Log-Based Testing

Alexander Elyasov
Dept. of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands
a.elyasov@uu.nl

***Abstract*—This thesis presents an ongoing research on using logs for software testing. We propose a complex and generic logging and diagnosis framework, that can be efficiently used for continuous testing of future Internet applications. To simplify the diagnosis of logs we suggest to reduce its size by means of rewriting.**

***Keywords*-log file analysis; instrumentation; rewriting**

I. Introduction

A. Testing and Logging of Internet Applications

Internet Applications (IAs) obviously play an important role in our life. People's requirements on reliability and safety of IAs are changing, and these qualities become more critical. Modern IAs often represent a conglomeration of independently functioning services, delegating certain tasks to separate modules as much as possible. They are composed of third party components deployed independently. These components are provided without source code and in many cases with only partial documentation. Such components may have unpredictable behavior and are often error prone. Even if an application is well documented and its source code is available, we can not firmly guarantee the absence of bugs. Therefore, testing of IAs is very challenging.

For decades software logs were extensively used for various kinds of analyses by different groups of users. In particular, many software testing techniques, such as dynamic model inference, classification tree generation, oracle learning, can exploit logs as a source of information about the behavior of an application. The FITTEST project [1] proposes to develop a continuous testing framework based on aforementioned testing techniques with logs in a central position. This framework aims to alleviate the problems of testing IAs.

B. Research Problem

The goal of my PhD research is to develop an efficient, expressive and generic logging and diagnosis framework that can be used to support IAs testing. Suppose, we have an application that needs to be extended with logging. To approach the above goal it is necessary to answer the following questions:

- Which information should be logged?
- How to structure the information in logs?
- Where to log in the application?
- How to add logging logic to the application?

Although some of these questions may seem to be technical, they lead to interesting research problems. For instance, we can decide to log the application's variables or events or to log the application's control flows. Since storage is limited and IO operations are expensive, we should find a compromise between the richness of log content and application performance. These questions and the logging solution we propose is discussed in Section III-A.

As we mentioned at the beginning of this section, our framework should also diagnose the application. That is, it should be able to identify certain application behavior as "suspicious" to predict potential errors. For that we need to have criteria of suspiciousness. The suspicious behavior then can be reported along with the log that contains it for subsequent analysis.

But, the reported log itself can consist of thousands entries. To simplify its analysis, the logical step is try to reduce the log while keeping it somehow "equivalent". The *log reduction* problem is discussed in Section III-B. Other issues in logging such as performance degradation produced by superfluous IO operations, storage consumption and privacy are also within the scope of this thesis.

II. Related Work

For many applications, the production of logs is still commonly required practice. However, despite the long history of exploiting logs for software development, there is no systematic research on the application of logs to software testing. Most of the work is either on new logging libraries proposals or on log analysis techniques. But none of them have been trying to approach the problem from both sides and develop a unified log-based analysis framework which can be instantiated for various classes of applications and different kinds of analyses.

Valdman [2] notice that the log based analysis approach has not been deeply studied and a universal log analysis tool would be practically useful. He discusses some characteristics of such a tool, but did not provide a concrete solution. Let us revisit the questions posed in Section I-B and present existing trends in logging and log based analysis.

A. Logging Approaches

The simplest way, and perhaps also the most commonly used, is manual insertion of *print*-like statements which write their arguments into a log file. This idea lies behind many logging libraries, for instance, log4j [3]. Though these libraries provide plenty advanced features, they still require manual insertion of logging statements. As a result, it makes this approach very tedious and clutters the source code, and consequently may cause the programmer to make more errors.

The second approach is based on aspect-oriented programming [4]. Logging statements are written as a separate "aspect", which is then weaved into the application. The separation of the logging concern makes this solution modular and allows to avoid multiple insertion of logging statements everywhere given function is called. The corresponding instrumentation can be done at either source or byte code level. While implementing each of these approaches we need to make sure that they do not change the intended behavior.

It worth to mention here that the definition of logging framework also requires to define a logging format. Such formats as Syslog [5] or CLF have already become logging standards in some application fields. But they are not generic by default to be used for various log-based analyses.

B. Applications in Testing

From analysis point of view, logs contain information about application behavior — real behavior of the SUT. This behavior should be validated against the application's specification. However, this specification is often only partly defined or completely absent; to some extent logs can take the role of a specification.

The term *log based analysis* was first introduced by Andrews in his seminal paper [6]. Andrews suggested to check logs against special oracles called *Log File Analyzers* (LFA), which essentially are finite state machines annotated with application events on the edges. Other examples of works in oracle specification language are [7], [8]. For instance, in [8] Barringer et al. reported on successful application of their tool LogScope for the testing of highly critical applications, which often employ logging. Logs are monitored online and a violation of the specification is immediately reported. The suggested oracle specification language is expressive enough to specify temporal properties over event sequences.

Oracle specifications by itself can also be inferred from logs if an application has a rich enough execution history. The Daikon family of tools [9], [10] use a set of common patterns of invariants over program data (variables, methods, fields) to generate oracles from application logs. Dynamic approaches like Daikon can potentially be strengthened by exploiting information provided by static analysis as in e.g. [11]. An alternative approach was taken by Mariani et al. in [12], where they inferred application models from logs and use it as oracles.

III. Solution

In this section we propose our solution of logging claimed to be developed in the Section I-B. We cover all four questions except log format, which can be found in [13]. Also we discuss the log reduction problem related to the diagnosis part of our framework.

A. Log Generation

Abstractly, logging can be considered as the process of recording application responses as the sequence of events while a user is interacting with the application. We classify application events as follows: *high level events* and *low level events*. Events from the first category help us to understand how an application is used. GUI events such as button click, field input and so on represent an events from this category. Events such as a function (method) call, throwing an exception, and a control flow transition fall into the second category of events. The final may contain both event types. Logging of high level events can trigger logging of multiple low level events.

To make this approach more attractive for developers and relieve them from exhaustive inserting of logging statements, we propose to separate logging from the development process. That is, an original application without logging facilities is instrumented, such that later on it will be able to produce logs. This requires identification of *logging points*, that is places in the program where logging statements need to be inserted. There are many predefined logging statements corresponding to different program constructions branches, loops, function call etc., grouped together as a library. The instrumentation either directly injects the log statement at a point, or else provides a way to later on execute the right logging statement at the right moment.

In addition to server-side logging that is rather common for IA, we strengthen our approach by logging of clients, which have become rich and complex. We employ two kinds of instrumentation. The first one is static instrumentation. It parses the application, directly injects logging statements and, finally, generates a new application, which has to be deployed for production.

Static byte-code instrumentation works well for facilitating the logging of low level events, but it does not work well for the logging of high level events. Quite often the GUI of an application is dynamically constructed, which means that it is in principle not possible to predict statically which events should be considered as high level events. So, we do the needed instrumentation dynamically. This means that at the run-time we scan the application GUI tree, and decide for each GUI component in the tree whether we want to log the component, and if so what kind of GUI events on the component should be logged.

The general structure of our framework is shown on Figure 1. The *automation framework* provides a bunch of *automation delegates* specifying which application events

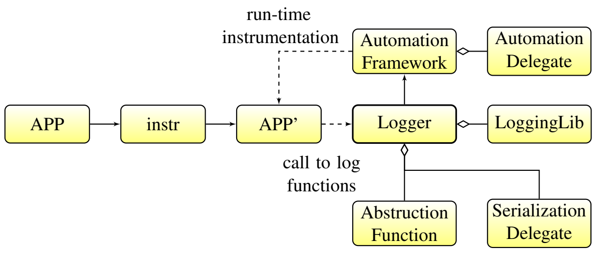

Figure 1. Logging Framework Architecture

should be logged. A set of *serialization delegates* tells the application how to serialize certain objects; an *abstraction function* projects the application state at a logging point. A set of functions explicitly responsible for log production is provided by *LoggingLib* library. The class *Logger* itself only contains configuration parameters, such as logging level and severity.

We have provided a concrete implementation of the above framework [13] together with the static instrumenter [14] for ActionScript applications. An analogous approach can be applied to other client programming languages e.g. JavaScript.

B. Log Reduction

In addition to generation of logs, our logging framework should also be able to provide diagnostic information about application behavior. Suppose, that we have developed a diagnostic module that can recognize suspicious behavior from logs and report the corresponding trace (piece of logs). This trace should be examined testers in order to decide whether it actually signalize real error or not.

According to our logging model presented in Section III-A, a log is an alternation of events and states. Assume, that the log L was reported as suspicious, for example, because it came to a suspicious state (or triggered a suspicious event). Because of continuous nature of our logging approach the reported log L can be very long; this makes it hard to analyze. We propose to reduce the log by rewriting it. The information contained in previously collected logs, called *history logs*, is used for inferring the rewrite rules that will guide our log rewriting process at a later stage. As a result of the reduction, we will replace the log L with a shorter one, which reproduces the same suspicious states (events) and can accordingly be analyzed instead of L.

The number of all possible rewrite rules that can be constructed from the finite set of events is potentially infinite. So we can limit the maximum length of the rewrite rule by a certain constant. In this paper we propose a quantitative log-based method of *micro-rewrite rule patterns* inference, where rules have the length at most four ($\tau \sim \sigma$, where $|\tau|, |\sigma| \leq 2$). To further limit the number of rewrite rules, we suggest to consider only the specific rewrite rule patterns

Table I
REWRIE RULES PATTERNS

Name	Pattern	Example
SKIP	$[e(\bar{p})] \sim []$	zoom();
ABSORB	$[d(\bar{q}); e(\bar{p})] \sim [e(\bar{p})]$	add(item);clear();
IDEM	$[e(\bar{q}); e(\bar{p})] \sim [e(\bar{p})]$	update(x);
IGNPAR	$[e(\bar{p}) \sim e(\bar{q})]$	setFont(font);
COM	$[e(\bar{p}); d(\bar{q})] \sim [d(\bar{q}); e(\bar{p})]$	setA(x);setB(y);

presented in Table I. These patterns are quite advantageous, among others, because of their simplicity and prevalence in applications. Of course, this approach is neither sound nor complete due to inherent deficiency of log-based approach. E.g. if we only log abstract states instead of real ones, we may produce false positives. If we fail to get enough evidence of a certain rewrite rule, we reject that rule even if it happens to be actually valid. To remedy the unsoundness, we propose to let a human expert inspect the inferred rules before applying them. That allows us to discard the wrong ones.

In order to reduce the number rewrite rules accepted by the algorithm because of the witnesses happened by coincident, we may assign a confidence level to the rewrite rule patterns. All rewrite rules whose confidence level is higher will be accepted.

As a rewriting engine we exploit the rewriting capabilities of the HOL proof assistant [15]. The log L and the used rewrite rules are converted to HOL terms, so that we can exploit HOL rewriting tactics. In the examples in Table I, the rewrite rules are symmetric. The resulting rewriting process is thus potentially nonterminating. Therefore we have to choose in which direction to apply these rules. But this strategy can end up with a log which is not shorter. Converting the original rewrite system to a monotonic one without a loop would guarantee the termination and reduction of the log rewriting. Up to now, this question has not been carefully investigated yet.

The whole structure of our framework is depicted on Figure 2. The proposed inference algorithm is independent over rewrite rule patterns so we can easily extend the list of suggested patterns. The algorithm is also independent of the used technology to build the application. Instead of HOL we could use any other system that provides facilities for term rewriting. Future research in this direction is possible. The suggested form of rewrite rule patterns are easy to check by human expert. In addition, the algorithm can be tuned by adding confidence levels and various abstract state comparison functions.

C. Contributions

Current contributions are: 1) the event-driven logging framework together with the ActionScript byte-code instrumentation tool have been developed; 2) the tools above have

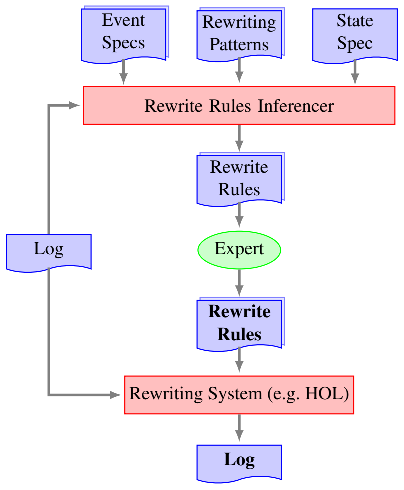

Figure 2. Log Reduction Framework

been used to provide logging facilities for some example application; 3) the log reduction framework has been implemented.

Future work. This is the first year of my PhD research, so many targets have not been explored. In particular, the following questions are still remain to be considered: log-based test coverage tracking, log-based diagnosis of errors and suspicious behavior, inferring oracles from logs, privacy of logged data, scalability of logging (e.g. performance and compression), continuous and self-adjusting log generation process.

Evaluation (future). Our logging approach targets IAs written in popular languages: ActionScript, PHP, and Java. The current case study is Habbo Hotel, which is a large Flash-based Internet application, written in ActionScript.

Acknowledgment

This work is part of the EU-funded FITTEST project No. 257574. Special gratitude to my colleagues W. Prasetya, A. Middelkoop and J. Hage for their significant contribution to this thesis.

References

[1] T. Vos, P. Tonella, J. Wegener, M. Harman, W. Prasetya, E. Puoskari, and Y. Nir-Buchbinder, "Future internet testing with fittest," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on.* IEEE, 2011, pp. 355–358.

[2] J. Valdman, "Log file analysis," 2001.

[3] C. Gulcu, "Log4j delivers control over logging," *Java World*, 2000.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97 — Object-Oriented Programming*, ser. Lecture Notes in Computer Science, M. Aksit and S. Matsuoka, Eds., 1997, vol. 1241, pp. 220–242.

[5] R. Gerhards, "The syslog protocol," RFC 5424, March, Tech. Rep., 2009.

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

[7] D. Tu, R. Chen, Z. Du, and Y. Liu, "A Method of Log File Analysis for Test Oracle," in *Scalable Computing and Communications; Eighth International Conference on Embedded Computing, 2009. SCALCOM-EMBEDDEDCOM'09. International Conference on.* IEEE, 2009, pp. 351–354.

[8] H. Barringer, A. Groce, K. Havelund, and M. Smith, "Formal analysis of log files," *AIAA Journal of Aerospace Computing, Information and Communications*, 2010.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, pp. 213–224, 2001.

[10] M. Boshernitsan, R. Doong, and A. Savoia, "From daikon to agitator: lessons and challenges in building a commercial tool for developer testing," 2006, pp. 169–180.

[11] J. W. Nimmer and M. D. Ernst, "Automatic generation of program specifications," 2002, pp. 232–242.

[12] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on.* IEEE, 2008, pp. 117–126.

[13] I. S. W. B. Prasetya, A. Middelkoop, A. Elyasov, and J. Hage, "D6.1: FITTEST Logging Approach, Project no. 257574, FITTEST Future Internet Testing," 2011.

[14] A. Middelkoop, A. Elyasov, and W. Prasetya, "Functional instrumentation of actionscript programs with asil," accepted for publication in Proceedings of IFL 2011.

[15] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: a theorem proving environment for higher order logic*. New York, NY, USA: Cambridge University Press, 1993.