What Logs should you Look when an Application Fails? Insights from an Industrial Case Study

Marcello Cinque, Domenico Cotroneo, Raffaele Della Corte, Antonio Pecchia
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione
Universitá degli Studi di Napoli Federico II
Via Claudio 21, 80125, Naples, Italy
Email: {macinque, cotroneo, raffaele.dellacorte2, antonio.pecchia}@unina.it

Abstract—**Event logs are the first place where to find useful information about application failures. Event logs are available at different system levels, such as application, middleware and operating system. In this paper we analyze the failure reporting capability of event logs collected at different levels of an industrial system in the Air Traffic Control (ATC) domain. The study is based on a data set of 3,159 failures induced in the system by means of software fault injection. Results indicate that the reporting ability of event logs collected at a given level is strongly affected by the type of failure observed at runtime. For example, even if operating system logs catch almost all application crashes, they are strongly ineffective in face of silent and erratic failures in the considered system.**

Keywords - event log; fault injection; failure analysis; middleware; air traffic control.

I. Introduction

Event logs play an important role in data-driven failure analysis because they contain a large amount of textual information about system behavior under real workload conditions [1], [2]. Understanding system failures through event logs is an important activity in many types of systems, to avoid severe consequences, such data and economical loss, damage to the environment. Event logs have been used over the past decades for either *post-mortem* [3], [4], [5], [6], [7] and *on-line* [8], [9] **failure analysis** and, more importantly, for characterizing the runtime behavior of industrial and critical systems [10], [11]. Moreover, the increasing use of Off-The-Shelf components, even in safety-critical domains, introduces new dependability challenges, due to unforeseen components interactions or wrong execution of operations processes [12].

Field experience suggests that event logs may be **inaccurate** to obtain information on software failures and their causes [13], [14]. For example, our previous study [4] indicates that event logs might be strongly inaccurate: we estimated that around 60% of failures do not leave any trace in the logs collected at application level. Leveraging logs collected at **different system levels**, such as application, middleware and operating system (OS), is a potential solution to increase the chance of detecting failures through event logs. This is especially true in complex systems, which consists of many software levels.

This paper outlines a study on the failure reporting capability of event logs collected at different level of an industrial system. This study is based on a dataset of 3,159 failures. Failures have been induced by injecting different ODC (Orthogonal Defect Classification) [15] fault types into a Data Distribution System (DDS) of an **industrial middleware** for the Air Traffic Control (ATC) domain. Event logs are collected at three different system levels, i.e., *application*-level, *middleware*-level and *OS*-level. Our study provides strong insights into the reporting ability of event logs as affected by the type of faults and failures induced in the system. We adopted a logistic regression model to analyze such relationships from a statistic perspective. This work suggests that looking at event logs produced at different system levels is extremely useful to complement information regarding operational failures. More specifically, key insights of the study are:

- According to our data, the reporting ability of collected logs is not influenced by the ODC fault type injected in the system. Results show that each system level is able to detect almost the same percentage of failures irrespectively of the type of injected fault.
- Event logs produced by different system levels show rather different ability at reporting a given type of failure. For example, OS-level logs report *crash* failures much more better then the other levels, while middleware-level logs exhibit a good reporting capability of *silent* failures (e.g., livelocks).

The rest of the paper is organized as follows. Section II provides background concepts and related works. Section III describes the target system and experiments conducted in the study. Section IV provides the analysis results, while Section V concludes the work.

II. Background

A. Log-based Failure Reporting

Understanding the relationship among faults, errors and failures is a key point to gain insights into the generation of textual entries in the event log. As shown in Figure 1, a **fault** can be activated by a specific sequence of input,

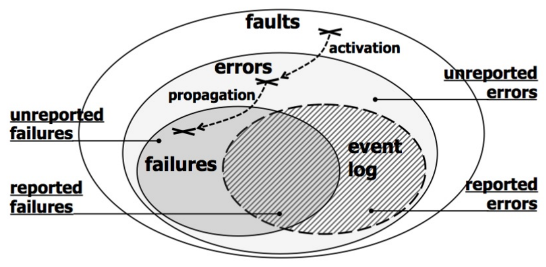

Figure 1: Failure reporting process.

environmental conditions, etc. Activation causes one or more **errors**, which can propagate in the system and may cause a **failure**, i.e., a deviation of the service provided by the system from the correct service [16]. Event logs report a subset of errors, i.e., *event log* in Figure 1 (for the sake of clarity, we only consider entries reporting error conditions in this discussion). As shown in Figure 1, logs might report errors that do not lead to failures (*reported errors*), and only a fraction of actual failures (*reported failures*), with many failures remaining unreported. These issues represent a serious threat when logs are used to analyze system failures [2], [3]. Unreported failures may lead to erroneous insights in the system behavior [17].

B. Related Work

Many works deal with the analysis of event logs to analyze failures occurred in a system. For example, in [3] are presented results of a failure data analysis of a LAN of Windows NT machines. Authors analyze event logs collected over a six-month period from the mail routing network of a commercial organization. In [6] the authors evaluate the reliability of web applications using both workload measurements and information extracted by event logs. In [18] it is proposed an approach which mines the execution logs of an application to flag anomalies from the dominant behavior. Finally, in [8] it is proposed a proactive prediction and control system for large clusters, which is based on the collection of event logs and system activity reports.

In the last decade, log analysis has been also used in **safety-critical and industrial domains**. For example, in [10] it is proposed a log analysis framework for Mars Science Laboratory flight software [19]. The proposed method is based on the extraction of simple events from application text logs. The extraction is implemented by means of regular expressions and it aims to obtain a well-structured log, which can be analyzed with appropriate patterns to detect failures. In [11] the authors proposed a failure detection and diagnosis framework for component-based distributed embedded systems, such as automotive systems. The framework is composed by a logging layer that allows to collect the stream of system events; events are then fed to Monitoring and Diagnosis layers.

Nevertheless, many works highlighted **inaccuracy** of event logs at detecting failures of software systems. For example in [5] the authors recognized that logs might be incomplete and ambiguous. Consequently, they analyzed the possibility of combining *wtmpx*[1] and *syslog* log files to achieve a better understanding of the target system. The work [14] indicates that event logs are able to detect failures caused by resource exhaustion; however, they are ineffective against software-related problems. The analysis is conducted by means of fault injection in the context of web applications. Work [4] shows that around 60% of failures caused by the activation of software faults are not reported by event logs.

We analyze the failure reporting ability of event logs collected at different levels of an ATC system. Failures are induced by means of fault injection in a controlled environment. Each experiment is characterized by the precise knowledge about the type of fault and failure induced in the system.

III. Experimental Setup

Experiments conducted in this study aim to exercise the logging mechanism of the application, middleware, and operating-system level under different combinations of fault and failure types. A fault injection approach has been adopted here to exercise the system behavior in a controlled environment. To this aim, we implemented the assessment framework proposed in [4]. In the following, we briefly describe the target system and how the experiments were performed.

A. Testbed and Fault Injection

The experimental campaign is conducted on a Data Distribution System (DDS) implementation[2], which has been used in a real-world Air Traffic Control system. The DDS ensures interoperability and integration of critical ATC components by means of the *publish/subscribe* paradigm. The source code consists of 739,490 lines. Faults have been injected into the source code of the DDS. Figure 2 shows the experimental testbed. The campaign consists of a sequence of software fault injection experiments. *For each experiment a single fault is introduced into the DDS*. The system is exercised with a workload to activate the fault-error-failure chain (Figure 1) and to collect event logs at different levels. A test DDS application implements the workload to exercise the system. The application consists of a *publisher* and a *subscriber* process (reported in Figure 2), which exchange messages through the DDS.

Software faults injected during the campaign are reported in Table I. Each fault represents a typical programming mistake, such as *missing variable initializations, missing*

[1]Binary files in which the SunOS/Solaris Unix operating system records information identifying the users login/logout

[2]The vendor is not disclosed due to confidentiality reasons

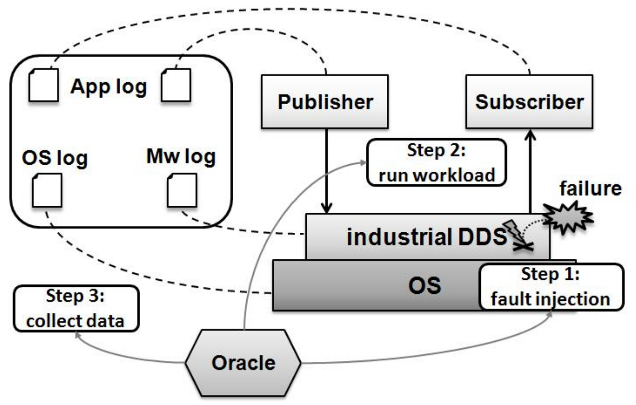

Figure 2: Testbed and experiment steps.

function calls, wrong values assigned to variables. Adopted faults represent a subset accounting for around 80% of representative faults, which have been identified by a reference work in the fault injection area [20]. The authors in [20] analyzed the fault distributions of many real-world software systems, and identified a subset of fault types representative of faults observed in the field. The fault types extend the Orthogonal Defect Classification for practical injection purposes. The ODC type covered by each fault is reported in the second column of Table I.

B. Experiments

The injection of a software fault is performed by means of the SAFE[3] tool [21]. The tool analyzes the source code of a given system and automatically infers the faults that can be introduced in the code along with their locations. For each fault identified by the tool, the experiment consists of **three steps** (reported in Figure 2):

1) The fault is introduced in the target software at the location established by the tool. The **injection** is implemented by means of changes of the source code, which emulate a programming mistake. The source code is compiled to obtain a faulty version of the target software.
2) The faulty version of the software is initialized and exercised with the test DDS application. The application exercises the publish/subscribe functionalities of the software with the goal of activating the injected fault.
3) Experiments results are collected when the workload completes or a predefined timeout expires (the timeout is tuned before the campaign by means of fault-free runs of the target system). Collected results include logs generated by (i) the applications that use the DDS (**App log**), (ii) the DDS middleware (**Mw log**), and (iii) the operating system (**OS log**). The original version of the source code is restored and logs are cleaned before a new experiment is started.

[3]http://www.critiware.com/safe.html

Table I: Faults adopted in the study [20]. (ALG-algorithm, ASG-assignment, CHK-checking, INT-interface)

Fault		ODC type
MFC	missing function call	ALG
MVIV	missing variable initialization using a value	ASG
MVAV	missing variable assignment using a value	ASG
MVAE	missing variable assignment with an expression	ASG
MIA	missing IF construct around statements	CHK
MIFS	missing IF construct plus statements	ALG
MIEB	missing IF construct plus statements plus ELSE before statement	ALG
MLAC	missing AND clause in expression used as branch condition	CHK
MLOC	missing OR clause in expression used as branch condition	CHK
MLPA	missing small and localized part of the algorithm	ALG
WVAV	wrong value assigned to variable	ASG
WPFV	wrong variable used in parameter of function call condition	INT
WAEP	wrong arithmetic expression in parameter of a function call	INT

The analysis focuses only on the experiments that caused a failure because we aim to assess the ability of collected logs to report software failures. Each experiment is characterized by the (i) type of fault induced in the system, (ii) type of failure caused by the fault, and (iii) event logs.

Experiments execution is automated by means of an **oracle program**, shown in Figure 2. At the end of each experiment, the oracle program labels each failure manifestations with one the following types:

- *crash*: the system has terminated unexpectedly during the fault injection experiment;
- *silent*: the system is up, but no output or functionality has been provided within the expected timeout;
- *erratic:* failures that are not *crash* or *silent*, e.g. the system produced a bad output.

The oracle analyze both operating system information (e.g. pid(s) of the process(es) executing the software under test, memory dumps), and workload-dependent data (e.g., output provided by the system under test) to determine the failure type. The oracle also establishes whether the failure has been reported, i.e, detected and notified with one or more entries in the event logs, by each layer. The oracle adopts a set of regular expressions, which have been developed by means of post-mortem analysis of collected logs, to identify entries in the log reporting failures conditions.

Table II: Application-level: number and percentage of reported failures (RF %) by fault and failure type.

	failure							
fault	CRASH	*RF %*	SILENT	*RF %*	ERRATIC	*RF %*	***total faults***	***RF %***
ALG	4	0.27	76	48.10	8	3.6	88	4.73
ASG	6	0.89	19	54.29	1	1.59	26	3.36
CHK	0	0.00	0	0.00	0	0.00	0	0.00
INT	5	1.14	8	80.00	0	0.00	13	2.77
total failures	15	0.57	103	50.49	9	2.85	**127**	**4.02**

Table III: Middleware-level: number and percentage of reported failures (RF %) by fault and failure type.

	failure							
fault	CRASH	*RF %*	SILENT	*RF %*	ERRATIC	*RF %*	***total faults***	***RF %***
ALG	210	14.17	93	58.86	45	20.27	348	18.69
ASG	130	19.23	30	85.71	16	25.40	176	22.74
CHK	9	21.43	1	100.00	1	9.09	11	20.37
INT	116	26.42	9	90.00	4	20.00	129	27.51
total failures	465	17.62	133	65.20	66	20.89	**664**	**21.02**

Table IV: OS-level: number and percentage of reported failures (RF %) by fault and failure type.

	failure							
fault	CRASH	*RF %*	SILENT	*RF %*	ERRATIC	*RF %*	***total faults***	***RF %***
ALG	1,481	99.93	74	46.84	6	2.70	1,561	83.83
ASG	676	100.00	6	17.14	2	3.17	684	88.37
CHK	42	100.00	0	0.00	0	0.00	42	77.78
INT	439	100.00	0	0.00	0	0.00	439	93.60
total failures	2,638	99.96	80	39.22	8	2.53	**2,726**	**86.29**

IV. Analysis Results

Overall 3,519 injection experiments caused a failure of the DDS. These failures compose the data set adopted in our study. Table V summarizes observed failures by fault and failure type. For example, the value 1,482 reported in the cell (*ALG, CRASH*) indicates that 1,482 algorithm faults, i.e. *ALG* type, caused a *CRASH* failure. We evaluate how the total amount of failures is reported by event logs collected across different system layers. Results are reported in Tables II to IV. These tables report number and percentage (with respect to the total number) of reported failures (RF %) by fault and failure type. For example, the pair of values 1,481 and 99.93% reported in the cell (*ALG, CRASH*) of the Table IV, indicate that 1,481 *CRASH* failures caused by algorithm faults are reported by OS log (out of the above discussed 1,482), which accounts for 99.93% of failure, i.e., 1,481/1,482. For the same fault/failure combination, i.e., *ALG, CRASH*, application-level logs report only 4 out of 1,482 failures, i.e., 0.27%. Furthermore, the rightmost columns of the Tables II to IV report the total number of faults detected by means of event logs. Similarly, the bottom rows of these tables aggregate the number and the percentage of reported failures.

Overall reporting capability of each system layer is shown by Figure 3. Reporting ability changes significantly across the considered layers, and is between 4,02% (application level) and 86,29% (for OS level). Such a strong difference is a relevant finding of the study.

Table V: Obtained failures by fault and failure type.

	failure			
fault	CRASH	SILENT	ERRATIC	***tot. faults***
ALG	1,482	158	222	1862
ASG	676	35	63	774
CHK	42	1	11	54
INT	439	10	20	469
tot. failures	2639	204	316	**3,159**

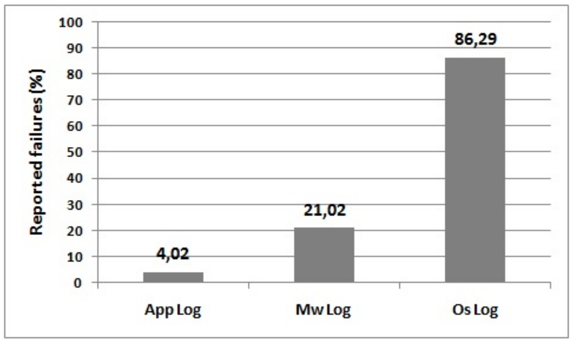

Figure 3: Percentage of reported failure by system levels.

We established whether there exists a relationship between the reporting ability of each system level and the type of fault/failure induced in the system.

Histogram shown in Figure 4 reports the percentage of **failures** by type and level. It can be observed that event logs produced by different levels have a rather different ability at detecting a given type of failure. For example, OS-level detects 99.96% *CRASH* failures, which is significantly greater than 0.57% and 17.62% of application- and middleware-level, respectively. In the same way, the middleware-level presents higher capability to report *SILENT* and *ERRATIC* failures than other levels. Moreover, we can notice that each level presents different reporting ability with respect to a given failure type. This finding suggests that there is a relation between reporting capability of each level and the type of failure occurred in the system.

Figure 5 shows the percentage of activated **faults** that are reported by event logs. Again, results are broken down by fault type and level. Percentages range between a minimum of 0%, for *CHK* faults at the application level, and a maximum of 93.6%, for *INT* faults at the OS level. Figure 5 shows that OS-level logs outperforms the others at detecting the activation of all types of fault. Moreover, results show that each system level is able to detect almost the same percentage of failures irrespectively of the type of injected fault.

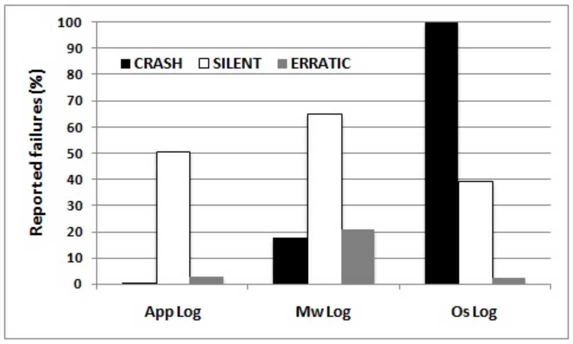

Figure 4: Percentage of reported failure by type and system levels.

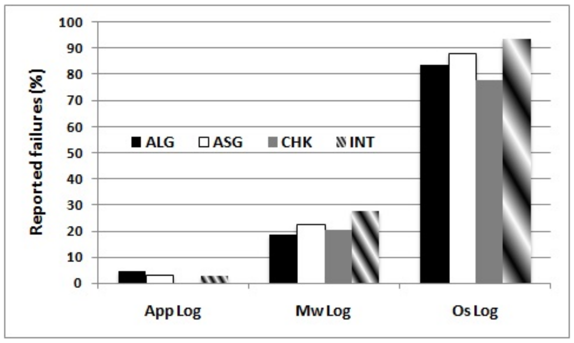

Figure 5: Percentage of reported failure by fault type and system levels.

A. Logistic regression analysis

The outcome of each experiment is a dichotomous variable representing whether the failure is reported or unreported by collected logs. The outcome is obtained by exercising the logging mechanism under different combinations of faults and failures. For this reasons, we modeled the experiment outcome as a dichotomous variable, which is characterized by two factor: *type of fault* injected in the system and *type of failure* that is caused by the activated fault. These factors represent two independent categorical variables. Precisely, the fault type is modeled by a four-level variable which assumes a value among *ALG*, *ASG*, *CHK*, *INT* whereas the failure type is modeled by a three-level variable which assumes a value among *ERRATIC*, *SILENT*, *CRASH*.

We performed a **logistic regression analysis** [22] to model the dependence between the experiment outcome and the fault and failure type. The strength of the relationship between the outcome and the independent variables is measured by the *p-value*[4]: the smaller the p-value, the higher the chance that the independent variable influences the reporting ability of event logs. Due space limitations, the data table with the analysis results is omitted; however, we summarize key findings of the analysis in the following.

The *p-value* is <0.0001 for both Mw and OS logs in the case of *failure* variable, which indicates a strong dependence between the failure type and the system level. On the contrary, the average *p-value* observed for the *fault* variable is above 0.49: according to our data, the type of injected fault does not seem to affect the reporting ability of collected logs. We also investigated, by means of logistic regression, the combined relationship between the failure reporting capability and the effect due to interaction between failure and fault type, i.e., if a given couple of fault-failure is better reported than others by a given log level. Results show that in no case there is a relationship between the type of injected fault and the resulting failure, in terms of reporting capabilities of logs (*p-value* above 0.1).

V. Conclusion

The paper proposed a study of failure reporting capability of three system level: application-level, OS-level and middleware-level. The considered middleware is an industrial middleware that enables the integration and the interoperability between critical systems for time and reliability.

[4]In the logistic model, the *p-value* is the conditional probability that a likelihood statistical test value (in particular, the *Likelihood-ratio chi-square test*) is equal to or greater than the actual value assumed by his value under null hypothesis.

The analysis reveals that different level presents different failure reporting capability. In particular, this capability is influenced by failure manifestation in 2 of the 3 levels considered in the study (middleware and operating system levels), while it is not significantly influenced by type of fault activated at runtime. Furthermore, results show that OS-level is the level with greater capacity to detect activated fault then other levels, in particular with respect to crash failures, whereas, middleware and application logs are useful to complement OS logs in the reporting of information related to silent and erratic failures. This findings suggest that it can be useful to combine event logs collected at different levels of a software system to increase failure detection, since, for example, failure not reported by a level can be reported by another level. Future works provide the extension of this analysis considering other failure data sources, such as software assertion, methods based on instrumentation of the source code, etc., to evaluate their capability to report software failure and to understand the usefulness of their possible combination.

Acknowledgment

This work has been partially supported by the TENACE PRIN Project (n. 20103P34XC) funded by the Italian Ministry of Education, University and Research, by the MINIMINDS PON Project (n. B21C12000710005) funded by the Italian Ministry of University and Research and by the project "EMBEDDED Systems" CUP B25B09000100007, POR Campania FSE 2007/2013.

References

[1] R. K. Iyer, Z. Kalbarczyk, and M. Kalyanakrishnan. Measurement-Based Analysis of Networked System Availability. *Performance Evaluation: Origins and Directions*, pages 161–199, 2000.

[2] A. J. Oliner and J. Stearley. What Supercomputers Say: A Study of Five System Logs. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 575–584. IEEE Computer Society, 2007.

[3] M. Kalyanakrishnam, Z. Kalbarczyk, and R. K. Iyer. Failure Data Analysis of a LAN of Windows NT based Computers. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, pages 178–187. IEEE Computer Society, October 1999.

[4] M. Cinque, D. Cotroneo, R. Natella, and A. Pecchia. Assessing and Improving the Effectiveness of Logs for the Analysis of Software Faults. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 457–466. IEEE Computer Society, 2010.

[5] C. Simache and M. Kaâniche. Availability Assessment of SunOS/Solaris Unix Systems Based on syslogd and wtmpx Log Files: A Case Study. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 49–56. IEEE Computer Society, 2005.

[6] J. Tian, S. Rudraraju, and Zhao Li. Evaluating web software reliability based on workload and failure data extracted from server logs. *Software Engineering, IEEE Transactions on*, 30(11):754–769, Nov 2004.

[7] A. Pecchia and S. Russo. Detection of software failures through event logs: An experimental study. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 31–40, Nov 2012.

[8] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 426–435, New York, NY, USA, 2003. ACM.

[9] Felix Salfner and M. Malek. Using hidden semi-markov models for effective online failure prediction. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 161–174, Oct 2007.

[10] H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal Analysis of Log Files. *Journal of Aerospace Computing, Information and Communication*, 7(11):365–390, 2010.

[11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Reflection: Dynamic model-based analyis of component-based distributed embedded systems. In *Modellierung von Automotive Systems*, 2006.

[12] D. P. Siewiorek, R. Chillarege, and Z. T. Kalbarczyk. Reflections on Industry Trends and Experimental Research in Dependability. *IEEE Transactions on Dependable and Secure Computing*, 1:109–127, April 2004.

[13] D. Cotroneo, S. Orlando, and S. Russo. Failure Classification and Analysis of the Java Virtual Machine. In *Proceedings of 26th International Conference on Distributed Computing Systems (ICDCS)*, 2006.

[14] L.M. Silva. Comparing Error Detection Techniques for Web Applications: An Experimental Study. *7th IEEE International Symposium on Network Computing and Applications*, pages 144–151, 2008.

[15] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.-Y. Wong. Orthogonal Defect Classification-A Concept for In-Process Measurements. *IEEE Transactions on Software Engineering*, 18:943–956, 1992.

[16] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan.-March 2004.

[17] M. Cinque, D. Cotroneo, and A. Pecchia. Event logs for the analysis of software failures: A rule-based approach. *Software Engineering, IEEE Transactions on*, 39(6):806–821, June 2013.

[18] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automatic identification of load testing problems. In *ICSM*, 2008.

[19] Mars Science Laboratory. http://mars.jpl.nasa.gov/msl.

[20] J.A. Duraes and H.S. Madeira. Emulation of Software Faults: A Field Data Study and a Practical Approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006.

[21] R. Natella, D. Cotroneo, J.A. Duraes, and H.S. Madeira. On Fault Representativeness of Software Fault Injection. *IEEE Transactions on Software Engineering*, 2011. PrePrint.

[22] David W. Hosmer and Stanley Lemeshow. *Applied logistic regression (Wiley Series in probability and statistics)*. Wiley-Interscience Publication, 2 edition, 2000.