Testing Non-Functional Requirements: Lacking of Technologies or Researching Opportunities?

Victor V. Ribeiro[1], Guilherme H. Travassos[1]

[1]COPPE – Federal University of Rio de Janeiro (UFRJ)
Caixa Postal 68.511 – 21.941-972 – Rio de Janeiro – RJ – Brazil
{vidigal, ght}@cos.ufrj.br

Abstract. Context: *Software testing aims to reveal failures due to the lack of conformity (defects) among functional and non-functional requirements and the implemented system. Thus, defects can be identified and fixed, improving software quality. However, despite several works emphasizing the importance of non-functional requirements (NFRs), there is an insufficient amount of software testing approaches dealing with them. The lack of NFR evaluation may be the cause of low-quality software that does not meet users need, influencing software project success.* ***Goal:*** *To organize a body of knowledge regarding NFRs and software testing approaches available in the technical literature and reveal the gaps between testable NFRs and software testing approaches.* ***Method:*** *To perform structured literature reviews to identify NFRs and software testing approaches dealing with testable NFRs. To combine both results, reveal research opportunities and organize a body of knowledge regarding NFRs and software testing approaches.* ***Results:*** *From 224 identified NFRs, 87 were described, and 47 software testing approaches observed. Only eight approaches are empirically evaluated. No testing approaches were identified for 11 testable NFRs. Furthermore, regarding the testing process, we did not observe any testing approach covering the test planning phase.* ***Conclusion:*** *Despite their importance, many testable NFRs seem not be tested due to the lack of appropriate software testing approaches yet. Also, the existing testing approaches do not cover all testing processes activities and, in general, lack empirical evidence about their feasibility and performance, making their use in software projects risky.*

1. Introduction

Software engineers use to classify software requirements into Functional and Non-functional. The first type usually represents problem domain properties expected in software and the second one represents solution domain concerns or restrictions. Despite some researchers discuss such classification weakness [Eckhardt, Vogelsang and Fernández 2016] [Glinz 2007], it is mostly used among practitioners, influencing how to handle requirements in practice [Ameller *et al.* 2012] [Borg *et al.* 2003] [Chung and Prado Leite 2009].

Non-functional requirements (NFRs) play a fundamental role on software systems success [Ameller *et al.* 2012] [Hammani 2014] [van Heesch 2011]. Moreover, contemporary software systems have NFRs as essential properties to be assured, e.g. energy efficiency, and portability are crucial features of mobile applications [Joorabchi,

Mesbah and Kruchten 2013] [Rashid, Ardito and Torchiano 2015]. However, there are no widely accepted approaches for handling this sort of requirement [Borg *et al.* 2012] [Svensson, Gorschek and Regnell 2009].

Due to the importance exerted by NFRs, software engineers need to ensure, with high confidence, that software is properly behaving in conformance with its NFRs. Software testing is a mechanism to observe such conformance [ISO-29119-1 2013]. However, it is not clear yet which software tests cover the NFRs and what is the intensity (testing phase, level, and technique) of such coverage.

This paper performs an investigation concerned with NFRs and their corresponding software testing approaches. Its motivation is to identify gaps among software testing approaches and NFRs and, therefore, organize a body of knowledge regarding the testing of NFRs and revealing research opportunities. The following investigation steps illustrate the research protocol.

At first, a structured literature review has been performed aiming at identifying what are the most frequent NFRs observed in the technical publications. The lack of consensus observed on NFRs definitions led to the execution of a coding procedure on the extracted data. Thus, the identified NFRs could be better understood, and they were organized on an initial body of knowledge.

Thereafter, it was necessary to know which the software testing approaches covers such identified NFRs. Thus, a second structured literature review was performed allowing identify software testing approaches, and which approach covers each NFR.

Finally, both NFRs and software testing approaches were combined revealing the relevant (most frequent) testable NFRs that are covered and not covered by software testing approaches.

Despite the importance of testing NFRs, the results indicate that the identified software testing approaches do not cover every more frequent and testable NFRs, and there are software testing approaches proposed to test NFRs not frequently observed. Furthermore, most of the software testing approaches are not empirical evaluated and do not fully cover the software testing phases (planning, design, implementation, execution, analysis), making their use risky.

The remainder of this paper is organized in five more sections. Section 2 shows the most frequent NFRs. Section 3 describes the identified software testing techniques and their corresponding NFRs. Section 4 combines both sets of information and map tested and non-tested NFRs. Section 5 presents threats to validity, and Conclusions are presented in Section 6.

2. Most Frequent Non-Functional Requirements

The first step to achieving the main objective of this research is to identify what are the most common NFRs. Therefore, a structured literature review (LR1) was performed aiming at finding other literature reviews presenting NFRs. LR1 was carried out in March 2015, naturally retrieving papers from 1996 to 2015, and driven by the following research question:

RQ1: What are the most frequent NFRs investigated in the technical literature?

To support the findings, a two parts search string was organized and executed in Scopus search engine. The first section of the search string aims to limit the results to find any literature review and the second part, restrict the reviews on NFRs.

("systematic review" OR "systematic literature review" OR "systematic mapping" OR "systematic investigation" OR "systematic analysis" OR "mapping study" OR "structured literature review" OR "evidence-based literature review" OR "survey" OR "review of studies" OR "structured review" OR "systematic review" OR "literature review" OR "systematic literature review" OR "literature analysis" OR "meta-analysis" OR "analysis of research" OR "empirical body of knowledge" OR "overview of existing research" OR "body of published knowledge") **AND** ("non-functional requirements" OR "non-functional software requirement" OR "non-behavioral requirement" OR "non-functional property" OR "quality attribute" OR "quality requirement" OR "software characteristic")

After the search string execution, the title and abstract of each paper were read by a researcher and classified on Included or Excluded based on two inclusion criteria: 1) Presents a literature review, a survey or similar study; AND 2) Identify non-functional requirements. Then, a second researcher analyzed the excluded papers set and reclassified them on Included or keep out. Table 1 shows the amount of papers of LR1. It is important to note one paper was manually included because it was not correctly indexed in the search engine.

Table 1. Amount of LR1 Papers

Papers Found	Excluded	Included	Manual Included	Total Included
266	252	14	1	15

The authors analyzed the 15 included papers, and extracted the following information:

- **Reference information:** it aims to identify the paper by title, author, and publisher.
- **Abstract:** it is intended to contextualize the research when to query the form.
- **Study Type:** it identifies the type of study, e.g. systematic literature review, survey, and so on.
- **System Domain/Type:** it identifies the system type or domain in which the research has been done.
- **Non-Functional Requirements:** it identifies the NFRs presented in the paper and their description when presented.

2.1. LR1 Results

After information extraction, it was possible to observe a significant amount of NFRs. Besides, during data extraction, we also noted the lack of agreement regarding NFRs names and descriptions. In some cases, different NFRs stands for the same description, for instance, fault tolerance [Yang *et al.* 2014] and robustness [Bajpai and Gorthi 2012] are equally described, i.e. they represent the same software property. In other cases, there are NFRs with the same name, but different descriptions, e.g. performance can be related to resource consumption [Ameller *et al.* 2015] and time behavior [Montagud, Abrahão and Insfran 2012].

This scenario makes the organization of a body of knowledge regarding NFR hard since it does not allow to know the straightforward meaning of an NFR. Thus, it was necessary to analyze the descriptions of each NFR to understand correctly the

software property concerned with it. Then, the NFRs identified were split into two groups, the ones with and the ones without descriptions, as it can be seen in Table 2.

Table 1. NFRs with and without Description

All Identified NFRs	224
NFRs with Description	87
NFRs without Description	137

2.1.1. Understanding and Organizing NFRs

The next step comprises the understanding of each NFR for organizing them into a body of knowledge. For the sake of comprehensibility, the group of NFRs without description was not considered at this point.

To organize all described NFR into a body of knowledge, we performed open coding, as described in Grounded Theory [Strauss and Corbin 1998]. Figure 1 shows an example of resulting code on performance definition where the first box is the final performance definition extracted from subject papers. For instance, the text highlighted in blue associate performance on resource consumption and the text highlighted in green to time behavior.

Coded performance definition: system capability to execute functionalities with specified amount of resource and properties related to system time behavior.

Performance describes the timeliness aspects of the software systems behavior. It, therefore, includes quality attributes such as latency, throughput and turn-around. Performance may also be used to describe the utilization of resources in order to provide a particular service (e.g. memory and CPU usage). [Ameller *et al.* 2015]

Requirements that specify the capability of software product to provide appropriate performance relative to the amount of resources needed to perform full functionality under stated conditions. [Mairiza, Zowghi and Nurmuliani 2010]

The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions. [Yang *et al.* 2014]

Deals with response time, which means the time taken by the system to load, reload, screen open and refresh times etc… [Bajpai and Gorthi 2012]

Performance is concerned with how long it takes the system to respond when an event occurs. Efficiency and throughput are examples that belong to performance. [Poort *et al.* 2012]

Figure 1. Open coding example

The coding process allowed identifying a hierarchical structure at which NFRs were organized. That structure is shown in Figure 2 in which the class **NFR** represents high-level abstract system properties such as Usability, Security, and Performance. These properties are perceived through a set of **Sub_NFR**, which is also an NFR but,

they represent more accurate system properties such as Navigability (Usability), Confidentiality (Security), or Resource Consumption (Performance). Moreover, some NFRs may allow **Operationalization** which are features that must be present on the system for it meets the NFR. For instance, the usage of an image compression algorithm is one operationalization of Resource Consumption.

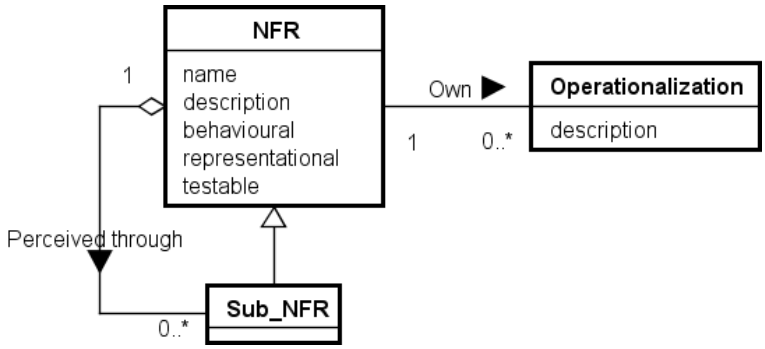

Figure 2. Body of knowledge structure

Besides the proposed structure, it was possible to identify additional NFRs' properties that help to understand better and define relationships among them. These properties are:

- **Description:** NFR definition. Usually, an NFR description explains some system's capability, e.g. performance: It is the system capacity to provide appropriate use of resources (memory, CPU) needed to perform full functionality under stated conditions.
- **Synonyms:** names having the same meaning, e.g. reliability is presented as a synonym of dependability.
- **Composed by:** other NFRs that are part of the main NFR, e.g. scalability and timeliness compose performance.
- **Target Object:** system element by with the NFR can be observed, e.g. performance: system performance (how the system is using memory during execution), function performance (what is the response time of specific function observing the messages among system functions), interaction with user performance (response time observing user request and time until response).
- **Observed Through:** how the NFR can be observed or how the software exposes it, e.g. performance can be observed through resources monitoring or time observation in execution time.
- **Specification Examples:** suggest how to specify an NFR, e.g. usability can be observed through user feedback.
- **Operationalization:** system characteristics, properties or features influencing an NFR, e.g. a reliability operationalization defines procedures to recovery the system for each type of failure.
- **Risks:** it is related to non-compliance with an NFR, e.g. risks of no compliance with availability requirement: loss of business opportunities or slow productivity.

- **Who is affected by:** the roles directly affected by the NFR, e.g. Internal Stakeholders, Owner, Manager, Software Engineer, Programmer, Final User.
- **Mentioned by:** list of papers identifying the NFR, but not describing it.
- **Defined by:** list of papers identifying and describing the NFR.

During the body of knowledge organization, it was observed that some NFRs should not be assessed through software testing because testing is a dynamic process meaning it performs verification on software properties that can be observed on execution time. So, properties that cannot be observed during system execution are not testable. Identify if an NFR is testable is essential for this research. It makes no sense to point towards a lack of testing approaches covering these NFRs knowing that they are not testable.

Thus, determining whether an NFR is testable is twofold. First, NFR must allow operationalization. Some NFRs represent abstract properties that cannot be directly operationalized, so cannot be observed through system execution, and then cannot be tested. For instance, Security does not allow operationalization but, it can be perceived through sub-NFRs Confidentiality, Auditability, and Vulnerability which in turn allow operationalization. Thus, Security cannot be directly tested although verification of system security can be performed by testing their sub-NFRs.

Second, NFR must represent a behavior. Some NFRs represent static software properties that cannot be observed during software execution and then cannot be tested, e.g. maintainability. Thus, we used the concept of behavior and representational NFR [Borg *et al.* 2012] by classifying them with the following properties:

- **Behavioral:** it defines if an NFR represents a software behavior, e.g. “system services must response every request at most one second”. Behaviors properties can be observed in execution time, therefore can be tested.
- **Representational:** it represents syntactical or technical software properties, e.g. “Software must use MySQL database”. Representational properties are static properties, and so they cannot be tested. However, it can be checked.

Therefore, an NFR can be classified as testable if it allows operationalization and represents a behavior property.

Next, to ease the exploration, we used a wiki format[1] to represent the open coding procedure results and facilitating the body of knowledge user navigation through hyperlinks. Figure 3 presents part of the NFR body of knowledge. It is possible to observe that it is hierarchically structured. For instance, usability is composed by understandability, accessibility, satisfaction, learnability, organization, and attractiveness. Furthermore, bolded NFRs allow operationalization and they are classified as behavior property meaning they are assessable through testing. The number inside the brackets is the amount of papers referencing the NFR and gives an NFR popularity indication.

[1] http://lens.cos.ufrj.br/nfrwiki

Clicking in each NFR is possible to access full information about selected NFR, as described in this section.

Figure 3. Catalog example

3. Software Testing Approaches for NFRs

The second structured literature review (LR2) aims to identify proposed software testing approaches concerned with NFRs and their testing covering. In this context, the testing covering is regarding software testing phases, levels, techniques, and kind of evaluation of the proposed approaches. Unlike LR1, LR2 does not look at other literature reviews because previous *ad-hoc* searches do not retrieve this sort of study concerning testing approaches for NFRs. LR2 was performed in March 2016, naturally retrieving papers from 1991 to 2005, and driven by the following research question:

RQ1: What are the software testing approaches used to test NFRs?

As in LR1, to support the finding of works, a two parts search string was organized and submitted to the Scopus search engine. The first section of the search string aims to limit the results to software testing approaches and the second one restricts the search to non-functional requirements.

("software test design" OR "software test suite" OR "software test" OR "software testing" OR "system test design" OR "system test suite" OR "system test" OR "system testing" OR "middleware test" OR "middleware testing" OR "property based software test" OR "property based software testing" OR "fault detection" OR "failure detection" OR "GUI test" OR "Graphical User Interfaces test" OR "test set" OR "non-functional testing" OR "model based testing" OR "test case" OR "testing infrastructure" OR "testing approach" OR "testing environment") **AND** ("non-functional requirements" OR "non-functional software requirement" OR "non-behavioral requirement" OR "non-functional property" OR "quality attribute" OR "quality requirement" OR "software characteristic")

The filtering process followed a similar procedure as in LR1. The inclusion criterion was: "The work presents software testing procedure, technique, or any other type of proposal concerned with the testing of non-functional requirements". Table 3 shows the amount of papers regarding LR2. Three papers were manually included because they were not directly available through the Scopus search engine.

Table 2. Amount of LR2 Papers

Papers Found	Excluded	Included	Manual Included	Total Included
331	287	44	3	47

The 47 papers were analyzed using an extraction form with the following fields:

- **Reference Information:** it aims to identify the paper by title, author, and publisher.
- **Abstract:** it aims to give an overall idea of the paper subject.

- **System Domain/Type:** it indicates whether the approach is proposed to specific software domain or type e.g. embedded systems, telecommunication systems.
- **Test Phase:** testing covering regarding the testing process phases: Planning, Design, Implementation, Execution, and Analysis.
- **Test Level:** testing granularity, with the options Unit, Integration, System, Acceptance, Not Informed, and Not Applied.
- **Test Technique:** with the options Structural, Functional, Fault Based, Not Informed and Not Applied.
- **Evaluation:** it represents how the software testing approach has been evaluated e.g. proof of concept, experiment, case study, simulation, not applied, and not informed. Evaluation values emerged from the subject papers.
- **Non-Functional Requirements Considered:** it represents the list of NFRs considered by the software testing approach with their description.

3.1. LR2 Results

Table 3 shows an overview of the test approaches identified including the publication year, system type, evaluation type, and covered NFRs. The complete list of characteristics extracted from each approach can be found on Wiki.

Table 3. Software Testing Approaches

#	Year	System Type	Evaluation	Non-functional Requirements
1	1999	Real-time distributed	Proof of Concept	Timeliness;
2	2001	Web application	Proof of Concept	General Approach;
3	2004	Service Oriented	Proof of Concept	Performance: Throughput; Reliability;
4	2005	Distributed component	Proof of Concept	Performance: Latency, Throughput, Scalability;
5	2007	Component-based	Experiment	Performance: Time, Efficiency;
6	2007	Component-based	Proof of Concept	General Approach;
7	2007	General	Case Study	Performance: Resource Consumption, Time; Reliability: Recoverability; Interoperability;
8	2008	General	Proof of Concept	Performance: Timeliness, Process capacity, Resource consumption; Reliability;
9	2008	Web Services	Proof of Concept	Reliability: Fault Tolerance;
10	2008	Web Services	Proof of Concept	General Approach;
11	2008	General	Proof of Concept	General Approach;
12	2009	General	Not applied	Performance: Execution Time; Quality of Service; Security: Vulnerability; Usability: Navigability; Safety;
13	2009	Component-based	Experiment	Performance: Time, Efficiency;
14	2009	Embedded real time	Proof of Concept	Performance: Timeliness: Response Time;
15	2009	Distributed component	Experiment	Performance: Response Time, Processing Time;
16	2009	Embedded	Case Study	General Approach;

17	2009	Component-based	Proof of Concept	Reliability;
18	2009	Web application	Proof of Concept	Performance: Scalability, Timeliness; Usability: Navigability;
19	2009	Distributed component	Case Study	Performance: Response Time;
20	2010	General	Proof of Concept	General Approach;
21	2010	General	Proof of Concept	General Approach;
22	2010	Web application	Proof of Concept	Security: Vulnerability;
23	2010	General	Experiment	General Approach;
24	2010	Object Oriented	Not informed	General Approach;
25	2010	Distributed	Not applied	Performance: Timeliness, Scalability; Security: Vulnerability; Correctness: Avoid deadlock, Checking conformance; Reliability;
26	2011	General	Experiment	Performance: Scalability;
27	2011	Embedded	Proof of Concept	Performance: Response Time;
28	2011	Reactive	Proof of Concept	Reliability: Fault Tolerance;
29	2011	Embedded	Proof of Concept	Reliability: Fault Tolerance;
30	2012	Not Specified	Case Study	Reliability: Fault Tolerance;
31	2012	Rich Internet Application	Proof of Concept	Usability: Accessibility;
32	2013	General	Proof of Concept	Performance: Response Time; Availability; Usability:Organization, Accessibility: Interactive;
33	2013	Service Oriented	Experiment	Security: Confidentiality, Integrity, Authenticity; Repudiation (non-repudiation); Reliability: Fault Tolerance, Availability;
34	2013	Embedded real time	Experiment	Performance: Resource Consumption;
35	2013	Elastic Computing	Simulation	Elasticity: Plasticity, Resonance;
36	2013	General	Proof of Concept	Performance: Response Time, Throughput;
37	2013	Using ASTERIX protocol	Experiment	General Approach;
38	2013	Interactive	Proof of Concept	Usability: Effectiveness, Efficiency;
39	2013	Using HTTP protocol	Case Study	Performance: Response Time;
40	2013	DB2 Database	Proof of Concept	Performance: Response Time, Execution time;
41	2014	General	Proof of Concept	Security;
42	2014	Real time	Case Study	General Approach;
43	2014	Embedded	Proof of Concept	Performance: Resource Consumption, Timeliness; Reliability: Fault Tolerance; Security: Vulnerability;
44	2014	General	Proof of Concept	Performance: workload, timeliness, think time, Rampup time, Startup delay; Reliability;
45	2015	Embedded	Not informed	Performance: Energy consumption;
46	2015	Web application	Proof of Concept	Performance: Response Time, Throughput, Simulated workload; Security: vulnerabilities;
47	2015	General	Case Study	General Approach;

Data extracted was analyzed to better understand the identified software testing approaches. First, Figure 4 summarizes the system types. It is possible to observe that most approaches are generic i.e. they are not intended to a particular system type. It is

important to note that some values could be joined e.g. systems using HTTP protocol presumably is a web application. However, we prefer to keep the original descriptions to avoid lack of precision in future approaches evaluation.

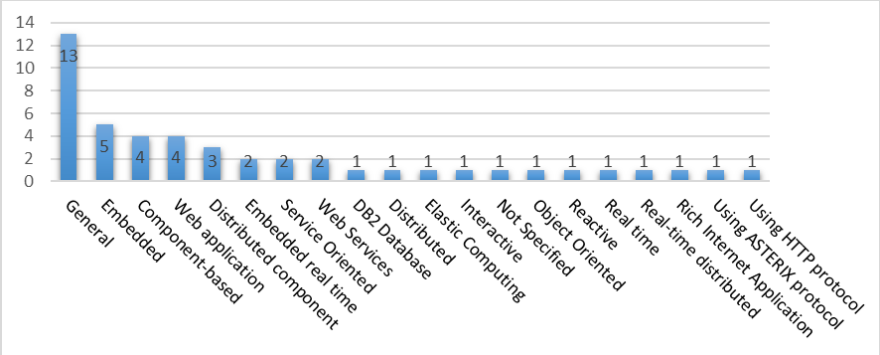

Figure 4. System type

The findings related to the type of evaluation are shown in Figure 5. It is possible to observe that most approaches are evaluated through proof of concepts, a strategy having a very low degree of confidence. It can result in uncertainty of benefits and risks related to practical use of such software testing approaches.

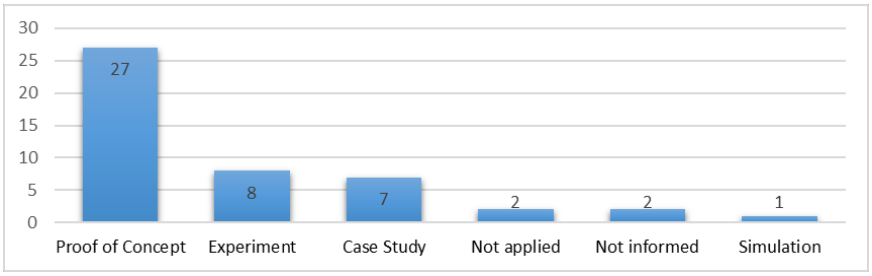

Figure 5. NFR Testing approach evaluation

In sequence, the analysis related to the testing phases is shown in Figure 6. It is possible to observe that most software testing approaches cover design and implementation, some of them cover execution and very few cover analysis. However, the test plan phase is the most alarming because none of the observed approaches is dealing with it.

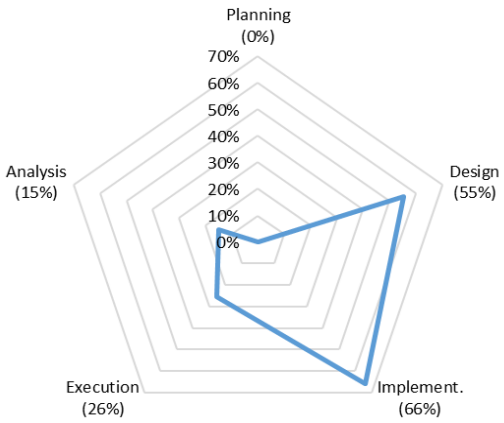

Figure 6. Testing phase

Regarding testing level, most of the identified approaches proposed to solve some aspect of system testing, as shown in Figure 7. It sounds like a warning because there is no information in the dataset about researches aiming at identifying what is the adequate level to test each NFR.

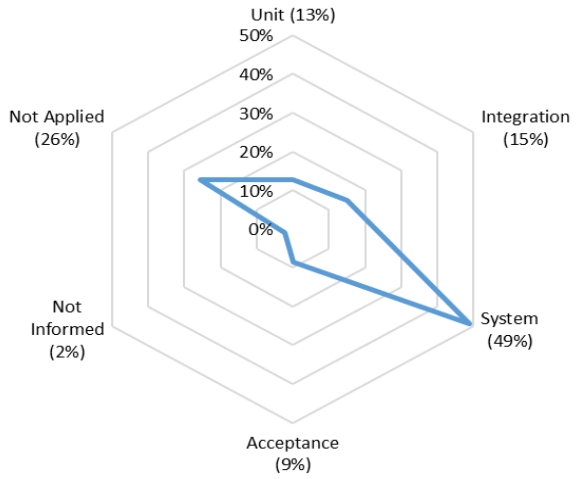

Figure 7. Testing level

Finally, Figure 8 indicates that functional testing is the most regular one. It is coherent with the results concerned with testing level since system testing level usually demands functional techniques.

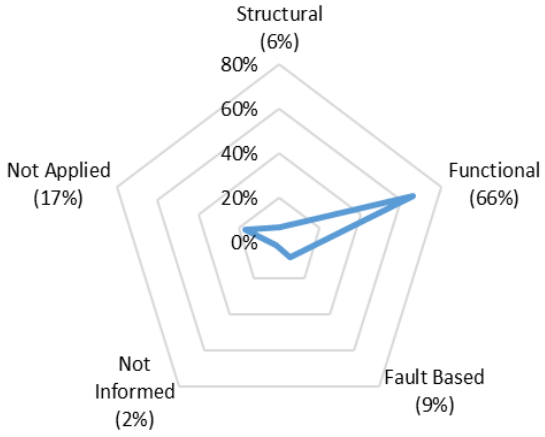

Figure 8. Testing technique

4. Combining LR1/LR2 Results

With LR1 and LR2 completed, the combination of their results can be done. The result is knowledge about what are the most frequent NFRs without corresponding software testing techniques. Figure 9 summarizes this matching where 11 of 87 testable NFRs miss a software testing approach. The first number inside the brackets is the amount of papers referencing the NFR, and the second one is the amount of software testing approaches dealing with the NFR.

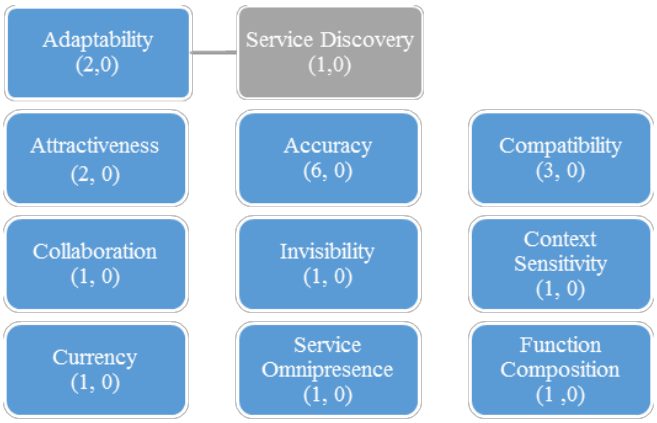

Figure 9. Testable NFR without testing approach

After that, the combination was done in the opposite direction to identify the software testing approaches dealing with less frequent NFRs. Figure 10 summarizes this result in which seven approaches are proposed to less frequent NFRs.

It is important to note that correctness and elasticity are not directly covered by any identified testing approach, but they are displayed to keep the structure presented in the paper proposing the software testing approach. The reasons for these proposed approaches need a more detailed analysis, but we can realize two explanations on them. First, the lack of clear research agenda to justify the investigation of such approaches, rather the self-motivation of researchers. It means that effort was expended in the creation of testing approaches covering less frequent (low interest) NFRs. The second one is that these approaches have been proposed for a particular context in which these NFRs were important and needed to be verified. However, it is not possible to claim any of them without more comprehensive information from the researchers.

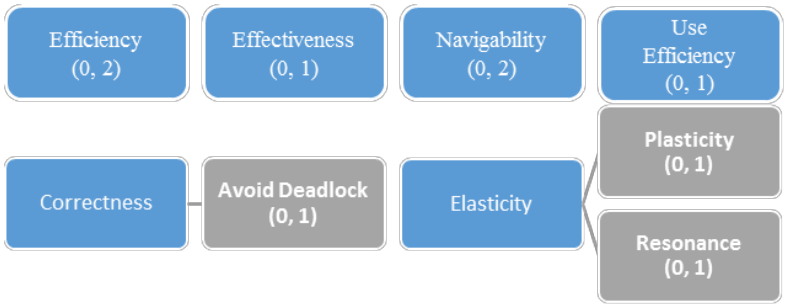

Figure 10. Less-frequent NFRs covered by software testing approach

4.1. NFRs with Software Testing Approaches

Although software testing approaches are dealing with most of NFRs, there is not enough evidence that these approaches sufficiently cover such NFRs testing. First, most of the approaches are not empirically evaluated. Second, software testing is not a simple process, depicted in different dimensions (phase, level, and technique). With this scenario in mind, data was analyzed to identify what are the testing phases covered for each NFR and the kind of empirical evaluation used.

An example regarding the analysis of Response Time and Fault Tolerance NFRs is shown in Table 4. Nine software testing approaches to deal with Response Time, none of them covering Planning, five covering Design, eight covering Implementation, four covering Execution, and three covering Analysis. Four of the approaches covering Design were evaluated through Proof of Concepts and just one through an Experiment. The one interested in further information can access the NFR body of knowledge at http://lens.cos.ufrj.br/nfrwiki.

Table 4. NFR testing approach coverage

	Testing Phase	Evaluation
Response Time:9	Planning:0	*Not applied*
	Design:5	Proof of Concept:4; Experiment:1
	Implementation:8	Proof of Concept:5; Experiment:1; Case Study: 2
	Execution:4	Proof of Concept:2; Experiment:1; Case Study:1
	Analysis:3	Proof of Concept:1; Experiment:1; Case Study:1
Fault Tolerance:6	Planning:0	*Not applied*
	Design:2	Proof of Concept: 2
	Implementation:4	Proof of Concept:3; Experiment:1
	Execution:1	Proof of Concept:1
	Analysis:0	*Not applied*

5. Threats to Validity

The main threats to validity are related to subjective evaluations carried out on this research. For instance, the open coding is an interpretative process, and it could have led us to a wrong non-functional requirements categorization. Moreover, the NFR body of knowledge was built based on NFR descriptions provided by LR1. Thus, further investigation on each particular NFR can result in the restructuring of the body of knowledge which in turn can lead to changes in this initial findings.

Furthermore, papers included in LR2 are not clear about test dimensions. Therefore, defining which a particular approach covers testing phases, levels, and techniques was an interpretative task.

Finally, another threat to validity is related to the use of a single search engine. We understand the importance to use more than one search engine to improve the coverage. However, our experience on undertaking other secondary studies indicates that Scopus can give a reasonable coverage saving much effort on removing duplicates and reading false positives. Further, LR protocol may be used to guide future executions using other search engines.

6. Conclusions

The importance of NFRs to assure software systems success increases their need for testing. This work is a preliminary initiative to understand which software testing approaches deal with NFRs, and, in consequence, to identify the gaps between frequent NFRs and software testing approaches.

The analysis of two structured literature reviews showed a set of NFRs not covered by software testing approaches. Additionally, regarding software testing phases (planning, design, implementation, execution, and analysis) and strategies (level and technique), among the identified software testing approaches no one covers the full testing process for any NFR. It is most critical when considering the planning phase that is entirely uncovered.

Furthermore, most of the identified software testing approaches are not empirically evaluated resulting in the lack of evidence on their benefits and risks in practical use. All of these issues can represent exciting research challenges.

7. Acknowledgments

This work is part of "CAcTUS - Context-Awareness Testing for Ubiquitous Systems" project partially financed by CNPq – Universal 14/2013 (484380/2013-3). Prof. Travassos is a CNPq Researcher (305929/2014-3).

References

Ameller, D., Galster, M., Avgeriou, P. and Franch, X. (2015) "A survey on quality attributes in service-based systems, Software Quality Journal", Kluwer Academic Publishers.

Ameller, D., Ayala, C., Cabot, J. and Franch, X. (2012) "How do software architects consider non-functional requirements: an exploratory study", 20th IEEE International Requirements Engineering Conference, 41–50.

Bajpai, V. and Gorthi, R. (2012) "On non-functional requirements: a survey", Conference on Electrical, Electronics and Computer Science: Innovation for Humanity, SCEECS.

Borg, A., Yong, A., Carlshamre, P. and Sandahl, K. (2003) "The bad conscience of requirements engineering: an investigation in real-world treatment of non-functional requirements", 3rd Conference on Software Engineering Research and Practice in Sweden (SERPS).

Chung, L. and do Prado Leite, J. C. S. (2009) "On non-Functional Requirements in Software Engineering", In conceptual modeling: foundations and applications, 363-379. DOI=10.1007/978-3-642-02463-4_19.

Eckhardt, J., Vogelsang, A. and Fernández, D. M. (2016) "Are 'non-functional' requirements really non-functional?", 38th International Conference on Software Engineering (Austin, Texas).

Glinz, M. (2007) "On non-functional requirements", 15th IEEE International Requirements Engineering Conference, 21-26.

Hammani, F. Z. (2014) "Survey of non-functional requirements modeling and verification of software product lines", IEEE Eighth International Conference on Research Challenges in Information Science (RCIS).

ISO/IEC/IEEE 29119-1. (2013) "Software and systems engineering - software testing".

Joorabchi, M. E., Mesbah, A. and Kruchten, P. (2013) “Real challenges in mobile app development”, ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 15-24. DOI=http://dx.doi.org/10.1109/ESEM.2013.9.

Mairiza, D., Zowghi, D. and Nurmuliani, N. (2010) “An investigation into the notion of non-functional requirements”, Conference of 25th Annual ACM Symposium on Applied Computing. (March 2010). 311-317.

Montagud, S., Abrahão, S. and Insfran, E. (2012) “A systematic review of quality attributes and measures for software product lines”, Software Quality Journal, Vol. 20 (3-4), pp. 425-486.

Poort, E.R., Martens, N., Van De Weerd, I. and Van Vliet, H. (2012) “How architects see non-functional requirements: Beware of modifiability”, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 37-55.

Rashid, M., Ardito, L. and Torchiano, M. (2015) “Energy consumption analysis of algorithms implementations”, ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). 1-4. DOI= http://dx.doi.org/10.1109/ESEM.2015.7321198.

Strauss, A. and Corbin, J. (1998) “Basics of qualitative research: techniques and procedures for developing grounded theory”, SAGE Publications, 2nd Ed. London.

Svensson, R. B., Gorschek, T. and Regnell. B. (2009) “Quality requirements in practice: An interview study in requirements engineering for embedded systems”, Foundation for Software Quality, volume 5512 of Lecture Notes in Computer Science. Springer.

van Heesch, U., Avgeriou, P. (2011) “Mature architecting - a survey about the reasoning process of professional architects”, Software Architecture (WICSA), pp. 260–269.

Yang, Z., Li, Z. C., Jin, Z. and Chen, Y. (2014) “A systematic literature review of requirements modeling and analysis for self-adaptive systems”, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Springer Verlag, Vol. 8396 LNCS, 55-71.