A BRIEF PROGRAM ROBUSTNESS SURVEY

Ayman M. Abdalla, Mohammad M. Abdallah and Mosa I. Salah

Faculty of Science and I.T, Al-Zaytoonah University of Jordan, Amman, Jordan

ABSTRACT

Program Robustness is now more important than before, because of the role software programs play in our life. Many papers defined it, measured it, and put it into context. In this paper, we explore the different definitions of program robustness and different types of techniques used to achieve or measure it. There are many papers about robustness. We chose the papers that clearly discuss program or software robustness. These papers stated that program (or software) robustness indicates the absence of ungraceful failures. There are different types of techniques used to create or measure a robust program. However, there is still a wide space for research in this area.

KEYWORDS

Robustness, Robustness measurement, Dependability, Correctness.

1. INTRODUCTION

The word "robustness" is widely used with different meanings. In one instance, it describes computer software. At other times, it may express a machine attribute, a mathematical equation, a medicine or a patient. The question is: What is Robustness? In this paper, we will answer this question from a software engineering point of view focusing on software robustness.

Before defining the meaning of software robustness, some expressions will be defined to help explain the meaning of software robustness. The concepts of correctness, dependability, and reliability will be clarified to differentiate between robustness and each of them. They can give multiple meanings that may cause robustness to become ambiguous.

2. CORRECTNESS, DEPENDABILITY, AND RELIABILITY

There is a relationship between software robustness and each of software correctness, software dependability, and software reliability. Software correctness may be considered as a robustness characteristic. Software correctness is defined in IEEE standards [1] in three different ways:

- The degree to which software, documents, or other items meet user needs and expectations, whether specified or not.
- The degree to which software, documents, or other items meet specified requirements.
- The degree to which a system or component is free from faults in its specification, design, and implementation.

The first two definitions discuss software correctness via input and output. Here, the only criterion that evaluates software correctness is requirements satisfaction, such as user and systems specifications. The third definition takes all sides of software; input, output, and the program body (code), considering their correctness to get partial or total software correctness.

Partial software correctness [1] implies that the "program's output assertions follow logically from the input assertions and proceeding steps and processing steps". In addition, the "program should terminate under all specified input conditions" to satisfy the total software correctness [1]. Every tool, algorithm, and technique needs a proof of correctness. A proof of correctness is "a formal technique used to prove mathematically that the computer program satisfies its requirements" [1]. The techniques used vary between different systems because they depend on system assertions, formal specifications, and requirements.

The opposite of correctness is failure, where the system has some faults. A fault [1] is: "a defect in a hardware device or component." In computer programs, a fault means "an incorrect step, process, or data definition. 'Bug' and 'error' are common use to express program fault" [1]. Device faults or program faults could cause a system failure [1], which is "the inability of a system or components to perform its required functions within specified performance requirements."

There are different classifications for faults. Laprie [2] classified faults depending on the viewpoints as phenomenological cause, nature, phase of creation or occurrence, situation with respect to system boundaries, and persistence.

In another aspect, software dependability in general is "the ability to deliver service that can justifiably be trusted" [3]. This means that the system can avoid failures and it is less likely to be broken or stopped. Furthermore, if System A depends on System B, the dependability of A is affected by the dependability of B [3].
Dependability was discussed in many places and it is integrated into the following attributes [3-7]:

1) Availability in a simple way where the system is able to run and deliver a service at any time.
2) Reliability, which is the "ability of a system or component to perform its requirements functions in a specified period of time" [1, 4]. Therefore, the main factor in reliability is time. In this research, however, robustness is not considering time as one of the robustness aspects.
3) Safety, where the system is judged based on whether it may cause any harm to users or the environment.
4) Confidentiality is concerned with preventing the system from having, giving, or dealing with unauthorised information.
5) Integrity is the absence of improper change in the system.
6) Maintainability is the ability of the system to be changed to meet new requirements.

Avizientis et al. [3] mentioned that robustness consists of "specialized secondary attributes" of dependability. It characterises the system reactions towards some faults.

3. ROBUSTNESS DEFINITION

The IEEE definition of robustness [1] is: "The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions".

In this definition, there are three main aspects; program response, input data, and system environment. Program response means that the system should respond rationally [8], but not necessarily correctly. It should not fail to reply or react illogically. The input data is one of the factors that affect the robustness of the program. A robust program can continue to operate correctly despite the introduction of invalid input [9].

The environment where the program is run includes hardware, other software systems, and the humans that run the program. These factors also affect program robustness.

Steven [10] mentions that robustness is “the ability of a system to continue to operate correctly across the wide range of operational conditions, and fails gracefully outside the range”. Robustness is required in critical programs, where program failure could cause massive extreme problems [11]. In the previous definitions, Steven did not disallow system faults, but the required condition is the system failing gracefully [10], which means that the failure of the system will not crash or hang. Steven’s definition covers hardware faults such as shortage of power supply. Hardware defects may be considered as stressful environment conditions.

4. ROBUSTNESS TECHNIQUES

Researchers who deal with software robustness try to develop internal techniques to produce a robust program. On the other hand, others try to find external techniques to certify the program and determine whether it is robust or not.

Robustness can be internal or external [12], where internal robustness is robustness of the code of the program including functions, classes, threads, etc., and external robustness is that of the surrounding environment [13].

Robustness is not limited to software programs. It is also a key issue in hardware sensors and chips and in hardware trust [14]. Robustness is essential for embedded systems security where hardware attacks such as fault injection attacks and hardware Trojans can be prevented if robustness were taken as a main factor in embedded systems design [15, 16].

Different techniques were tried to satisfy software robustness. These techniques also utilize different theories and methods to measure software robustness. In this section, an assessment of the software robustness techniques used to build and measure robustness will be discussed.

4.1 Fault Tolerance

Fault tolerance and robustness have the same objective; ensuring that system faults do not cause system failure. Both fault tolerance and robustness are needed in all systems especially in critical ones, where system failure can cause massive problems.

Fault tolerance that provides service complying with the specification in spite of faults is less costly than other redundancy techniques, but it has the same problem of increased code size and reduced performance [17].

Fault tolerance can be hardware fault tolerance or software fault tolerance. Hardware fault tolerance considers the correctness of hardware (physical) parts of the system. Software fault tolerance discusses the correctness of the code. Therefore, fault tolerance techniques have the advantage that they can be used to validate any kind of systems [18].

Fault tolerant systems can continue in operation after the occurrence of some system faults. Fault tolerance has four aspects [7]:

1) Fault detection: faults that could cause system failure will be detected.
2) Damage assessment: the affected system parts will be detected.

3) Fault recovery: can be done in two ways. “Backward error recovery” where the system will return to last constant state (Safe state). “Forward error recovery” where the system repairs the faults and keeps running.

Fault repair: includes the faults that are not cured in the fault recovery aspect.

Implementation of fault tolerance is possible by including checks and recovery action in the software. This is called defensive programming. Defensive programming cannot effectively manage the system faults, which occur through interaction between software and hardware. Software fault tolerance has two approaches [7]:

1) N-version programming is using at least three versions of software that should be consistent in the event of a single failure. In this approach, software is run in parallel on different computers. Using a voting system, the system compares the output and invalid output or late output will be rejected.

Recovery blocks are dynamic techniques. The system adjusts the output during execution depending on acceptance test and backward recovery, where the system returns to the last acceptance stage before the fault happens.

Pullum [9] mentions that the relationship between software robustness and fault tolerance is focused mainly on the techniques that can handle the: Out of range input, Input of the wrong type and Input in the wrong format.

Michael et al. [21] added that robust systems mark faults to make it easier for other programs to fix them. In addition, software robustness may share some features with fault tolerance techniques, such as testing the input type, testing the control sequence, and testing the functions of the process.

Different techniques were developed to measure software robustness. The next step in measuring robust software is to test whether or not the techniques were successful in producing robust software.

4.2 Redundancy

Redundancy is one of the ideas applied in building component robust software [19]. The idea of redundancy is to add a different component, but one that will be equivalent in functionality with old ones. Consequently, if one part fails to perform correctly, it will be replaced with another that can provide the same services.

Redundancy was used in hardware systems such as NASA satellites by duplicating critical hardware subsystems. Nevertheless, in software systems redundancy, it cannot be applied the same way because identical software subsystems will fail in the same identical ways. Thus, redundancy must be applied to software in a different way [19, 20]. The challenge in software systems is to design subsystems that can perform and behave in equivalent functionality, but do not fail in the same situations [19, 21].

4.3 Agent Systems

Michael et al. [21] used agent techniques (an agent may be a client, a friend, or a service provider agent [22]). Agents have the same idea of redundancy, but differ in implementation. They have equivalent functionality, but they are not identical to each other. To overcome redundancy technique problems, agent software is used. Agents present a convenient level of granularity. Agents by design know how to deal with each other, so they can cooperate and cover each other's faults [13].

Applying agents in a system fulfils the following benefits [23]:

- Agents can be added to the system one at a time. Software modification will be allowed during its lifetime.
- Agents represent multiple viewpoints and can use a different decision procedure in every distinct agent, which is the main idea of redundant techniques.

All redundancy techniques cause a complexity increase in programs by increasing the code length, and thus affect many aspects including program maintenance and testing. In addition, the normal run-time system will execute only one of the functions while the others remain idle, which reduces efficiency.

4.3.1. Self-adaptive systems

Another method applied to obtain robust software is self-adaptive software where the program has the ability to fix itself. The mechanism is easy to understand, but difficult to apply. The self-adaptive system can evaluate its work, and change the behaviour when the evaluation result shows there is an error. Moreover, a self-adaptive system can fix itself by applying an alternate behaviour.

Self-adaptive systems struggle in evaluating functionality and performance at run time. In addition, they may manage to get close to the solution of a problem but not close enough to solve it precisely [24].

Another adaptive system called "Self-controlling software model" was developed. This system contains three loops. The feedback loop adjusts system variables to meet the quality of service requirement. The adaption loop evaluates the behaviour and performance of the model and triggers change when necessary. The reconfiguration loop runs the adaption loop request. This loop is costly compared with feedback and adaption loops [25].

4.3.2 Event-driven programming

Event-driven programming is applied in many applications such as user interfaces, discrete systems and business module simulations [13]. "An important characteristic of event-driven computation is that control is relinquished to a library that waits for events to occur. Each event is then dispatched to the application by invoking a handler function or a handler object for appropriate action" [26]. Event-driven programming may be applied in three ways [26]:

- Event loops: explicitly dispatching on an event to raise the appropriate application code.
- Callback functions: implicitly (table-based) dispatching based on an association between a callback function and the type of event. Callback is registered when a program cannot complete an operation because it has to wait for an event. A callback executes indivisibly until it hits a blocking operation and then it registers a new callback and returns [13].

- Listener objects: callback on objects with hook methods that are invoked on the occurrence of an event. Listener objects are more powerful than callback function since they rely on ad-hoc mechanisms to take the history of event occurrences into account.

Event-driven programming is a technique that the user can use to trigger a program in an arbitrary order [27]. The characteristics of event-driven programming encouraged software engineers to use it for obtaining robust programs.

Frank et al. [13] modified an asynchronous programming library to take advantage of multi-processes. The modified library presents a model for execution on multiple CPUs, thus avoiding most of the synchronization complexity of a threaded programming model.

Event-based programming can provide a convenient programming model, which can also be extended to take advantage of multi-processors. Events are better for managing I/O concurrency in server software than threads because events have less complexity and produce more robust software. In addition, event-driven programming has an advantage over threads since it provides a convenient programming model that is naturally robust. The event-driven model can be extended to exploit multi-processors with minor changes to the code. However, event-driven program structure has a series of small callback functions, which rely heavily on dynamic memory.

5. ROBUSTNESS TESTING

Robustness testing checks whether robust programming techniques succeed in satisfying the robustness requirements of the program. The main terms that are used in robust testing are mentioned in [1, 20]:

Testing can only reveal robustness errors in successful test cases. A robust error is defined as an inrobust reaction a test case produces during its execution. Inrobust reactions are observed when test objects crash or hang. A test object in this context is a software component that may be a procedure, a function, or a method via a non-empty list of parameters.

The importance of software robustness drives researchers to develop different techniques and tools to test software robustness. Some testing software robustness tools and techniques are discussed below.

5.1 Interface Robustness testing tools

Fuzz [28] is a simple automatic method where a random input stream is used as a robustness testing method. Nine versions of UNIX operating system and X-Window applications were tested using this method. The failures identified and categorized were crash (with core dump) or hang (infinite loop).

The results show that, in the worst case, over 40% of the basic programs and over 25% of the X-Window applications crashed or hanged. They were neither able to crash any of the network services that they tested nor any of the X-Window servers.

The Riddle tool [29] was used to test the robustness of Windows NT. Two different approaches were examined to generate data (generic data generation and intelligent data generation) to be used for automated robustness testing. They concluded that this tool is useful for constructing both generic data and intelligent data where they discovered a new kind of failures.

Ballista [30, 31] is an automated robustness testing tool designed to exercise commercial off-the-shelf software components. Ballista is a methodology and a web server that remotely test software modules in linkable object code form. Ballista's purpose is to identify sets of input parameters that cause robustness failure in the software components being tested. Ballista testing begins with identifying the data types used by an application programming interface under test. Application-specific data types can inherit base test cases from predefined data types in the Ballista testing tool set. Then, the Ballista test harness generator is given the signature for a function to be tested in terms of those data types, and it generates a customized testing harness. The test harness composes combinations of test values for each parameter and reports robustness testing results.

Interface robustness testing is where the success criteria in most of the cases is “if it does not crash or hang, then it’s robust.” Another kind of robustness testing method used is the dependability benchmark explained next.

5.2 Dependability benchmark robustness testing tools

The dependability benchmark defines benchmarks to characterize the system behaviour under normal loads and faults. The goal of benchmarking the dependability of computer systems is to provide generic ways for characterizing their behaviour in the presence of faults [32]. Some tools that used this technology to develop a robustness tester tool include DBench and IBM Autonomic Computing benchmarks. The DBench project aimed at defining a conceptual framework and an experimental environment for dependability benchmarking [32]. The IBM Autonomic Computing benchmark focuses on the resiliency of the system against various disturbance [33, 34].

6. Robustness Measurement

Measurement as an activity is used in everyday life, such as in supermarkets, clothing stores, and driving trips, where the prices, sizes, distances, etc. are measured to help the making of decisions. Software measurement is often known as software matrices, software engineering matrices, or software metrication [35].

In general, measurement is “the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules” [36]. The entities in the definition are the objects to be measured, and the attributes are the objects, the entity features, or entity properties.

By applying the previous definition on software, software measurement is defined as “a quantified attribute of a characteristic of a software product or the software process. It is a discipline within software Engineering” [37]. The software matrices term is related to software measurement. Software matrices measure some properties of a piece of software or its specifications. There are many software matrices, where the varieties of these matrices depend on the variety of the targeted software attributes to be measured.

The robustness, as a part of software characteristics, can also be measured. By using the measurement definition, the robustness measurement can be defined as “the process by which relative numbers are assigned to robustness degree of a C program in such a way to describe them according to MISRA C2 rules and their Weights.” [45]

6.1 Software Measurement Classification

There are different measurement classifications depending on the features or the programs being measured. There are measurement techniques for software quality, software complexity, software validity, and for object-oriented programs. Measurements can be classified into static and dynamic measurements [38]. In this paper, only robustness measurement is discussed further.

6.2 Robustness Measurement

Critical programs must be robust to avoid the problems that could be caused by failures [39]. The C language standards were introduced to avoid code misinterpretation, misuse, or misunderstanding. The IEEE presented the ISO/IEC 9899:1999 standard [40], which was later used by MISRA to produce MISRA C1 and C2. This in turn led to Jones producing "The New C Standard: An Economic and Cultural Commentary" [41]. The LDRA company uses MISRA C rules in addition to 800 rules it created to assess programs. Other C standards such as "C programming language coding guideline" [42] are less frequently used.

Measuring the application of a language standard to a program is one technique of program robustness measurement. Several techniques were used to measure program robustness. Software measurement could mean estimating the cost, determining the quality, or predicting the maintainability [36]. Arup and Daniel [43] presented features, such as portability, to evaluate some existing benchmarks of Unix systems. As a result, they built a hierarchy-structured benchmark to identify robustness issues that were not detected before. Behdis and Shokat [44] introduced a theoretical foundation for robust matrices that reduce the uncertainty in distributed system. Arne et al. [45] used some robustness criteria such as input date rate and CPU clock rate to create multi-dimensional robustness matrices and then use them to measure the robustness of a system.

A robustness hierarchy is a relative scale to find the robustness characteristics that need to be added to programs. It is a technique used to build a robust program. The hierarchy starts with a non-robust program as the first step and then adds robust features before reaching a robust program in the highest level of the Hierarchy [46].

The above software measurement techniques do not give the developer a fully detailed measurement. Furthermore, they do not specify the parts of a program that need to be modified to raise the quality of the program. Therefore, the robustness grid was developed [45]. It utilizes a full description of all robustness features and their satisfaction degree. It also allows the developer to specify the code lines that need to be modified to improve the program robustness degree [45].

7. Conclusions

Program robustness is necessary for all types of programs, especially the critical one. It ensures that the program will either work or fail gracefully. Many techniques were used to create robust programs. These techniques differ depending on what and how they do it. The robustness of a program can be measured by some tools or methods. Such methods aim to find the defect in a program to help the maintainers fix it and make it more robust.

References

[1] IEEE Standard Glossary of Software Engineering Terminology, 1990.

[2] J. C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definition and analysis of hardware- and software-fault-tolerant architectures," Computer, vol. 23, pp. 39-51, 1990.

[3] 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, vol. 1, pp. 11-33, 2004.

[4] W. S. Jawadekar, Software Engineering: Principles and Practice: Mcgraw Hill Higher Education, 2004.

[5] J. C. Laprie, "Dependable computing: concepts, challenges, directions," in Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International, 2004, p. 242 vol.1.

[6] R. S. Pressman, Software Engineering: A Practitioner's Approach, Seventh edition ed.: McGraw Hill Higher Education, 2009.
[7] I. Sommerville, Software Engineering: Addison-Wesley, 2006.
[8] D. M. John, I. Anthony, and O. Kazuhira, Software reliability: measurement, prediction, application: McGraw-Hill, Inc., 1987.
[9] L. L. Pullum, Software fault tolerance techniques and implementation: Artech House, Inc., 2001.
[10] D. G. Steven, "Robustness in Complex Systems," presented at the Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, 2001.
[11] G. M. Weinberg. (1983) Kill That Code! Infosystems. 48-49.
[12] D. John and Philip J. Koopman, Jr., "Robust Software - No More Excuses," presented at the Proceedings of the 2002 International Conference on Dependable Systems and Networks, 2002.
[13] D. Frank, Z. Nickolai, K. Frans, M. David, res, and M. Robert, "Event-driven programming for robust software," presented at the Proceedings of the 10th workshop on ACM SIGOPS European workshop, Saint-Emilion, France, 2002.
[14] Y. Bi, J. Yuan, and Y. Jin, "Beyond the Interconnections: Split Manufacturing in RF Designs," Electronics, vol. 4, p. 541, 2015.
[15] Y. Bi, X. S. Hu, Y. Jin, M. Niemier, K. Shamsi, and X. Yin, "Enhancing Hardware Security with Emerging Transistor Technologies," presented at the Proceedings of the 26th edition on Great Lakes Symposium on VLSI, Boston, Massachusetts, USA, 2016.
[16] Y. Bi, K. Shamsi, J.-S. Yuan, P.-E. Gaillardon, G. D. Micheli, X. Yin, X.S. Hu, M. Niemier, Y. Jin, "Emerging Technology-Based Design of Primitives for Hardware Security," J. Emerg. Technol. Comput. Syst., vol. 13, pp. 1-19, 2016.
[17] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M. Violante, "Soft-Error Detection through Software Fault-Tolerance Techniques," in IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems, 1999.
[18] R. L. Michael, H. Zubin, K. S. S. Sam, and C. Xia, "An Empirical Study on Testing and Fault Tolerance for Software Reliability Engineering," presented at the Proceedings of the 14th International Symposium on Software Reliability Engineering, 2003.
[19] N. H. Michael and T. H. Vance, "Robust Software," IEEE Internet Computing, vol. 6, pp. 80-82, 2002.
[20] M. Dix and H. D. Hofmann, "Automated software robustness testing - static and adaptive test case design methods," in Euromicro Conference, 2002. Proceedings. 28th, 2002, pp. 62-66.
[21] N. H. Michael, T. H. Vance, and G. Rosa Laura Zavala, "Robust software via agent-based redundancy," presented at the Proceedings of the second international joint conference on Autonomous agents and multiagent systems, Melbourne, Australia, 2003.
[22] T. Rajesh and N. H. Michael, "Multiagent Reputation Management to Achieve Robust Software Using Redundancy," presented at the Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2005.
[23] V. T. Holderfield and M. N. Huhns, "A Foundational Analysis of Software Robustness Using Redundant Agent Collaboration," in Agent Technologies, Infrastructures, Tools, and Applications for E-Services. vol. 2592/2003, ed Berlin / Heidelberg: Springer, 2003, pp. 355-369.
[24] R. Laddaga. (1999, May/June) Creating Robust Software through Self-Adaptation. IEEE Intelligent systems. 26-30.
[25] M. K. Mieczyslaw, B. Kenneth, and A. E. Yonet, "Control Theory-Based Foundations of Self-Controlling Software," vol. 14, ed: IEEE Educational Activities Department, 1999, pp. 37-45.
[26] C. Petitpierre and A. Eliëns, "Active Objects Provide Robust Event-Driven Applications," in SERP'02, Las Vegas, 2002, pp. 253-259.
[27] G. C. Philip, "Software design guidelines for event-driven programming," Journal of Systems and Software, vol. 41, pp. 79-91, 1998.
[28] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, J. Steidl, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services," Report: University of Wisconsin, 1995.
[29] M. Schmid and F. Hill, "Data Generation Techniques for Automated Software Robustness Testing," in Proceedings of the International Conference on Testing Computer Software, 1999, pp. 14-18.
[30] J. P. DeVale, P. J. Koopman, and D. J. Guttendorf, "The Ballista Software Robustness Testing Service," presented at the Tesing Computer Software Coference, 1999.
[31] P. Koopman. (2002, 2nd September). The Ballista Project: COTS Software Robustness Testing. Available: http://www.ece.cmu.edu/~koopman/ballista/index.html

[32] K. Kanoun, H. Madeira, and J. Arlat, "A Framework for Dependability Benchmarking," presented at the The International Conference on Dependable Systems and Networks, Washington, D.C., USA, 2002.
[33] A. B. Brown and P. Shum, "Measuring Resiliency of IT Systems," presented at the SIGDeB Workshop, 2005.
[34] A. B. Brown, J. Hellerstein, M. Hogstrom, T. Lau, S. Lightstone, P. Shum, M. Peterson, "Benchmarking Autonomic Capabilities: Promises and Pitfalls," in International Conference on Autonomic Computing (ICAC'04), Los Alamitos, CA, USA, 2004, pp. 266-267.
[35] H. Zuse, A Framework of Software Measurement: Walter de Gruyter, 1998.
[36] N. E. Fenton and S. L. Pfleeger, Software Metrics, A Rigorous and Practical Approach, 2 ed.: PWS Publishing Company, 1997.
[37] ISO/IEC 15939: Systems and software engineering -- Measurement process, ISO/IEC, 2007.
[38] K. Kaur, K. Minhas, N. Mehan, and N. Kakkar, "Static and Dynamic Complexity Analysis of Software Metrics," Empirical Software Engineering, vol. 56, pp. 159-161, 2009.
[39] D. M. Jones, The New C Standard: A Cultural and Economic Commentary, 1st edition ed.: Addison-Wesley Professional, 2003.
[40] International Standard ISO/IEC 9899, 1999.
[41] D. M. Jones, The New C Standard: An Economic and Cultural Commentary, 2002.
[42] C programming language coding guidelines, www.lrdev.com, 1998.
[43] M. Arup and P. S. Daniel, "Measuring Software Dependability by Robustness Benchmarking," vol. 23, ed: IEEE Press, 1997, pp. 366-378.
[44] B. Eslamnour and S. Ali, "Measuring robustness of computing systems," Simulation Modelling Practice and Theory, vol. 17, pp. 1457-1467, 2009.
[45] H. Arne, R. Razvan, and E. Rolf, "Methods for multi-dimensional robustness optimization in complex embedded systems," presented at the Proceedings of the 7th ACM & IEEE international conference on Embedded software, Salzburg, Austria, 2007.
[46] M. Abdallah, M. Munro, and K. Gallagher, "Certifying software robustness using program slicing," in 2010 IEEE International Conference on Software Maintenance, Timisoara, Romania, 2010, pp. 1-2.

Authors

Dr. Ayman M. Abdalla has been a member of the Faculty of Science and Information Technology at Al-Zaytoonah University of Jordan since 2001, where he held different positions including the Chair of the Department of Multimedia Systems. He received his Ph.D. in computer science from the University of Central Florida, FL, USA; and his Master's and Bachelor's degrees in computer science from Montclair State University, NJ, USA. He has experience in research and teaching in the USA and Jordan in addition to working in software development in a company in the USA.

Dr. Mohammad Abdallah has been a member of the Faculty of Science and Information Technology at Al-Zaytoonah University of Jordan since 2012, where he held different positions including the Chair of the Department of Software Engineering, and now, he is the Chair of the Department Computer Science. He received his Ph.D. in 2012 from Durham University, UK; his Master's degree in software engineering from the University of Bradford, UK; and his Bachelor's degree in computer science from Al-Zaytoonah University of Jordan, Amman, Jordan.

Mr. Mosa Salah has been a member of the Faculty of Science and Information Technology at Al-Zaytoonah University of Jordan since 2008. He received his Master's degree in computer science from the Arab Academy for Finance and Banking, Amman, Jordan; and his Bachelor's degree in computer science from Al-Zaytoonah University of Jordan, Amman, Jordan.