Deriving State-Based Test Oracles for Conformance Testing

James H. Andrews
Department of Computer Science
University of Western Ontario
London, Ontario, CANADA N6A 5B7

Abstract

We address the problem of how to instrument code to log events for conformance testing purposes, and how to write test oracles that process log files. We specifically consider oracles written in languages based on the state-machine formalism. We describe two processes for systematically deriving logging code and oracles from requirements. The first is a process that we have used and taught, and the second is a more detailed process that we propose to increase the flexibility and traceability of the first process.

1. Introduction

Testing can be made more automated and reliable by the use of *test oracles*, programs that check the output of other programs. In situations where it is infeasible to capture program input/output directly, the software under test (SUT) must write text log files of events. The oracles that process these log files are then referred to as log file analyzers.

In contrast to other dynamic analysis tasks, the SUT instrumentation needed for writing log files is often dependent on the requirements, and thus cannot be added automatically. The same is true for the log file analyzers. We are therefore faced with the problem of how to insert logging instrumentation manually and how to write log file analyzers. This paper addresses these problems.

We have recently been studying a method of log file analysis (LFA) in which oracles are written in a language (LFAL) based on state machines. Our experiences of writing these oracles have led to a recommended process for moving from requirements to oracles and logging instrumentation. We describe this process in Section 3 below, and report on our experiences of using and teaching it. We have noticed some deficiencies in this process, however, and for this workshop we propose a new, more detailed process (Section 4) with the advantages of greater flexibility and traceability.

We do not believe that LFA testing can or should replace all traditional verification and validation activities, but rather that it can act as a complement to traditional methods, enhancing them by enhancing the reliability of test result checking. We take account of this in our proposed processes. Section 5 discusses the potential benefits and problems of LFA testing, especially with reference to the proposed development processes. We begin, however, with a discussion of the background of this paper.

2. Background

Testing involves selecting test cases, running the test cases on the SUT, and checking the results. This paper deals primarily with test result checking. Here we answer the questions of why test oracles are necessary, what advantages we get by running test oracles on log files, and what motivated our decision to create a state-based language for writing log file analyzer test oracles. We also describe the log file analysis language LFAL and discuss work related to this paper.

2.1. Why Test Oracles?

Test oracles [18] are needed in several common situations. The first is when test output is too complex to be checked by a human. This is the case for applications as diverse as communication protocol software and safety-critical control software.

When a test case has been run once and the output has been confirmed as correct, a common practice is to store that output as a "gold file" for regression testing. When the same test case is run on a new version of the software, the new output is checked to see if it is the same as the "gold" output. However, it may be valid that the new output is different. This can be the case, for instance, if the relative timing of distributed events changes slightly.

Finally, modern computer systems are often subjected to random testing, stress testing or load testing. In such cases, input and system behaviour may not be completely predictable, and the volume of output may be high.

In all of the cases mentioned above, test result checking must typically be more complex than a simple equality check of output against stored output. The phrase "test oracle" is usually reserved for programs that do such more complex analysis of output.

2.2. Why Log Files?

Although test oracles are useful, it is often difficult to capture software input and output directly, and difficult to extract relevant information from captured I/O. The use of text log files addresses these difficulties. Text log files are already in wide use in industry, where they are sometimes referred to as "debug files" or "debug logs".

Modern software has many diverse inputs and outputs, including mouse input, graphical output, and network and file I/O. This can be difficult to monitor directly unless the software is launched within a platform-specific OS-level sandbox that intercepts all I/O. If instead the SUT itself logs relevant information to a text file, inputs and outputs of diverse devices can be recorded indirectly.

The volume and complexity of I/O can cause problems for direct I/O capture as well. If only some aspects of correctness are to be checked by a test oracle, it may be that only a small part of the actual I/O of a system is needed for checking. Directly captured I/O, such as TCP/IP output of a program, may need to be re-parsed and re-interpreted to see whether given high-level events to be checked for have happened. If, instead, selected high-level events are logged to a text file, a smaller amount of focused, easily-parsed information is available.

2.3. Why State-Based Log File Analyzers?

We refer to a test oracle that processes only log files as a *log file analyzer*. Log file analyzers can be written in any programming language, but we have come to believe that languages based on the state machine formalism are the best fit for the task, for three main reasons.

First, we observed that log file analyzers often had to store information about past events in order to detect conformance violations when future events happen. This was sometimes information about which of several discrete states the SUT was in, and sometimes more complex information about numeric and string values appearing in log file lines. This suggests a programming language based on state machines, although the need to also store more complex information suggests that something more than simple finite state machines (FSMs) is needed.

Second, we observed that log files often contain many interleaved streams of information about the SUT, but that checking any one requirement typically involved only a subset of this information. This suggests a programming language in which the checking of separate requirements is assigned to separate state machines.

Third, the state machine formalism is widely-known and used in software engineering in other contexts, such as UML state activity diagrams. The extensions needed for storing more complex information and specifying more than one machine are not major.

2.4. LFAL

We have developed a simple domain-specific language called LFAL (Log File Analysis Language) for writing log file analyzers [5]. LFAL is based on the state machine formalism; however, an LFAL analyzer is *not* an FSM, but rather a collection of (infinite-)state machines running in parallel, in which the states can be any first order terms [6], and in which each machine makes transitions from state to state based on first order terms representing complete log file lines. Conditions on source states and triggering log file lines can be placed on transitions.

LFAL analyzers assume that each log file line starts with a keyword and continues with any number of keywords, strings, integers and real numbers, separated by spaces. Each analyzer machine typically checks conformance to one SUT requirement or a group of related requirements, and notices only a subset of the lines in the log file. If an analyzer machine notices a log file line but has no valid transition on it, it reports an error. We write the state machines for a given analyzer so that this happens if and only if the log file being analyzed shows that a requirement has been violated. (An example of an LFAL analyzer will be developed in Section 3.)

2.5. Related Work

The relation of log file analysis to other work in distributed systems debugging, formal specification, test oracle generation, and assertions is explored in detail in [5]. Because our focus here is on the process by which an LFA test oracle is developed from requirements, we compare this work to similar work in development of formal specifications.

Many papers have dealt with the issue of deriving formal specifications from informal requirements. The target formal specification technologies have included tabular notations [16], the Z specification language [11], the SCR specification methodology [12], and statecharts [9]. These works share a general pattern of describing how to move from informal requirements systematically to the notation or technology in question. In this paper, we follow a similar pattern, concentrating on state machines and providing more detail about intermediate steps required. In addition, the artifact that we end with (the log file analyzer) can be viewed as a formal specification, but is also a program that can be compiled and run for the purpose of test result checking.

Work has been done on deriving requirements and oracles from traces produced by automatic instrumentation [8], although as yet the requirements produced are relatively simple. Some criteria have also been stated for inserting logging instrumentation for the specific purpose of performance profiling [14].

Finally, Cleanroom and other processes based on it [15, 17] share with the processes described here the practice of generating a traceable sequence of artifacts of

increasing formality from requirements. In Cleanroom, however, the final artifact is the code itself, and here we are concerned only with a test oracle, which may represent only some of the requirements.

3. Big-Step Process

In this section, we describe a process for deriving logging instrumentation and state-based test oracles from requirements. This process is a distillation of practices that we have followed on previous projects. We call this process the *big-step* process because it involves users taking bigger steps of inference between artifacts than in the small-step process to be described later.

We first describe an example we will use in this paper for expository purposes. We then describe the central artifact of the big-step process, the SPFEs (Situations with Permitted and Forbidden Events), and then go on to describe the process as a whole. We then report on experiences we have had with using and teaching the big-step process, and point out some issues that we have with it.

3.1. Example Software and Requirements

The example software that we will use in this paper for expository purposes is a hypothetical controller for an elevator. We assume that the controller controls both the doors and the motion of the elevator, and we consider the following two requirements.

- R1. The doors are never open when the elevator is in motion.
- R2. Under normal conditions, the elevator door never stays open more than 30 seconds.

The phrase "under normal conditions" in requirement R2 is deliberately vague; we will use it to illustrate how the big-step and small-step processes handle uncertainty in requirements.

3.2. SPFEs

Figure 1 summarizes the big-step process. The central artifact of the process is a list of *Situations with Permitted and Forbidden Events (SPFEs)*. The SPFEs form a link between the language of the requirements and the concepts of state machines.

Each SPFE consists of a situation that the software or its environment may be in, a possibly empty list of events that are permitted in that situation, and a possibly empty list of events that are forbidden in that situation. We use the word "situation" here instead of "state" to avoid confusion with the concept of state-machine states, although we expect that situations in the SPFEs will have a close correspondence with states in the log file analyzer.

SPFEs are best illustrated with some examples. For requirement R1 listed above, a possible set of SPFEs is as follows.

- SPFE1.
  - Situation S1: The elevator door is open.
  - Permitted event P1.1: The door closes.
  - Forbidden event F1.2: The elevator starts moving.
- SPFE2.
  - Situation S2: The elevator is moving.
  - Permitted event P2.1: The elevator stops moving.
  - Forbidden event F2.2: The door opens.
- SPFE3.
  - Situation S3: The elevator is stopped and the door is closed.
  - Permitted event P3.1: The elevator starts moving.
  - Permitted event P3.2: The door opens.

Requirement R2 listed above can be captured by a single SPFE.

- SPFE4.
  - Situation S4: The door last opened at time T_1 and is still open.
  - Permitted event P4.1: The door closes at time T_2, where $T_2 - T_1 \leq 30$.
  - Forbidden event P4.2: The door closes at time T_2, where $T_2 - T_1 > 30$.

Note that in writing the above SPFE, we have implicitly assumed that we are going to use the log file analyzer only when we test the SUT under the "normal conditions" mentioned in requirement R2. This assumption is made more explicit and traceable under the more detailed process described in Section 4.

3.3. Process

The flow of information and the sequence of artifacts produced in the big-step process follows the general pattern of Figure 1. The steps of the process are:

1. From the requirements of the system, derive the SPFEs. The SPFEs should not contradict the requirements, although they can represent only a subset of the requirements if the LFA testing is not intended to cover all requirements.
2. Based on the SPFEs, write a logging policy (LP). The logging policy should specify what events the source code should log and how it should log them. This should include:

Abbr	Expansion
Reqs	Requirements
SPFEs	Situations with Permitted and Forbidden Events
LP	Logging Policy
LFAP	Log File Analyzer Program

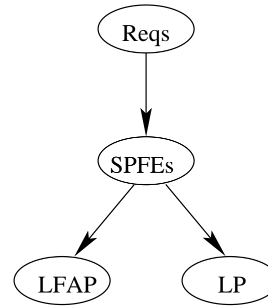

Figure 1. Big-step process summary. Left: Artifact abbreviations and their expansions. Right: Information flow. An arrow indicates that the arrow source is a primary source of information for the arrow destination.

(a) All events that will allow us to determine, for each SPFE, whether we are in the described situation;

(b) All events that are mentioned as "permitted" or "forbidden" in any SPFE.

3. Instrument the source code consistent with the LP.

4. Based on the SPFEs and the logging instrumentation, write and validate the log file analyzer.

In step 1, we do not require every possible event relevant to an SPFE to be listed as either permitted or forbidden, although such a requirement would make the SPFEs more precise. We do this in order to make this step less constrained. If there are particular events that are clearly permitted or clearly forbidden in given situations, then they can be listed as such in the SPFEs, in order to give guidance during steps 2-4. The other events can temporarily be left with their permittedness undefined. Whether they are permitted or forbidden can then be decided when the analyzer is written in step 4.

We now expand upon steps 2, 3 and 4 above. Step 2 requires us to write a logging policy, and step 3 requires us to implement this policy. In our example, if we were considering only SPFE2, we would only need to determine whether the elevator is currently moving or not (S2), whether the elevator has stopped moving (P2.1), and whether the door has opened (F2.2). A sufficient set of events to be logged would be:

- Events in which the elevator starts moving. (Needed to determine whether we are in S2.)
- Events in which the elevator stops moving. (Needed to determine whether we are in S2, and whether P2.1 has occurred.)
- Events in which the door opens. (Needed to determine whether F2.2 has occurred.)

However, for all of SPFE1-4, more events are needed and more data is needed about events in which the door opens. The following set of events is sufficient:

- Events in which the elevator starts moving. (Needed for S2, S3, F1.2, P3.1.)
- Events in which the elevator stops moving. (Needed for S2, S3, P2.1.)
- Events in which the door opens, together with the time the door opens. (Needed for S1, S3, S4, F2.2, P3.2.)
- Events in which the door closes, together with the time the door closes. (Needed for S1, S3, P1.1, P4.1, P4.2.)

The logging policy should state explicitly what format the given events should be logged in. For example, we could say that the SUT must record the above events by logging lines of the form `start_move`, `stop`, `door_open` *t*, and `door_close` *t* respectively, where *t* is a timestamp.

Step 4 of the big-step process requires us to write a log file analyzer. Generally, each analyzer machine is likely to correspond to one or a group of SPFEs, with the Situations corresponding to states of the machine. For example, it is possible to check all of SPFE1-3 with a single LFAL state machine:

```
machine door_safety;
  initial_state closed_stopped;
  from closed_stopped, on start_move,
    to moving;
  from moving, on stop,
    to closed_stopped;
  from closed_stopped, on door_open(T),
    to open;
  from open, on door_close(T),
    to closed_stopped;
  final_state Any.
```

The three states of this machine (`open`, `moving`, and `closed_stopped`) correspond to the Situations in SPFE1-3 respectively. As an example of the treatment of forbidden events, there is no transition on the log file line `door_close` *t* from the state `open`, because that log

file line corresponds to an event which is Forbidden by SPFE1; however, there is a transition on the log file line `door_open` *t*, which is a Permitted event. It is possible to check SPFE4 with one additional LFAL state machine:

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

Note that the state pattern `open(T1)` contains not only the information that the door is open, but also the time at which the door was opened as a parameter of the state.

3.4. Experiences

We used early, informal versions of the big-step process for several previous projects [4, 2, 13]. The largest analyzer developed and validated was based on the first eight pages of Abrial's Steam Boiler specification [1], and was 333 net lines of LFAL code, containing 19 state machines having a total of 141 transitions. Based on our experiences with those projects, we codified the process for training purposes.

In [10], we reported on an exploratory study of learning and initial use of LFA testing and LFAL. The process for developing log file analyzers which we taught to the learners in this study was the big-step process. We did not monitor how closely they actually followed the process in the study. However, we found that they performed well at the tasks of creating a logging policy, instrumenting code with logging instrumentation, and writing an LFAL log file analyzer.

3.5. Issues

Several issues have come up during our use and teaching of the big-step process that we feel are not handled well by the process. One issue is that it contains no recommendations for what to do when not all the requirements will be checked by LFA testing. We expect that LFA testing will often be used as a complement to traditional testing; that is, not to test all the requirements of a system using LFA, but only a subset, and those only under given conditions. With the big-step process, the decisions made in this regard (e.g. the decision to test the example system only "under normal conditions") are nowhere explicit.

A related issue is lack of documentation and traceability. There is a lack of documentation of the requirements to be checked, the conditions under which LFA testing will take place, and why given events were chosen to be logged. This makes it more difficult to validate big-step process artifacts, e.g. in artifact inspection sessions [7].

Finally, we have noted that in some cases it is difficult to find places in the SUT code at which to log the events needed for the SPFEs. In some cases, the events needed are in a sense "abstract", not able to be matched directly to locations in the code but rather indicating a general pattern of things that have occurred. In these cases we have found that the logging policy is difficult to match up with the SPFEs, the instrumentation in the source code, or both.

These issues are not major if the number of events to be logged or the number of requirements to be checked is small, or in cases of non-critical software. We expect them to be more significant as the number of events or requirements grows and the SUT is more safety-critical.

4. Small-Step Methodology

In this section we propose a process for deriving state machine oracles and logging instrumentation that avoids some of the pitfalls of the big-step process. Because it takes smaller steps toward the production of the final artifacts, we refer to it in this paper as the *small-step* process. It is derived from practices that we have used and observed in the past for making the big-step process more manageable.

We expect the small-step process to be more useful the more the following conditions hold in the development project in question.

- The project is safety-critical, and we want high assurance that requirements are reflected in the test oracle.
- Only a subset of the requirements is to be tested using LFA, and/or LFA testing is only to be done under certain conditions.
- The number of events we need to log is high.
- The requirements to be checked are complex.
- The developers are unfamiliar with using log file analysis for conformance testing, and want to follow a detailed process so that they can learn it.

Figure 2 summarizes the small-step process. The process is similar to the big-step process, but with some important additions for the sake of traceability and generality:

- *RCs (Requirements to be Checked)*: the subset of the original requirements that is to be checked with the LFA testing.
- *CAs (Checking Assumptions)*: the assumptions under which the LFA testing will take place.
- *AEs (Abstract Events)*: all the events that will allow us to tell whether or not we are in the Situations in the SPFEs, and whether or not the Permitted and Forbidden events of the SPFEs have happened.

Abbr	Expansion
RCs	Requirements to be Checked
CAs	Checking Assumptions
SPFEs	Situations with Permitted and Forbidden Events
AEs	Abstract Events
CEs	Concrete Events
LP	Logging Policy
LFAP	Log File Analyzer Program

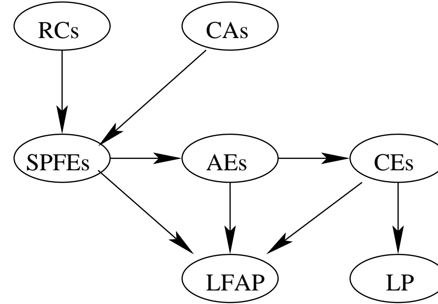

Figure 2. Small-step process summary. Left: Artifact abbreviations and their expansions. Right: Information flow. An arrow indicates that the arrow source is a primary source of information for the arrow destination.

- *CEs (Concrete Events)*: events that are expected to be easily identifiable and loggable at the source code level, that allow us to tell whether or not any of the Abstract Events (AEs) have taken place.

4.1. Small-Step Process Artifacts

We now discuss the major differences between the artifacts generated by the small-step and big-step processes.

RCs (Requirements to be Checked) and CAs (Checking Assumptions). The RCs and the CAs together allow us to inspect the SPFEs for completeness, not just correctness, in cases where LFA testing is used for only some requirements and/or under some testing conditions. The SPFEs should restate all the information in the RCs, given the CAs.

For the elevator controller example, we might state as checking assumptions that we will perform LFA testing of the elevator controller only under the condition that the door is never obstructed by some object, and that this is what "under normal conditions" means in requirement R2. The SPFEs chosen above under the big-step process can then be justified by pointing out that they encompass all the requirements to be checked, given the checking assumptions we have made.

The small-step process is therefore more traceable than the big-step process. Each RC should correspond to one or more SPFEs and vice-versa, and missing details or subcases not considered in the SPFEs should be able to be justified by appealing to the CAs.

AEs (Abstract Events), CEs (Concrete Events), and LP (Logging Policy). AEs are "requirements-level" events, in the sense that their descriptions should be close to the language and terminology of the SPFEs. In contrast, the CEs are "code-level" events, in the sense that it should be possible to match each with one or more locations in the SUT source code at which they should be logged. The AEs and CEs are not necessarily disjoint; there may be some events that are both sufficiently abstract to give information about the SPFEs and also sufficiently concrete to be logged. Such events would be listed as both AEs and CEs.

For our example, we might consider the doors to be open the moment that a command is sent to the door lock actuator to release the door from the closed position. In this case it would be appropriate to take "door open at time t" as an AE, and take "door lock actuator release command at time t" as the CE that will be logged.

Because the CEs are explicitly documented in the small-step process, the logging policy (LP) in the small-step process needs only to document how and when each CE will be logged and the format in which it will be logged. For example, there may be a paragraph in the LP which states:

- When a RELEASE command is sent to the door lock actuator, the line `door_lock_release_cmd` t must be logged, where t is the current timestamp in seconds obtained from `gettimeofday()`.

Again, the small-step process has the advantage of greater traceability over the big-step process. Each paragraph in the LP has to do with one or more CEs; each CE is chosen to allow us to tell whether one or more of the AEs has taken place; and each AE is chosen to allow us to check specific situations and events in the SPFEs.

LFAP (Log File Analyzer Program). The analyzer written in the small-step process is similar to that written in the big-step process, with one exception. We expect the analyzer to contain state machines of two main types: event transducers and SPFE checkers.

Event transducers are relatively simple machines that read CEs from the log file and "convert" them into AEs. In LFAL, we can write state machines that do this by specifying on transitions that the machine "sends" messages standing for the AEs to all the other machines. For instance, a simple single-state machine that translates the "door lock release command" CE into the "door open" AE might be written as follows in LFAL:

```
machine transducer_door_open;
  initial_state null;
  from null, on door_lock_release_cmd(T1),
    to null, sending door_open(T1);
  final_state null.
```

SPFE checkers are machines that check one or more of the requirements, as expressed in the SPFEs. The LFAL state machines described for the big-step process in Section 3.3 are SPFE checkers, and the machines given there would still be appropriate under the small-step process.

4.2. Process

The small-step process starts by generating each of the artifacts mentioned in the table at the right of Figure 2 in that order. Any or all of these steps may cause change requests to be filed against previously-generated artifacts. In particular, we expect the creation of the SPFEs to cause change requests against the RCs and CAs, the creation of the AEs and CEs to cause change requests against the SPFEs, and the creation of the LP and LFAP to cause change requests against all of the SPFEs, AEs, and CEs.

We believe that a thorough development process should include document inspections [7]. Each artifact should be inspected for consistency with upstream artifacts when first created. If numerous change requests have been made against artifacts since the last time they were inspected, a maximally careful process would conduct further inspections to ensure that all artifacts remain consistent with each other. To inspect and validate the LFAP, we have developed tools and techniques for animating the state machines in the analyzer, to help inspectors check that the analyzer is not too strict or lenient [3].

5. Potential Benefits and Problems

Every new software engineering technology brings with it potential benefits and potential problems. Here we enumerate what we see as the most important of both for LFA testing.

5.1. Potential Benefits

Improved reliability. LFA testing assists direct, automatic confirmation of the conformance of software to requirements. This automatic test result checking can be done in complement to traditional human checking and regression checking of test results. The log file analyzer can be written in a language such as LFAL designed for the purpose, and as such can take an abstract, concise view of the requirements.

Note also that, in contrast to other formal methods-based techniques for increasing reliability, no assumptions are made as to the development language or platform of the SUT. As long as the SUT can write a text file, and that file can be transported to a platform on which it can be analyzed, LFA can be used in testing.

Flexibility. LFA testing can be used to check for either a small or a large number of requirements. An elevator controller, for instance, may be a large and complex system having many requirements; however, if only the two example requirements are to be checked by LFA testing, we have shown (Sections 3 and 4) that this can be done by logging only four classes of events and writing as little as 22 lines of LFAL code.

A developer or development team that is unsure about whether LFA testing is appropriate can therefore use it for part of a testing effort, and later evaluate to what extent they wish to use it in the future.

Scalability. Our experience suggests that the number of SPFEs generated by either the big-step or the small-step process is linear in the number of requirements to be checked, and that the size of the final log file analyzer program is linear in the number of SPFEs. We therefore believe that as the number of requirements to be checked by LFA testing increases, the amount of effort to implement LFA testing scales up reasonably.

Traceability. Particularly when using the small-step process proposed here, the log file analyzer program (LFAP) and the logging instrumentation can be traced back directly to the requirements. This traceability can aid in achieving high assurance, in situations in which it is required, for instance in safety-critical projects.

5.2. Potential Problems

False negatives and positives. When an analyzer announces that a log file indicates a fault in the SUT, the cause might actually be a fault in the analyzer. This "false negative" is the equivalent of an invalid expected result in regression testing. A more serious problem is that a faulty analyzer could give a "false positive" by passing a log file that does actually indicate an error.

To address this problem, we have advocated the use of inspections for log file analyzers, and have provided tools for validating them [3]. Of course, the use of these techniques and tools increases the weight of the development process (see below).

Instrumentation maintenance. The logging instrumentation added to the SUT for LFA testing is extra code that must be maintained. Changes to the code for other maintenance reasons must take account of the logging instrumentation.

This problem exists already in the large body of software that generates log files. However, existing log files are often used only for debugging, and are not an intimate part of the test result checking effort; the logging instrumentation therefore does not have to be kept in step with the rest of the code as strictly as with LFA testing.

Process "weight". The benefits that we get out of LFA testing increase as we follow one of the processes outlined in this paper more closely. However, naturally this makes the process more heavyweight and brings with it problems such as developer frustration and process overhead. The more heavyweight the LFA testing process, the more safety-critical the project would have to be to justify it.

6. Conclusions

Whether the benefits of LFA testing outweigh the problems, and under what conditions, are subjects for future research. One of the purposes of this paper is to propose processes for future study that can enhance the benefits and address the problems. We plan to continue by applying the new small-step process to publically-available and industrial requirements, measuring the amount of time taken, the size of the resulting artifacts, and the effectiveness of the resulting testing.

7. Acknowledgements

This research has been generously supported by the Natural Science and Engineering Research Council of Canada (NSERC) and Nokia Corporation. This paper was written while the author was visiting Dan Hoffman and the Computer Science Department at the University of Victoria, to whom we are grateful for the use of their facilities.

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] X. An. Steam-boiler control system – simulation and testing using log file analysis. Master's thesis, Department of Computer Science, University of Western Ontario, London, Ontario, Canada, September 2000.

[3] J. H. Andrews, R. Fu, and V. D. Liu. Adding value to formal test oracles. In *Proceedings of the 17th Annual International Conference on Automated Software Engineering (ASE 2002)*, pages 275–278, Edinburgh, Scotland, September 2002.

[4] J. H. Andrews and Y. Zhang. Broad-spectrum studies of log file analysis. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 105–114, Limerick, Ireland, June 2000.

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

[6] G. S. Boolos and R. C. Jeffrey. *Computability and Logic*. Cambridge University Press, Cambridge, UK, 2 edition, 1980.

[7] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, July 1976.

[8] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 25th International Conference on Software Engineering*, pages 60–73, Portland, Oregon, May 2003.

[9] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, January 2001.

[10] G. Huang and J. H. Andrews. Learning and initial use of a software testing technology: An exploratory study. In *Proceedings of the 8th International Conference on Empirical Assessment in Software Engineering (EASE 2004)*, Edinburgh, Scotland, May 2004. To appear.

[11] J. Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.

[12] J. Kirby, Jr., M. Archer, and C. Heitmeyer. SCR: A practical approach to building a high assurance COMSEC system. In *15th Annual Computer Security Applications Conference*, pages 109–118, Phoenix, Arizona, December 1999.

[13] V. D. Liu. Conformance analysis of communications protocol software using log files. Master's thesis, Department of Computer Science, University of Western Ontario, London, Ontario, Canada, April 2002.

[14] E. Metz and R. Lencevicius. Efficient instrumentation for performance profiling. In *Proceedings of the Workshop on Dynamic Analysis, ICSE 2003*, pages 10–12, Portland, Oregon, May 2003.

[15] H. D. Mills, M. Dyer, and R. Linger. Cleanroom software engineering. *IEEE Software*, 4(5):19–24, September 1987.

[16] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12:251–257, February 1986.

[17] B. D. Tackett and B. V. Doren. Process control for error-free software: A software success story. *IEEE Software*, 16(3):24–29, May/June 1999.

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