log file analyzer can be seen as a specification of a test oracle, or of the system under test itself. Hence it is most appropriate to compare this work with work on test oracles.

Several researchers [PP84, RAO92, ORD96] have worked on generating test oracles from formal specifications. However, in their work, formal specifications of the complete system under test have to be prepared in complex graphical or logical notations. Here, an oracle is specified directly using a straightforward state-machine notation which is directly executable.

Customized test oracle specifications have been used in some application areas, such as protocol testing. Bochmann et al. [vBDZ89] report on one scheme in which ESTELLE specifications are translated into Pascal. This paper can be seen as a generalization and formal definition of such techniques.

Finally, Chechik and Gannon [CG94] consider C programs annotated with expressions in a symbolic language related to SCR-style specifications [Hen80]. Some of the annotations are comparable to the write statements needed to produce a log file, and others put conditions on the states of variables; the result is a general technique for formal program verification. However, their work requires information about source code structure obtained from such tools as `gcc`, whereas we are concerned with techniques for analyzing log files produced by programs regardless of source language.

7 Conclusions and Future Work

We have given the theoretical foundations of a framework for analyzing debugging and monitoring logs. We have defined a log file analyzer as a set of state machines, and have described a language for specifying analyzers, and its associated implementation.

Further work includes improvements to the implementation, linking the implementation with test case generation, and a study of the application of log file analysis to larger systems, such as protocol conformance testing and GUI testing. We eventually hope to catalogue the nature of the situations in which log file analysis is applicable, and develop a methodology for its use.

8 Acknowledgments

Thank you to Martin Stanmore, Phil Maker, Jeff Joyce of Hughes Aircraft, Jack Morrison of Sun BOS Product Assurance, and Janette Wong of IBM Toronto for informal discussion of the use of log files in their respective organizations and projects.

The ideas reported on in this paper were formulated while the author was working on the FormalWare project at the University of British Columbia, Hughes Aircraft of Canada Limited Systems Division (HCSD), and MacDonald Dettwiler (MDA). Thanks especially to Richard Yates of MDA for our many discussions concerning these ideas. Thanks also to Mike Donat, Phil Gray, Dan Hoffman, Jeff Joyce, Hanan Lutfiyya, Phil Maker, and Gail Murphy for helpful comments and suggestions.

The FormalWare project is financially supported by the BC Advanced Systems Institute (BCASI), HCSD, and MDA. The author is currently supported by a grant from NSERC and by his startup grant from the Faculty of Science, University of Western Ontario.

References

[CG94] Marsha Chechik and John Gannon. Automatic verification of requirements implementation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 1994.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hen80] K. Heninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.

[Mor98] Jack Morrison. Personal communication. February 1998.

[O'K90] Richard A. O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, Mass., 1990.

[ORD96] T. Owen O'Malley, Debra J. Richardson, and Laura K. Dillon. Efficient specification-based oracles for critical systems. In *Proceedings of the California Software Symposium*, 1996.

[PP84] Dennis K. Peters and David L. Parnas. Using test oracles generated from program documentation. In *Proceedings of the International Symposium on Software Testing and Analysis*, 1984.

[RAO92] 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*, Melbourne, Australia, May 1992.

[vBDZ89] Gregor von Bochmann, Rachida Dssouli, and J. R. Zhao. Trace analysis for conformance and arbitration testing. *IEEE Transactions on Software Engineering*, 15(11), November 1989.

[WP94] Yabo Wang and David Lorge Parnas. Simulating the behavior of software modules by trace rewriting. *IEEE Transactions on Software Engineering*, 20(10), October 1994.

[Yu97] Sheng Yu. Regular languages. In *Handbook of Formal Languages*, Berlin, 1997. Springer.