IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 16, 2016, accepted March 7, 2016, date of publication March 22, 2016, date of current version May 11, 2016.

Digital Object Identifier 10.1109/ACCESS.2016.2544951

Robustness Testing of Embedded Software Systems: An Industrial Interview Study

SYED MUHAMMAD ALI SHAH[1], DANIEL SUNDMARK[1,2], BIRGITTA LINDSTRÖM[3], AND STEN F. ANDLER[3]

[1]Swedish Institute of Computer Science, Swedist ICT AB, Kista 164 29, Sweden
[2]Mälardalen University, Västerås 721 23, Sweden
[3]University of Skövde, Skövde 541 28, Sweden

Corresponding author: S. M. A. Shah (shah@sics.se)

This work was supported by the Knowledge Foundation through the Testing of Critical System Characteristics under Grant 20130085.

ABSTRACT Embedded software is at the core of current and future telecommunication, automotive, multimedia, and industrial automation systems. The success of practically any industrial application depends on the embedded software system's dependability, and one method to verify the dependability of a system is testing its robustness. The motivation behind this paper is to provide a knowledge base of the state of the practice in robustness testing of embedded software systems and to compare this to the state of the art. We have gathered the information on the state of the practice in robustness testing from seven different industrial domains (telecommunication, automotive, multimedia, critical infrastructure, aerospace, consumer products, and banking) by conducting 13 semi-structured interviews. We investigate the different aspects of robustness testing, such as the general view of robustness, relation to requirements engineering and design, test execution, failures, and tools. We highlight knowledge from the state of the practice of robustness testing of embedded software systems. We found different robustness testing practices that have not been previously described. This paper shows that the state of the practice, when it comes to robustness testing, differs between organizations and is quite different from the state of the art described in the scientific literature. For example, methods commonly described in the literature (e.g., the fuzzy approach) are not used in the organizations we studied. Instead, the interviewees described several *ad hoc* approaches that take specific scenarios into account (e.g., power failure or overload). Other differences we found concern the classification of robustness failures, the hypothesized root causes of robustness failures, and the types of tools used for robustness testing. This paper is a first step in capturing the state of the practice of robustness testing of embedded software systems. The results can be used by both researchers and practitioners. Researchers can use our findings to understand the gap between the state of the art and the state of the practice and develop their studies to fill this gap. Practitioners can also learn from this knowledge base regarding how they can improve their practice and acquire other practices.

INDEX TERMS Testing, interviews, robustness, embedded systems, survey, state of the practice, state of the art.

I. INTRODUCTION

Among various techniques used to verify and validate a software system, testing is most frequently used for evaluating the quality of software [1]. If properly conducted, testing may provide an efficient and rigorous way for error identification [2]. Testing of embedded software systems is a great challenge [1], as there are various characteristics of such systems that need to be considered while performing testing. In particular, Qian and Zheng [3], identify four characteristics that distinguish testing of embedded software system from testing of general software.

First, embedded software systems are developed to perform a specific task in a specific environment, and the challenge for the tester is to test the system in a host-based or target-based context. Second, interaction is an important characteristic in the operation of embedded software systems. Embedded systems typically operate in, and interact with, an external environment by collecting data through sensors and acting upon this collected data through actuators. Failure of recreating this interaction during testing may lead to inadequate or erroneous conclusions. The third characteristic of embedded software systems is the

development practice, where various interfaces and supporting platforms play important roles. The fourth distinguishing characteristic of embedded software systems is timeliness, meaning that the correctness of the embedded systems' behavior depends not only on *what* the system does but *when* it does so. Hence, timeliness adds an extra dimension to testing. Traditional software testing methods are useful for identifying functional errors and attaining a high test coverage in embedded software systems, but may not be comprehensive enough to uncover robustness problems that occur because of environment errors such as unexpected or erroneous input values [4]. Robustness is defined by the IEEE standard 610.12-1990 as "*The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions*" [5].

Characteristics like robustness are particularly challenging for embedded software systems as their execution environments cannot fully be foreseen at the time of development [2]. Robustness testing requirements can range from very general considerations to more specific requirements. General considerations are, for example, no run-time errors, no crashes or no deadlocks. More specific requirements are, for example, the capability of a system to always return to a nominal state after entering a degraded state, or that some system resources remain available for high-priority tasks [2]. However, there could be many interpretations of robustness testing and hence, many differences in how it is performed in various contexts. As described by Huhns and Holderfield [6], robustness testing of software system concerns how to test its ability to avoid crash. Fault tolerance as a concept is also widely used in robustness. Fault tolerance techniques are used "to meet design dependability requirements" [7]. Therefore, robustness testing often aims to assess the systems' fault tolerance.

Considering most of the definitions related to robustness, the main aim is to test the capability of the software system to handle adverse situations caused by internal or external factors [8]. In addition, the relation of robustness to other dependability attributes such as reliability, stability, availability, and performance is disputed [8]. For example, in the EMISQ model [9], robustness is described as part of the reliability of a system, even though it is not explicitly mentioned in the ISO 9126 standard [10].

The understanding of robustness within the scientific community seems relatively consistent. The knowledge on state of the practice in robustness testing of embedded systems is however not very mature. To our knowledge, Eldh and Sundmark [8] performed the only industrial case study so far to identify the common practices of robustness testing in large-scale telecom systems. They identified key challenges related to the understanding of robustness at different levels such as design level, unit level, and system level. From their work it could be emphasized that there is sometimes a lack of understanding of robustness in industry and that the view of robustness may vary considerably between companies and industrial contexts. As we know so little about state of the practice in various contexts, we do not have a clear view of the gap between state of the art and state of the practice. Because of this, there is a risk that researchers do not address relevant and urgent problems or that they fail to define feasible solutions.

The objective of this study is to explore state of the practice of robustness testing of embedded software systems and to compare this to the state of the art. The contribution of this study is, therefore, twofold: First, it highlights the industrial practices of robustness testing of embedded software systems, and contributes in building the knowledge about robustness testing. Second, it allows us to highlight the gaps and differences between the theoretical (academic) and the practical (industrial) knowledge of the subject.

This paper presents the results of an empirical study conducted in seven different embedded software system domains. The study is exploratory and aims at investigating the current state-of-the-practice robustness testing of embedded software systems. Data acquisition is performed through semi-structured interviews with one or multiple interviewees for each industrial domain, and the data is then analyzed through a qualitative data analysis technique. Results of the study indicate that the industrial practice of robustness testing is clearly distinct from what has been previously described in the scientific literature. We describe the state of the practice and identify the gap between state of the art and state of the practice of robustness testing.

The paper is structured as follows. In Section II, related work is presented in detail. In Section III, the research method is presented, and Section IV discusses results with a focus on various challenging areas as described. In Section V the discussion on results is made and validity of the findings is analyzed. Finally, in Section VI the conclusions are summarized with indications of future research.

II. RELATED WORK

This section describes related work on robustness testing of embedded software system.

A. DEFINITION OF ROBUSTNESS

Robustness is often considered to be a quality for achieving higher dependability [11]. In the scientific literature, software robustness is defined explicitly, e.g. by the 610.12-1990 IEEE standard glossary of software engineering terminology [5]. However this not the only definition of robustness; considering the work of Lei et al., the notions such as invalid inputs and stressful environment conditions can be ambiguous depending on the value of the invariant and the precondition before execution [12]. Consequently, alternative definitions exist: Lussier et al. [13] define robustness as "*the delivery of a correct service in implicitly-defined adverse situations arising due to an uncertain system environment (such as an unexpected obstacle or a change in lightning condition affecting sensors)*". Shahrokni and Feldt define robustness considering the industrial context by stating that "*robustness is interpreted as stability in presence of*

erroneous input and execution stability in presence of stressful environment created by external services or modules'' [11]. In general, the widespread informal definition of robustness is that a system should show acceptable behavior in spite of exceptional or unforeseen operating conditions [2].

B. FAULT INJECTION

Fault injection has been used as a common practice for robustness testing [14], [15]. Fault injection is a technique for testing the response of a system to a fault that is artificially induced. Two approaches have mainly been used when testing robustness by means of fault injection. The *fuzz* approach tests software with random and crafted streams of inputs, looking for system crashes. The aim of this approach is to quantify the software systems from an interactive user's viewpoint [14]. The other approach, called *benchmarking*, uses fault injection by passing a combination of exceptional inputs as a parameter through the API of the software under test to detect crashes and hangs [15].

C. COMMON ROBUSTNESS FAILURES AND REASONS

In the scientific literature, robustness testing of particularly operating systems has been studied exclusively [4], [15]–[19]. Notably, failures in an operating system context have been characterized by means of the CRASH benchmark. CRASH is a five-level categorical description of robustness failures i.e. Catastrophic, Restart, Abort, Silent and Hindering. The CRASH failures are defined in [15] as:

- Catastrophic failure: *"The Catastrophic class of failure occurs when a failure is not contained within a single task. In other words, this level of failure means that a call to an OS function has caused other tasks, or even the system itself, to crash or hang. A Catastrophic failure typically requires a hardware reset of the entire system, but may possibly be limited to a warm restart of the OS"*.
- Restart failure: *"The Restart class of failure occurs when a single task hangs, resulting in the need to kill and restart that task to return to normal execution"*.
- Abort failure: *"The Abort class of failure occurs when a single task experiences an abnormal termination. A typical abnormal termination is caused by a segmentation violation, in which the task attempts to access memory to which it does not have access permissions (for example, by dereferencing a null pointer)"*.
- Silent failure: *"The Silent class of failure occurs when invalid parameters are submitted to an OS call, but neither an error return code nor other task failure is generated. For example, a call to open a file with a NULL filename might return a success flag, instead of an error flag"*.
- Hindering failure: *"The Hindering class is so named because the OS is hindering correct diagnosis of a problem by providing an incorrect error code. For example, an invalid memory access code returned when the only erroneous input is an invalid file handle value would be a hindering-class failure"*.

While many studies characterize robustness failures according to CRASH, some of these studies also highlight the reasons for these failures. Concerning the 'Catastrophic' failures, Lei et al. [20] studied state-based robustness testing of components and found that if the service replies with an error message or exception that indicates the occurrence of an unexpected internal problem, then this might lead to a catastrophic failure. In addition they also found that an unchecked exception thrown to a script without being declared, and user-defined checked exceptions, which indicates the component is in an incorrect state and not performing its functionality, can lead to catastrophic failures. Schmid et al. [21], found after evaluating the robustness of Windows NT software that if a program fails to handle an exception thrown by an OS function or a divide-by-zero exception it is most likely to crash. Koopman and DeVale [18] studied the exception handling effectiveness of the POSIX operating system and found that system calls with exceptional parameter values can lead to catastrophic failures.

Concerning the 'Restart' failure, Fernsler and Koopman [19], found that internal errors such as failure of any function that required killing any task, lead to restart failure. Similarly, concerning the 'Abort' failure, Koopman and DeVale [16] in their study of comparing various POSIX operating systems based on robustness found that data types such as invalid file pointers (excluding null), null file pointers, invalid buffer pointers, minint integers, and maxint integers are associated with abort failures. In addition Koopman and DeVale in another study [18] found that if a signal is sent from the system call or library function to itself, causing an abnormal task termination, this could lead to the abort failure. Concerning 'Silent' failures, Koopman and DeVale [18] also found that if exceptional inputs to a module under test resulted in erroneous indication of successful completion, it might lead to a silent failure. It should however be noted that the above results may be highly context-specific, and it is difficult to draw any general conclusions regarding the extent to which certain types of faults are more prone to cause robustness failures.

In the scientific literature, there are some identified causes of robustness failures that have not been linked to any particular category of CRASH. For example Fernsler and Koopman found in their study [19] that internal exception handling errors (actually a segmentation violation caught semi-gracefully), unknown exceptions (exception-handling software defects) and segmentation faults (exceptions that evaded the exception handlers) are reasons for robustness failures. Miller et al. [14], found many reasons for robustness failures. For example, errors in the use of pointer and array subscripts dominate to produce such failures. They found that dangerous input functions, such as the gets() function is a common root cause of robustness failures, because gets() has no parameter to limit the length of the input data. In addition they also found that conversion of numbers from one size to another would result in a robustness failure. Lei et al. [12] found that particular conditions such as division by zero,

integer overflow null reference, and array out of bounds access can result in a robustness failure.

D. ROBUSTNESS TESTING TOOLS

There are some tools that have been used and studied for robustness testing. The most often studied tool is the Ballista tool, which is used to test the robustness of operating systems [16]–[19]. In addition, the tool Riddle has been used to test robustness of Windows NT [22]. For the robustness of UNIX utilities and services, the tool Fuzz has been used [14], [23]. It should be noted that most of these tools are quite dated. Considering various application domains, different tools have been developed and studied in the scientific literature. For example, for Java-based applications, the automatic Java-based JCrasher tool is used to test the robustness of Java programs [24]. In addition Robut, a stated-based robustness testing tool has been used to test robustness of Java components [20]. The automated tool WebSob has been used for robustness testing and response analysis of web services [25].

E. ROBUSTNESS TESTING IN DIFFERENT DOMAINS

In the scientific literature, most evidence shows that robustness is mainly performed and studied for operating systems with the objective to test the operating systems' dependability [4], [14], [26]. In addition, some studies perform a dependability comparison between operating systems based on their robustness [15]–[18]. There are few studies that focus on testing robustness in other domains than operating systems. One of the few examples is from the telecommunication systems domain, where Eldh and Sundmark [8] performed a case study on how robustness testing is performed in mobile telecommunication systems. Another example from the same context is Johansson et al. [27], who developed T-Fuzz, a novel fuzzing framework for testing robustness of telecommunication protocols. Concerning middleware systems, two studies [28], [29] describe a methodology for executing robustness tests on high availability middleware solutions. Lei et al. [12] highlighted a method to perform robustness testing of components using a semantic model. Similarly, Ali et al. [30] model the system using the aspect-oriented modeling technique to support robustness testing. They use a video conferencing system for their studies. Concerning web based projects, Laranjeiro et al. [31] use text classification algorithms that are applied to test the web services' robustness. Finally, Belli et al. [32] propose a model using event sequence graphs (ESG) and decision tables (DT) for robustness testing of web based system.

In summary, it can be concluded from the literature that:

- Most existing work in robustness testing focus on the robustness of software, but very few of these studies focus on testing the robustness of embedded software systems,
- Except one, all studies focus on the state-of-the-art perspective. Not much is known about the state of the practice of robustness testing.
- State-of-the-art only defines the robustness testing of specific domains. Therefore it is challenging to have this information from different domains.

III. RESEARCH METHOD

In this section, the research method is presented. The description includes the research objectives, the research method, the preparation, the selection of participants and study instruments, and how data from the interviews have been collected, extracted and analyzed.

A. RESEARCH OBJECTIVE AND DESIGN

1) DEFINITION OF OBJECTIVES

The overall objective of this study is to understand the state of the practice of robustness testing of embedded software systems, and build empirical knowledge about it. Particularly this study aims to address the following research question:

RQ: What is the state of the practice in robustness testing of embedded software systems?

2) SELECTION OF METHOD

The research question tries to explore what is happening in industry concerning robustness testing, seeking hidden and new insights, and based on that generating new ideas and hypothesis for future research. Therefore to answer the research question, a research method that is exploratory in nature is required [33].

3) DEFINITION OF CASE STUDY

An exploratory multiple case study design has been used. When conducting this study we have used the guidelines described by Runeson and Höst [33], detailing five major process steps that we walk through as shown in Table 1.

B. PREPARATION

In this section, the strategies for data collection (what, why, how, when) are defined.

1) DEFINING OF INTERVIEW QUESTIONS

To define interview questions we adopted steps described in [34] to work in a structured and iterative manner. Initially, as summarized in Section II, the state-of-the-art knowledge was surveyed to understand the existing knowledge of robustness testing. Next, key areas were identified and interview questions were developed by considering the main aspects of robustness testing found in scientific literature. The interview questions were developed in iterative fashion, exploiting the perceptions, opinions, experiences and beliefs of all the co-authors of this paper. The development of the interview questions took three months of time during which all authors contributed in the review and iterative refinement of the interview questions. The work was coordinated by the first author of this paper. The interview questions were designed such that the answers would provide insight into the state of the practices in robustness testing of embedded software systems. Most of the interview questions were open-ended to give an

TABLE 1. Overview of the study.

Objective and Design	Preparation	Evidence Collection	Evidence Analysis	Reporting
• Definition of objectives • Selection of method • Definition of case study	• Defining of interview questions • Selection of interviewees • Pilot interview	• Interviews • Evidence collection Method	• Notice, Collect and Think technique	• Textual summary

opportunity to ask follow-up questions and thereby encouraging exploratory discussions. Table 2 presents the interview questions used in this study, categorized by important aspects of robustness testing of embedded software systems.

2) SELECTION OF INTERVIEWEES

Two channels were used in selection of interviewees, some from industrial contacts in our network and some from browsing profiles of experts of robustness testing on LinkedIn.[1] We consequently made use of a convenience sampling for the sake of interviewee selection. The focus was to select experts located in a sufficient proximity to the first author (Stockholm, Sweden) to allow for face to face interviews.

The interview subjects were selected based on their experience of robustness testing of embedded software system. Software testers and test managers are considered in this regard to make this study authentic and reliable. In total thirteen interviews were conducted. Twelve of the interviewees were male and only one was female. All interviewees have at least five years' experience of testing embedded software in general and at least three years' experience of robustness testing of embedded software systems in particular, as shown in Table 3.

3) PILOT INTERVIEW

One pilot interview was conducted, with the intention to evaluate that the interview questions are sufficiently easy to understand. The pilot interview did not result in any major changes to the interview questions, as the subject expressed no difficulties in understanding the questions and answering them. However the interviewee made us aware of some minor improvements with respect to the interview protocol and pre-interview information. These were subsequently altered accordingly.

C. EVIDENCE COLLECTION

This section describes how the evidence is collected. The conduction of the interviews as well as the analysis of the responses are described in detail.

1) INTERVIEW

The interview was conducted in two sessions; the *training session*, and the *interview execution session*. The training session lasted for about ten minutes in which the brief agenda of conducting this particular study and associated research objectives were explained to the interviewees. In addition, we explained the existing definitions of software robustness present in the scientific literature. Afterwards, the interviewees were requested to ask clarifying questions in order to remove any ambiguity related to the interview. The execution session was the one in which the formal interview was conducted. The open-ended interview questions gave opportunity to lead a good discussion and gave us enough freedom to ask follow-up questions. This also led to further exploration from the contextual viewpoint, where the interviewees started to give examples from their specific context.

2) EVIDENCE COLLECTION METHOD

The evidence was collected by two means, manually taking notes of each interview question and also by recording the interviews using a digital recorder. This method of evidence capturing was chosen to not lose any relevant information and to allow for all authors to hear the responses and participate in the analysis work. All interviews were conducted in English. The first author conducted all interviews. Eleven interviews were conducted face to face, one interview was conducted by phone and one was by Skype. The duration of the interviews varied between 35 and 60 minutes.

D. EVIDENCE ANALYSIS

The evidence from the interviews consists of the written notes and the recorded audio. The sequence number and categorization of interview questions helped us to first process the interview questions in a spread sheet. Thereafter we added the written notes of each interview for each corresponding interview question.

We reviewed all the audio contents of the interviews and added relevant information that were not captured in the handwritten notes. In some cases, a few statements directly constituted the answers to the interview questions and these answers were extracted from the transcribed interviews and inserted in the spreadsheet as they were. However for the detailed answers and examples we applied the ''Notice, Collect, and Think'' technique on the transcribed outputs of the interviews [35]. This is a non-linear qualitative analysis model, which consists of three phases; noticing, collecting and thinking phases. The analysis work was divided among

[1] https://www.linkedin.com/

TABLE 2. Interview questions.

Robustness Testing Aspects	Interview Questions
Definition of robustness	"610.12-1990 IEEE "*The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions*" SI1: Do you agree with the given IEEE definition of robustness? If not, what is your definition? SI2: What are the key aspects of a robust software system in your domain?
Requirements for robustness testing	SI3: What are robustness requirements in general? SI4: Do you have specific examples of robustness requirements? SI5: Where do they come from? Customer? Regulatory standards? Own organization?
Robustness test design	SI6: How do you ensure that you have designed a test for robustness? What would you do in testing e.g. models, policies etc.
Performing robustness testing	SI7: Describe how you perform software system robustness testing in practice? SI8: Do you have specific staff member(s) in your organization who work on robustness (testing?) of a software/system? In what roles? If not, who is responsible? SI9: Do you explicitly measure robustness in your software system? If yes, how? S10: If you test specifically for robustness, do you run these tests on the target hardware, in a non-target software testing environment, or both?
Robustness failures	SI11: In your system(s), what would you say are the most common types of robustness failures? SI12: Do you classify failures? How? SI13: What things (faults) can cause such failures?
Tools used for robustness testing	SI14: Which tools do you use for test execution, design, preparation and verdict for robustness testing? SI15: What are the functions of tools that are most useful, with respect to robustness testing? SI16: What are the functions in tools that are missing? SI17: What type of support (tool or other) would you ideally like to use with respect to robustness testing?
Additional	SI18: Is there something that you want to add additionally considering the asked questions?

the authors so that at least two researchers independently listened to the interviews and contributed to the analysis of each robustness testing aspect (see Table 2). The results described in Section IV are based on a mutual agreement for each such aspect, i.e., in each subsection. In the 'Notice' phase, the detailed points, reasoning, and examples as highlighted by the interviewees are considered. In the 'Collect' phase, the similar and different answers of each domain are arranged and grouped together. In the 'Think' phase, analysis was carried out by applying critical thinking on the grouped answers to extract meaningful findings. To avoid subjective findings from the interviews at least two researchers independently contributed to the analysis of each result subsection that laid the ground for the mutual agreement of findings.

TABLE 3. Overview of interviews, companies, and participants.

ID	Company		Interviewee Role	Experience (years)	
	Domain	Size (S,M,L)		Robustness testing	Total testing
1	Telecom	L	Test Manager	5+	15+
2	Telecom	L	Program Manager	5+	10+
3	Telecom	S	Tester	3+	5+
4	Telecom	M	Test Manager	15+	25+
5	Automotive	L	Test Manager	4+	7+
6	Automotive	M	Test Director	5+	20+
7	Multimedia (IPTV)	L	Tester	3+	5+
8	Multimedia (IPTV)	M	Tester	3+	5+
9	Multimedia (IPTV)	S	Tester	3+	5+
10	Aerospace	L	Test Manager	5+	15+
11	Banking	L	Test Director	5+	20+
12	Consumer Products	S	Tester	3+	5+
13	Critical Infrastructure (Energy Distribution)	L	Test Manager	5+	25+
			Average	5+	12.5+

IV. RESULTS

Below, we provide the summarized synthesis of the answers to the interview questions, organized by the previously derived key aspects of robustness testing.

A. DEFINITION OF ROBUSTNESS TESTING

The IEEE 610.12-1990 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*") is generally accepted by the respondents, but in most cases we could see that the respondents either did not use any definition in their organization or that they had additions. In particular, the following statements were made by the interviewees:

- *"It is acceptable if the system fails but it is troublesome if the system accepts everything and does not fail. The system failing and then recovering in all circumstances is robustness to me"*
- *"Robustness is not only handling a stressful environment, but handling a misbehaving environment including any connected equipment"*
- *"Whatever the input is, the system should never provide faulty output, but rather go in safe or idle state"*

Moreover, several interviewees mentioned the system's ability to cope with noise, attacks and malicious data. The responses from the aerospace and automotive domain stated that they normally do not use the term robustness. Robustness, in these domains, is a property that is connected to safety and the system should end up in a safe state in response of a failure. Robustness is handled implicitly by following standards such as DO-178B, rather than addressed explicitly as a non-functional property. The interviewees from the multimedia and telecom domains stated that they use the term robustness even though they, in most cases, did not use the IEEE definition. Robustness for most of the interviewees in these domains is a property that is connected to the system's ability to make a quick recovery and that the system's behavior when failing is predictable. One of the interviewees (telecom) stated that they make a distinction between robustness and redundancy, which are two different aspects when it comes to failure scenarios. This view on robustness might differ between multimedia and telecom since another interviewee (multimedia) stated that fault-tolerance also is part of robustness.

B. KEY ASPECTS OF ROBUST EMBEDDED SOFTWARE SYSTEMS

In the domains of telecom and multimedia, the interviewees pointed out *no single point of failure* and *graceful degradation* as the key aspects. For example, one of the interviewees mentioned that it is tolerable if some transactions are rejected but the already accepted transactions should be completed and the primary functions should always work. The system should always respond and be available for remote recovery. The two interviewees from the automotive domain also pointed out graceful degradation as a key aspect. In addition they mentioned the importance of *predictable behavior over*

time and the need of having a backup plan that handles problems as an option to *transfer the system into a safe state* whenever there is a need for this. The interviewee from the critical infrastructure domain focused on no single point of failure and mentioned the use of a hot stand-by computer, which is ready to take over whenever there is a problem with the main computer. The interviewee from the aerospace domain highlighted such requirements as no catastrophic failures for 10^9 flight hours. End user behavior and experience are other aspects that some of the interviewees pointed out as important in their organization.

C. REQUIREMENTS FOR ROBUSTNESS TESTING

The general requirements for robustness spans from the critical safety requirement of no catastrophic failures for 10^9 flight hours in the aerospace domain to requirements on user experience and being robust enough to be accepted by customers. Availability seems to be a key aspect in all domains since this was mentioned by all in various terms such as no single-point of failure, hot standby, quality of service and quality of experience. Several of the interviewees also mentioned that they had requirements to handle specific scenarios such as overload, malicious traffic, or repetitive occurrences of restarts and failovers. Requirements specifying data integrity, failure rates, latency and response time were also mentioned. These requirements were elaborated by the interviewees who gave specific examples such as availability of 99.999% over time, systems achieving safety level 4, restoration time to be less than 2 seconds, and response time in milliseconds. When it comes to the general requirement of no failures for 10^9 flight hours in the aerospace domain, this requirement applies to all critical parts of the system. For example, the control system is critical and must meet the requirement and so must therefore, all components between the control system and the pilot.

The origins of robustness requirements were limited to different combinations of three sources: *customers*, *regulatory standards,* and the *own organization*. Most of the subjects explicitly stated that requirements come from customers, either directly or by taking into account customer's experiences, behaviors and needs, for example, by monitoring customers' behavior while visiting a web site. Some of the interviewees considered requirements to come from all three sources depending of the type of requirement. For example, requirements concerning safety usually come from regulatory standards while requirements on quality of service usually come from the own organization or agreements on the service level. A few of the subjects mentioned that they only have requirements from their own organization.

D. DESIGN FOR ROBUSTNESS TESTING

Indications of any robustness-specific test design were quite limited among our interview subjects. Most of what is considered to fall under the category of robustness testing seems to take place on system level. With a few exceptions, the test design is ad hoc and experience-based rather than systematic. The few systematic test design techniques that are mentioned include boundary-value analysis and search-based simulation testing for identifying violations of resource consumption requirements. A few of the subjects highlight the usage of an exploratory testing strategy for robustness testing. The main argument in favor of exploratory testing for robustness is that exploratory testing relies more on the experience and domain expertise of the tester, than on what is explicitly stated in the requirements. This argument is based on an assumption that many robustness requirements are implicitly assumed rather than explicitly stated, and thus not covered by traditional requirement-based testing. In addition, several interviewees (primarily in the telecom domain) mention the use of code inspection, and static and dynamic analysis tools (like Lint[2] and Valgrind[3]) in the context of robustness testing. These analyses are primarily performed on lower levels of integration for identifying things like buffer overflows and memory leaks. While not solely targeting robustness, such issues are considered likely to impact the overall robustness of the system under test.

E. PERFORMING ROBUSTNESS TESTING

Based on the responses of the interview subjects, we can identify three different organizational approaches to robustness testing:

A. Organizations where no dedicated robustness testing staff or teams exist, and where no robustness-specific testing is mandated by standards or internally defined procedures.
B. Organizations where no dedicated robustness testing staff or teams exist, but robustness-specific testing is mandated by standards or internally defined procedures.
C. Organizations where dedicated staff or teams do exist for robustness testing, and where robustness-specific testing is mandated by standards or internally defined procedures.

Note that in approach A, organizations may still perform robustness testing in practice, but they may not think about this activity as robustness testing per se. Robustness-relevant tests may be considered as performance, maintainability or stability testing. Consequently, there is a gray scale between the A and B approaches, and most of the organizations of our interviewees seem to fall somewhere within this scale. In fact, out of our studied organizations, only the telecom organizations fall in the C approach. Robustness tests typically originate from considering potential unintended, unwanted or unexpected scenarios occurring in the intended environment of the system under test. Such scenarios may include system restart, overload, power failure, cable failure, failure of peripherals or system misuse. The purpose of dedicated robustness tests is to evaluate the capability of the system under test to cope with these scenarios. In the cases where

[2]http://pubs.opengroup.org/onlinepubs/7908799/xcu/lint.html [Accessed 17 November 2015]
[3]http://valgrind.org [Accessed 17 November 2015]

robustness is measured by a generic metric during testing, this is typically done from the perspective of an end-user system-quality attribute. To some extent, these quality attributes are domain specific. However, for most of the studied organizations, different measures of availability and uptime seem to be the primary attributes. Ideally, the availability of the system under test should not be unreasonably negatively affected by a robustness stressor (like the restart of a subsystem). Some interviewees also mention lower-level measurements, like communication performance and memory consumption over time, as indicators of robustness problems. While all interviewees state that robustness testing is conducted on the target platform (i.e., actual hardware, actual software, but often simulated environment), a few of the interviewees state that some robustness testing is also conducted on a virtual platform. Most of our interviewees claim that they have, or are working towards a high level of test execution automation.

F. ROBUSTNESS FAILURES

The results show that there are many different types of robustness failures. Among these, crashes are the most common (almost all of the interviewees). Other common robustness failures are caused by performance related issues, restarts, as well as CPU and memory related issues.

In general, it is not a common practice to classify the failures. Except for two, all interviewees indeed stated that they do not classify their failures. However the two interviewees, who do classify failures, mentioned that they classify as they concern safety, i.e. which failures lead the system to enter a safe or unsafe state. Failures may be classified as crashes and reboot, memory failures, performance problems, response/latency (telecom, multimedia), or timing constraints (automotive, multimedia). Others classify by system reliability, in terms of catastrophic or non-catastrophic failures, or by the severity of impact on the system. Some rule out the possibility of catastrophic failures in the final product by design (telecom, aerospace). The overwhelming reason for robustness failures was quoted as the sheer complexity of the system. Other possible reasons for robustness failures were given as lack of understanding of the environment, improper memory addressing, fragmentation issues, configuration issues, integration issues, and issues related to geometrical transformation.

G. ROBUSTNESS TESTING TOOLS

The results show that in general the tools used for robustness testing are built in-house, open-source customized in-house, proprietary or open source. The useful functionality in these robustness-testing tools is to provide help in monitoring, such as status reports and visualization of trends over a chain of activities, generation and control of large streams of data, providing flexibility in adding and editing scenarios, and the ability to replay tests. The main challenge associated with robustness testing tools is a universal tool that handles the whole system and considers different scenarios, simulates the environment and replays the tests. Table 4 describes the summary of robustness testing tools in practice.

V. DISCUSSION

This section summarizes the results from the interviews and discusses the differences between state of the art and state of the practice. We also discuss the threats to validity for our study in section V-C.

A. SUMMARY OF RESULTS

The conduction of this case study in an industrial context allows us to publish information on state of the practice of robustness testing. We inquired about different aspects of robustness testing and the answers can be summarized for a particular sample of interviewees with the following pieces of evidence:

The IEEE definition of performing robustness is commonly accepted among interviewees, however there are some additional considerations concerning robustness. The most commonly brought up aspect of robust embedded software system is availability. However no single point of failure, graceful degradation, ability for remote recovery, predictable behavior over time, and return to a safe state are the key aspects of embedded software systems. The common requirement of robustness testing of embedded software systems is the availability of the system in various terms. There are some requirements to handle specific scenarios and some requirements specifying data integrity, failure rates, latency and response time. Usually the requirements of robustness testing comes from three sources, the own organization, regulation standards and the customers. However, there is evidence in some domains that some requirements could come from just one or two of these sources. Robustness-test design is to a large extent ad hoc and experience based rather than systematic. Various organizations have different approaches to robustness testing, ranging from no dedicated teams and no defined process, to dedicated teams and defined process. Robustness tests typically originate from different scenarios, where the purpose is to evaluate the capability of the system and measure different quality attributes such as availability. Mostly robustness testing is conducted on the target platform (i.e., actual hardware, actual software, and often simulated environment). Crashes are the most common robustness failures compared to performance related issues, restarts, or CPU and memory related issues. In general, robustness failures are not classified into categories.

The common causes of robustness failures are found to be complexity of the system, lack of understanding of environment, memory addressing, fragmentation issues, configuration issues, integration issues and geometrical transformations. The tools used for robustness testing are built in-house, open source customized in-house, proprietary or open source. The tools provide help in monitoring tests by visualizing, controlling, revising, and replaying them.

The main challenge associated with robustness testing tools is a universal tool that handles the entire system and

TABLE 4. Summary of robustness testing tools.

Issue	Domain			
	Telecom	Automotive, Critical infrastructure	Multimedia	Aerospace, Banking, Consumer prod.
Robustness testing tools used	90% in-house or open source and in-house tools. Tools that generate traffic load, in-house tools and open source. Akilles Satellite, from Valtech. Jenkins and Erlang-based tool.	Hardware-in-the-loop lab and Python scripts. MUnit, CUnit, and JUnit, as well as in-house tool. Automated availability tools.	Mixed open source and in-house. PureLoad traffic generator. Mixed proprietary software and combinations of open source and customized in-house tools.	Mostly open source and in-house built tools. No specific tool. Open-source test execution tool Mocha.
Most useful functions of the tools	Ability to control large data streams, flexibility in traffic generation and data volume. Traffic generation, load profile, ability to generate malicious traffic. Competent, useable and configurable tool. Proper visualization. Erlang concurrent programming language interface.	Monitoring inputs/outputs, inserting errors. All parts are needed, chain of activities, defining specifications and reusing tests. Automatic system testing and system coverage are useful and necessary.	Load generation. Statistics on sequences of requests and replies; model of traffic, manually provoking the system. Load and stress testing other performance measures.	Productivity, version control system, certifying separate pieces. The ability to define many conditions, to scan full range, and cross-check; a configurable tool that can treat external events and input from environment in all combinations.
Missing functions of the tools	The ability to handle physical events, such as pulling cables and cards, or changing the temperature. Reflecting dependency on components; traceability, replay and record features. The ability to allow hands on testing; editing and adding scenarios. No missing function.	The ability to get a complete overview, after a test; ability to analyze. Ability to automate testing for various hardware on the system level; ability to reuse tests. Ability to condense the complete system in a specification; to make it less expensive to maintain and test the system.	Need a single tool, handling all types of systems, with various kinds of traffic. Fuzzing tools to add fuzzing parameters in load test. We develop customized in-house tools using open-source tools when we need specific results.	The ability to simulate the whole system including all component; the ability to detect unwanted functionality. Flexibility and configurability; ability to model natural variations in in the environment such as signals, voltages, vibrations, etc.
Support desired for robustness testing	The ability to see what is happening in reality; creating a track model and environment model, that keeps up with new platforms. A better traffic generator; tools are not silver bullets, proper system design implementation is important. A tool that would be flexible in editing and adding scenarios. We use mostly in-house and open source; we are flexible, as the interface is a programming language.	A synced instrument cluster display to support playback, with visualization of all variables including deviations from desired signals. Support for test automation. Full system functional coverage; automate all system testing; complete system overview in one tool.	A multi-purpose tool, handling different sorts of traffic. The testers would only need to learn one tool. Being able to define more flexible and reproducible tests, based on API definitions; a mix between fuzzer and load test. We are flexible in developing our tools. This complements tools with expensive licenses, also for test automation.	Need support for whole system level behavior by simulation, to detect how. Implicit assumptions may surface as unwanted behavior; we have considered using Matlab, as a good developing method. Ability to make changes in the framework more easily.

considers various scenarios. The common causes of robustness failures are found to be the complexity of the system, lack of understanding of the environment, memory addressing, fragmentation issues, configuration issues, integration issues and geometrical transformations.

B. DISCUSSION OF SUMMARY

Our interviews provide us with a sample of the state of the practice based on thirteen experienced testers from various industrial domains. Such a study can, of course, never be more than a snapshot view of the activities and methods used at the companies where the interviewees work. With that said, there are some clear differences between the state of the art as described in Section II and the results we got from our interviewees, which we discuss here.

Concerning robustness definition, the state-of-the-art IEEE definitions is accepted by most of the interviewees. The argument of one interviewee that robustness to him is not only a stressful environment but a misbehaving environment and connected equipment is very close to the definition proposed by Lussier et al. [13]. In addition, robustness in terms of the system entering into safe or unsafe state resembles the robustness definition proposed by Fernandez et al. [2]. The most common state-of-the-art approaches of performing robustness testing, i.e. the fuzz approach [14] and the benchmarking approach [15], are unfamiliar to the interviewees. Although they may be following a similar procedure of performing robustness testing to some extent, none of the interviewees explicitly mentioned following such published approaches. On the other hand interviewees are performing robustness testing taking into account scenarios that may include things like system restart, overload, power failure, cable failure, failure of peripherals or system misuse that is not evident in the state of the art. In state of the art, the robustness failures are classified according to the CRASH failure categorization [15]. In practice, however, all the interviewees stated that they do not classify failures based on any such categorization

This does not imply that the categorization is wrong, only that it does not seem to be used in practice. Concerning the causes of robustness failures, the state of the art [12], [14], [19] highlights some particular causes such as internal exception handling errors, unknown exceptions and segmentation faults. However, according to interviewees, complexity of the system, lack of understanding of the environment, and memory addressing are the common reasons of robustness failures. This may be obvious, as most of the state-of-the-art work only deals with robustness testing of operating systems, while our interviewees are performing robustness testing in several other contexts. Most of the state-of-the-art tools deal with the robustness testing of operating systems [16], [18], [19], [22] and Java-based software [20], [24]. These tools are well known and available. However, the tools used by interviewees are mostly built in-house, and to some extent use open source but are customized in-house.

C. VALIDITY THREATS

There are many threats to the validity of our findings as in any exploratory case study. We follow the guideline of Wohlin et al. [35] to identify potential threats.

Construct validity refers to the extent to which the study focuses on what it is intended to focus on, e.g., whether the study instruments (interview questions) adequately capture the concepts we want to study. To reduce the threat related to the study instrument, we have designed the interview questions based on a primary survey of literature (to identify what should be explored) and an iterative refinement of the study instruments among all authors of this study. We have further selected interviewees that have at least three years of robustness testing experience with more than five years of general testing experience. Another potential threat related to construct validity is that the interview is answered by guessing what the researcher has in mind rather than answering the question. To reduce this threat we used open-ended questions, asking the interviewee explicitly to give their answers in terms of describing examples from their fields and made sure to not intervene during their answers. Finally, there is a potential threat related to the misinterpretation of interviewees' answers during the analysis phases. To reduce this threat, we have used a formal process of data extraction and analysis as described in sections III D, where the formal procedure helps to not misinterpret or lose any information. In addition all interviews were recorded and each recording was analyzed by at least two of the co-authors.

Internal validity is often less sensitive in exploratory studies [36], however there are still some threats relevant to mention. The sample size of each domain under study was relatively small, in most cases it is only one, as shown in Table 3, and does not allow for any quantitative analysis. However, with stratified selection of interviewee subjects, we ensured that selection covers a broad area of different domains. Moreover, the interviewees share among them a fair amount of practical experience, which gives a high level of confidence for their answers to be representative when it comes to state of the practice in robustness testing.

External validity concerns how the results can be generalized, and it is very specific for a case study, such as whether the conclusion can be generalized to the same and different domains. In our study we did include representations of various industrial domains of embedded software systems. There are some common practices and characteristics we found in the same domain and sometimes across various domains that are used. However we cannot say how commonly they are used since we only can provide a snapshot from the companies we studied. Further studies, including more interviewees from different contexts are required to increase the confidence for our results. Reliability relates to the replication of the study and arriving at the same results. Replication requires a well-documented design, a structured data collection process and a formal analysis procedure and this is the case for our study. It is, therefore, possible to keep

the same design when the study is replicated and used to increase the empirical knowledge base for the subject.

VI. CONCLUSION

This study investigates the state of the practice of robustness testing by conducting a multiple case study, where thirteen experienced practitioners in different organizations are interviewed. We provide a knowledge base of the state of the practice in robustness testing of embedded software systems and compare this to the state of the art. Although the state of the art describes issues and methods in robustness testing, state of the practice has not been explored and described to this extent in literature before. Moreover, robustness testing, as described in literature, focuses on the software alone and in most cases only on operating systems, rather than embedded software. Our study shows that the state of the practice when it comes to robustness testing is quite different from the state of the art described in the scientific literature. For example, the methods commonly described in literature (e.g., the fuzz approach) are not used in the organizations we studied. Instead, the interviewees describe several ad-hoc approaches that take specific scenarios into account (e.g., power failure or overload). Other differences we found concerns classification of robustness failures, the hypothesized root causes of robustness failures and the type of tools used for robustness testing. The knowledge base is useful for researchers as well as practitioners. Knowledge of the state of the practice is essential for researchers in order for them to create solutions that are feasible for industry and adaptive to industrial approaches. Practitioners can use the knowledge base to incorporate new approaches into their own test environment.

A suggestion for future work is to extend the study. Our study only focuses on organizations situated in Sweden and, even though these organizations are mostly multi-national, it would be interesting to interview testers from other parts of the world. Another extension of this study is to include testing of other non-functional properties than robustness, such as performance efficiency or security.

ACKNOWLEDGEMENT

The authors are thankful to the entire TOCSYC team for involvement in the early design of this study and providing us feedback on study instruments and data analysis.

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.
[5] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Standard 61012-1990, Dec. 1990, pp. 1–84.
[6] M. N. Huhns and V. T. Holderfield, "Robust software," *IEEE Internet Comput.*, vol. 6, no. 2, pp. 80–82, Mar./Apr. 2002.
[7] 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.
[8] S. Eldh and D. Sundmark, "Robustness testing of mobile telecommunication 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.
[9] R. Plosch *et al.*, "The EMISQ method—Expert based evaluation of internal software quality," in *Proc. 31st IEEE Softw. Eng. Workshop (SEW)*, Mar. 2007, pp. 99–108.
[10] *Software Engineering—Product Quality—Part 4: Quality in Use Metrics*, Standard ISO/IEC TR 9126-4:2004, International Organization for Standardization, 2004.
[11] A. Shahrokni and R. Feldt, "A systematic review of software robustness," *Inf. Softw. Technol.*, vol. 55, no. 1, pp. 1–17, Jan. 2013.
[12] B. Lei, X. Li, Z. Liu, C. Morisset, and V. Stolz, "Robustness testing for software components," *Sci. Comput. Program.*, vol. 75, no. 10, pp. 879–897, Oct. 2010.
[13] 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.
[14] 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.
[15] 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.
[16] P. Koopman and J. DeVale, "Comparing the robustness of POSIX operating systems," in *29th Annu. Int. Symp. Fault-Tolerant Comput., Dig. Papers*, Jun. 1999, pp. 30–37.
[17] C. P. Shelton, P. Koopman, and K. Devale, "Robustness testing of the microsoft Win32 API," in *Proc. Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2000, pp. 261–270.
[18] 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.
[19] K. Fernsler and P. Koopman, "Robustness testing of a distributed simulation backplane," in *Proc. 10th Int. Symp. Softw. Rel. Eng.*, Nov. 1999, pp. 189–198.
[20] 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.
[21] M. Schmid, A. Ghosh, and F. Hill, "Techniques for evaluating the robustness of Windows NT software," in *Proc. DARPA Inf. Survivability Conf. Expo. (DISCEX)*, vol. 2. Jan. 2000, pp. 347–360.
[22] 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.
[23] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.
[24] C. Csallner and Y. Smaragdakis, "JCrasher: An automatic robustness tester for Java," *Softw.–Pract. Exper.*, vol. 34, no. 11, pp. 1025–1050, Sep. 2004.
[25] 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.
[26] M. Acharya, T. Xie, and J. Xu, "Mining interface specifications for generating checkable robustness properties," in *Proc. 17th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Nov. 2006, pp. 311–320.
[27] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano, "T-Fuzz: Model-based fuzzing for robustness testing of telecommunication protocols," in *Proc. IEEE 7th Int. Conf. Softw. Test., Verification Validation (ICST)*, Mar. 2014, pp. 323–332.
[28] 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.
[29] 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.
[30] 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.

[31] 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.
[32] F. Belli, A. Hollmann, and W. E. Wong, "Towards scalable robustness testing," in *Proc. 4th Int. Conf. Secure Softw. Integr. Rel. Improvement (SSIRI)*, Jun. 2010, pp. 208–216.
[33] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Eng.*, vol. 14, no. 2, pp. 131–164, Apr. 2009.
[34] K. Peffers, T. Tuunanen, M. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *J. Manage. Inf. Syst.*, vol. 24, no. 3, pp. 45–77, 2007.
[35] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. C. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*, vol. 6. Norwell, MA, USA: Kluwer, 2000.
[36] R. K. Yin, *Case Study Research: Design and Methods*, 5th ed. Newbury Park, CA, USA: Sage, 2013.

SYED MUHAMMAD ALI SHAH received the M.Sc. degree in software engineering from the Blekinge Institute of Technology, Sweden, and the Ph.D. degree in software engineering from the Politecnico di Torino, Italy. He was an ERCIM (Marie-Curie) Post-Doctoral Fellow with the Swedish Institute of Computer Science (SICS). His research focus is on empirical software engineering, software testing, and quality and software start-ups. He was a Research Assistant with the Software Engineering Laboratory, Politecnico di Torino, for approximately three years. He is currently a Researcher with the Software and Systems Engineering Laboratory, SICS.

DANIEL SUNDMARK received the M.Sc. degree in information technology from Uppsala University, Uppsala, Sweden, in 2001, and the Ph.D. degree from Mälardalen University, in 2008. He has served as a Senior Researcher with the Swedish Institute of Computer Science, and a Lecturer with Mälardalen University. He serves as a Professor with Mälardalen University, where he co-leads a research group on software testing. His research primarily revolves around empirical studies of industrial software engineering, in particular focusing on different aspects of software testing.

BIRGITTA LINDSTRÖM received the Ph.D. degree in topic testability of dynamic real-time systems from Linköping University, in 2009. She is leading the Distributed Real-Time Systems Research Group with the University of Skövde. She is also a Program Manager with TOCSYC, a distributed research environment, where researchers from three Swedish universities and a research institute collaborate in search for cost-effective techniques to test non-functional properties in embedded systems. Her research is focusing on model-based and mutation-based software testing.

STEN F. ANDLER received the M.Sc. and Ph.D. degrees in computer science from Carnegie Mellon University, Pittsburgh, PA, USA, in 1976 and 1979, respectively, and the Ph.D. degree in computer science from the Chalmers University of Technology, Göteborg, Sweden, in 1979. Before joining the university, he served as a Research Staff Member with the IBM Research Division, Almaden Research Center, San Jose, CA, USA, from 1979 to 1992. He has been a Professor of Computer Science with the University of Skövde since 1992. His research primarily revolves around software engineering of embedded systems, including testing, information fusion, and distributed systems issues. He currently serves as a member of the Board of Directors of the International Society of Information Fusion and a member of the Editorial Board of *Innovations in Systems and Software Engineering*.