Benchmarking Software Requirements Documentation for Space Application

Paulo C. Véras[1], Emilia Villani[1], Ana Maria Ambrósio[2], Rodrigo P. Pontes[1], Marco Vieira[3], and Henrique Madeira[3]

[1] Department of Mechanical Engineering, Instituto Tecnológico de Aeronáutica, Praça Marechal Eduardo Gomes, 50, 12228-900, São José dos Campos-SP, Brazil
{pcv,evillani,rpastl}@ita.br
[2] Ground System Division, National Institute for Space Research, Av. dos Astronautas, 1758, 12227-010, São José dos Campos-SP, Brazil
ana@dss.inpe.br
[3] DEI/CISUC, University of Coimbra, 3030, Coimbra, Portugal
{mvieira,henrique}@dei.uc.pt

Abstract. Poorly written requirements are a common source of software defects. In application areas like space systems, the cost of malfunctioning software can be very high. This way, assessing the quality of software requirements before coding is of utmost importance. This work proposes a systematic procedure for assessing software requirements for space systems that adopt the European Cooperation for Space Standardization (ECSS) standards. The main goal is to provide a low-cost, easy-to-use benchmarking procedure that can be applied during the software requirements review to guarantee that the requirements specifications comply with the ECSS standards. The benchmark includes two checklists that are composed by a set of questions to be applied to the requirements specification. It was applied to the software requirements specification for one of the services described in the ECSS Packet Utilization Standard (PUS). Results show that the proposed benchmark allows finding more with a low effort.

Keywords: benchmark; software requirements quality; space systems; ECSS standards; Packet Utilization Standard.

1 Introduction

Writing a high quality software requirements specification (SRS) is one of the hardest phases of the development life cycle of a software system [1]. Ill-defined requirements contribute to significant schedule delays and cost increases [2]. Furthermore, there are evidences that errors in the requirements can lead to serious problems during software development and usage [3].

E. Schoitsch (Ed.): SAFECOMP 2010, LNCS 6351, pp. 112–125, 2010.

Some problems in the specification of requirements, such as the occurrence of conflicts, can be resolved by using formal specification languages or formal methods. However, they are hard to be understood by non-experts, which limits their practical application to some restricted domains [4]. Despite the many problems of using natural languages, such as the lack of formality and ambiguity, this is still the most used mean to express software requirements and to communicate those requirements to customers [5]. Software for space systems is no exception and requirements specification based on natural languages are still widely used.

The European Cooperation for Space Standardization (ECSS) [6] provides a set of standards to support the development of space products. These standards cover a broad range of application areas, such as mechanical, software engineering, control engineering and ground system. Among the ECSS standards, ECSS-70-41A [7] proposes the Packet Utilization Standard, also known by its acronym PUS, which addresses the utilization of telecommand and telemetry for the remote monitoring and control of spacecrafts. It defines a set of services that covers all the fundamental requirements for spacecraft operation. PUS defines the protocol for the communication between the spacecraft and the ground segment (i.e. the control centre in Earth).

Leveson [8] studied in detail five software-related accidents in space systems. Among the factors that contributed to the accidents, there are some related to software engineering, such as poor or missing specifications, and inadequate review activities. Common sense is that these problems can be mitigated by applying a standardized process of assessing or reviewing the quality of the requirements. This work proposes a starting point to define a benchmark to be applied to the software requirements specification of space systems that adopt the PUS.

Typically, a benchmark is a systematic procedure to assess measures related to the behaviour of a computer system or computer component, and aims at comparing alternative solutions or evaluating its characteristics against a reference model (e.g., a standard). While a performance benchmark is composed by a workload and measurements, a dependability benchmark adds an additional component: a faultload that represents real faults experienced by systems in the field. The system under benchmarking (SUB) and the benchmark target must be well defined, as well as the benchmarking rules and procedures. The benchmark shall be validated taking into account its representativeness, repeatability, reproducibility, portability, non-intrusiveness, scalability, time (that has to be as short as possible) and cost (the perceived value shall be higher than the associated costs).

As the target of the benchmarking approach proposed in this work is a document, there is no workload to run or faultload to inject. Instead, a checklist composed by questions replaces the workload and is used to obtain measures that portray specific characteristics of the software requirements specification. The purpose of this benchmark is to guide/help the review of the requirements for the onboard computer software. This review is typically performed at the beginning of the space software development process. The proposed benchmark provides a standardized way for assessing the quality of the requirements and their accomplishment regarding the PUS ECSS standard.

The proposal of a benchmark for software requirements is motivated by the high costs of conducting a software requirement review over low quality software requirements specifications and by the very high costs of the rework caused by poorly written

requirements. In fact, problems detected at the end of the software development cycle may compromise the entire space mission timeline. Moreover, applications with high complexity, like space systems, demand the use of standards in order to guide the whole development life cycle. Naturally, if a project is in accordance with key standards, the overall software quality tends to increase.

Particularly, the paper discusses the definition of the benchmark checklist. As this checklist greatly influences the benchmark representativeness, we use two sources of information for defining two different checklists. The first is based on the description of the services in the PUS and basically aims at guaranteeing that the SRS complies with the standard (regarding its content). The second is based on the CoFI methodology [9], which aims at verifying whether the SRS covers system failure situations.

To show the feasibility and applicability of the proposed benchmarking approach, we use the telecommand verification service described in PUS as a case study. The two checklists are applied to a concrete software requirements specification of a space product. The results are analysed and discussed in detail. The idea of the whole work is to propose a benchmark to assess the overall SRS by defining checklists that cover all of the services described in PUS. Although the proposed benchmark is restricted to software that follows the PUS, the methodology we used to create it can be easily extended to embedded software that follows other standards.

The paper is organized as follows. Section 2 reviews related work. Section 3 presents the definition of the two checklists. Section 4 presents the concrete example for the telecommand verification service and discusses the results obtained from the application of the two proposed checklists. Section 5 concludes the paper.

2 Related Work

Many works aim at studying attributes of software requirements quality, such as: correctness, completeness, consistency, clarity, and feasibility. Halligan [10] 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). Davis [11] 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 [10]. This approach provides a score that reflects the quality of the overall requirements document. However, Davis does not define the point from which the document is considered good enough to proceed to next phase of the development process. Knauss [12] performed a study based on the metrics defined by Davis in order to find out a threshold that determines whether the requirements document can be considered good enough to serve as a foundation for project success.

Hofmann and Lehner [13] conducted a field study to identify requirements engineering practices that contribute to project success. Boehm [14] and Wilson [15] 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 [4] proposed an approach for systematically identifying and managing requirements conflicts based on requirement partition in natural language.

Gilliam [16] 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. Sheldon [17] discusses the validation of a SRS based on natural language

in terms of completeness, consistency and fault-tolerance. A method for detecting semantic level inconsistency in software requirements based on state transition is described in [18].

The works presented above (and many others) deal with SRS quality assessment in general and try to perform this by using some new method, some combination of existing methods or by proposing some new process of assessment. Although there is a considerable number of works in the literature, to the best of our knowledge none target specifically critical embedded software, neither systems that follow some specific standard. Furthermore, none of the existing works are based on a benchmarking process, with well-defined metrics and a very well contextualized scenario.

The Transaction Processing Performance Council (TPC) [19], an organization composed by major vendors of database and transaction processing software, has a long-standing tradition in proposing and managing performance benchmarks. Although, dependability measures have been largely absent of the TPC benchmarking effort, the TPC benchmarks have influenced many dependability benchmarks proposed so far (see, for example, the book edited by K. Kanoun and L. Spainhower [20]). Dependability benchmarking has been mainly focused in the evaluation and comparison of the dependability of COTS (Component Off-The-Shelf) and COTS-based systems in embedded, real-time and transactions systems. Both academy and industry have proposed benchmarks focusing on a wide range of types of systems [20]. Examples of benchmarks for embedded systems are [21] and [22]. The former focused on real-time kernels for on-board space systems and the latter addressed automotive systems.

Existing benchmarks aim at assessing and/or comparing computer systems or components. The purpose is to assess products, i.e., the systems under benchmarking are final products and they do not take into account the quality of the requirements documentation or the impact of low quality requirements in the final product. Our work opens a new research direction by extending the benchmarking concept to software documentation, namely requirements specifications for space systems.

3 Benchmarking Approach for Software Requirements

Although based on the concepts of dependability benchmarking, which aim at assessing and comparing key features of the behaviour of a computer system or component, the purpose of our approach is to assess software requirements specification, thereby requiring the redefinition of the main elements of a typical dependability benchmark [20], [23]. Besides allowing comparing some quality attributes of software requirements specifications, it allows comparing a given SRS against a reference model (in our case, the PUS standard).

In our approach, the workload and faultload are replaced by a checklist that consists of a set of questions to assess key features of the software requirements specification under benchmark. The purpose of these questions is to verify whether the SRS is in accordance with the PUS [7] (PUS-based checklist), as well as to verify whether the SRS describes the actions that the system shall perform in the case of a failure (CoFI-based checklist). All of these questions accept just “yes” or “no” as answer.

The measure of the proposed benchmark is the number of questions answered "yes" when applied to the software requirements document under benchmarking. The amount of time necessary to the specialist to execute the benchmark is also assessed in order to provide an idea of the ease of application of the benchmark.

Although the checklists proposed for this benchmark takes into account only the telecommand verification service of the PUS, they can be easily extended to the other 15 basic services of the PUS, which constitute the capabilities to be implemented on-board a satellite along with the corresponding monitoring and control facilities on the ground. The telecommand verification service provides the capability for explicit verification of each distinct stage of the execution of a telecommand packet, from the on-board acceptance through the completion of the telecommand execution. This service consists of the following stages: (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, (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 of the same stages if this has been requested in the acknowledgment flags in the telecommand packet header. These reports shall provide auxiliary data for the ground system to fully understand the report (e.g. to identify the nature and cause of a telecommand failure).

As mentioned before, the proposed benchmarking checklists were defined by using two different methods, which allows comparing the results obtained from the application of each one. The first is directly based on the telecommand verification service of the PUS and on the description of the telecommand packets structure. The second was generated by applying the CoFI methodology to the PUS.

3.1 Checklist Based on the PUS

The questions that compose this checklist were defined by analysing the specification of the telecommand verification service. For each mandatory statement of the standard, one or more questions were defined. When necessary, other parts of the standard were consulted to gather more information to define the questions (e.g., when the telecommand verification service makes reference to the structure of the telecommand and telemetry packets). For example, there is a question to verify if the SRS defines the type and size of each field of the telecommand and telemetry packets.

The resulting checklist was reviewed by a developer with a large experience (more than 6 years) that works on space application software in the context of PUS and by a researcher on space systems (whose knowledge is described in section 4.1 of this paper, specialist 3). The suggestions and recommendations received from the specialists were then incorporated into the final checklist.

This checklist verifies whether the SRS follows the PUS by taking into account the content of the PUS itself and of the SRS. The checklist is composed of a set of 92 questions that accept as answer only "yes" or "no", where "yes" means that a given requirement specification complies with the PUS and "no" means that the requirement specification does not comply with the PUS or has some ambiguity in the context of the question. Three examples of questions are:

- Does the requirement specification define the telecommand verification service type as 1?
- Does the specification state that this service shall check whether the Packet Length field value is at least 16 octets and at most 65535 octets?
- Does the requirement specification state that the code 0 of the failure acceptance report means "illegal APID (PAC error)"?

As an example, let's take the first question above. Each one of the services defined in PUS has a number that represents its service type and that distinguishes each service in a unique manner. The telecommand verification service shall have its type defined as 1. If the answer given to this question is "yes", this means that the SRS follows the standard. The full PUS-based checklist is available at [25]. An important aspect is that the PUS does not state all the features to be mandatory. This standard has some points that are optional and 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. Therefore, the resulting percentage of "yes" answers is referred to the applicable questions considered by each specialist.

3.2 Checklist Based on the CoFI Methodology

The second checklist was defined by using the CoFI methodology. CoFI stands for Conformance and Fault Injection as it drives the conformance and robustness test cases generation. This methodology guides a tester to create simple finite state machines (FSMs) starting from a textual description, such as the software requirements specifications. Instead of relying on a single behaviour model of the system, it guides the creation of a set of small FSMs representing partial behaviours to cover test objectives. The first step is to identify the set of services that the system provides and then create different FSMs for each service, taking into account the following classes of inputs: (i) normal, (ii) specified exception, (iii) inopportune input (i.e., corrects but in wrong moments), and (iv) invalid inputs caused by external faults. Thus, decomposition of the system complexity in small FSMs is driven in terms of: (i) the services and (ii) types of behaviour, namely, normal, specified-exception, sneak-path, and, fault-tolerance. Once the FSMs are defined, they are submitted to the ConData tool [24], which can automatically generate test cases, as those used for protocol testing. In our work, the description of the service in PUS is used to create the FSM for the normal, specified exception and inopportune input behaviours. Then, instead of using the FSM of CoFI to generate test cases, we used it to generate questions to compose the benchmark checklist.

As mentioned before, this checklist does not take into account the content of the SRS and of the PUS. Instead, it considers the PUS as a kind of "black box", it just considers the functional behaviour that the software shall have by verifying the output response to the provided inputs. It does not verify the steps that the software performs to accomplish what the standard defines.

The questions derived from the CoFI methodology look for evidences in the requirements specification that show that the developer considered not only the normal behaviour but also all the important cases of invalid inputs or sneak paths. The rationale behind it is to discover potential 'holes' in the requirement specification that would lead to the identification of failures when testing of the final software product.

The FSM models the behaviour of the telecommand verification service when communicating with both the ground station and the on-board application process. The ground station sends a telecommand to be executed on-board and wait 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 station via telemetry.

Each possible 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.

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 RepSuccAcc,

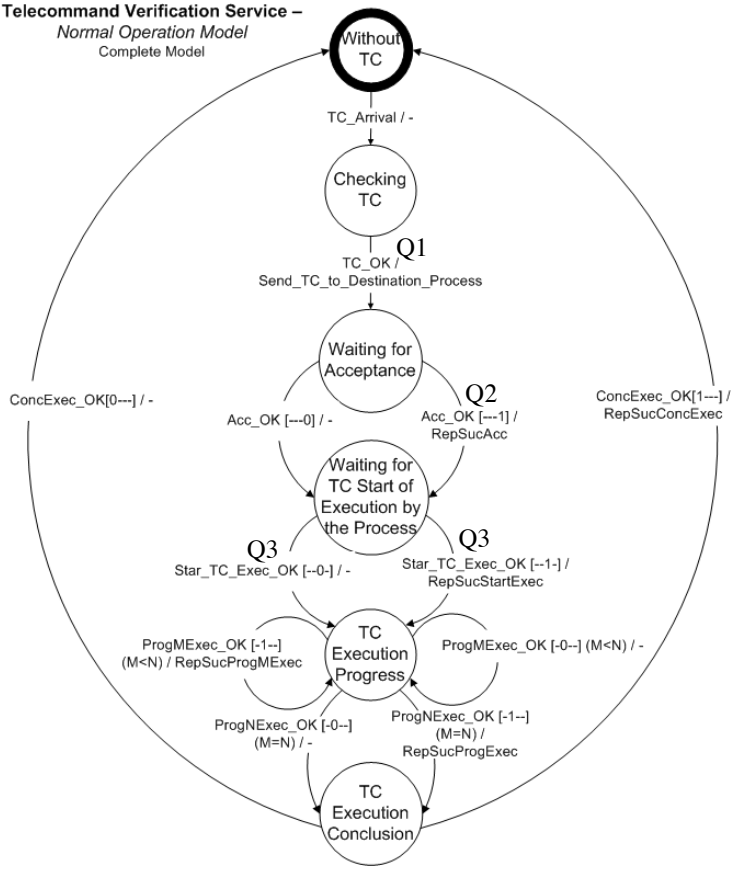

Fig. 1. Normal behaviour FSM

RepSucProgExec are different reports carried into the telemetry and sent to the ground station. The events Acc-OK, Start_TC_Exec_OK 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 for the FSM of Fig. 1 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)
- 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)
- 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)

Fig. 2 illustrates the sneak paths behaviour. Basically, the sneak paths model considers the case of receiving a valid response from the application process at the wrong 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)
- 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)
- 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)

By using the CoFI methodology, a list with 36 questions was generated. Additional questions were added to verify sensible points indirectly suggested by the system modelling. Some 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?

The full CoFI-based checklist is available at [25]. By analysing both PUS and CoFI-based checklists, we conclude that the major contribution of using the CoFI methodology is the definition of the questions based on the sneak path FSM. The PUS-based checklist contemplates almost all of 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. Therefore, the portion of

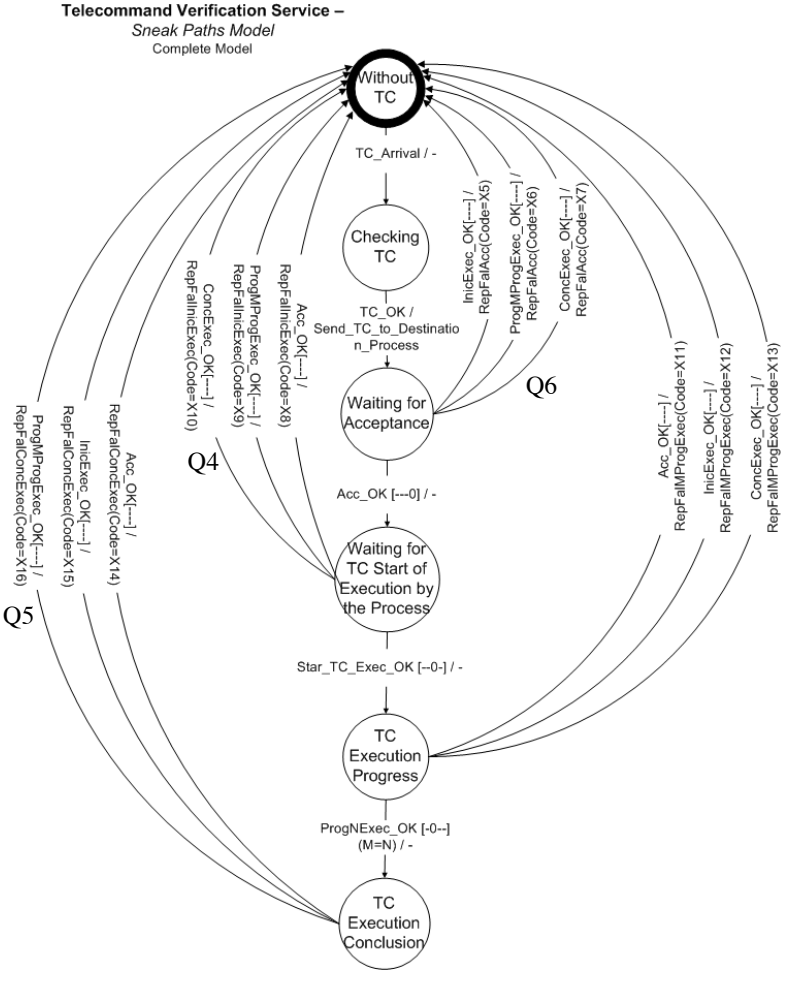

Fig. 2. Sneak path FSM

the CoFI-based checklist defined from the sneak path FSM can be considered complementary to the PUS-based checklist.

4 Example of Application

This section presents a concrete example of the application of the benchmark and discusses the results obtained.

4.1 Case Study

The example used in this work consists of a software requirements specification that describes the functional requirements of the telecommand verification service. This

example is taken from a real space project under development at INPE (Instituto Nacional de Pesquisas Espaciais, Brazil). This specification does not take into account the whole software to be implemented, but just the piece of software that implements the telecommand verification service. Some examples of requirements are:

- R4.3 - When a failure occurs during the acceptance of a telecommand, the OBDH (On-Board Data Handling) software shall generate a report informing this occurrence.
- R4.3.1 – The report mentioned in R4.3 shall have subtype 2.
- R4.3.2 – The format of the packet data field of the report mentioned in R4.3 shall conform to the format specified in clause 6.3.2 of the ECSS-E-70-41A standard.

The specification has a total of 39 functional requirements and is limited to the telecommand verification service. Some requirements make reference to specific sections of the PUS standard. When applying the checklists, these sections of the PUS standard were considered as part of the requirements specification.

Four specialists applied independently the benchmark to the requirements specification. 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 possible interpretations from the different specialists. The specialists were also allowed to register additional comments to each question of the checklist.

All the specialists have knowledge in the PUS ECSS standard. Specialist 1 has been a professor and researcher in mechatronics and aerospace systems for 5 years and has experience in modelling, validation and verification of systems. Specialist 2 is a MSc student in aerospace systems and has worked with space projects that adopt ECSS standards. Specialist 3 has been researcher in space systems for 25 years and has worked with ground segment system, having also a broad experience in systems independent validation and verification. Specialist 4 is a PhD student and has worked with aerospace systems for 5 years, both in the academy and industry. All specialists had a participation in the discussion about this work since its preliminary steps.

Each specialist read the software requirements document one time before start answering the questions of the checklists. Particularly, specialist 2 is the one who actually wrote the requirement specification. The total time needed to apply the checklist (including the first read of the document) was measured. In practice, the attributes measured in the application of each of the checklists are the number of answers “yes” and the total time necessary to apply it.

4.2 Results and Discussion

The results obtained with the application of the PUS-based checklist by the four specialists are shown in Table 1, where the column A.Q. contains the number of Applicable Questions considered by the specialist.

By analysing the answers of the four specialists, we observed that 68 questions had the same answers (either “yes” or “no”) for all specialists, and 7 other questions had the same answers for only three of the specialists. These results indicate that the remaining 17 questions may be ambiguous and may need to be revised.

Table 1. Result of the application of the PUS-based checklist

	"Yes"	A.Q.	"Yes" (%)	Time (hh:mm)
Specialist 1	76	92	82.6	00:44
Specialist 2	74	87	85.1	00:37
Specialist 3	68	89	76.4	01:29
Specialist 4	68	87	78.2	00:42
Average	**71.5**	**88.7**	**80.6**	**00:53**

The analysis of the comments provided by the specialists allowed understanding the reason for the discrepancy in the positive answers. Specialists 3 and 4 were more rigorous in the interpretation of the SRS, while specialists 1 and 2 were more flexible. One example is the case of Specialist 4 considering two requirements in conflict. As a consequence, he answered "no" to four questions of the checklist. The other 3 specialists answered "yes" to the same questions and detected no conflict. Even in the presence of some discrepancy among the answers of the specialists, we can see that the maximum difference between the average percentage of "yes" (80.6%) and the percentage of "yes" for any specialist is less than 5%, which is quite acceptable once that the checklist and SRS have a margin to interpretation.

The mean time necessary to apply the PUS-based checklist to the SRS is 53 minutes, what gives us an idea of the ease of application of the 92 questions. In average, only three questions were considered not applicable to the SRS under assessment. In general, the questions that were considered not applicable are questions about aspects that are optional in the PUS. Thus, if the SRS does not intend to implement that point, the question about it is not applicable.

The results obtained with the application of the checklist generated by using the CoFI methodology are listed in the Table 2. In the case of this checklist, there is no evaluation of the applicability of the questions because all of them are applicable. The total number of questions whose answers were the same for the four specialist is 17 (again either "yes" or "no"), and the total number of (remaining) questions whose answers were the same for only three of the specialists is 14.

Table 2. Result of the application of the CoFI-based checklist

	"Yes"	"Yes" (%)	Time (hh:mm)
Specialist 1	13	36.1	00:16
Specialist 2	14	38.9	00:27
Specialist 3	7	19.4	00:30
Specialist 4	12	33.3	00:22
Average	**11.5**	**31.9**	**00:23**

Except for the questions generated by the sneak paths FSM, whose answers were all the same ones, the application of the questions to the requirements list were interpreted in different ways by the 4 specialists. Furthermore, the specialist 3 was much more rigorous than the other specialists. The main reason for the different interpretation is that the requirements are strongly based on the PUS standard and the PUS

standard leaves implicit the interaction with the application process. As a consequence, the requirements also leave this part poorly detailed. Some specialists considered the implicit interaction and answered "yes", while other considered that there was no clear answer to the same question and answered "no".

A good example is the question: "*Does the requirement specification state that the successful report of telecommand execution starting shall be generated after the confirmation of the execution starting sent by the target application process?*". The requirements contain no information about a confirmation that should be received from the target application process. On the other hand, it says explicitly that the successful report should be sent after the execution start. This has resulted in different answers from the specialists.

Despite the SRS used in this work is from a real project, the number of answers "yes" given in the application of the CoFI-based checklist is very low. This can be explained by the fact that this checklist looks for system failure situations that the PUS does not cover. In this way, the SRS does not describe this kind of situation. The idea of applying these checklists during the software requirements definition phase is exactly to find this kind of weakness in the SRS.

As can be seen in tables 2 and 3, the mean time required to apply the PUS-based checklist is more than two times the time required to apply the CoFI-based checklist. This is mainly due to the number of questions of each checklist (92 question of the former, against 36 questions of the later).

It is important to emphasize that the questions in the PUS-based checklist aim to verify whether the software requirements specifications complies with the PUS. On the other hand, the CoFI-based checklist goes beyond the standard. Through the development of the FSMs, this methodology provides a mean for verifying key aspects that are not explicitly approached by the standard, such as the sequence of messages changing between the telecommand verification service and the target application process. In addition, the sneak path FSMs provides a way for thinking about operational conditions not handled by the PUS and allows verifying the robustness and dependability of the system.

This is the reason why the average percentage of "yes" answers in the CoFI checklist (31.9%) was much smaller than in the PUS-based checklist (80.6%). The software requirements specification used as case study does not describe how the system shall behave in the presence of faults. As the PUS-based checklist did not contemplate this, the number of "yes" answers was higher. On the other hand, the CoFI-based checklist is not as detailed as the PUS-based checklist.

Results show that the proposed checklists are quite complementary and can be merged to form a more comprehensive and representative checklist. Also, the definition of a representative checklist for a benchmark for requirements specifications must take into account different sources of information. It is important to emphasize that the results obtained by applying the proposed benchmark can be used as a feedback to the development team. In fact, those results provide a measure of the completeness, robustness and accomplishment with the followed standards (ECSS standards). The negative answers given to the checklist can be used to improve these aspects in the requirements specification.

5 Conclusion and Future Work

This work proposed a benchmarking approach for software requirements specifications for space applications. This benchmark is based in two checklists that help

assessing specific characteristics of a requirements specification. As a starting point, the proposed work is restricted to software that implements the telecommand verification service of the PUS, an ECSS standard.

The definition of the first checklist was based on the analysis of PUS specification and the second was based on the CoFI methodology. The former evaluates whether the software requirement specification complies with the PUS, the latter goes beyond this and verifies whether the document handles situations such as the presence of faults. Although the PUS-based checklist does not approach this kind of situation, it has detailed questions about telecommand and telemetry data that verify the compliance of the software requirements with the standard.

The proposed approach has been applied to a software requirements specification of a real space project currently under development at INPE. Four specialists have applied the proposed benchmark to provide a more consistent evaluation of the study. Results showed that the piece of the CoFI-based checklist that aims at verifying whether the SRS handles situations of presence of faults is complementary to the PUS-based checklist. The PUS-based checklist already contemplates the remaining parts of the CoFI-based checklist.

To each question of the checklist criticality could be associated, according to the severity of the aspect treated by that question. As a future work, we are planning to define how to measure the result of the benchmark considering that each question has a weight. Additionally, a minimum threshold value to decide whether the requirement specification is good enough to pass to next project phase needs to be determined.

Acknowledgment

This work was partially supported by CAPES – Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, AEB – Agência Espacial Brasileira, and by CISUC – Centro de Informática e Sistemas da Universidade de Coimbra.

References

1. Brooks, F.P.: No silver bullet: essence and accidents of software engineering. IEEE Computer 20(4), 10–19 (1987)
2. Mission critical systems: defense attempting to address major software challenges. US General Accounting Office (1992)
3. Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: Automated consistency checking of requirements specifications. ACM Transactions on Software Engineering and Methodology 5(3), 231–261 (1996)
4. Kim, M., Park, S., Sugumaran, V., Yang, H.: Managing requirements conflicts in software product lines: a goal and scenario based approach. Data and Knowledge Engineering 61(3), 417–432 (2007)
5. Davis, A.M.: Predictions and Farewells. IEEE Software 15(4), 6–9 (1998)
6. ECSS system: description and implementation, ECSS-S-00A Standard (2005)
7. ECSS space engineering: ground systems and operations – telemetry and telecommand packet utilization, ECSS-E-70-41A standard (2003)

8. Leveson, N.G.: The role of software in spacecraft accidents. AIAA Journal of Spacecraft and Rockets 41(4), 564–575 (2004)
9. Ambrosio, A.M., Martins, E., Vijaykumar, N.L., Carvalho, S.V.: Systematic generation of test and fault cases for space application validation. In: 9th ESA Data System in Aerospace (DASIA), Edinburgh, Scotland, May 30 - June 2, ESA Publications, Noordwijk (2005)
10. Halligan, R.J.: Requirements metrics: the basis of informed requirements engineering management. In: Complex Systems Engineering Synthesis and Assessment Technology Workshop (CSESAW 1993), Calvados, MD, USA (1993)
11. Davis, A.M.: Just enough requirements management: where software development meets marketing. Dorset House Publishing Company (2005)
12. Knauss, E., Boustani, C., Flohr, T.: Investigating the impact of software requirements specification quality on project success. Product-Focused Software Process Improvement 32 Part 2, 28–42 (2009)
13. Hofmann, H.F., Lehner, F.: Requirements engineering as a success factor in software projects. IEEE Software (July/August 2001)
14. Boehm, B., In, H.: Identifying quality-requirement conflicts. IEEE Software 13(2), 25–35 (1996)
15. Wilson, W.M., Rosenberg, J.H., Hyatt, L.E.: Automated analysis of requirement specifications. In: 19th international conference on Software engineering, Boston, Massachusetts, United States, May 17-23, pp. 161–171 (1997)
16. Gilliam, D.P., Wolfe, T.L., Sherif, J.S., Bishop, M.: Software security checklist for the software life cycle. In: Proceedings of the 12th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, pp. 243–248 (June 2003)
17. Sheldon, F.T., Kim, H.Y., Zhou, Z.: A case study: validation of guidance control software requirements for completeness, consistency and fault tolerance. In: Eighth Pacific Rim International Symposium on Dependable Computing (PRDC 2001), Seoul, Korea, December 17-19, IEEE Computer Society, Los Alamitos (2001)
18. Zhu, X., Jin, Z.: Detecting of requirements inconsistency: an ontology-based approach. In: Proceedings of the Fifth International Conference on Computer and Information Technology (CIT 2005), Shanghai, China, September 21-23 (2005)
19. Transaction Processing Performance Council, `http://www.tpc.org`
20. Spainhower, L., Kanoun, K. (eds.): Dependability Benchmarking for Computer Systems. Wiley-IEEE Computer Society Press, Hoboken (2008) ISBN: 9780470230558
21. Madeira, H., Some, R., Moreira, F., Costa, D., Rennels, D.: Experimental evaluation of a COTS system for space applications. In: The International Conference on Dependable Systems and Networks, Bethesda, Maryland, USA (2002)
22. Ruiz, J.C., Yuste, P., Gil, P., Lemus, L.: On benchmarking the dependability of automotive engine control applications. In: IEEE/IFIP International Conference on Dependable Systems and Networks, Florence, Italy (2004)
23. Koopman, P., Madeira, H.: Dependability benchmarking & prediction: a grand challenge technology problem. In: 1st IEEE Int. Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems, Phoenix, Arizona, USA (November 30, 1999)
24. Martins, E., Sabião, S.B., Ambrosio, A.M.: ConData: a tool for automating specification-based test case generation for communication systems. Software Quality Journal 8(4), 303–319 (1999)
25. Véras, P.C., et al.: Checklist of the software requirements documentation benchmark for space application, `http://eden.dei.uc.pt/~mvieira`