INTERNATIONAL JOURNAL OF ADVANCE RESEARCH, IDEAS AND INNOVATIONS IN TECHNOLOGY

ISSN: 2454-132X
Impact factor: 6.078
(Volume 7, Issue 1)
Available online at: https://www.ijariit.com

A detailed survey and evaluation of various aspects of embedded software robustness testing

Chiranjeevi Aradhya
chiranjeevi.aradhya@collins.com
Collins Aerospace, Charlotte, North Carolina, USA

ABSTRACT

The focus of the current and future automotive, multimedia, and industrial automation systems is embedded software. A reliability of the embedded software system determines the performance of virtually every industrial application; one approach for determining a system's reliability is to measure its robustness. The aim of this paper is to provide a framework to understand the existing state-of-the-art practice in the robustness testing of built-in software systems. In robustness tests from different industrial domains, I've collected details on the state of practice on telecommunication, automotive, multimedia, critical infrastructure, aerospace, consumer products, and banking. I investigate various aspects of robustness testing, such as the general view of robustness, engineering and design of specifications, test results, failure and equipment. I emphasise the state of understanding of the robustness testing practice of integrated software systems. The survey outlines many ad-hoc techniques, including power failure and overload situations. Other discrepancies I found concern the classification of robustness defects, the assumed root causes of robustness defects and the types of devices for testing robustness. This article is an initial step in the evaluation of the robustness testing practice of embedded software systems.

Keywords: *Embedded System Robustness, Robustness Testing, Robustness Testing Tools*

1. INTRODUCTION

Testing is most commonly used in software quality assessments among different techniques used to verify and validate the software system [1]. Testing can be an effective and thorough way to correctly classify mistakes, if done properly [2]. Embedded information systems research is a major challenge [1] as there are different features that need to be taken into account when conducting tests. In particular, Qian and Zheng[3] describe four features that differentiate between testing embedded software and testing of general software. First, embedded software systems are built in a specific environment for a certain purpose, and the challenge faced by the test system is to test the system in a host or target setting. The tests are carried out in a specific environment. Second, interaction is a key aspect of the way embedded software systems work. Embedded systems usually work in an external environment and communicate with it by collecting data via sensors and acting on that data via actuators. Inadequate or incorrect assumptions may arise from the failure to replicate this interaction during the test. Embedded software systems are distinguished by programming practice in which different interfaces and platforms have important roles. The fourth function of integrated software is timeliness, which means that the accuracy of the actions of embedded systems not only depends on what the system is doing but also when it does. Timeliness therefore adds to the testing an extra dimension. In order to assess functional failures and achieve a high degree of test coverage within embedded software systems, standard software testing techniques are helpful but not sufficiently rigorous to detect problem robustness resulting from environmental errors such as unintended or misleading inputs [4]. Robustness has been defined as a 'degree to which a device or part can operate correctly when inputs are invalid or when environmental conditions are stressful' by the IEEE standard 610.12-1990 [5]. Features such as robustness are particularly demanding for embedded software systems, because at the time of development their operating environments cannot be fully predicted [2]. The specifications of robustness testing can vary from very general to particular requirements. For starters, no runtime errors, no crashes or deadlocks are general considerations. For example, the ability for a system to return to a nominal state after a degraded state has been reached, or that certain system resources remain available for high priority tasks [2]. However, multiple interpretations of robustness testing and thus of the way it is carried out in different contexts could occur. The robust testing of the software system is, as defined by Huhns and Holderfield, about the way it checks its ability to avoid crashes. Often robustness is commonly used as a definition of fault tolerance. Technology of fault tolerance is used to comply with requirements for design reliability [6]. Robustness testing therefore also helps to measure the fault tolerance of the systems. The main goal is to measure the ability of the software system to manage adverse circumstances arising from internal or external influences in the majority of robustness definitions [7]. The relationship between robustness and other attributes of reliability, such as stability, availability and efficiency is also contested [7]. Robustness is defined for example in EMISQ [7] as part of a system's reliability, even if it is not expressly referred to in the standard of ISO 9126 [8]. However, the information about practice state is not very reliable in solidity testing of embedded systems. To our knowledge, Eldh and Sundmark[7] have done only an industrial test in which the typical robustness testing practices in broad-scale telecommunications systems have been established. They identified major challenges in understanding robustness at various levels, such as architecture, unit and device level. Their work should stress that the robustness of industry is often lacking in understanding and that the view of robustness can differ considerably between enterprises and industrial contexts.

The aim of this study is to analyze and compare the state of the art of robustness testing for embedded software systems. There is, therefore, a twofold benefit from this study: firstly, to emphasize industrial robustness testing practices of embedded software systems and help to improve information on robustness testing. Secondly, it helps us to illustrate discrepancies and disparities in knowledge of the topic from theoretical to realistic.

This paper presents the findings of an empirical analysis of seven distinct areas of embedded computing. The goal of the survey is to analyze the current robustness test of embedded software systems. The findings of the study show a strong difference between industrial practice of robustness assessment and the scientific literature mentioned above. I am discussing the state of the art and the difference between state of the art and robustness testing practice.

2. LITERATURE SURVEY

2.1. Robustness

The consistency to achieve higher reliability is also regarded as robustness [9]. Software robustness is specifically defined in scientific literature, such as the IEEE Standard Glossary (610.12-1990)[5]. But this is not the sole description of robustness; in view of Lei et al., definitions like invalid feedback and stressful conditions in the setting can be ambiguous depending on the invariant's meaning and the precondition before performance [11]. There are thus alternative definitions: Lussier and al.,[11] describe robustness as "the provision of a correct service to an uncertain system environment, such as an unexpected barrier or a change of lighting condition affecting sensors," which is implicit in some adverse circumstances. Shahrokni and Feldt describe robustity in the context of the industry as 'robust in the presence of incorrect feedback and stability of execution in the presence of challenging conditions generated by external resources or modules' [9]. In general, the standard informal definition of robustness is that, under extraordinary or unexpected operating conditions, a system should demonstrate reasonable behavior [2].

2.2. Fault Injection

The common practice for robustness checking was fault injection [12],[13]. Fault injection is a method used to assess a system's reaction to an artificially induced failure. In measuring robustness by means of defect injection two methods have mainly been used. The fuzz method measures applications with random and elaborate input streams that search device crashes. This approach attempts to measure information systems from the point of view of interactive users[12]. The other method, known as benchmarking, uses fault injection to detect crashes by passing a combination of exceptional inputs as a parameter through the test program API[13].

2.3. Reason for Robustness Failures

Robustness testing of operating systems in particular have been examined primarily in scientific literature. In particular, the CRASH benchmark defined the failures in the operating system sense. Robustness failures, i.e., are represented categorically in 5 levels by CRASH. Disaster, resumption, abortion, silence and obstruction. In [13] the following are described as:

- **Catastrophic failure:** "'The Catastrophic failure class happens when a failure isn't contained in one mission. In other words, this degree of failure leads to other tasks or even the machine itself dropping or hanging from a call to an OS feature. Usually, a catastrophe failure involves a hardware reset of the whole device, but can be limited to warm resetting of the OS.
- **Restart failure:** "If there is a single task, the failure restart class occurs and the task needs to be killed and rebooted to return to normal performance."
- **Failure to abort:** "The failure abortion class occurs when a single mission is abnormal. A typical abnormal termination is due to a segmentation infringement, which attempts to access a memory that has no access rights (for example, by dereferencing a null pointer)"
- **Silent failure:** If invalid parameters are sent to an OS call the Silent failure class occurs. But no error return code or other task failure is produced. For instance, a call to open a NULL file might return an error flag rather than a success flag.
- **Failure impediment:** The impede class is so called because the OS prevents a problem from being properly diagnosed by incorrect error code**.** For instance, if the only mistaken input is an invalid file handle value, a memory access code is returned that would be an obstacle class failure.

While several studies describe CRASH robustness failures, some of these studies also illustrate the causes. With regard to the "catastrophic" errors, Lei et al. [17] analyzed state-based robustness testing for elements and concluded that the services could result in a catastrophic failure if they replied with an error message or with a derogation suggesting an unforeseen internal problem. They also found that an unchecked exception thrown without declaring it in a script, and user-defined managed exceptions, which indicating that the component is not working correctly, can lead to disastrous failure. Schmid et al., [18] found that if a program fails to cope with an exception thrown by an OS feature or an exception divides by null it would most likely crash after determining Windows NT software's robustness. Koopman and DeVale [15] examined the efficiency of exceptional management of POSIX and found that system calls with exceptional parameter values could lead to catastrophic failures.

Fernsler and Koopman [16] described internal errors like failure of any feature which required the killing of any task leading to restart failure with respect to the "Restart" failure. Likewise, Koopman and DeVale [14] have found that data types such as invalid file pointers, null file pointers, invalid buffer pointers, minint integers, and maxint integers are correlated with error errors, in their study on comparing various POSIX operating systems based on robustness. Koopman and DeVale also discovered in another study[15] that if a signal is sent to itself from the system call or library feature, resulting in an irregular work termination, the abortion may result. Koopman and DeVale[15] noticed, with regard to "silent" failures, that when exceptional inputs in a test module were misled to show that they were successful, it could lead to a silent breakdown. Nevertheless, the findings described above may be highly context-specific, and it is difficult to draw general conclusions about how more resilient some types of defects are.

There are some known robustness triggers in scientific literature that are not relevant to any unique CRASH group. In their study[16], for example, Fernsler and Koopman have found that internal exceptions, unknown exceptions and segmentation errors are the reason for robustness failures. The reasons for robustness failure were found by Miller et al. [12]. For instance, the use of pointer and array subscripts is dominant in errors. The results revealed that dangerous input functions like gets() are a common root cause of robustness failures, because gets() has no parameter to limit the input data length. They also found that translating numbers from one size to another would result in a lack of robustness. Lei et al. [10] have found that unique conditions such as zero-division, zero overflow and boundary access array can lead to a failure of robustness.

2.4. Testing Tools
Some methods for robustness testing are used and researched. The instrument used most frequently to measure the power of operating systems [14] is the Ballista tool. The Riddle tool was also used for checking Windows NT robustness [19]. The Fuzz tool has been used [12] to support the robustness of UNIX utilities and services. It is important to remember that most of the tools are dated. Different methods were developed and studied in scientific literature with respect to various application fields. For instance, the Java automatic JCrasher tool is used in Java-based applications to test Java programs for robustness [20]. Robout has also been used to test the robustness of Java components with a stated robustness test tool [17]. WebSob is an automated platform for robustness testing and web service response analysis [21].

2.5. Robustness Testing in Different Domains
In scientific literature, most evidence suggests that the robustness of operating systems is primarily achieved and studied for the checking of the dependability of operating systems [4]. In addition, several studies have compared operating systems to their robustness based on reliability [13]. In other fields than in operating systems there is little research that concentrates on measuring robustness. One of the only examples of this is the domain of telecoms systems, where Eldh and Sundmark[7] have carried out a case study of how to test robustness in telecommunications systems. Johansson et al.[22], who has developed T-Fuzz, a new furrowing system for the testing of telecommunicative protocols, is also an example from the same context. Two studies [22][23] identify a method for the execution of robustness tests for high-disposability middleware solutions with regard to middleware systems. The approach for rough testing components using a semantical model has been illustrated by Lei et al.[10]. Ali et al.[24] likewise model the framework using aspect-oriented modelling approaches to enable testing of robustness. For their research, they use a video conference device. Laranjeiro et al. [25] use algorithms for text classification for projects based on a Web basis which measure the robustness of web services. In conclusion, Belli et al. [26] proposes a web-based system robustness testing model using event sequence graphs (ESG) and decision tables (DT).

In short, from the literature it can be inferred that:
- Most current robustness testing initiatives concentrate on software robustness, but very few such studies focus on testing the robustness of built-in software systems
- All research, except one, concentrate on new technology. The state of practice of robustness testing is not very well known.
- Only robust control in certain domains is specified by state-of-the-art. This knowledge from various domains is therefore a challenge.

3. ROBUSTNESS TESTING
The survey result has definition of robustness i.e., the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions. Lastly, the survey was done based on these following questions.
- "It is acceptable if the system fails. If the system is reliable enough. The system is robust and capable of withstanding unexpected events."
- "One must cope with the unexpected, not only the predictable."
- "We must never have faulty output in our system."

The system is really impressive; it can deal with noise, attacks and harmful data without any difficulty. The responses from these industries revealed that they do not use the word robustness. The system should end up in a safe state when there is a failure and end up in the safe state. Although robustness can be addressed explicitly as a non-functional property, it is handled implicitly by following a technical standard such as DO-178B. Surveys revealed that multimedia or telecom organizations use the term robustness, while many of them did not use the IEEE definition. Robustness of the system is important because the system recovers quickly and its failure behaviors are widely known. An annual survey indicated that telecom companies differentiate between robustness and redundancy, two distinct types of system failure scenarios. There is a considerable difference in telecommunications and computing as the view of robustness can be contradicting.

3.1 Key Aspects of Robust Embedded Software Systems
In the domain of telecommunication and multimedia, the survey found that there is no single point of failure and graceful deterioration a crucial feature. In fact, a survey found that certain transactions should not be refused but the approved transactions should be accomplished and the primary purpose of the bank should always function. The device should still be available for retrieval. Both surveys pointed out graceful deterioration as significant feature. In addition, they discussed the importance of stable behavior over time, and the necessity of providing a backup system to move the system into a safe state whenever necessary. From

the perspective of infrastructure security, resilient systems are important. The survey from the aerospace domain indicates that the values of longevity need to be appropriate for 109 hours. It is also notable that the interviewees reflected on the importance of end user actions and experience in their organization.

3.2 Requirements for Robustness Testing

Especially in aerospace domain, there are critical safety requirements that must be upheld even after 109 flight hours. Availability plays an important role in this as this is mentioned by all domains in various manners. A number of the studies reported recurrent events such as overload, malicious traffic, or redundant recovery of servers. Also mentioned was the need for information on data integrity, failure rate, latency and response time. These requirements were listed by surveys as availability of 99.999% of uptime, systems on net achieve safety level 4, restoration time for systems with SLA is less than 2 seconds, and response time is below milliseconds. In order to meet the general requirement of no failures for 109 flight hours, every critical component must have no failures. The control system is a critical component and so must therefore, all components between the control system and the pilot. The robustness requirements were limited to different combinations of three sources. The critical combinations included customers, operational standards, and propriety requirements. Majority of the respondents say that their needs were influenced by customers to some extent. Requirements will differ according to the type of requirement. For example, regulatory requirements for safety come from legislation while standards concerning quality of service are set by the company itself. There are subjects that only have concerns from their own organization.

3.3 Design for Robustness Testing

Most of the robustness-specific design attributes were unfortunately unpublished or unavailable. The greatest amount of research is done on system level. Test design is focused on the experience of individuals rather than using rigorous analysis. Few proposed systematic test design techniques include boundary value analysis and search-based simulation tests for measuring constraint breaches. Any of the questions in the pretest highlights the use of the system's robustness testing technique. The strongest point in favor of robustness testing for exploratory testing is that exploratory testing depends on the testers' experience and domain knowledge rather than requirements. This is statement based on supposition that certain robustness criteria are implicitly believed rather than specifically articulated. Code verification, static analysis and dynamic analysis are widely used in robustness testing in the telecom domain. These analyzes are usually conducted on lower levels of integration, for example stack buffer overflows and memory leaks. While not seeking robustness, problems such as atrocities are likely to affect the reliability of the system.

3.4 Performing Robustness Testing

We have identified three common approaches to robustness testing.

a) Organization whose test-driven architecture was not completing robust. Any high-risk item must occur with personnel and dependable testing protocols in place.
b) Organizations where no robustness testing teams or workers exist to monitor and maintain robustness in their systems and applications.
c) Those organizations in which robustness testing was explicitly recorded by requirements or regulations.

Without understanding, organizations can still regard their testing activities as standard, but robustness testing. A testing scheme related to robustness can be established as performance, maintainability or stability testing. This can explain why most of the organizations interviewed by us had more of a combination of A and B strategies. In reality, only telecom organizations fall into the "C" approach in our analysis. Robustness assessments usually derive from the possible unwanted, undesired or unforeseen impacts that may happen within the environment of the device under evaluation. Failure scenarios can include device reset, crash, power outage, cable failure, hardware malfunction, system misuse, etc. Dedicated robustness tests are designed to determine the ability of the device being tested to withstand such catastrophic events. When system robustness is calculated using a generic metric during development, this is usually done in the perspective of a system user. These vary in type and level. However, for most of the studied organizations, availability is still more critical than uptime. Ideally, a robustness stressor does not induce the availability of the device under evaluation (like the restart of a subsystem). There are memory and connectivity problems as an example of the robustness problems. A number of surveys state that robustness testing has also been performed on a virtual platform. The majority of our surveys state that they have, or are working toward high levels of automated testing.

3.5 Robustness Failures

The findings show there are a number of different forms of resilience failures. Crashes are the most common form of traffic incidents. Errors from both technological and non-technical causes are widely seen.

In general, deficiencies of systems should not be graded. Both interviewees admitted that they do not mark their shortcomings. However, the two questionnaires, which categorize failures, came up with the information, which concerns safety, i.e. which failures lead the system to enter a safe or unsafe state. There are examples of failures which are categorized as crashes and reset, memory failures, performance issues, response/latency (telecom, multimedia), or timing constraints (automotive, multimedia). There are different types of types of systems failure or breakdown. Actually this architecture prevents such catastrophes from happening (telecom, aerospace). The reason the system is less stable is that it's become so complex. The robustness failures may be a variety of reasons including lack of understanding of the environment, inappropriate memory addressing, not finishing sequences, configuration issues, geometrical transformation errors, and integration problems.

4. SUMMARY OF SURVEY

I gather the following to be true about robustness testing:

The IEEE concept of computing robustness is generally acknowledged, but there are further reservations concerning computing robustness. The most widely cited attribute of embedded systems is their availability. The program does not come with any flaw that can make it vulnerable. The necessity of robustness testing of embedded software systems is the availability of the system to the conditions of use. There are some system specifications for managing the circumstances and some for improving efficiency, speed and reliability. Typically, robustness-testing criteria come from the development organization, policy guidelines and users. However, there is some literature that some specifications may come from only one or two of these sources. Robustness-test design is an inductive rather than a deductive analysis. Testing companies with multiple approaches to testing robustness. In experiments, various situations originate so as to determine the capability of the systems and quantify different qualities. Robustness research is usually performed on the network (i.e., actual hardware, actual software, and often simulated environment). The crash rate is the most common hardware failure compared to performance, restarts, Processor or memory related problems. In general, catastrophic failures are not divided into various groups.

According to the widely agreed position, the causes of robustness failures are those that are created by complexity of the system, lack of understanding of environment, memory addressing, fragmentation issues, configuration issues, integration issues and geometrical transformations. There are various types of methods which could be used in the concept of robustness testing. The tools will allow for the visualization, monitoring, revision, and replaying of the test.

The biggest problem involved with robustness testing is that all of the testing must be performed with a single instrument for the whole system. The loss of robustness is often caused by different factors, one of which is sophistication of the system, lack of understanding of the environment, memory addressing, fragmentation issues, configuration issues, integration issues and geometrical transformations.

5. CONCLUSION

I survey current experience in robustness testing and compare it to that of exiting system in pratice. So far, the state of the science has not been discussed as much in literature on robustness testing. As seen in literature, robustness testing of a software project would focus only on operating systems (OS), rather than embedded or real time system. My research found that the procedure when it comes to robustness testing is different to practice when it comes to science. For example, these kinds of advice are used occasionally or never. Surveys explained strategies to deal with special situations (e.g., power failure or overload). Other discrepancies I found involves the proposed definition of root cause of the failures, the hypothesized root causes and types of methods used in robustness research. This informs professionals at various levels as well as in different geographical regions. Knowledge of current practices is an essential and must aspect of study. Knowledge learned in this course can then be applied in the real test environment.

6. REFERENCES

[1] S. P. Karmore and A. R. Mahajan, ‘‘Universal methodology for embedded system testing,’’ in Proc. 8th Int. Conf. Comput. Sci. Educ. (ICCSE), Apr. 2013, pp. 567–572.

[2] J.-C. Fernandez, L. Mounier, and C. Pachon, ‘‘A model-based approach for robustness testing,’’ in Testing of Communicating Systems (Lecture Notes in Computer Science), vol. 3502. Montreal, QC, Canada: Springer, 2005, pp. 333–348.

[3] H. M. Qian and C. Zheng, ‘‘A embedded software testing process model,’’in Proc. Int. Conf. Comput. Intell. Softw. Eng. (CiSE), Dec. 2009, pp. 1–5. [4] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek, ‘‘Automated robustness testing of off-the-shelf software components,’’ Proc. 28th Annu. Int. Symp.Fault-Tolerant Comput. (FTCS), Jun. 1998, pp. 230–239.

[4] IEEE Standard Glossary of Software Engineering Terminology, IEEE Standard 61012-1990, Dec. 1990, pp. 1–84.

[5] W. Torres-Pomales, ‘‘Software fault tolerance: A tutorial,’’ Dept. Comput. Program. Softw., Langley Res. Center, Hampton, VA, USA, Tech. Rep. NASA/TM-2000-210616, 2000.

[6] S. Eldh and D. Sundmark, ‘‘Robustness testing of mobile telecommunica- tion systems: A case study on industrial practice and challenges,’’ in Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation (ICST), Apr. 2012, pp. 895–900.

[7] Software Engineering—Product Quality—Part 4: Quality in Use Met- rics, Standard ISO/IEC TR 9126-4:2004, International Organization for Standardization, 2004.

[8] A. Shahrokni and R. Feldt, ‘‘A systematic review of software robustness,’’Inf. Softw. Technol., vol. 55, no. 1, pp. 1–17, Jan. 2013.

[9] B. Lei, X. Li, Z. Liu, C. Morisset, and V. Stolz, ‘‘Robustness test- ing for software components,’’ Sci. Comput. Program., vol. 75, no. 10, pp. 879–897, Oct. 2010.

[10] B. Lussier, R. Chatila, F. Ingrand, M.-O. Killijian, and D. Powell, ‘‘On fault tolerance and robustness in autonomous systems,’’ in Proc. 3rd IARP- IEEE/RAS—EURON Joint Workshop Tech. Challenges Dependable Robots Human Environ., Oct. 2004, pp. 1–7.

[11] B. P. Miller et al., ‘‘Fuzz revisited: A re-examination of the reliability of UNIX utilities and services,’’ Dept. Comput. Sci., Univ. Wisconsin-Madison, Madison, WI USA, Tech. Rep., 1995.

[12] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, ‘‘Comparing operating systems using robustness benchmarks,’’ in Proc. 16th Symp. Rel. Distrib. Syst., Oct. 1997, pp. 72–79.

[13] P. Koopman and J. DeVale, ‘‘Comparing the robustness of POSIX oper- ating systems,’’ in 29th Annu. Int. Symp. Fault-Tolerant Comput., Dig. Papers, Jun. 1999, pp. 30–37.

[14] P. Koopman and J. DeVale, ‘‘The exception handling effectiveness of POSIX operating systems,’’ IEEE Trans. Softw. Eng., vol. 26, no. 9, pp. 837–848, Sep. 2000.

[15] K. Fernsler and P. Koopman, ‘‘Robustness testing of a distributed simu- lation backplane,’’ in Proc. 10th Int. Symp. Softw. Rel. Eng., Nov. 1999, pp. 189–198.

[16] B. Lei, Z. Liu, C. Morisset, and X. Li, ''State based robustness testing for components,'' Electron. Notes Theoretical Comput. Sci., vol. 260, pp. 173–188, Jan. 2010.

[17] M. Schmid, A. Ghosh, and F. Hill, ''Techniques for evaluating the robust- ness of Windows NT software,'' in Proc. DARPA Inf. Survivability Conf. Expo. (DISCEX), vol. 2. Jan. 2000, pp. 347–360.

[18] A. K. Ghosh, M. Schmid, and V. Shah, ''Testing the robustness of Windows NT software,'' in Proc. 9th Int. Symp. Softw. Rel. Eng., Nov. 1998, pp. 231–235.

[19] C. Csallner and Y. Smaragdakis, ''JCrasher: An automatic robustness tester for Java,'' Softw.–Pract. Exper., vol. 34, no. 11, pp. 1025–1050, Sep. 2004.

[20] E. Martin, S. Basu, and T. Xie, ''Automated testing and response analysis of Web services,'' in Proc. IEEE Int. Conf. Web Services (ICWS), Jul. 2007, pp. 647–654.

[21] A. Kövi and Z. Micskei, ''Robustness testing of standard specifications- based HA middleware,'' in Proc. IEEE 30th Int. Conf. Distrib. Comput. Syst. Workshop (ICDCSW), Jun. 2010, pp. 302–306.

[22] Z. Micskei, I. Majzik, and F. Tam, ''Robustness testing techniques for high availability middleware solutions,'' in Proc. Workshop Eng. Fault Tolerant Syst., 2006, pp. 1–12.

[23] S. Ali, L. C. Briand, and H. Hemmati, ''Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems,'' Softw. Syst. Model., vol. 11, no. 4, pp. 633–670, Oct. 2012.

[24] N. Laranjeiro, R. Oliveira, and M. Vieira, ''Applying text classification algorithms in Web services robustness testing,'' in Proc. 29th IEEE Symp. Rel. Distrib. Syst., Oct./Nov. 2010, pp. 255–264.

[25] F. Belli, A. Hollmann, and W. E. Wong, ''Towards scalable robustness test- ing,'' in Proc. 4th Int. Conf. Secure Softw. Integr. Rel. Improvement (SSIRI), Jun. 2010, pp. 208–216.