2015 41st Euromicro Conference on Software Engineering and Advanced Applications

Testability and Software Robustness: A Systematic Literature Review

Mohammad Mahdi Hassan
Karlstad University, Sweden
mohammad.hassan@kau.se

Wasif Afzal
Mälardalen University, Sweden
wasif.afzal@mdh.se

Martin Blom
Karlstad University, Sweden
martin.blom@kau.se

Birgitta Lindström
University of Skövde, Sweden
birgitta.lindstrom@his.se

Sten F. Andler
University of Skövde, Sweden
sten.f.andler@his.se

Sigrid Eldh
Ericsson AB, Sweden
sigrid.eldh@ericsson.com

***Abstract*—The concept of software testability has been researched in several different dimensions, however the relation of this important concept with other quality attributes is a grey area where existing evidence is scattered. The objective of this study is to present a state-of-the-art with respect to issues of importance concerning software testability and an important quality attribute: software robustness. The objective is achieved by conducting a systematic literature review (SLR) on the topic. Our results show that a variety of testability issues are in focus with *observability* and *controllability* issues being most researched. *Fault tolerance*, *exception handling* and *handling external influence* are prominent robustness issues in focus.**

***Index Terms*—Software testability; Software robustness; Systematic literature review**

I. Introduction

Software testing dynamically verifies and validates that a program or a system behaves as expected when subjected to a finite set of test cases, usually selected from an infinite execution domain [1]. According to Voas and Miller [2], software testing is one of the three pieces that developers must complete to assess reliability of software, the other two being software testability and formal verification. Software testability, the topic of interest in this paper, has a number of different interpretations. While we shall provide an overview of such interpretations shortly, it is important to differentiate between software testing and software testability. While software testing aims to assess the quality of software produced, software testability is not concerned with whether the software is producing correct or incorrect results [3]. Rather, software testability concerns the characteristics of the software that affect the effort needed to test the software. Put differently, the higher the testability is, the easier it is to perform testing activities such as designing, executing and analyzing tests.

Freedman [4] defines a program testable if it has no input-output inconsistencies and that it has the properties of observability (of outputs) and controllability (of inputs). A different interpretation of testability is given by Bache and Müller [5] where testability is determined by the coverage achieved by a test strategy such as branch coverage for control flow testing strategies. A probabilistic view on software testability is given by Voas and Miller [2] and Bertolino and Strigini [6], looking at the probability that the code will fail if it is faulty.

While software testability has been investigated according to the above interpretations, the functional correctness of the software has been or is assumed to be the focus. Little is known regarding what software testability issues impact non-functional properties, how is testability estimated for such properties and what measured impact software testability has on them? Thus there is an ample opportunity to investigate the relationship between software testability and different non-functional properties. In this paper, we restrict ourselves to investigate the relationship between software testability and software robustness. Software robustness is an especially important property for critical software systems and is defined as the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions [7]. Robustness is also considered an element in achieving higher dependability in systems [8]. In order to provide a state-of-the-art with respect to issues of importance concerning software testability and software robustness, we have performed an extensive systematic literature review (SLR) according to the well-established guidelines of Kitchenham and Charters [9]. We have categorized the evidence in the area in terms of issues of importance for software testability and software robustness. We then present a descriptive synthesis of the primary studies with a specific focus on software robustness issues.

The rest of the paper is organized as follows. Section II describes the process followed in conducting the SLR. Section III presents a synthesis of obtained primary studies, organized with respect to software robustness issues. Sections IV and V present a discussion on results and conclusions respectively.

II. Method

A systematic literature review (SLR) is a form of secondary study since it synthesizes evidence from already existing primary studies. After the need for an SLR has been identified, the most important part of an SLR is the specification of research questions.

A. Research question

In order to capture the existing views on testability and software robustness, we have formulated the following research

978-1-4673-7585-6/15 $31.00 © 2015 IEEE
DOI 10.1109/SEAA.2015.47

TABLE I: Count of papers before and after duplicate removal among different publication sources.

Source	Search count	After duplicate removal
Springer Link	9933	8551
IEEE Xplore	1161	748
ACM Digital Library	5683	3422
ISI Web of Science	617	578
Scopus	5103	4059
ScienceDirect	3651	1658
Wiley Online	5673	4343
Sub-total	31821	23359
Exact phrase search	786	174
Total	32607	23533

question:

RQ: What is the state-of-the-art on issues related to testability and software robustness?

In terms of PICOC (Population, Intervention, Comparison, Outcome, Context) criteria for structuring research questions [9], our research question has no limitation with respect to 'comparison' and 'context' but has the following elements:

- Population: software.
- Intervention: testability and robustness.
- Outcomes: Issues of importance concerning testability and software robustness.

B. Generating a search strategy

The search strategy was agreed upon after several rounds of trial searches using various combinations of search terms. Due to the broad scope of our research question, we finalized four search terms: *software testability*, *software testable*, *software untestable* and *software non testable*. We did not include the term *robustness* and its synonyms in search due to two reasons: (1) it is difficult to find synonyms of robustness and (2) we plan to investigate the relationship between testability and other non-functional properties like performance as future work. These search terms were used separately in the following databases: Springer Link, IEEE Xplore, ACM digital library, ISI web of science, Scopus, ScienceDirect and Wiley Online Library. This initial search was complemented with an exact-phrase search (in full-text/other fields) whereby the four search terms were used with double quotation marks. The exact phrase search was carried out in databases where this search option was available (IEEE Xplore, ACM digital library, Springer Link, ISI web of science and Scopus).

We did not restrict the search results based on publication year as we wanted to be as inclusive as possible. Thus the default settings for the start year were used for each database.

Table I shows the number of hits for each database. We got a total of 32607 papers after the initial and exact-phrase search. After duplicate removal based on title and abstract, we were left with a total of 23533 papers.

C. Study selection criteria

An important step in the study selection process is to list exclusion and inclusion criteria. We decided to exclude studies that:

- do not relate to software engineering/computer science,
- do not relate to software testability,
- merely mention testability in a cosmetic/cursory manner, lacking any credible research on it,
- have a focus on hardware/system testability (such as digital circuit testability analysis),
- are book reviews,
- are not written in English language,
- are editorial papers written for special issues of different journals,
- represent academic theses,
- are books/book chapters,
- are only discussing software testability without relating it to software robustness.

We included all those studies that:

- address software testability and its relation to software robustness.

The study selection was done in multiple steps:

1) First a total of 2089 papers were discarded based on automatic removal by keywords, done by using a SQL query. We removed papers with keywords that suggested them not to be relevant to software testability and falling in our exclusion criteria. Examples of such keywords include VLSI, microchips, CMOS, circuit design, cell array, voltage, transistor, flipflop, microprocessor, nanometer, DRAM and SRAM.
2) The second step of the study selection involved reading the titles and abstracts of remaining 21444 papers and excluding papers not relevant to software testability. The papers were distributed among authors and for each paper we classified it as being either *relevant*, *non-relevant* or *not clear*, based on the stated exclusion criteria. Each paper was read by two authors. In case of disagreement among the two authors, the paper was marked as *not clear*. As a result of this step, we were left with 1422 *not clear* and 413 *relevant* papers.
3) The third step of study selection involved deciding on the *not clear* papers based on skimming the full-text of each paper to see if it relates to software testability. The skimming process for each paper was done in several steps: (1) reading the introduction and conclusion sections (2) searching for term *testability* in the full text and (3) reading sections if found relevant for decision-making. After the full-text skim, we were left with 807 relevant papers.
4) The fourth step of the study selection involved deciding on which of the software testability papers relate to software robustness. We again skimmed the full-text of 807 papers, similar to the previous step, but now searching for software robustness. After this full-text skim for software robustness, we were left with 75 papers.
5) The fifth step of study selection was done to read full-text of the 75 papers. As a result of this step, we were left with 23 relevant papers.

6) The set of 23 relevant papers were complemented with additional 4 papers recommended by an expert on the subject. These papers were either not captured in our search or were discarded during step 2 because of not being explicit about their relation to testability. In the end, we had a total of 27 primary studies for our SLR. The primary studies are indicated in bold titles in the references section.

D. Study quality assessment

We did not assess the quality of included studies using any pre-designed quality instrument due to our quest of being as inclusive as possible since we expected the focus on testability and software robustness to be still in its infancy, with less chances of finding large-scale empirical studies. Also our research question does not aim at finding the strength of inferences where study quality assessment is regarded as valuable.

E. Data extraction

The purpose of data extraction is to record information obtained from primary studies in a pre-designed data extraction form. The data extraction was done by three authors. Besides the general information about paper ID and title, the following specific information was gathered: testability method/ technique; robustness method/technique; testability issue in focus; robustness issue in focus; testability metric; robustness metric; and measured positive/negative impact of testability on robustness.

III. Data synthesis

Data synthesis involves combining the results of included primary studies [9]. Due to diversity in the context of the primary studies, a quantitative synthesis is not a possibility in this SLR. We therefore present a descriptive (non-quantitative) synthesis. The data extraction forms of individual primary studies are analyzed for finding patterns of issues of importance that answer our stated RQ.

In Figure 1, we show a categorization of the *software testability issues* discussed in our set of primary studies. These issues are concerned with *observability*, *controllability*, *diagnosability*, *testing effort*, *automation* and *miscellaneous issues*. Below we present a brief description of these categories along with related primary studies:

- Observability: the ability to observe output/internal states of a component or a software under test ([6], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21]).
- Controllability: the ability to control input and execution of a component/software under test as required for testing ([13], [14], [15], [16], [17], [19], [20], [22]).
- Diagnosability: the ease with which a fault is isolated after the occurrence of a failure ([11], [15], [23]).
- Testing effort: the ability to reduce testing effort and to promote ease of testing ([15], [24], [25], [26], [27]).
- Automation: the extent to which software testability aspects can be automated (e.g., using an automated testing framework and built-in tests) ([15], [28], [29], [22], [30]).

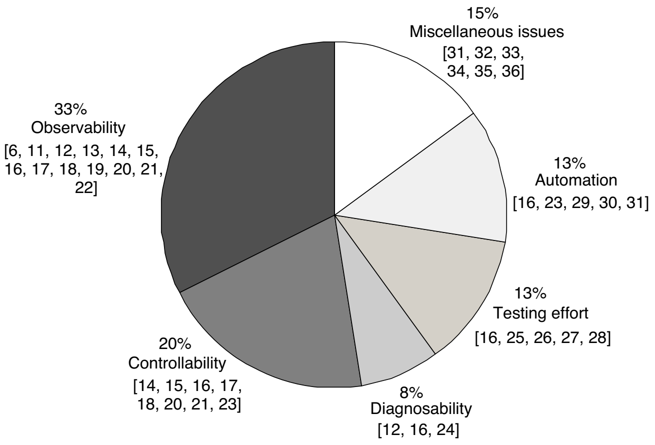

Fig. 1: A categorization of software testability issues and associated primary studies.

- Miscellaneous issues: the issues concerning testability and requirements traceability [31], testability transformation [32], fault-detection capability of tests [33], testability improvement through a software process methodology [34], [30] and verifiability [35].

It is evident from Figure 1 that more than half of the primary studies (53%) deal with observability and controllability aspects in software testability issues impacting software robustness. In Figure 2, we show a categorization of the *software robustness issues* discussed in our set of primary studies. These

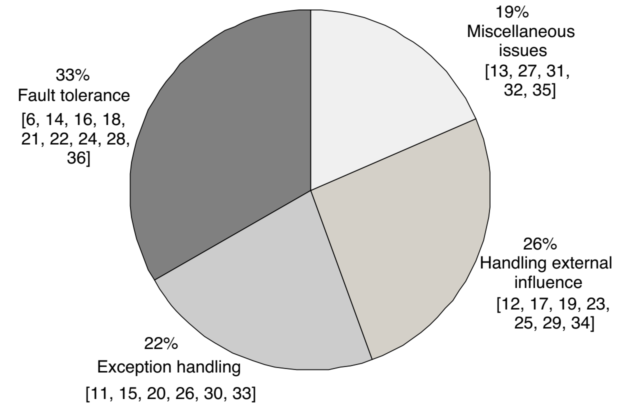

Fig. 2: A categorization of software robustness issues and associated primary studies.

issues are concerned with *fault tolerance*, *exception handling*, *handling external influence* and *miscellaneous issues*. Below we present a brief description of these categories along with related primary studies:

- Fault tolerance: the ability to avoid critical failures in the presence of faults and errors ([6], [13], [15], [17], [20], [21], [23], [27], [35]).
- Exception handling: the ability to handle exceptions ([10],

[14], [19], [25], [29], [32]).

- Handling external influence: the ability to handle faults due to external events, such as contract breaches [11], external dependencies for components/subcomponents [16], hostile environment [18], failure in communication framework [28] and protocols [24], invalid input [22] and externally infected provider classes [33].
- Miscellaneous issues: the issues concerning hazard analysis [12], introducing malfunctions [34], handling of unforeseen situations using a knowledge-based system [30], handling faults due to an extended period of execution [26] and redundancy [31].

It is clear from Figure 2 that fault tolerance, exception handling and handling external influence are the main robustness issues discussed in our set of primary studies.

The following subsections summarize software testability issues for each of the identified software robustness issues in our set of primary studies.

A. Fault tolerance

Voas and Miller [21] argue that testability analysis can be applied during testing to enhance fault tolerance. They extend the earlier described propagation analysis [36] as a way to isolate program regions that show data state errors and thus will result in certain types of software failures. The authors present two extended propagation analysis algorithms that produce a set of locations generating particular type of software failures. Such locations can then include fault tolerant mechanisms to be placed and thus make the software more robust.

Schütz [20] suggests that distributed real-time critical applications need high dependability which can be achieved through some form of a fault tolerant mechanism such as redundancy or recovery blocks. On the other hand, testability issues such as observability can be highly challenging for such a system due to "probe effect". To improve observability, techniques such as code instrumentation may introduce delay and thus affect the outcome of a real-time execution. The author suggests three solutions: to use dedicated hardware; integrate all monitoring and test support with the system; use a combination of both. To improve controllability, the main issue is reproducibility which is difficult to achieve in a distributed system due to nondeterminism in its hardware, software or operating system. To solve it, the author suggests defining precisely a particular execution scenario in terms of significant events and then applying control over the execution to reproduce the same scenario deterministically.

Kopetz [27] compares two different design paradigms for real-time systems: Time-triggered (TT) and event-triggered (ET). Such systems must be robust and fault-tolerant. This is typically achieved by redundancy. The major difference between TT and ET systems is the predictability in the time domain since detailed plans for the behavior of the system in the time domain is available for TT systems while ET systems creates execution schedules dynamically based on the actual demand, meaning that there are variations in the timing of tasks. Moreover, these variations have consequences for the timing of other tasks in different nodes. To gain confidence in the robustness of a real-time system, it is necessary to perform tests with rare events (e.g., a serious fault) on simulated loads and check whether the event is handled properly within time bounds. However, while the load patterns for a TT system is known, it is very hard to know whether the simulated load pattern in an ET system is representative for the load patterns that will develop in the real application.

Metsä et al. [13] propose using code-level aspects for testing of non-functional properties in order to not mix SUT code with purely test-related code. The aspects were implemented using `AspectC++`. The study shows that it is possible to encode non-functional requirements using aspects, hence making non-functional testing easier and more manageable. Robustness is only mentioned as one-of-many non-functional properties.

Kopetz et al. [17] suggest that fault tolerance is an important approach to ensure dependability of service in a hard real-time system.[1] Regarding testability, the authors identify three major aspects of testability: test coverage, observability and controllability. They further suggest a closed loop approach for robustness testing where the output of the system is fed to a model of the environment that dynamically computes and returns a feedback as input to the system. This simulated environment itself is a real-time application, which ensures testing of unforeseen and hypothesized errors of a system in its environment.

Alanen and Ungar [15] describe software design for testability (DFT) practices while comparing them to hardware DFT. They suggest that a designer can promote software testability by focusing on DFT issues in early software development phases. Regarding fault tolerance, they suggest redundancy in the form of component diversity where different versions of the component are developed in different languages or algorithms. Besides, wrappers can be used to detect faults related to input data. They also suggest recovery mechanisms (such as checkpoints) to restore a system to a previous safe state in case of failures. They also suggest using fault injection to test fault handling mechanisms as well as rare failure scenarios which are difficult to replicate.

According to Laprie [35], fault tolerance can be promoted by either error processing (removing errors from the computational state before failure occurrence) or fault treatment (deactivating the faults permanently). The author mentions computational redundancy (through multiple channels) as well as structural redundancy as being widely used in a fault tolerant system but with the drawback of creating agreement problems. The author's view on design for testability suggests facilitating verification and further suggests using modeling and fault injection to evaluate fault tolerant systems.

Baudry et al. [23] measures the benefit of a design by contract approach on software robustness and software diagnosability. Software robustness is defined as the ability to detect

[1]The authors define a hard real-time system as one in which the consequences of a failure are potentially catastrophic, such as in a flight control system.

internal anomalies during execution while diagnosability is defined as the degree to which the software allows an easy and precise location of a fault when detected. The authors present a model for measuring global robustness of a system that is calculated based on local robustness of components in the system. The local robustness of a component in a system is defined as the probability a fault in the component is detected either by its own contracts or by the contracts it interacts with. Practically, the local robustness is measured by calculating the mutation score of the individual class and its test dependent classes. The mutation score is calculated as the percentage of mutants the class' contracts are able to detect. A similar analysis is done for measuring diagnosis effort (size of the indistinguishability set[2] in which the faulty statement must be located) and global diagnosability of a system (diagnosis precision obtained with a certain proportion/quality of contracts in the system compared to the same system with no contracts).

Bertolino and Strigini [6] argue that Voas and colleagues define testability as the conditional probability that the program fails [2], without taking into account undetected failures due to an imperfect oracle and that faults can be revealed in the absence of failures (by observing the internal state of a program). Therefore, according to Bertolino and Strigini [6], there are three elements in making a program testable (and reliable): (1) reduce the probability of operational failure given an error exists (that is to make the program more robust[3]) (2) improve specification of an oracle or improve the observability of the internal state of the program (3) improve the input distribution used in testing so as to increase the probability of fixing errors given a fault exists. The authors discuss a number of ways to improve program robustness and testability, especially highlighting software design with a self-checking capability provided by software fault tolerance and defensive programming. This includes techniques such as executable assertions with exceptional handling, recovery blocks, N-version or self-checking programming, use of robust data structures and audit programs [6].

B. Exception Handling

According to Salva and Rabhi [14], `BPEL` (Business Process Execution Language) supports fault handling activities. One of the problems in structured BPEL activities is that such activities are nested, including the fault-handler activities. To understand the problem, the authors transform `ABPEL` (Abstract BPEL) specification to `STS` (Symbolic Transition System) which is more suitable to analyze testability criteria such as observability and controllability. They found that in `ABPEL` specifications, non-identical "catch" creates an observability problem, whereas the "fault handling" activity gathering two identical "catch" activities is not controllable. They propose making the "catch" activities distinct and ensuring fault distinction in fault handlers, to promote testability.

Binder [19] describes characteristics of the implementation as one of the six factors contributing to testability in object-oriented systems. Within characteristics of the implementation, Binder discusses testability of exception handling and argues that exceptions are typically less controllable than other application functions and may require simulation of failure modes. Also "testable exception handling would require consistent usage of language-supported features and a related design strategy". The author continues to describe a strategy for Built-in test (BIT) capability that provides explicit separation of test and application functionality.

Chen et al. [25] highlight the importance of exception handling in avoiding abnormal interruption or system crashes. They argue that programming constructs have an influence on software quality, therefore they analyze four different languages (`C++, Ada, Smalltalk, Pascal`) to investigate their impact on software quality. According to their analysis, `Ada` is a better choice in terms of exception handling and testability but it degrades execution efficiency.

Embedding built-in testing capabilities into a component under test is emphasized as a way to improve testability in Brito et al. [29]. According to the authors, existing development methodologies do not emphasize testing of exceptional behavior of a software system. The authors continue to describe a testable component architecture that augments the component with test facilities (built-in tests), which are accessed by the user through a standard interface. These facilities include runtime verification of component contracts, following the Design-by-Contract approach, which are used as test oracle [29]. Using the testable component architecture (including interface specification (tracking mechanisms) and `OCL` (Object Constraint Language) contracts), contract verification mechanisms are introduced in the intermediate code of the component under test.

Fraser and Arcuri [32] use the `EvoSuite` tool to detect failures in terms of undeclared exceptions (i.e. exercising automated oracles) and their results show that testability transformation[4] revealed additional failures without negatively affecting coverage. The main purpose of testability transformations done in the study is to guide the fitness function used in automated search towards maximizing violations of undeclared exceptions.

Briand et al. [10] present templates in `AspectJ` for automatic instrumentation of contracts and invariants in `Java`. The templates are for checking pre- and post-conditions as well as invariants and post-conditions can be checked after a return statement or after an exception is thrown. The `AspectJ` code specifies the assertions and insertion points. Since the instrumentation of contracts is implemented in aspects, in separate files, the actual manipulation of the software takes place on a byte code level.

[2]An indistinguishability set represents a set of statements that are indistinguishable from each other if they are bounded by consecutive contracts in an execution flow.

[3]The authors define software robustness as the ability of the software – accidental or intentionally designed-in – to tolerate internal errors.

[4]A testability transformation is a source-to-source program transformation that seeks to improve the performance of a test data generation technique [37].

C. Handling external influence

Paige [18] describes the concept of "software design for testability" with an example. The author describes a situation where two modules, each having the algorithmic code and robustness code (or protection/exception-handling code) are integrated, resulting in some overlap of robustness code. This overlap can give rise to situations where robustness code of one module excludes data cases from the algorithmic code of the other module. In order to solve this situation, the author proposes two sets of building blocks for the code to offer some visibility into program branch points. These two blocks are algorithmic-oriented and protection-oriented. Such modular development practice suggests inducing some degree of robustness into the code to protect each module against hostile environments.

In [16], the authors propose extending the contract-based built-in-test technique to hierarchical components, i.e. an assembly of components. The authors define three categories of contracts specific to component-based software: Library (defined at the level of object classes and interfaces), Interface (defined to express bindings between a required interface and a provided one) and composition (defined to express external interfaces of component and its subcomponents). Library and interface contracts can be made self-testable according to authors' contract-based built-in test framework called `STclass-Java`. However composition contracts use external dependencies for components/sub-components (a case of integration testing) therefore `STclass` cannot be applied directly unless the component hierarchy or dependencies are extracted.

Baudry et al. [33] discuss trust in object-oriented components using the design-by-contract approach where the specification is systematically transformed into executable assertions (invariant properties, pre/post conditions of methods). They propose a testing-for-trust methodology whereby a component is said to be trustable if the component unit test cases are effective in detecting anomalies in implementation. The robustness of a self-testable component is defined as its ability to detect faults due to external infected provider classes, e.g., due to integration or evolution.

Tappenden et al. [22] argue that security vulnerabilities exploited by malicious inputs can compromise robustness in a modern web-based system. They propose three ways to develop secure (and hence robust) web-based systems: modeling of security requirements, employing a highly testable architecture, and describing and running automated security tests using `HTTPUnit`. In the proposed highly testable architecture, the authors emphasize the need to embed test layers for each layer in the layered architecture of a web application such as presentation, process/control, business entity and data services layers. The authors further describe automated testing using `HTTPUnit` for testing the robustness of a system. They emphasize that automated testing needs to bypass the presentation layers and interact directly with the web application server via `HTTPUnit`. The resulting test cases can then focus on server-side input validation (such as bypassing the predefined input selection criteria and bypassing built-in `HTML` length validation), testing for unexpected state transition and `SQL` injection.

Vuong et al. [24] discuss testability issues from the perspective of protocol engineering life-cycle in distributed systems. They promote a synthesis/analysis approach where synthesis denotes a process that specifies a design specification meeting testability requirements and analysis denotes the process of examining a design specification to check whether it meets certain testability requirements. For each phase in the life-cycle, they list a set of issues such as non-determinism with respect to ordering and effects of events. They also discuss what is needed from a design for testability perspective to handle the issues, e.g., formal methods, models, modularization, compatibility checking and self-stabilization. One testability issue that authors mention for robustness has to do with the fact that limitations affecting robustness (e.g., the size of a message queue) are often un-specified in the model. Another issue is that unexpected events also are un-specified and should be handled at the implementation level according to the semantics of the formal method used.

King et al. [28] propose a self-testing approach with two validation strategies: Replication with Validation (RV), which tests adaptive changes using copies of the managed resources of the system; and Safe Adaptation with Validation (SAV), which tests adaptive changes in-place, directly on the managed resources of the system. `Cobertura` is used to instrument the managed resources during testing to calculate line and branch coverage. The robustness focus is on finding unexpected inputs or events (anomaly detection), such as unsupported user requests or the occurrence of external events such as failure in a framework, thereby requiring self-healing. They also address the challenges associated with validating adaptive changes to autonomic software at runtime.

Briand et al [11] propose using executable software contracts to both replace manually coded oracles and to lower the effort of locating faults once a failure has been identified. The contracts are defined in `OCL` that is included in the `UML` framework and the instrumentation tool used is `Jcontract`. They measure testability in general by mutation testing, i.e. how many mutants are killed with and without contracts. The authors also define diagnosability through Diagnosis Flow, which measures the sequence of methods one has to investigate from the detection of a failure to the location of the faulty statement. The mutation score is lower than with a manually coded oracle, but the diagnosability of the software, i.e. the decrease in effort to locate faults after a failure has occurred, is increased almost one order of magnitude. Indirectly this translates into increased robustness as diagnosability improves fault removal.

D. Miscellaneous issues

Pau [30] argues that failure detection and testing are knowledge-intensive and experience-based tasks and thus are suitable for a knowledge-based systems (KBS) system. KBS can be used to handle unforeseen situations if it possesses detailed knowledge on system functionality, general failure modes propagation and the follow-up repair actions. Regarding testability, KBS can be a part of a built-in test system to identify intermittent faults, reduce false alarms and carrying out recalibration.

Soubies et al. [34] suggest that following strict programming rules can produce better testable software for safety critical systems. For robustness study, they use simulation base testing where they identify safety critical components and introduce malfunctions targeting those components.

Vincent et al. [26] discuss Built-In-Test (BIT) techniques for Run-Time-Testability (RTT) in component-based software systems. They highlight difficulties in testability of COTS, as they are typically supplied as binaries with the internal mechanisms being unknown and lacking standardized test facilities. This increases the chances of residual defects that may show up after extended periods of execution and may compromise system robustness. BIT provides a standardized way in which to incorporate test facilities into software components. This is useful for detecting and localizing defects. Also it is useful for the propagation of these error conditions to a system component having responsibility for error handling and/or recovery.

Bozzano et al. [12] present a methodology for design of complex systems. The methodology is supported by a formal framework, which is integrated with verification tools and supports a number of formal analyses. The authors present a modeling language in which not only the normal hardware and software operations can be described but also properties that are interesting from a testability perspective. It is possible to specify faults (probabilistic), how these faults propagate, error recovery, and failure modes. In order to enable modeling of partial observability, the System-Level Integrated Modeling (`SLIM`) language allows the designer to explicitly define a set of observables. Hence, it is possible to perform observability analyses on such models.

Beer and Heindl [31] focus on a formal, non-invasive technique for requirement traceability and use of standardization (`UML` & `TTCN-3`). The main issues are tracing of requirements and formal diagrams that lead to certainty about the consistency and completeness of test cases. They use a generic approach to testability and robustness, where all tests have to be performed on the original hardware, and compliance as well as robustness testing is prescribed. Robustness is achieved using redundancy, and the system is characterized by dedicated hardware with triple redundancy (2-out-of-3) for automatic fault detection.

IV. Discussion & threats to validity

In this study, we have gathered available evidence on testability and software robustness. We identified and classified issues relevant for software testability and software robustness. Since software testability and software robustness are impacted by a variety of factors, we observed that the authors have taken a number of different research foci on the topic. Nevertheless, observability and controllability are the most researched testability concerns in our primary studies while fault tolerance, handling external influence and exception handling are the most researched robustness concerns. Regarding the extent of evaluation in the research area, our general impression is that a number of studies report initial proposals and proof-of-concept examples but lack large-scale industrial case studies.

We also observed that testability and software robustness is a difficult and a rather less researched area. While design-for-testability is a mature concept in hardware testing, it has not attracted similar attention from the software testing community. As part of our data extraction, we also gathered evidence on testability and robustness metrics, but do not have enough space to discuss them in detail. However, to briefly describe the situation, although there are studies that mention robustness or testability metrics individually, not many papers mention them together as a single concern, i.e. testability and software robustness. In other words, when researchers investigate testability and software robustness, quantifying these two qualities is not a priority. In our view this is partly because of the current immature state of affair in this area of research where focus has been on ideas and proposal generation. A part of our data extraction also deals with the stated measured impact of testability on robustness. Generally, authors report a positive relationship between software testability and software robustness, i.e. improving one property as a consequence of improving another. An exception to this is given by Bertolino [38] where it is argued that an increase in testability increases the confidence in the absence of faults but on the other hand, it reduces the robustness of a program, if faults do remain. We plan to separately analyze this impact of testability on software robustness in detail as an extension to this study.

Following the guidelines for conducting SLRs [9] helped us address majority of validity threats to this study. We did not assess the quality of included primary studies using an explicitly defined instrument. We argue in favor of this decision in Section II-D. To ensure the consistency of data extraction, a subset of primary studies was used to extract data for the second time. In case of a disagreement among authors in study selection, a third person acted as an arbitrator. Lastly, we believe that our extensive search of relevant studies has given us a representative set of primary studies.

V. Conclusions and future works

This SLR has gathered research evidence on testability and software robustness. A variety of testability and robustness issues have been gathered. *Observability* and *controllability* represent the two most researched testability issues while *fault tolerance*, *exception handling* and *handling external influence* are the most researched robustness issues. We also present a synthesis, structured according to each of the identified robustness issues. Several interesting research directions are discussed, notably contract-based built-in-test mechanisms, propagation analysis to identify code locations to include fault tolerant mechanisms, redundancy in the form of component diversity, simulation of failure modes to test exceptions, deterministic execution of significant events and timeliness of event-handling with simulated loads in real-time systems.

There can be several interesting extensions of this work, as hinted at in Section IV. We also plan to conduct a similar study to understand testability and other important non-functional qualities such as efficiency.

VI. Acknowledgment

The authors are thankful to Thomas Ostrand for involvement in early part of this study. This work was funded by The Knowledge Foundation (KKS) through the project 20130085: Testing of Critical System Characteristics (TOCSYC).

References

[1] IEEE Computer Society, "Guide to the software engineering body of knowledge (SWEBOK) v3.0," 2014.

[2] J. M. Voas and K. W. Miller, "Software testability: The new verification," *IEEE Software*, vol. 12, no. 3, pp. 17–28, 1995.

[3] J. Voas, "Testability, testing, and critical software assessment," in *Proceedings of the 9th Annual Conference on Computer Assurance, Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security (COMPASS'94)*, 1994.

[4] R. S. Freedman, "Testability of software components," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 553–564, 1991.

[5] R. Bache and M. Müllerburg, "Measures of testability as a basis for quality assurance," *Software Engineering Journal*, vol. 5, no. 2, pp. 86–92, 1990.

[6] A. Bertolino and L. Strigini, "**On the use of testability measures for dependability assessment**," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 97–108, 1996.

[7] IEEE, "IEEE international standard ISO/IEC/IEEE 24765 for systems and software engineering vocabulary," 2010.

[8] A. Shahrokni and R. Feldt, "A systematic review of software robustness," *Information and Software Technology*, vol. 55, no. 1, pp. 1–17, 2013.

[9] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in Software Engineering," Keele University and Durham University Joint Report, Tech. Rep. EBSE 2007-001, 2007.

[10] L. Briand, W. Dzidek, and Y. Labiche, "**Instrumenting contracts with aspect-oriented programming to increase observability and support debugging**," in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005.

[11] L. C. Briand, Y. Labiche, and H. Sun, "**Investigating the use of analysis contracts to improve the testability of object-oriented code**," *Journal of Software: Practice and Experience*, vol. 33, no. 7, pp. 637–672, 2003.

[12] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Nguyen, T. Noll, and M. Roveri, "**The COMPASS Approach: Correctness, modelling and performability of aerospace systems**," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, B. Buth, G. Rabe, and T. Seyfarth, Eds. Springer Berlin Heidelberg, 2009, vol. 5775, pp. 173–186.

[13] J. Metsä, S. Maoz, M. Katara, and T. Mikkonen, "**Using aspects for testing of embedded software: Experiences from two industrial case studies**," *Software Quality Journal*, vol. 22, no. 2, pp. 185–213, 2014.

[14] S. Salva and I. Rabhi, "**A preliminary study on BPEL process testability**," in *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW'10)*, 2010.

[15] J. Alanen and L. Y. Ungar, "**Comparing software design for testability to hardware DFT and BIST**," in *Proceedings of the 2011 IEEE International conference on automated test (AUTOTESTCON'11)*, 2011.

[16] P. Collet and R. Rousseau, "**Contract-based testing: From objects to components**," in *Proceedings of the 1st International Workshop on Testability Assessment (IWoTA'04)*, 2004.

[17] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner, and W. Schütz, "**The design of real-time systems: From specification to implementation and verification**," *Software Engineering Journal*, vol. 6, no. 3, pp. 72–82, 1991.

[18] M. R. Paige, "**Software design for testability**," in *Proceedings of the 1978 Hawaii International Conference on System Sciences (HICSS'78)*, 1978.

[19] R. V. Binder, "**Design for testability in object-oriented systems**," *Communications of the ACM*, vol. 37, no. 9, pp. 87–101, 1994.

[20] W. Schütz, "**Fundamental issues in testing distributed real-time systems**," *Real-Time Systems*, vol. 7, no. 2, pp. 129–157, 1994.

[21] J. M. Voas and K. W. Miller, "**Dynamic testability analysis for assessing fault tolerance**," *High Integrity Systems*, vol. 1, no. 2, pp. 171–178, 1994.

[22] A. Tappenden, P. Beatty, J. Miller, A. Geras, and M. Smith, "**Agile security testing of Web-based systems via HTTPUnit**," in *Proceedings of the 2005 Agile Conference (AGILE'05)*, 2005.

[23] B. Baudry, Y. L. Traon, and J.-M. Jézéquel, "**Robustness and diagnosability of OO systems designed by contracts**," in *Proceedings of the 7th International Symposium on Software Metrics (METRICS'01)*. Washington, DC, USA: IEEE Computer Society, 2001.

[24] S. T. Vuong, A. A. F. Loureiro, and S. T. Chanson, "**A framework for the design for testability of communication protocols**," in *Proceedings of the 6th International Workshop on Protocol Test Systems VI (IFIP TC6/WG6.1'94)*. North-Holland Publishing Co., 1994.

[25] D.-J. Chen, W.-C. Chen, S.-K. Huang, and D. T. Chen, "**A survey of the influence of programming constructs and mechanisms on software quality**," *Journal of Information Science and Engineering*, vol. 10, no. 2, pp. 177–201, 1994.

[26] J. Vincent, G. King, P. Lay, and J. Kinghorn, "**Principles of built-in-test for run-time-testability in component-based software systems**," *Software Quality Journal*, vol. 10, no. 2, pp. 115–133, 2002.

[27] H. Kopetz, "**Event-triggered versus time-triggered real-time systems**," in *Operating Systems of the 90s and Beyond*, ser. Lecture Notes in Computer Science, A. Karshmer and J. Nehmer, Eds. Springer Berlin Heidelberg, 1991, vol. 563, pp. 86–101.

[28] T. M. King, A. A. Allen, Y. Wu, P. J. Clarke, and A. E. Ramirez, "**A comparative case study on the engineering of self-testable autonomic software**," in *Proceedings of the 8th IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems (EASe'11)*, 2011.

[29] P. H. da S. Brito, C. R. Rocha, F. C. Filho, E. Martins, and C. M. F. Rubira, "**A method for modeling and testing exceptions in component-based software development**," in *Proceedings of the 2nd Latin-American Conference on Dependable Computing (LADC'05)*. Berlin, Heidelberg: Springer-Verlag, 2005.

[30] L. Pau, "**Survey of expert systems for fault detection, test generation and maintenance**," *Expert Systems*, vol. 3, no. 2, pp. 100–110, 1986.

[31] A. Beer and M. Heindl, "**Issues in testing dependable event-based systems at a systems integration company**," in *Proceedings of the 2nd International Conference on Availability, Reliability and Security (ARES'07)*, 2007.

[32] G. Fraser and A. Arcuri, "**1600 faults in 100 projects: Automatically finding faults while achieving high coverage with EvoSuite**," *Empirical Software Engineering*, pp. 1–29, 2013, To appear.

[33] B. Baudry, V. Le Hanh, J. Jezequel, and Y. Le Traon, "**Building trust into OO components using a genetic analogy**," in *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*, 2000.

[34] B. Soubies, M. L. Meur, J. Y. Henry, and J. Boulch, "**Evaluation methods for the I and C safety softwares of nuclear power plants**," Specialists' meeting on software engineering in nuclear power plants: Experience, Issues and Directions. International working group on Nuclear Power Plant Control and Instrumentation (IWG NPPCI), 1992.

[35] J.-C. Laprie, "**Dependability—Its attributes, impairments and means**," in *Predictably Dependable Computing Systems*, ser. ESPRIT Basic Research Series, B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, Eds. Springer Berlin Heidelberg, 1995, pp. 3–18.

[36] J. M. Voas, "PIE: A dynamic failure-based technique," *IEEE Transactions on Software Engineering*, vol. 18, no. 8, pp. 717–727, 1992.

[37] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, 2004.

[38] A. Bertolino, "Software testing for dependability assessment," in *Objective Software Quality*, ser. Lecture Notes in Computer Science, P. Nesi, Ed. Springer Berlin Heidelberg, 1995, vol. 926, pp. 236–248.