Title: A Benchmarking Process to Assess Software Requirements Documentation for Space Applications

Author: Paulo C. Véras Emilia Villani Ana Maria Ambrosio Marco Vieira Henrique Madeira

PII: S0164-1212(14)00240-4
DOI: http://dx.doi.org/doi:10.1016/j.jss.2014.10.054
Reference: JSS 9409

To appear in:

Received date: 27-12-2013
Revised date: 29-10-2014
Accepted date: 30-10-2014

Please cite this article as: Véras, P.C., Villani, E., Ambrosio, A.M., Vieira, M., Madeira, H.,A Benchmarking Process to Assess Software Requirements Documentation for Space Applications, *The Journal of Systems and Software* (2014), http://dx.doi.org/10.1016/j.jss.2014.10.054

Highlights

- We propose a benchmark to assess the quality of space software specifications.
- The benchmark includes 3 checklists, each composed by a large number of questions.
- Two checklists verify compliance with PUS standard and faulty behaviour.
- One checklist verifies the absence of typical documentation errors.
- We validate the benchmark using the concept of fault injection.

A Benchmarking Process to Assess Software Requirements Documentation for Space Applications

Paulo C. Véras, Emilia Villani, Ana Maria Ambrosio, Marco Vieira, and Henrique Madeira

Abstract—Poorly written requirements are a common source of software defects and, in application areas like space systems, the cost of malfunctioning software can be very high. This work proposes a benchmarking procedure for assessing the quality of software requirements that adopt the Packet Utilization Standard (PUS) defined by the European Cooperation for Space Standardization (ECSS) standards. The benchmark uses three checklists that aim at guaranteeing that the specifications comply with the PUS standard, consider faulty behaviour, and do not include errors typically found in this type of documents. The benchmark is defined for two services of the PUS standard: the telecommand verification and on board operating scheduling. A benchmark validation approach is also proposed in the paper. It uses the concept of fault injection to insert known errors in software requirements specification documents. The benchmark validation is performed through its application to three projects from different countries. Results show that our proposal provides a simple and effective way for identifying weaknesses and compare the degree of maturity of requirements documents.

Index Terms—benchmarking, ECSS standards, software requirements quality, space systems

1 INTRODUCTION

This work addresses the problem of assessing the quality of software requirements specifications for space applications.

The proposed approach is based on a benchmark for software requirements documentation, whose purpose is to guide/help the review of the requirements for onboard computer software. The benchmarking concepts in this paper are adapted and expanded from previous works on dependability benchmarking, in particular from [35], [19] and [25]. The focus on dependability benchmarking concepts is motivated by the fact that the proposed approach aims at detecting specification problems that may compromise the dependability of the software product under development. A dependability benchmark is a standardized procedure that allows assessing and comparing dependability attributes of computer systems. It typically includes dependability metrics (used to characterize the system under benchmarking), a workload (that allows exercising the system), a faultload (injected during the workload execution to test dependability features), and an experimental procedure (that defines the set of steps needed to implement and run the benchmark and calculate the metrics) [19].

Our benchmark can be performed at the beginning of the space software development process to prevent defects from propagating to later phases. Unlike other benchmarking works that target the assessment of computer programs (e.g. [26], [32]), the target of the proposed benchmark is a document: the software requirements specification.

Unlike other benchmarks for computer systems described in the literature, in our work there is no workload to run. Instead, a checklist composed of questions replaces the workload and is used to obtain measures that portray specific characteristics of the software requirements specification document. As the checklists greatly influence the benchmark representativeness, we use three sources of information for defining three different checklists with three different purposes. The first is based on the description of the services provided by the Packet Utilization Standard (PUS) [11] defined by the European Cooperation for Space Standardization (ECSS), and basically aims at guaranteeing that the software requirements specification (SRS) complies with the standard (regarding its content). The second is based on the CoFI methodology [3]. CoFI stands for Conformance and Fault Injection and it is a model-based methodology to drive the conformance and robustness test cases generation. In this work, we use the models of CoFI to generate questions for the checklist with the aim of verifying whether the SRS covers system and operation failure situations. The third checklist, based on realistic errors found in SRSs of real-world projects of space systems, aims at assessing if the document has typical specification mistakes.

A. Problem Domain and Motivation

In the last decades, the number of functions implemented by onboard space software has increased, as well as the need for processing data onboard and for adopting efficient protocols to communicate with the ground control system. The traditional approach of focusing the verification and validation efforts on the extensive testing of the software at the end of the development lifecycle has become inefficient and costly. In fact, the late detection of specification errors contributes to significant schedule delays and brings a high cost in terms of rework [27]. Additionally, there are many evidences that errors in the requirements can lead to serious problems during software development and usage [17]. Ultimately, problems detected at the end of the development cycle may compromise the entire space mission timeline.

This work provides a systematic process for characterizing the quality of requirements specifications of embedded software for space applications that adopt the PUS. The central goal of the work is to provide a way for comparing requirement specifications and the following use cases are envisaged: (1) comparison among documents provided by different suppliers; (2) comparison of a document against a threshold in order to approve it or not before proceeding to a formal review; (3) comparison of the document with itself in different development phases; and (4) unbiased comparison between the requirements

- *Paulo C. Véras, Emilia Villani:Department of Mechanical Engineering, Instituto Tecnológico de Aeronáutica, Pr. Mar. Eduardo Gomes, 50, São José dos Campos, SP, 12228-900, Brazil. E-mail: pauloveras@gmail.com, evillani@ita.br.*
- *Ana Maria Ambrosio: Space System Division, National Institute for Space Research, Av. dos Astronautas, 1758, São José dos Campos, SP, 12227-010, Brazil. E-mail: ana.ambrosio@inpe.br .*
- *Marco Vieira, Henrique Madeira: Department of Informatics Engineering, University of Coimbra, 3030, Coimbra, Portugal. E-mail: mvieira@dei.uc.pt, henrique@dei.uc.pt.*

document of a current mission with documents from previous successful missions.

B. Research Goal and Research Questions

The research goal of this paper is to define a benchmark for assessing the quality of requirement documents based on PUS standard. In order to fulfil this goal, we proposed a set of checklists and we define the following research questions that are addressed in the paper:

- Does the set of checklists identify errors in the requirement specification document?
- Are these errors representative of the quality of requirement documents?
- Does the set of checklists fulfil the properties of a benchmark?

C. Scope of the Work

The proposed benchmark is restricted to software that follows the PUS standard (ECSS-70-41A [11]). The PUS is part of the ECSS standards and addresses the utilisation of telecommand and telemetry for the remote monitoring and control of a spacecraft. It defines a set of services that covers all the fundamental requirements for spacecraft operation. Space industry and academia/research institutes in Europe, as well as in other countries around the world, including Brazil, have adopted ECSS standards. Embedded software for space systems that do not follow the PUS are not addressed in this paper. Moreover, the verification of the SRS is limited to the services described in the PUS standard. If the SRS includes other functionalities, these are not covered by the benchmark proposal.

The benchmark considers that the requirements are written in natural language, as usual in space projects. In the definition of the benchmark, we do not consider any approach that imposes modifications in the way that requirements are written. We assume that a set of experts with some knowledge of the PUS standard are available for the benchmark application, and do not address the problem of automatizing the process to make it independent of the experts' ability of processing natural language.

D. Contributions of This Work

Comparing to our previous works [37][38], this paper brings the following new contributions:

- We added two new checklists for the on board operation scheduling service. This service was chosen because it is the most complex one of PUS. The successful application of the benchmark to the on board operations scheduling service allows us to have some assurance that it can also be applied to other services.
- We added a checklist based on the most frequent errors identified in [38]. The purpose of this checklist is to verify whether the SRS under benchmarking is free of some of the most frequent errors found in software requirements documents of the same type of system. The questions that compose this checklist cover the following types of errors: lacking of traceability, requirement incompleteness, incorrectness, internal conflict/inconsistency.
- We propose a validation approach for the benchmark and illustrate this approach with the results presented in this paper. The fault operators were used to guide the injection of faults in software requirements documents with the purpose of obtaining requirement specifications with known problems that could be used to demonstrate and validate the benchmark. We used three external space projects, developed by different teams from different countries. We injected a set of realistic errors in the documents of each project and applied the benchmark. We then measured the percentage of injected errors that are detected by the procedure, and the sensibility of the results regarding the specialist that applies the benchmark. Such results are used for discussing the benchmark properties, such as representativeness, portability and repeatability, and show that it can be used in real environments, with low cost and effort.

This paper extends and integrates the works presented in [37] and [38], which are related to this paper in the following way. In [37], we presented the PUS and CoFI checklists for the Telecommand Verification service of PUS. We discussed the results obtained with the application of the checklists to an in-house project that had not yet been submitted to a formal revision process. In [38], we presented a survey on software requirements specification review reports. In this case, the goal was to characterize the most frequent errors found in software requirements documents of space systems. Once the software requirements errors were identified and classified, a set of fault operators was defined. Such fault operators allow the injection of realistic errors in software requirements documents.

E. Structure of the Text

The paper is organized as follows. Section 2 reviews related work. Section 3 presents the benchmark, in particular the definition of the three checklists. Section 4 describes the case study used to demonstrate our proposal, which includes two PUS services: the telecommand verification and the on-board operations scheduling services. Section 5 presents and discusses the results. Finally, Section 6 concludes the paper and introduces potential future work.

2 Related Work

This paper covers a wide range of subjects, such as benchmarking, software requirements quality assessment, and field studies on software requirements quality.

The dependability benchmarking problem has been studied in the last decade with the goal of evaluating and comparing the dependability of COTS (Component Off-The-Shelf) and COTS-based systems in embedded, real-time and transactional systems. Both academy and industry have proposed benchmarks focusing on a wide range of types of systems (see [35] for an extensive review on dependability benchmarking).

One example of a benchmark for embedded system is presented in [26]. The benchmark targets real-time kernels for on-board space systems and allows evaluating the impact of transient errors in the operating system (OS) of a COTS-based system (a board with two PowerPC processors running LynxOS) and quantifying their effects at both the OS and the application level. The benchmark faultload is based on the injection of faults on some key features of the OS, with the goal

of understanding the impact in aspects such as data integrity, error propagation, application termination, and correctness of application results. The ultimate goal was to provide a way for assessing the impact of space radiation in the embedded system of space applications.

Another example of a dependability benchmark for embedded systems is presented in [32] for the context of automotive systems. It proposes a practical approach to characterize the impact of faults on the behaviour of control software embedded in engine ECUs (Electronic Control Units). This was done by capturing the essential features of such type of applications in a general model, which was then exploited in order to specify a standard procedure to assess dependability measures.

Both [26] and [32] had as target characterizing the dependability of embedded systems (specifically, the embedded software). Our work also aims at assessing the impact of faults in the system behaviour. However, in our case we want to anticipate the application of the benchmark to an early phase - the requirements specification.

The assessment of the quality of requirements has been the purpose of many works, focusing on attributes that are commonly associated with the requirement quality, such as correctness, completeness, consistency, clarity, and feasibility.

Halligan [15] presents a structured methodology for measuring the quality of requirements individually and collectively, based on each requirement statement (which provides a score for individual requirements). The current work does not only evaluate individual requirements but also evaluates the specification as a whole, in order to identify missing requirements. Davis [9] proposes metrics to measure the quality of software requirements following an approach based on the assessment of each requirement according to quality attributes similar to the ones proposed in [15]. This approach provides a score that reflects the quality of the overall requirements document. Knauss [21] performed a study based on the metrics defined by Davis with the goal of finding out a threshold that helps determining whether the requirements document can be considered good enough to serve as a foundation for project success. The assessments that these three works present do not take into account the most common types of errors found in the specific domain of the software under development. In other words, they consist of generic processes that can be applied to any software requirements specification. Moreover, these works do not aim at making an extensive search for errors like in our proposal.

Suleiman et al. [36] propose the use of comprehensive integrated checklist that covers the different phases of software development, including requirements specification. The checklist was created based on a survey of literature from academia and industry and includes 183 requirements engineering items and 263 project management items. In contrast to our work, the items check if some action on some particular artefact has been performed. In our case the checklist questions verifies the content of the requirements.

Porter [30] compares three different methods for analysing software requirements specifications: ad hoc, checklist-based and scenario-based. The results showed that the scenario-based method has a higher fault detection rate than both the ad hoc and checklist-based methods. In fact, in the scenario-based study reviewers were more effective at detecting the faults, and checklist reviewers were not more effective than ad hoc reviewers. Cheng [8], in turn, compared the scenario-based technique used in [30] with inspection strategies self-set by the inspection team prior to the inspection but after they have seen the documents to be inspected. This comparison was carried out with a system developed for a commercial sector. Cheng concluded that the commercial scenarios developed for the experiment were not superior to self-set strategies. Likewise, Biffl [5] compared defect detection effectiveness and efficiency of different reading techniques in the context of requirements documents. This was done by a controlled experiment aimed at comparing a general reading technique (using checklist-based reading) to a systematic scenario-based reading technique. On the individual level, the general reading technique was found to have a higher effectiveness, while the focus of the systematic reading technique lead to a higher yield of severe defects. Relating to works presented in [30], [8] and [5], our proposal can be considered a mix of checklist-based and scenario-based methods. The scenario-based method is indirectly used when we apply the CoFI methodology to generate the checklist. The scenarios considered by CoFI are the normal behaviour, specified exceptions and sneak paths.

Hofmann and Lehner [18] identified requirements engineering practices that contribute to project success based on a survey performed with 15 requirements engineering teams. The authors identified practices that contribute to project success in terms of team knowledge, resource allocation and process. Boehm [6] and Wilson [43] developed tools to help developers analysing requirements and identifying conflicts among them, as well as tools to assess requirements by searching for terms that are quality indicators. Kim [20] proposed an approach for systematically identifying and managing requirements conflicts based on requirement partition in natural language. Using a controlled experiment, a metric-based reading technique for inspecting requirements was proposed in [4]. All of these works are either based on generic types of errors that can occur in software requirements specifications, or are focused in only one type of problem, such as requirements conflict. None of these works is based on a study about the most frequent types of problems in software requirements.

Gilliam [14] focused his work on the development of a software security checklist for the software life cycle, including, among others, the requirements gathering and specification process. The problem is that this checklist is not focused on the software requirements quality, but on the security of the process. Sheldon [34] discusses the validation of a SRS based on natural language in terms of completeness, consistency and fault-tolerance. It uses formal models to test the specified behaviour and identify problems. Our proposal is similar to [34] in the sense that we also elaborate a formal model. However, we use the PUS standard as a reference (instead of the specification document), which we consider correct and use to check the specification.

A study on faults and failures based on empirical data is presented in [16]. This study analysed the fault and failure data of two large, real-world case studies and presented a distribution of these data throughout the software lifecycle. However, the study does not take into account the specific type of faults in each phase, thus does not allow understanding the issues related to the requirements specification phase (which is precisely our

goal). Alshazly et al. [1] present a taxonomy for requirements defects and the causes of their occurrence. The main difference of this taxonomy and that of Walia [42] is that this taxonomy also includes the source of the defects. The authors propose a combined-reading technique for analysing SRS documents. As in our approach, the technique uses yes/no questions to investigate special kinds of defects. The list of questions is based on the previous knowledge of frequently occurring defects.

Lutz [23] analysed the root causes of software errors detected at the integration phase of two spacecrafts. He compared the causes/effects of safety related errors with those of non-safety related errors and showed that problems with requirement specifications are a key root cause. As a conclusion he proposed a set of recommendations for the specification phase of space projects, such as the use of formal specification techniques, the inclusion of requirements for extreme conditions or extreme values. In another work [24], the same author proposes a safety checklist for the verification of SRS documents of critical embedded system, including spacecrafts. The checklist aims at reducing interface problems and improving robustness in the case of anomalous circumstance. It was developed based on the identification of typical problems in this kind of embedded system.

Although there is a considerable number of works in the literature about requirements specification inspections and field studies, none of the existing works are based on a benchmarking process, with well-defined metrics and a well-defined scope. As a consequence, these works cannot be used to compare a specification document under development with existing documents of other space missions, or to compare different versions of the same document (to understand quality evolution over time).

Following a different line, a number of works analyses the use of formal methods, such as model checking, for verifying the correctness of a set of requirements. Usually, the specification is translated to a state transition model and verified against a set of formal properties. The model checking tool explores exhaustively the state space of the model in order to determine whether the system satisfies the properties or not. Rozier [31] presents a survey about linear temporal logic (LTL) symbolic model checking for critical applications, including works in aerospace domain. The author points that model checking contributes to the delivery of safer and reliable software, but the scalability of model checking is a problem that limits its application at industrial scale. A method for detecting semantic level inconsistency in software requirements based on state transition and model checking is described in [44]. Ogawa et al. [29] propose a method based on a goal-oriented analysis of the requirement specification. Goals specified in a natural language are systematically refined into linear temporal logic formulae. The authors use the model checker SPIN and the properties are specified in LTL. Among the problems discussed by the authors is how to specify a set of properties that is able to assure the model correctness. Pontes et al. [28] analyse the contributions of model checking using UPPAAL and CoFI testing methodology. The analysis is based on experimentation using two software products that implements two services from the PUS standard. The authors conclude that model checking contributes to the revision of the software specification, identifying ambiguities and inconsistencies. The CoFI methodology complements the model checking approach and detects failures, which are usually associated with the sneak path model of CoFI.

Comparing formal methods with the proposal of this work, model checking usually provides a larger coverture of the system behaviour, as it is based on its exhaustive exploration. However, the efforts required for the application of formal methods are significantly larger than the ones required by our approach. Moreover, in order to perform an adequate verification of the requirements, the number of details that must be incorporated into the model can turn unfeasible its verification due to the state explosion problem.

3 Benchmarking Approach for Software Requirements Specification

Our benchmark aims at assessing the quality of software requirements of space systems that follow the standards provided by ECSS. In practice, the goal is to provide a standardized way for assessing the quality of the requirements and their accomplishment regarding the PUS ECSS standard.

The proposed benchmark shall be applied during or before the System Requirement Review (SRR), which typically occurs in the early phases of the software development process [13]. In fact, the measures provided by the benchmark are useful to evaluate the quality and completeness of the requirements, showing the weak points of the documents (that need improvement). Moreover, they can be used to decide whether the requirements specification is good enough for the project to proceed to the next phase. The benchmark provides an unbiased way for comparing the requirements specification of a project with that of other projects or comparing requirements specification of different suppliers. It can also be used by several project stakeholders, including the contractor of an onboard satellite software product, the developer team (or entity), or the system integrator.

Although based on the concepts of dependability benchmarking, which aims at assessing and comparing key dependability features of computer systems or components, the purpose of our approach is to assess software requirements specifications, thereby requiring the redefinition of the main elements of a typical dependability benchmark [35], [22]. Fig. 1 illustrates the main component/elements of a benchmark.

Fig. 1. Benchmark components.

In the context of our work, the benchmark target is the requirements specification under assessment, which is a formal document specifying, in a hierarchical manner, the functional and non-functional behaviours of the system to be developed (according to the PUS standard). The benchmark metrics are quantitative and obtained by experimentation/analysis. The metrics portray the requirement specification under three different perspectives: 1) the adherence to the PUS standard, 2) the completeness and consistence of the specification when describing abnormal conditions of operation, such as failures, and 3) the absence of common mistakes.

The proposed benchmark has no workload to run or faultload

to guide the injection of faults in the system. Instead, the proposed approach is composed of three checklists, each one consisting of a number of questions that should be applied to the software requirements document under benchmarking. Each checklist is used to obtain a benchmark metric. As a result, the benchmark provides three metrics, each one calculated from the application of a given checklist. As each metric evaluates the document from a different perspective, they are not combined. According to his purposes, the user can decide to use the complete set or a subset of the 3 metrics/checklists.

The first checklist aims at verifying whether the SRS is in accordance with the PUS (PUS-based checklist). The second aims at verifying whether the SRS describes the actions that the system shall perform in the case of a failure (CoFI-based checklist). Finally, the goal of the third checklist is to verify whether the SRS has some of the most common errors found in requirements of space applications (error-based checklist). An important aspect is that a dedicated PUS-based checklist and a dedicated CoFI-based checklist have to be defined for each PUS service. On the other hand, the error-based checklist is generic and can be applied to any PUS service (and, in general, to any software requirements specification in the domain).

The benchmark experimentation is a procedure that defines the set of steps that should be conducted to apply the checklists and calculate the metrics. To calculate a given metric, the benchmark user should respond to the questions of the corresponding checklist. Each question accepts as answer just "yes" or "no". The outcome is the total number of questions answered "yes" when the checklist is applied to the software requirements document under benchmarking. An important aspect is that, if the specification does not completely satisfy the verifications defined by a given question then the answer shall be "no" (as the target of the benchmark is critical systems, there cannot be margins for errors in the assessment of the requirements). According to Eq. 1, each metric (M) is defined as the percentage of "yes" answer (YES) out of the total number of applicable questions (APP) (some questions may not be applicable, as some PUS statements are optional, depending on the specificities of the mission).

$$M = \frac{\sum YES}{\sum APP} \quad (1)$$

The definition of weights for the questions was avoided because prioritisation among the questions is subjective and may vary among benchmark users. We also avoided the definition of a threshold associated with the benchmark proposal. The acceptable threshold depends on the context of the benchmark application. For example, the requirements specification of an industrial project may adopt a lower threshold than one of a university project. It is thus a responsibility of the benchmark user to define the adequate thresholds.

Another important aspect is that we considered two of the PUS services for the definition, application and validation of the benchmark: the telecommand verification service and the on-board operations scheduling service. The former is an essential service that in general all satellites shall provide in order to validate the telecommands that are received from the ground control system. This service is implemented practically in all satellite on-board software. The latter service was chosen because it is the most complex one. The successful application of the benchmark to the onboard operations scheduling service allows us to get some guarantees that it can also be applied to other services.

The telecommand (TC) verification service provides the capability for the explicit verification of each distinct stage of the execution of a telecommand, from the on-board acceptance through the completion of the telecommand execution. The service consists of the following stages (see ECSS-E-70-41A [11] for details): (1) acceptance of the telecommand by the destination application process, which includes syntax, validity, and feasibility checking, (2) telecommand execution start, (3) intermediate stages of execution progress, and (4) telecommand execution conclusion. The telecommand verification service shall generate a report if a telecommand fails at any of its identified stages of execution. It shall also generate a report of successful completion at the same stages if this has been requested in the acknowledgment flags of the telecommand packet header. The on-board operations scheduling service provides the capability for the satellite to execute a telecommand in a moment later than the one that the telecommand arrived to the satellite. This service maintains a schedule with the telecommands that shall not execute at the moment of their arrival. Among the functionalities provided by the TC verification service we can find: inserting and deleting telecommands in the schedule, enabling and disabling the release of telecommands, and time shifting the scheduled telecommands. The release of a telecommand for execution can be interlocked with the success or failure of other telecommands. The release time of a telecommand can be specified as an absolute on-board time or as a time relative to the release of another telecommand. The schedule has a set of attributes that ensures its correct management, such as its maximum size, the list of sources that can add telecommand to the schedule, and the list of process that can receive a telecommand that has been released. The assurance that the service shall refuse to perform a request if this creates an inconsistency in the telecommand schedule is one of the critical requirements specified by the PUS standard.

It is important to emphasize that, although the benchmark proposed in this paper takes into account only these two services, it can be easily extended to the other 14 basic services (which define the capabilities to be implemented on-board a satellite along with the corresponding monitoring and control facilities on the ground). For each service, the steps performed for the definition of the first and second checklists shall be repeated. However, the third checklist, related to frequent errors in space software requirement specifications, is the same for all PUS services. In the next sections, we illustrate the definition of the checklists using as an example the telecommand verification service. Details about the definition of the checklists for the on-board operations scheduling service can be found in [39].

A. *Checklist Based on the PUS*

The questions that compose the PUS-based checklist were defined by analysing the specification of the telecommand verification and onboard operations scheduling services. Algorithm 1 was used in the definition of the checklist (and should be used for other services).

Algorithm 1 – PUS-based checklist

Select the section of the standard related to the service
For each statement do
    Create one or more questions
    If the statement is optional, **then** insert an if clause
Select the general sections of the standard
For each statement do
  If the statement is related to the service **then**
    Create one or more questions
    If the statement is optional **then** insert an if clause
For each question in the checklist **do**
  Define if the question is optional or if it is mandatory

For each mandatory statement in the telecommand verification and onboard operations scheduling services, one or more questions were defined. Additionally, to each field of the structure of the telecommand and telemetry packets described in the general section of the standard, a question was defined. Let us consider the following example to better explain the approach: one of the fields of the telecommand is the source ID. The following text describes this field in the PUS standard:

"This field indicates the source of the telecommand packet.
EXAMPLE: Different control centres on the ground or a given on-board source.
This field shall be systematically omitted if the mission has only one telecommand source or has no requirement to distinguish between sources."

One question was generated to verify whether this statement is accomplished by an SRS under benchmarking:

- If applicable to the mission, does the telecommand verification service specification state that this service shall check whether the Source ID field corresponds to a valid source of the telecommand packet?

The final PUS-based checklist of the TC verification service is composed of a set of 91 questions and the PUS-based checklist of the on-board operations scheduling service is composed of 260 questions. As mentioned before, a "yes" answer means that a given requirements specification complies with the PUS and a "no" means that the requirements specification does not comply with the PUS or has some ambiguity in the context of the question.

The structure and type of questions generated using the described approach depends on the content of the service. In the case of the TC verification, the questions of the PUS checklist can be organized in the following classes:

- Questions that verify whether the structure of telecommands and telemetries complies with the PUS standard. Example: "Does the requirement specification state that the second field of the Source Data field of the failure telecommand execution start report shall have 2 octets?"
- Questions that verify whether the value of telecommands and telemetries fields complies with the PUS standard. Example: "Does the requirement specification state that the second field of the Source Data field of the successful acceptance report shall be the Packet Sequence Control?"
- Questions that verify whether the functional behaviour of the service complies with the PUS standard (i.e., whether the service provides the functionalities specified in the PUS standard). Example: "Does the requirement specification state that this service shall generate a report if a telecommand fails at any of its identified stages of execution?"

An important characteristic is that the PUS has some aspects that are defined as optional and are specific to the mission. In this kind of situation, the user can mark the question in the checklist as not applicable to the SRS under benchmarking (APP=0). Therefore, the resulting percentage of "yes" answers (∑YES) is referred to the set of applicable questions considered by each specialist (∑APP).

In order to validate the proposed PUS checklist, an example of application was carried out with four experts who have deep knowledge about the PUS ECSS standard. The results obtained are described in [37]. The answers provided by the specialists were analysed and the questions that had different answers (by at least two specialists) were improved in order to minimize the influence of interpretation of the benchmark user. The findings of this preliminary application also helped improving the procedures of the benchmark. The full PUS-based checklist is available at [40].

B. *Checklist Based on the CoFI Methodology*

The questions based on the CoFI methodology allow verifying whether the SRS covers situations related to software failures. CoFI stands for Conformance and Fault Injection and it drives the conformance and robustness test cases generation [2]. The central idea of the CoFI methodology is to generate models (Finite State Machines - FSMs) from an SRS and, from these FSMs, automatically create test cases (in the case of the present work, manually create questions). In order to avoid state explosion in the representation of the system behaviour, these FSMs are divided into normal behaviour, specified exception, sneak path, and fault tolerance. In this work, we consider the first three classes of FSMs. The last class, fault tolerance, refers to the behaviour of the system under the occurrence of hardware faults and is not considered as it depends on the hardware used for the implementation.

A key aspect is that, instead of generating test cases, we create questions from the FSMs. Furthermore, although the CoFI methodology states that the FSMs should be generated from the software requirement specification, in this work we generate them from the PUS standard. Then, from the FSMs we generate the CoFI based checklist.

As both the CoFI-based and the PUS-based checklist are generated from the PUS standard, the same question may appear in both checklists. In this case, the question must be removed from the CoFI-based checklist. Duplicates appear mainly among the questions generated from the normal behaviour FSM and specified exception FSM. The first one describes the sequences of events that shall occur when a telecommand without any error is received from the ground control system. The second one describes the sequence of events that shall occur when a telecommand with errors that are specified in the PUS standard is received.

The sneak-path FSM describes what happens when an event occurs in an unexpected moment. This behaviour is either implicit or missing in the PUS standard but should be explicit in the SRS. The FSM representing a sneak path is created with the purpose of originating questions to verify whether the SRS de-

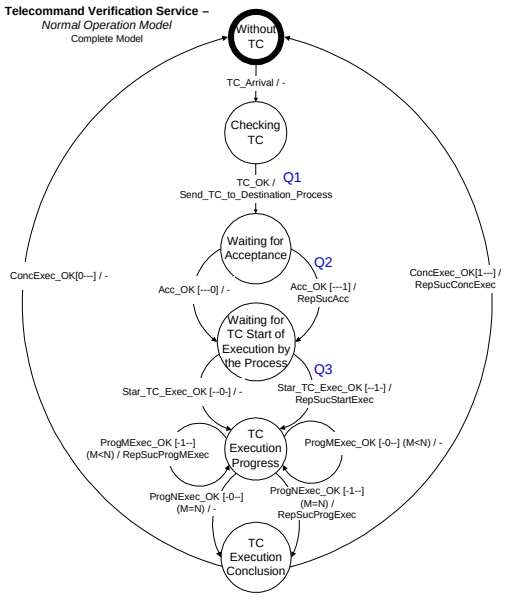

Fig. 2. Normal behavior FSM.

fines the actions the software should take in these abnormal conditions. The goal is to look for evidences in the software requirements specification documents that show that the developer team considered not only the normal behaviour of the software but also all the important cases of invalid inputs or unexpected events (sneak paths). The rationale behind this is to anticipate problems by discovering early gaps in the specification document that would potentially lead to failures.

To demonstrate the approach we have modelled the telecommand verification service of the PUS. The FSMs models the behaviour of this service when communicating with both the ground control system and the on-board application processes. The ground control system sends a telecommand to be executed on-board and waits for responses about the execution status. The telecommand verification service receives the telecommand and sends it to the application process. The application process is the part that actually executes the command and informs the telecommand verification service about its status. Based on the information provided by the application process, the telecommand verification service generates success or failure reports that are sent to the ground control system via telemetry.

Each transition of a FSM represents an expected input/output relationship and originates a question. The question must characterize the initial state of the transition and the expected input/output, as well as the specific conditions under which the transition occurs. As an example, Fig. 1 presents the FSM for the normal behaviour of the telecommand verification service. Events like TC_Arrival represent the arrival of a telecommand sent by the ground station to the telecommand verification service. Actions like RepSucAcc, RepSucProgExec are different reports carried into the telemetry and sent to the ground station. The events Acc_OK (report of success acceptance), Star_TC_Exec_OK (TC execution starting) are related to the communication between the telecommand verification service and the application process. The numbers, such as [1---] and [-0--], are reference to the criteria specified in the PUS for generating a particular report of success.

Some examples of the questions defined from the FSM in Fig. 2 are:

- Does the requirement specification state that the telecommand verification service shall send the telecommands received from the ground to its destination process after its checking? (Q1 in Fig. 2)
- Does the requirement specification state that the telecommand verification service shall send a report of success acceptance to the ground station if this is requested through the first bit set? (Q2 in Fig. 2)
- Does the requirement specification state that the verification of the TC execution starting shall occur after the acceptance confirmation by the destination application process? (Q3 in Fig. 2)

Fig. 3 illustrates the FSM representing the sneak paths behaviour. Basically, the sneak paths model considers the case of receiving a valid response from the application process at an unexpected moment. Some examples of questions defined from the sneak path FSM are:

- Does the requirement specification state the action of the telecommand verification service if it receives a confirmation of execution conclusion from the application process when it should receive a confirmation of execution start? (Q4 in Fig. 3)
- Does the requirement specification state the action of the telecommand verification service if it receives a confirmation of telecommand execution progress from the application process when it should receive a confirmation of execution conclusion? (Q5 in Fig. 3)
- Does the specification state the action of the telecommand verification service if it receives a confirmation of telecommand execution conclusion when it should receive a confirmation of successful acceptance? (Q6 in Fig. 3)

To complement the checklist, additional questions were added related to: (1) the assurance that the software can distinguish different input events (i.e., different transitions in the FSM), and (2) the absence of expected events. Examples are:

- Does the specification state that the confirmation of the progress given by the target application process shall identify the concerned step number?
- Does the requirement specification define the action of the service if some answer of the application process is not received?

By comparing the PUS and the CoFI-based checklists, we can see that the major advantage of using the CoFI methodology is the definition of the questions based on the sneak path FSM. In fact, the PUS-based checklist already includes almost all the questions based on the normal and the exceptional behaviour FSMs. However, none of the questions generated from the sneak path FSM are defined in the PUS-based checklist. Thus, the portion of the CoFI-based checklist defined from the sneak path FSM can be seen as a complement to the PUS-based checklist.

The CoFI-based checklist of the TC verification service has a total of 35 questions, while the CoFI-based checklist of the on-board operations scheduling service has 23 questions. The full checklist is available at [40], while the CoFI models for the on-board operations scheduling is available at [41].

Similarly to the PUS-based checklist, the CoFI-based checklist was revised taking into account the results of an example of application using the telecommand verification service. In this case, four specialists with experience in the PUS standard applied the checklist to the SRS of a space project. The results were compared and the questions that received different answers from the specialists were revised based on the comments of the specialists. The details of this example are described in [37].

C. *Error-based Checklist*

The last checklist of the benchmark was defined based on a survey aimed at characterizing the most frequent errors found in software requirements documents of space systems [38]. Following the definition of survey provided by [33], the work described in [38] consists of a collection of standardized information from a specific population. The study analysed the reports produced during the Software Requirement Review (SRR) of 3 different real projects of space systems. These reviews were performed by different teams of independent experts, whose goal was to assess the quality of the document by analysing key aspects of the document, such as its completeness, correctness, consistency and feasibility. The teams of experts are independent of the teams that wrote the documents, and belong to a different company that was contracted to perform the documents review. The output of the review process is a report that describes the errors found in the software requirements document.

The survey analysed a total of 7 reports, in which the errors found by the reviewers are reported as review item discrepancies (RIDs). For the 2,188 requirements analysed by the verification teams, 209 RIDs were found, which gives an average of almost one RID per 10 requirements. To classify the errors in distinct classes, the taxonomy proposed by Walia and Carver [42] was adapted. The RIDs were classified in the following classes: external conflict/inconsistency, lacking of traceability, external incompleteness, incorrectness, application knowledge, readability, non-usage of standard. The final results of this study, that indicate the most frequent types of errors found in SRSs of space applications, are listed in Table I (see details on the study in [38]).

Regarding the benchmark checklist, initially, the intent was to create questions about the most frequent errors on space software requirements. However, the first and third most frequent errors are related to discrepancies between the requirements in the document under assessment and the requirements contained in other documents (external conflict/inconsistency, and external incompleteness). The projects to which we had access to their SRSs did not give us access to related documents, such as system requirements documents or interface requirements documents. Thus, we could not use the mentioned types of errors in our benchmarking implementation. As a consequence, we created questions about the other most frequent types (the second, fourth, fifth and sixth most frequent errors found in the survey, respectively):

- Lacking of traceability: inadequate/poor requirement traceability.
- Requirement incompleteness: requirement that lacks some information to be completely understood. This includes requirement that has TBDs – To Be Defined, TBCs – To Be Confirmed, TBWs – To Be Written, and lack some numeric value.
- Incorrectness: requirement containing wrong information (related to the requirement itself) or it is not written correctly (i.e. the intended meaning is not what is in fact written).
- Internal conflict/inconsistency: conflicting requirements, or requirements that are inconsistent with other information (such as a figure) of the same requirements document.

TABLE I
MOST FREQUENT TYPES OF ERRORS IN SRSS OF SPACE APPLICATIONS

Type of Error	Percentage
External conflict/inconsistency	26.7%
Lacking of traceability	22.5%
External incompleteness	16.6%
Requirement incompleteness	15.0%
Incorrectness	6.4%
Internal conflict/inconsistency	4.8%
Application knowledge	3.2%
Readability	2.7%
Domain knowledge	1.1%
Non-usage of standard	1.0%

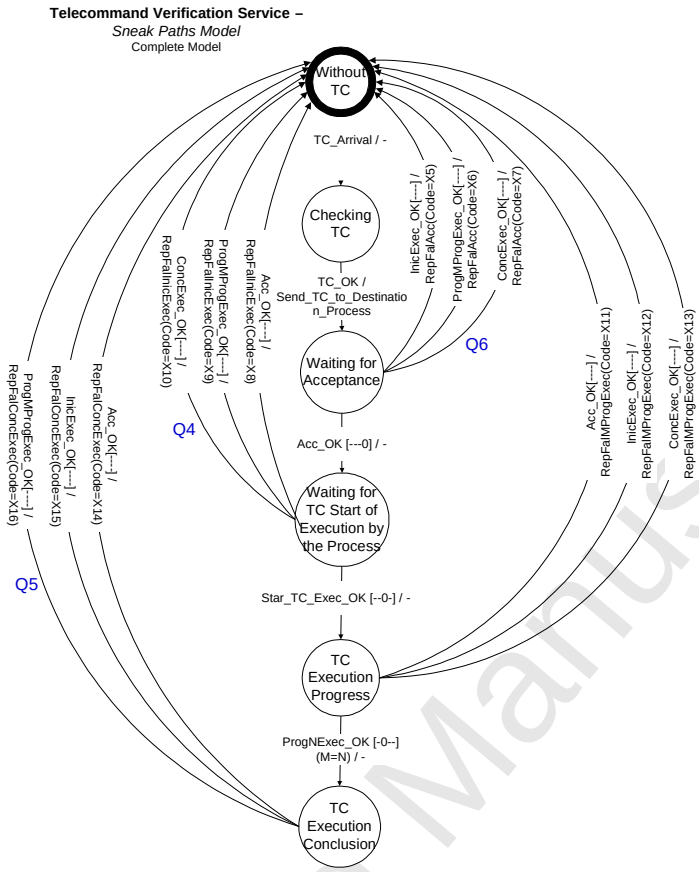

Fig. 3. Sneak path FSM

Once we have selected the types, we used the descriptions of the errors obtained from the reports used in the survey to generate the questions. The descriptions of the errors provide an idea on the kind of error that was encountered in the SRSs, so we could establish questions to identify these types of errors in new SRSs. Some examples of description and their corresponding questions are provided in the following.

Title of the RID: traceability matrix incomplete
Problem description: the traceability matrix is inconsistent. The following requirement from [AD-13] are missing: R-3.1.11; R-3.3.30; R-3.3.114
Question elaborated from this RID: In the traceability matrix of the software requirements document, are all the software requirements traced to at least a system or an interface requirement?

Title of the RID: TBC still exists in issue 5 of the software requirement document
Problem description: The TBCs should not be present in the requirements document, the TBCs must be confirmed
Questions elaborated from this RID: Are all the requirements without "TBC"?

Title of the RID: Requirement not properly captured
Problem description: Req-15900 requirement is not possible to validate. Expressions like "in preference to" should not be used
Questions elaborated from this RID: Are all the requirements without the expression "in preference" indicating a non-mandatory feature of the system?

The full error-based checklist is composed of 22 questions and is available in [40]. Table II summarizes the number of questions for each checklist.

TABLE II
NUMBER OF QUESTIONS PER CHECKLIST

Checklist	TC Verification Service	On-board Operations Scheduling Service	General Questions
Based on PUS	91	260	-
Based on CoFI	35	23	-
Error-based	-	-	22

4 CASE STUDY

To demonstrate the proposed approach, a total of 6 experts were asked to apply the benchmark to 3 software requirements specification (SRS) documents, where one of them is an SRS of an university satellite, prepared by students under professors' supervision (project 1), and the other two are SRSs of industrial projects of space applications (project 2 and project 3). For confidentiality reasons, the names of the real-world space projects cannot be provided.

TABLE III
PROFILE OF EXPERIENCE OF THE REVIEWERS

Reviewer	Writing requirements	Reviewing requirements	Participating in formal reviews	Familiarity with PUS	Familiarity with space systems
Specialist 1	Medium	Medium	Low	Medium	Low
Specialist 2	Low	Low	Low	High	Medium
Specialist 3	High	High	Low	Low	High
Specialist 4	Low	Medium	Medium	Low	High
Specialist 5	Medium	High	Medium	High	High
Specialist 6	Medium	Medium	Low	High	Medium

The number of requirements in each document is as follows:

- Project 1: 345 (17 pages)
- Project 2: 39 (3 pages)
- Project 3: 58 (13 pages)

Note that these numbers comprise only the requirements of the telecommand verification and on-board operations scheduling services and some general requirements about the structure of the telecommand and telemetry packets. The SRS of Project 1 includes in the text of the requirements the statements of the PUS, while the SRSs of Project 2 and Project 3 make direct references to the PUS. Consequently, these SRSs have a discrepancy in the quantity of requirements. As these SRSs make reference to other applicable documents or even to the PUS, the reviewers also examined these documents during the execution of the benchmark.

In order to evaluate the application of the benchmark, we injected errors in the requirements documents using a set of fault operators. The concept of fault operator has already been used successfully in traditional software fault injection [10]. In the case of requirement documents, the fault operators represent a structured description of the concrete changes that should be introduced in a requirements document in order to emulate a specific error class. A detailed description of the fault operators for requirement documents can be found in [38] and [39]. Some examples of the fault operators are:

1. Delete a reference from a software requirement to a system requirement from the traceability matrix;
2. Change the reference to a function in the software requirement text;
3. Omit a numeric value defined in a software requirement;
4. Change the type of a variable for an inconsistent type for the variable in question (e.g. change from “m/s” to “s”).

By applying the operators a set of faults was injected in the documents, as shown in the following:

- SRS1: document of Project 1 with 10% of the requirements with injected faults
- SRS2: document of Project 2 with 10% of the requirements with injected faults
- SRS3: document of Project 2 with 40% of the requirements with injected faults
- SRS4: document of Project 3 with 40% of the requirements with injected faults

These percentages were chosen based on the two results of the survey: (1) the average percentage of defective requirements for all the reports analysed in the survey, which is 9.55%; (2) the maximum percentage of defective requirements in a single report of the survey, which is 41.30%. As shown, the SRS of Project 1 has a large number of requirements, thus, it was chosen to be the SRS with the fewer percentage of injected faults.

It is important to state that we have tried to inject the faults in the documents with the same distribution in terms of types of faults, i.e. the same percentage of each type in each pair of SRS. In some cases it was not possible to maintain the proportions due to the types of requirements in the documents that do not allow the injection of some specific error types. As an example we can cite the lacking of traceability error type, which demands removing some kind of traceability information from the requirement.

Table III summarizes the experience of each expert according to the relevant characteristics presented in the first line of the table. The 6 experts conducted the benchmarking process as follows (obviously, the reviewers were not aware that there were faults injected in the SRSs):

- Specialist 1: SRS1
- Specialist 2: SRS1
- Specialist 3: SRS3
- Specialist 4: SRS2
- Specialist 5: SRS2
- Specialist 6: SRS4

The 6 specialists applied independently the benchmark to the SRSs. The purpose was to compare the results of the specialists in order to verify if the process is easily applicable, repeatable and if its application is independent of possibly different interpretations from the specialists.

5 RESULTS AND DISCUSSION

This section presents the results of the application of the proposed benchmark. Due to space restrictions, we do not present all the data, but only the relevant data and the relationships among them.

A. *OVERALL RESULTS*

Table IV lists the percentage of questions that received different answers from the specialists (and additionally from the first author of this paper). Unfortunately, specialists 1 and 2 did not answered the error-based checklist, thus we do not have these data for comparison.

The percentage of questions with different answers is an indicative of the accuracy of the metric provided by the benchmark. It should be taken into account when comparing documents or when defining thresholds, especially when different specialists conduct the benchmark.

TABLE IV
DIFFERENT ANSWER (D. A.) BETWEEN THE SPECIALISTS IN %

Reviewers (document)	PUS-based checklist	CoFI-based checklist	General checklist
Specialist 1 and 2 (SRS1)	19.8	25.7	-
Specialists 4 and 5 (SRS2)	11.9	42.9	19.0
Specialist 1 and author (SRS1)	17.6	11.4	-
Specialist 2 and author (SRS1)	28.6	37.1	-
Specialist 4 and author (SRS2)	9.5	0.00	19.0
Specialist 5 and author (SRS2)	21.4	42.9	19.0
Specialist 3 and author (SRS3)	21.3	35.0	9.5
Specialist 6 and author (SRS4)	9.2	53.7	9.5

In order to compare the checklists among them, we consider the mean value (of all specialists) obtained with each checklist as representative of the quality of the checklist. The checklist based on the CoFI methodology presents the worst result (mean of 31.1%). The PUS-based checklist, with a mean of 17.4, followed it. The error-based checklist had the best results with a mean value of 15.2%.

It is clear that the specialists had more different interpretations regarding the questions of the CoFI-based checklist. This indicates that these questions may be elaborated in a clearer and more objective way. Another matter about the questions of the CoFI-based checklist is how the documents shall be analysed to find the answer. Some of these questions are not as direct or objective as the questions of the PUS-based checklists. An illustrative example of this situation is the following question (of the CoFI-based checklist for the telecommand verification service):

"Does the document state that the starting confirmation sent by the destination application shall be different from the other confirmations sent by it?"

The SRSs used in this work do not have a requirement stating this condition using these words. However, the SRSs describe each one of the confirmations that shall be sent by the destination application process. Thus, the reviewers that had a wider look to the documents usually answered this question as "yes", while other specialists answered as "no" (stating that the information was not found in the SRS under review).

The errors detected by the reviewers, corresponding to the 'no' answers, can be associated with injected faults or previous existing errors of the SRSs. The percentage of 'no' answers associated with injected faults is 8.3%. All the remaining 91.7% of the 'no' answers where motivated by previous existing errors in the SRSs. The large percentage of previous errors detected by the benchmark has different reasons according to the checklist. For each case, the distribution of the 'no' answers per checklist is presented in Table V. All the checklists detected both injected faults and previous errors.

TABLE V
PERCENTAGES OF NO ANSWERS CAUSED BY INJECTED FAULTS OR PREVIOUS ERRORS

Injected faults			Previous errors		
PUS-based checklist	CoFI-based checklist	General checklist	PUS-based checklist	CoFI-based checklist	General checklist
2%	3%	3%	56%	32%	4%

The large percentage of previous errors detected by the PUS checklist retracts the fact that, frequently, the team that elaborates the SRS assumes that many statements of the PUS standard are known and do not need to explicitly stated as a requirement. This is a dangerous assumption because the verification of these statements will depend on the experience of the testing team. Also, there will be no traceability register for these implicit requirements. Regarding the percentage of errors detected by the CoFI checklist, most of the errors are related with the sneak path FSM, which specifies the system's behaviour when known events occur at unexpected moments. Previous works related to the application of CoFI ([28]) have shown that this kind of behaviour is usually not explicitly included in SRS. The sneak path model usually finds errors even in systems that have already been submitted to testing, verification and reviewing processes.

B. *COVERAGE RESULTS*

The percentages of the faults injected in the documents that the specialists were able to found during the benchmark execution are presented in Table VI.

TABLE VI
PERCENTAGES OF INJECTED FAULTS FOUND BY THE REVIEWERS

Reviewer (document)	Injected faults found (%)
Specialist 1 (SRS1)	47.4
Specialist 3 (SRS3)	57.1
Specialist 4 (SRS2)	33.3
Specialist 5 (SRS2)	0.0
Specialist 6 (SRS4)	65.0

Even though there are questions whose purpose is to find out all the types of injected faults in the documents, we can see that in average, the percentage of the injected faults that were not found is around 40%. The total number of injected faults not found by any of the reviewers is 20, with the following distribution:

- Incompleteness: 60%
- Missing of traceability: 20%
- Incorrectness: 20%

Such distribution was expected, since incompleteness is a very difficult type of error to find (it is much easier to find errors related to incorrectness or missing of traceability, for example). In fact, pointing out what is missing in a software requirement specification is a very hard task, even to the most experienced reviewers.

Table VII groups the average percentage of "yes" answers given by all the specialists to all the checklists in all the SRSs. As one of the purposes of a benchmark is the comparison between various solutions to the same problem, these data can be used to make this analysis. However, we cannot compare all the SRSs directly with each other, as they have different percentages of injected faults. Therefore, the pairs that we can compare are SRS1 with SRS2 (both with 10% of the requirements with injected faults), and SRS3 with SRS4 (both with 40% of the requirements with injected faults).

As we can see, SRS2 had more positive answers than SRS1, which indicates that SRS2 is a document with more quality (or fewer errors) than SRS1. In fact, as described in Section V.A, SRS1 is a document of an academic project, while SRS2 is a document of a real-world industrial project, which was written by experienced professionals and was already submitted to a review process. Comparing SRS3 and SRS4 (both documents of real-world industrial projects), the obtained data suggest that SRS4 has fewer errors than the document SRS3.

TABLE VII
PERCENTAGES OF "YES" ANSWERS GIVEN IN EACH SRS

Document	Total of "yes" answers (%)
SRS1 (project 1 with 10% of injected faults)	58.0
SRS2 (project 2 with 10% of injected faults)	70.4
SRS3 (project 2 with 40% of injected faults)	41.9
SRS4 (project 3 with 40% of injected faults)	68.3

C. BENCHMARK PROPERTIES

The meaningfulness of the results of the application of a benchmark requires the validation of its properties. In this section we discuss the validity of those properties in the context of the benchmark proposed.

Representativeness

The representativeness property is concerned with how well the checklist and metrics are defined/obtained.

For the PUS-based checklist, the representativeness is related to how well the questions cover all of the capabilities stated in the services adopted by the software. Thus, the representativeness of this checklist is strongly dependent on the way the questions are created. Considering the process defined to create the questions, the representativeness is guaranteed since it consists of creating questions for all the statements of the standard.

For the CoFI-based checklist, this property is related to the failure cases considered during the definition of the questions. Taking into account the procedures adopted in the CoFI methodology, we consider that the representativeness of this checklist shall be reasonably guaranteed since the CoFI methodology considers the faults specified in the PUS, all the possible events of the operation of the software happening in each possible state of the software, and the sneak paths of the scenarios of functioning of the software.

For the error-based checklist, the representativeness is related to how the questions are elaborated and in what errors they are based. The questions of this checklist were elaborated based on realistic errors in the application area of the benchmark. In fact, the proposed checklist is based on the errors found in a survey about the most frequent types of errors in SRSs of real-world space projects.

Portability

A benchmark process is said to be portable if it is applicable to different targets within the application area, considering the constraints of this target.

The proposed benchmark is applicable to any SRS that describes requirements of space applications that adopts the PUS as standard to describe the communication between the satellite and the ground control system.

Repeatability

This property guarantees that the same results are obtained when the benchmark is applied several times and by different people. The approach proposed in this thesis is dependent on the interpretation of the benchmark user. Thus, it is expected a fair difference on the results of the benchmark when executed by different people.

One important factor that influences the repeatability of the benchmark is the dedication of the performer to the benchmark execution task. When the performer is in doubt about a question or a requirement, if he is a meticulous person, he tends to investigate further. If he is a generalist person, he tends to answer without further investigation, based on his general impression about the document. Comparing the results of Table V with those of Table IV, the specialist that had the smallest percentage of different answers compared with the author, for the PUS and error-based checklists (Specialist 6), is also the one that detected the largest percentage of injected errors. He adopted a meticulous approach when performing the benchmark. On the other hand, Specialist 5 had the largest percentage of different answers compared with the author and detected no injected fault. He adopted a generalist approach during the benchmark. In order to mitigate the influence of this factor on the results, training of the benchmark performer should be considered.

Another important factor that influences the repeatability is the maturity of the SRS. From the three documents used in this study, two are SRSs of industrial space projects (that were already submitted to a review activity) and one document is a SRS of an academic project (whose maturity is not very high). The results obtained from the SRS of the industrial projects were, in general, more consistent.

Finally, we observed that the way the questions were elaborated (sometimes in a subjective manner) led to different interpretations by the reviewers. Additional work is needed to improve the readability, precision, completeness, etc., of the benchmark questions, especially in the case of the CoFI based checklist.

If we consider the checklist based directly on the PUS, the discrepancies among the answers varied between 10% and 20%. In the CoFI-based checklist, this value ranged between around 25% and 40%. For the error-based checklist, it was between 9% and 19%. Taking into account that this property depends upon many variables (performer profile, SRS maturity, etc.), we consider that these values are acceptable.

Scalability

To be scalable, a benchmark process shall be applicable to systems of different sizes. In general, this is strongly related to the applicability of the workload and faultload to targets of different dimensions (in our approach, the applicability of the checklists). The following points discuss the scalability aspects for each of the proposed checklists:

- PUS-based checklist: The questions generated for this checklist include all of the statements of the adopted services of the standard. In this way, there is no need to scale up this part of the checklist inside the service.
- CoFI-based checklist: Similarly to the PUS-based checklist, to scale down this checklist, the performer of the benchmarking can mark the unnecessary question(s) as not

applicable. Thus, the CoFI-based checklist can also be easily scaled down.

- Error-based checklist: The scalability of this checklist does not depend on the growth or reduction neither of the standard nor of the SRS. As this checklist is made of questions about general errors on requirements, it can be applied to any SRS of any size. This was demonstrated in the application of our proposal to different SRSs of different sizes (with different number of requirements).

Non-intrusiveness

This property is related to the nature of the proposed benchmark itself, which does not require its application to be intrusive. The process of assessing the target is by reading the requirements and answering the checklist questions about these requirements. This process does not change the software requirements specification documents under benchmarking. Thus, this property is validated.

Simplicity-of-use

Roughly speaking, the user of the benchmark just needs to read the questions and analyse whether the target accomplishes these questions. The major difficult encountered during the application of this process is about the proper interpretation of the questions and the search for the answer in the document under benchmarking. Nevertheless, this fact does not turn the application of the process itself into a hard activity, but it may compromise the result accuracy, which can be resolved with the application by more than one specialist. The only requisite is that the performer should be a specialist in space products, with a background on space systems and ECSS standards. A non-specialist may have difficulty in executing the process. This difficulty is natural; however, despite the longer time needed a person can yet apply the benchmark with no experience in ECSS standards.

D. THREATS TO VALIDITY

Following the recommendations from [33], the threats to the validity of our case study are analysed under four aspects:

i. Construct validity, i.e., "to what extent the operational measures really represent what is investigated in research questions".

The main measure adopted in the case study is the number of errors (questions of the checklist that received 'NO' as answer). The process used to define the checklist assures that these answers are related to errors, inconsistencies or gaps in the SRS. However, there is no way of assuring that these are the main problems of the SRS under analysis. One possibility to address this threat is to compare the results of the benchmark application with the results of a traditional review performed by a board of experts (as discussed in Section 6), but we already know from a controlled experiment (briefly described in [39]) that the traditional review process is strongly dependent of the board of experts that review the documents. The set of RIDs generate by a group of experts can be totally different from the ones generated by another group. Indeed, this problem is one of the main motivations of this work.

ii. Internal validity, i.e., "when causal relations are under investigation, there is a risk that the results are also affected by a third factor".

We supposed that the ability for detecting known errors (errors injected in the SRS) is indicative of the ability for detecting any previous errors of the SRS. One threat to this assumption is the fact that the fault operators are derived from the same survey that derived the third checklist. However, the results presented in Table V show that the detected errors are not limited to the injected errors. Moreover, it shows that all the three checklists are able to detect both injected and previous errors.

iii. External validity, i.e., "to what extent it is possible to generalize the findings".

The analysis of this aspect is related to the portability and scalability properties of the benchmark, both discussed in the previous section. The proposed approach is not applicable to software requirements specification documents that do not follow the PUS. However, it can be adapted to other standards. In fact, by following the same procedures to elaborate the questions based on the PUS, it is possible to elaborate a checklist based on other standards.

iv. Reliability, i.e., "to what extent the data and the analysis are dependent on the specific researchers".

The analysis of this aspect is related to repeatability property of the benchmark. The execution of the benchmark is highly dependent on the human factor, i.e. the interpretation of the reviewers to the questions and the way that they assess the documents has a high influence in the final result. Thus, it is expected that the results obtained from the application of the benchmark to the same SRS by different specialists have a discrepancy in the answers, which in the case of our study can have a mean varying from 15.2% to 31.1%.

6 CONCLUSIONS AND FUTURE WORK

This work presented a benchmarking approach for software requirements specifications for space applications. This benchmark is based on three checklists that help assessing specific characteristics of a requirements specification document. The first checklist verifies the adherence to the PUS standard, the second checklist complements the first one regarding the occurrence of unexpected events, and the third one checks for typical errors of requirement specifications.

This work is an integration and extension of the work presented in [37] and [38]. While the focus of [37] and [38] were the proposal of the first two checklists and the discussion of a survey on SRS errors, the focus of this paper is the benchmark validation. The benchmark approach was applied to three SRSs of three different projects (one academic project and two of real-world industrial projects). Six specialists performed the benchmark and the obtained measures were compared and discussed. Results show that the specialists answered some of the questions differently (21.2% of the total of the questions), which means that the proposed approach is dependent of the interpretation of the performer. This is a key aspect that has direct influence on the validation of the repeatability property.

The specification of the proposed benchmark has also addressed the validation of the key properties: representativeness,

portability, repeatability, scalability, non-intrusiveness, and simplicity of use. Except for the repeatability, all these properties were validated within the case study conducted in this work.

Once the checklists are elaborated and ready to be used, they can be reused in other projects without any change (with the constraint of being a project that adopts the PUS). With the feedback of the reviewers, we concluded that benchmarking documents can be a little confusing, since the reviewers shall read three documents (or even more, if the SRS makes reference to another documents, such as the PUS) at the same time: the procedures, the checklist, and the SRS. In the cases where the checklist needs to be adapted to other kinds of documents, the cost (in terms of effort) to elaborate the checklists is certainly lower than the cost saved by using the benchmarking process to verify documents.

To demonstrate the benchmark we used fault operators to support the process of injecting realistic faults in the SRSs under study. Therefore, the implementation of the benchmark carried out in this work used documents with injected faults with the purpose of verifying whether the reviewers would find these faults. Although the proposed checklist has a set of questions whose purpose is to find the injected faults, the average percentage of them that was found by the specialists was 40.6%. Analysing the remaining injected faults, 60% of them are errors related to lack of completeness in the requirements, which is a very difficult kind of error to be found.

The directions of future are numerous. One important step for assessing the advantages obtained with the practical use of the benchmark is its comparison with a traditional review process, usually performed by a board of external advisors. The comparison should evaluate not only the performance of the two approaches but also required time and cost.

Another possible future work would be to detail the benchmarking process by assigning a criticality level to the questions of the checklists according to the severity of the consequences of a failure in the software due to an error in the SRS. Since the criticality levels would be assigned to the questions, this could be used for defining a pass/fail criteria to the SRS based on the results of the checklist associated with the criticality of the questions.

The influence of human factors in the benchmark results is another important topic for future investigation and proposals. In order to minimize the percentage of different answers between specialists, it would be important to perform experiments with specialists that have been submitted to training activities. For the experiments presented in this paper, the specialists received a set of instructions but no formal training was provided to them. The effect of the human factor could also be minimized through the partial automation of the benchmark application process. One direction is to develop a tool that, based on the lexical interpretation of both the SRS and the checklist, helps the person to find the answers to the checklist questions. This tool could be developed considering either SRS written in natural language or in semi-formal languages.

The focus of the proposed work is restricted to documents of software requirements specification. Another possible extension of this work is to propose a benchmark to be applied in other documents of the software development life cycle, such as the software design document or the document that specifies the interface among the software items. The procedures and metrics used in our proposal can be reused in this new possible approach. The elements that should be redefined would be the benchmark target and the checklists.

ACKNOWLEDGMENT

The authors would like to thank Critical Software and all the voluntary reviewers that participated in this work. This work was partially funded by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Agência Espacial Brasileira (AEB), and Centro de Informática e Sistemas da Universidade de Coimbra (CISUC).

REFERENCES

[1] A.A. Alshazly, A.M. Elfatatry, M.S. Abougabal, "Detecting defects in software requirements specification", Alexandria Engineering Journal, 2014 (article in press) DOI: 10.1016/j.aej.2014.06.001

[2] A. M. Ambrosio, E. Martins, N. L. Vijaykumar, and S. V. Carvalho, "Systematic generation of test and fault cases for space application validation", in *Proc. 9th ESA Data System in Aerospace - DASIA*, Edinburgh, Scotland, ESA Publications, 30th May – 2nd June, 2005.

[3] A. M. Ambrosio, E. Martins, N. L. Vijaykumar, and S. V. Carvalho, "A Conformance Testing Process for Space Applications Software Services", *Journal of Aerospace Computing, Information, and Communication* (JACIC), USA, vol. 3, n. 4, pp. 146-158, 2006.

[4] B. Bernárdez, M. Genero, A. Durán, and M. Toro, "A controlled experiment for evaluating a metric-based reading technique for requirements inspection", in *Proc. of the 10th International Symposium on Software Metrics* - METRICS, Washington, DC, USA, 11th-17th Sept. 2004, pp. 257-268.

[5] S. Biffl, and M. Halling, "Software product improvement with inspection. A large-scale experiment on the influence of inspection processes on defect detection in software requirements documents", in *Proc. Euromicro Conference*, Maastricht, The Netherlands, 2000, vol. 2, pp. 262-269.

[6] B. Boehm, and H. In, "Identifying quality-requirement conflicts", *IEEE Software*, vol. 13, n. 2, pp. 25--35, Mar. 1996.

[7] B. Brown, et al., "Experience with evaluation human-assisted recovery processes", in *Proc. IEEE/IFIP The International Conference on Dependable Systems and Networks* - DSN, Florence, Italy 28th June – 1st July 2004, pp. 405-410.

[8] B. Cheng, and R. Jeffery, "Comparing inspections strategies for software requirement specifications", in *Proc. Australian Software Engineering Conference*, Melbourne, Australia, 1996, pp. 203-211.

[9] A. M. Davis, "Just enough requirements management: where software development meets marketing", *Dorset House Publishing Company*, 2005.

[10] J. Durães, and H. Madeira, "Emulation of software faults: a field data study and a practical approach", *IEEE Transactions on Software Engineering*, vol. 32, n. 11, pp. 849-867, Nov. 2006.

[11] *ECSS space engineering: ground systems and operations – telemetry and telecommand packet utilization* (Standard style), ECSS-E-70-41A standard, 2003.

[12] *ECSS system: description and implementation* (Standard style), ECSS-S-00A Standard, 2005.

[13] *ECSS Project Management: Project Phasing and Planning*, ECSS-M-30A Standard, 1996.

[14] D. P. Gilliam, T. L. Wolfe, J. S. Sherif, and M. Bishop, "Software security checklist for the software life cycle", *in Proc. of the 12th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Jun. 2003, pp. 243-248.

[15] R. J. Halligan, "Requirements metrics: the basis of informed requirements engineering management", in *Proc. Complex Systems Engineering Synthesis and Assessment Technology Workshop*, Calvados, MD, USA, 1993.

[16] M. Hammil, and K. Goseva-Popstojanova, "Common trends in software fault and failure data", *IEEE Transactions on Software Engineering*, vol. 35, n. 4, pp. 484-496, 2009.

[17] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated

consistency checking of requirements specifications", *ACM Transactions on Software Engineering and Methodology*, vol. 5, n. 3, pp. 231--261, Jul. 1996.

[18] H. F. Hofmann, and F. Lehner, "Requirements engineering as a success factor in software projects", *IEEE Software*. Jul./Aug. 2001.

[19] K. Kanoun, et al. (Editors), "DBench dependability benchmarks", LAAS-CNRS, 2002. 235 p. (DBench Project, IST 2000-25425). Available at <http://www.laars.fr/DBench> Accessed on: April 28th 2008.

[20] M. Kim, S. Park, V. Sugumaran, and H. Yang, "Managing requirements conflicts in software product lines: a goal and scenario based approach", *Data and Knowledge Engineering*, vol. 61, n. 3, pp. 417-432, Jun. 2007.

[21] E. Knauss, C. Boustani, and T. Flohr, "Investigating the impact of software requirements specification quality on project success", *Product-Focused Software Process Improvement*, vol. 32, Part 2, pp. 28-42, 2009.

[22] P. Koopman, and H. Madeira, "Dependability benchmarking & prediction: a grand challenge technology problem", in *Proc. 1st IEEE Int. Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems*, Phoenix, Arizona, USA, Nov. 1999.

[23] R. R. Lutz "Analyzing software requirements errors in safety-critical embedded systems", in *Proc. of IEEE International Symposium on Requirements Engineering*, San Diego, USA, 4th-6th Jan 1993.

[24] R. R. Lutz "Targeting safety-related errors during software requirements analysis", *Journal of System and Software*, vol. 34, pp. 223-230, 1996.

[25] H. Madeira, et al., "Preliminary dependability benchmark framework", LAAS-CNRS. 2001. 44 p. (DBench Project, IST 2000-25425) Available at: <http://www.laas.fr/DBench/deliverables.html> Accessed on: April 28th, 2008.

[26] H. Madeira, R. Some, F. Moreira, D. Costa, and D. Rennels, "Experimental evaluation of a COTS system for space applications", in *Proc. The International Conference on Dependable Systems and Networks* - DSN, Bethesda, Maryland, USA, 23rd-26th June 2002, pp. 325-330.

[27] "Mission critical systems: defense attempting to address major software challenges", *US General Accounting Office*, 1992.

[28] R. P. Pontes, P. C. Véras, A. M. Ambrosio, E. Villani. "Contributions of model checking and CoFI methodology to the development of space embedded software." *Empirical Software Engineering*, vol. 19, n. 1, pp. 39-68, 2014.

[29] H. Ogawa, F. Kumeno, S. Honiden. "Model checking Process with Goal Oriented Requirement Analysis". Proceedings of 15th Asian-Pacific Software Engineering Conference, p. 377-384, 2008.

[30] A. A. Porter, L. G. Votta, and V. R. Basili, "Comparing detection methods for software requirements inspections: a replicated experiment", *IEEE Trans. On Software Engineering*, vol. 21, no. 6, pp. 563-575, Jun. 1995.

[31] K. Y. Rozier, "Linear Temporal Logic Symbolic Model Checking", *Computer Science Review*, vol. 5, n. 2, pp. 163-203, 2011.

[32] J. C. Ruiz, P. Yuste, P. Gil, and L. Lemus, "On benchmarking the dependability of automotive engine control applications", in *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks* - DSN, Florence, Italy, 28th June – 1st July 2004, pp. 857-866.

[33] P. Runeson, M. Höst, "Guidelines for conducting and reporting case study research in software engineering", *Empirical Software Engineering*, vol. 14, n. 2, pp. 131-164, 2009.

[34] F. T. Sheldon, H. Y. Kim, and Z. Zhou, "A case study: validation of guidance control software requirements for completeness, consistency and fault tolerance", in *Proc. Eighth Pacific Rim International Symposium on Dependable Computing*, Seoul, Korea, 17th-19th December 2001, IEEE Computer Society.

[35] L. Spainhower, K. Kanoun, (Editors), "Dependability Benchmarking for Computer Systems", *Wiley-IEEE Computer Society Press*, Hoboken, New Jersey, ISBN: 9780470230558, 2008.

[36] H. Suleiman, "Comprehensive Integrated Checklists for Requirements Engineering and Software Project Management". *Proc. of Information Science and Applications (ICISA)*, Suwon, 2013.

[37] P. C. Véras, et al.: Benchmarking software requirements documentation for space application. In: International Conference on Computer Safety, Reliability and Security (Safecomp), 14th-17th September 2010, Vienna, Austria, pp. 112--125 (2010)

[38] P. C. Véras, et al., "Errors on space software requirements: a field study and application scenarios", in *Proc. IEEE 21st International Symposium on Software Reliability Engineering* - ISSRE, 1st-4th November 2010, San Jose, CA, USA, pp. 61-70.

[39] P. C. Véras, Benchmarking software requirements documentation for space application, PhD thesis, Aeronautics Institute of Technology - ITA, 2011.

[40] P. C. Véras, et al., "Checklist of the software requirements documentation benchmark for space application", http://eden.dei.uc.pt/~mvieira/BenchReqCheckLists.zip.

[41] P. C. Véras, et al., "Finite State Machines of the On-Board Operations Scheduling Service", http://eden.dei.uc.pt/~mvieira/BenchReqModels.zip.

[42] G. S. Walia and J. C. Carver, "A systematic literature review to identify and classify software requirement errors", *Information and Software Technology*, vol. 51, n. 7, pp. 1087-1109, Jul. 2009

[43] W. M. Wilson, J. H. Rosenberg, and L. E. Hyatt, "Automated analysis of requirement specifications", in *Proc. 19th International Conference on Software Engineering*, Boston, Massachusetts, USA, May 1997, pp. 161-171.

[44] X. Zhu, and Z. Jin, "Detecting of requirements inconsistency: an ontology-based approach", in *Proc. of the Fifth International Conference on Computer and Information Technology*, Shanghai, China, 21th-23th Sept. 2005.