2010 IEEE 21st International Symposium on Software Reliability Engineering

Errors on Space Software Requirements: A Field Study and Application Scenarios

Paulo C. Véras, Emilia Villani
Department of Mechanical Engineering
Instituto Tecnológico de Aeronáutica – ITA
São José dos Campos – SP, Brazil
{pcv,evillani}@ita.br

Ana Maria Ambrosio
Ground System Division
National Institute for Space Research – INPE
São José dos Campos – SP, Brazil
ana@dss.inpe.br

Nuno Silva
Dependability & Safety
Critical Software S.A.
Coimbra, Portugal
nsilva@criticalsoftware.com

Marco Vieira, Henrique Madeira
Department of Informatics Engineering
University of Coimbra – UC
Coimbra, Portugal
{mvieira,henrique}@dei.uc.pt

***Abstract*—This paper presents a field study on real errors found in space software requirements documents. The goal is to understand and characterize the most frequent types of requirement problems in this critical application domain. To classify the software requirement errors analyzed we initially used a well-known existing taxonomy that was later extended in order to allow a more thorough analysis. The results of the study show a high rate of requirement errors (9.5 errors per each 100 requirements), which is surprising if we consider that the focus of the work is critical embedded software. Besides the characterization of the most frequent types of errors, the paper also proposes a set of operators that define how to inject realistic errors in requirement documents. This may be used in several scenarios, including: evaluating and training reviewers, estimating the number of requirement errors in real specifications, defining checklists for quick requirement verification, and defining benchmarks for requirements specifications.**

.Keywords-requirements; space systems; space software requirements quality; field study

I. INTRODUCTION

The lifecycle of critical embedded software is characterized by formal reviews, including requirement reviews, architecture reviews design review and others. One example is the development cycle of spacecraft on-board software, which is detailed in the standard ECSS-E-40 Part1B [1]. Another example is aeronautics, where the development of critical software systems and components follows a very similar approach. In fact, ranging from automotive systems to medical devices, it is easy to find examples of application domains where formal reviews play a major role.

Formal reviews consist of submitting a set of artifacts (including code, which is a document written in a programming language) to a group of experts that analyze them following a very structured approach in order to find the highest number of problems possible. The results of the review provide the required feedback for the software development team to decide whether the project should move to the next phase (or not). Although reviews are essential during the development of critical software, they are largely influenced by human skills and judgment limitations, being at the same time very expensive and time consuming.

The process of gathering and writing a good quality requirement specification is the hardest part of the development lifecycle of a software system [2]. In fact, ill-defined requirements typically lead to schedule delays and increased cost [3]. Furthermore, errors[1] in the requirements can lead to failures that may cause major accidents [4]. In the space area, for example, Leveson [5] analyzed the direct and indirect causes behind five famous accidents that resulted in the lost of the spacecraft. A key observation was that all the accidents analyzed were some how due to poor specification of software requirements and inadequate review activities.

In this paper we present an extensive field study on real errors found in space software requirements documents. In this study we analyzed a total of 209 errors found in the requirement documents of three real projects in the space area. The errors were classified using an extension of the Walia and Carver [9] requirement error classification, a well-known taxonomy that was enlarged to allow a more thorough analysis of the error requirements really observed in the field study. Results provide a characterization of the most frequent

[1] To keep consistency with [9], in this paper we use the term "error" to refer to faults in the requirements.

1071-9458/10 $26.00 © 2010 IEEE
DOI 10.1109/ISSRE.2010.30

types of errors observed in real software requirement specifications and can be directly used by development teams to improve development processes in such way that reduces the occurrence of requirement errors.

The documents used in this study were written by different companies (with a large experience in the space domain) and the structured verification of the requirements was performed independently by an external organization, as part of the ISVV (Independent Software Verification and Validation) process [10]. In practice, the errors were found by the ISVV teams, which means that they have escaped to the quality verification mechanisms implemented during the development process. The main input for our study were the requirement error reports produced by the ISVV teams, which contain enough detailed information to characterize and classify each error.

In addition to the implicit and obvious utilization of the field study results to improve the software development process, the paper also proposes a set of fault operators that allow the emulation of realistic requirement faults. This is the equivalent of fault injection at the requirement level, and can be useful in several scenarios, including: evaluating and training requirement reviewers, estimating the number of errors in real requirement specifications, and defining checklists for quick requirement verification.

Obviously, these application scenarios are concrete and immediate applications of the results of the field study and of the fault operators presented in this paper. However, we also envisage a more speculative application scenario (which actually motivates our research in the medium/long term): **dependability benchmarking for requirement specifications**. In fact, the fault operators proposed may constitute the basis for the definition of faultloads for future benchmarks for requirement specification documents. The idea is to define a benchmarking procedure that helps making decisions (e.g., deciding whether a specific requirement document is ready for being reviewed, which avoids reviewing low quality documents). Obviously, the key idea here is not to compare different alternatives (as it is the case in typical benchmarking scenarios), but to benchmark the attributes of a document against a quality model.

Although applying dependability benchmarking to software documentation, namely to requirement specification documents, requires the redefinition of the main elements of a typical dependability benchmark (i.e., metrics, workload, faultload, and procedures and rules) [7][8], a first and decisive question has to be answered to assess the feasibility of the concept. Obviously, the question is whether it is possible (or not) to define a faultload for software requirements. Answering to this question is actually one of the goals of the present paper.

The paper is organized as follows. Next section presents background and related work. Section III reviews the Walia and Carver taxonomy. Section IV details the field study, presents an extension of the Walia and Carver taxonomy, and discusses the results. Section V proposes a set of fault operators for the injection of realistic requirement errors and section VI discusses several application scenarios. Finally, section VII concludes the paper.

II. BACKGROUND AND RELATED WORK

Many works (e.g., [11], [12], [13] and [14]) aim at studying the quality attributes of software requirements, such as correctness, completeness, consistency, clarity, and feasibility.

Halligan [11] 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) and on quality attributes such as the one mentioned above. Likewise, Davis [12] proposes metrics to measure the quality of software requirement. The approach is based on the assessment of each requirement according to quality attributes similar to the ones proposed in [11]. Knauss [15] performed a study based on the metrics defined by Davis in order to find out a threshold that determines whether a requirements document can be considered good enough to serve as a foundation for a successful project.

Hofmann and Lehner [16] identified requirements engineering practices that contribute to project success based on a field study. Using a controlled experiment, a metric-based reading technique for inspecting requirements was proposed in [17]. A study on faults and failures based on empirical data is presented in [18]. This study analyzed 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 fault in each phase, thus does not allow understanding the issues related to the requirement specification phase (which is precisely our goal).

Although there is a considerable number of works in the literature about requirement specification inspections and field studies, none of them was performed specifically for critical embedded software. In fact, the characterization of the type and distribution of faults in critical embedded software requirements is still an open question. This is indeed a key aspect to improve the quality and reduce the huge time and cost typically associated to the process of reviewing requirements for critical embedded software.

As already mentioned, a research goal motivating this work is the use of dependability benchmarking to reduce the time and cost of verifying critical embedded software requirements. Dependability benchmarking has been the focus of attention of researchers and practitioners in the recent years [7]. The goal of dependability benchmarking is to provide a standard procedure to characterize a computer system or component, by assessing dependability related metrics such as availability, integrity and reliability. The standardized and practical nature of benchmarking is in fact a key motivation to apply it to a new field: benchmarking the quality of critical embedded software requirement specification documents.

III. ERRORS CLASSIFICATION TAXONOMY

To classify errors in requirements we used the taxonomy proposed by Walia and Carver [9], which have carried out a

TABLE I. REQUIREMENT ERROR CLASSIFICATION FROM WALIA AND CARVER [9].

People Errors	
Class	***Description***
1. Communication	Inadequate or poor communication between teams, sub-teams or among the member of a team; changes in requirements not communicated, and other form of human communication errors.
2. Participation	No involvement of all stakeholders, lack of involvement of users at all times during requirement development.
3. Domain Knowledge	Lack of domain or system knowledge, lack of adequate training or experience on requirement engineering.
4. Specific Application Knowledge	Misunderstandings of particular aspects of the problem domain such as hardware and software interface specification, software interfaces with the rest of the system, timing issues, data dependency and event constraints, and mistakes in the expression of the end state or output expected.
5. Process Execution	Mistakes in the execution of actions of the requirement engineering process, regardless their adequacy.
6. Other Cognition Errors	Mistakes caused by adverse mental states, loss of situation awareness, ergonomics or environmental conditions.
Process Errors	
Class	***Description***
7. Inadequate method of achieving objectives	Incomplete knowledge leading to poor planning on achieving goals, mistakes in setting goals, error in choosing the wrong method or action to achieve the goals.
8. Management	Poor management of people and resources, lack of management leadership, problems in the assignment of resources to different tasks.
9. Elicitation	Inadequate requirement gathering, lack of awareness of all the sources of requirements and of proper methods for collecting requirements.
10. Analysis	Incorrect model while trying to construct and analyze solutions, mistakes in developing models for analyzing requirements, misunderstanding of the feasibility and risks associated with requirements, inability to consider all cases to document exact behavior of the system.
11. Traceability	Inadequate/poor requirement traceability.
Documentation Errors	
Class	***Description***
12. Organization	Poor organization of requirements.
13. No Usage of Standard	No use of standard format for documenting requirements or use of different technical standards or notations by sub teams.
14. Specification	Missing checks (item exists but forgotten), carelessness while documenting requirements, and other human mistakes or omissions while documenting requirements.

comprehensive literature review of 149 papers addressing errors in software requirements. Walia and Carver taxonomy is divided in three high-level error types: people errors, process errors, and documentation errors. People errors are related to mistakes performed by the individuals involved in the project. Process errors are related to the inadequacy of the requirement engineering process and to mistakes in selecting the means for achieving goals. Documentation errors are caused by mistakes in organizing and specifying the requirements independently of whether the developer understood the requirements or not. As shown in Table I, each of these high level groups is further divided in several sub-groups [9].

As our field study is based on the analysis of software requirements review reports some Walia and Carver [9] error classes are not applicable, as the corresponding errors cannot be confidently inferred from the review reports. For this reason, the following error classes were not considered in our field study:

- Communication
- Participation
- Process execution
- Other cognition
- Inadequate method of achieving objectives
- Management
- Elicitation
- Analysis

As a concrete example, we provide an error found in one of the reports analyzed: *Software requirement #5.12.7 is not aligned with system requirement #7.14.2 in the definition of the initial value of the variable 'InitRegs'*. In our analysis, we are not able to state if this error is a *Communication*, *Participation* or *Management* error, for example. It could be an error caused by poor communication among the development team members. This way, in the next section we present an extension to the Walia and Carver taxonomy that better fits the requirements of our analysis.

IV. FIELD STUDY

This section presents the field study: It includes an overview of the reports analyzed, the analysis methodology, and a detailed discussion of the results.

A. Reports Analyzed and Methodology

In the study we used the software requirement specifications of three real satellite projects developed by three different companies. This way, due to confidentiality reasons, limited information about these projects can be given here. The three satellites have scientific purposes: one of them has been finished and two are currently under development.

The reports analyzed describe the result of the Independent Software Verification and Validation (ISVV) of software requirement documents for the three satellite projects. Satellite systems are usually divided in sub-systems each one with a specific purpose. In general, there are three sub-systems with embedded software: Attitude and Orbit Control System (AOCS), On Board Data Handling (OBDH) System, and payload equipment control. The first is responsible for controlling and keeping the attitude of a satellite in the required position. The second is in charge of managing and monitoring the satellite data and healthy. The third controls and monitors the payload equipment.

The requirement documents analyzed in our study are related to software contained within the OBDH and payload equipment control systems. Note that, each document has a very specific scope, in general, the verification of software requirement specifications requires the use of other applicable documents, such as the system requirement document, the interface requirement document, and the software requirement documents of other subsystems.

The process used by the verification team to assess the software requirement documents was based on the guide provided by the European Space Agency (ESA) [19]. The ISVV process described in that guide is divided in several activities, including verification, validation and management activities. Broadly speaking, the independent technical specification analysis is the activity of verifying the software requirements. This activity is performed by an independent organization (i.e., by a team that belongs to an organization that is fully independent from the development team) and aims at verifying the software requirements correctness with respect to the system requirements. The result of such verification is a **technical specification analysis report**, which is the source used in our field study to characterize and classify the errors produced in the requirements phase of space software.

The independent technical specification analysis includes a thorough verification that comprises the following aspects [19]: consistency of the software requirements; software requirements completeness; robustness and reliability aspects of the requirements; readability of the software requirements; testability of the software; feasibility of producing an architectural design; trace matrix for the software requirement; and trace matrix for the interface requirements.

As mentioned before, the requirement documents where the errors analyzed in this field study have been found are final documents produced by the development team. This means that the errors found by the ISVV team are representative of real requirement errors that may escape from the quality assurance activities conducted in a careful and well-structured development process such as the one proposed by the ECSS-E-40 Part 1B Standard [1].

The errors found by the ISVV team during the Technical Specification Analysis are reported as review item discrepancies (RIDs). Each RID includes the following information:

- **RID title**: a very summarized description of the RID.
- **Traced documents**: documents related to that RID (it may be the requirements document or another one, if the discrepancy is related to another applicable document).
- **Discrepancy**: describes the non-conformance found.
- **Recommendation**: brief recommendation aiming at the resolution of the RID.

In summary, the documents analyzed in our field study (technical specification analysis reports) are documents written in natural language (English) that contain the description of the requirements errors following a RID format.

A total of seven reports from the three different projects were analyzed, containing a total of 209 RIDs. The methodology used in the analysis of the reports consisted of carefully reading and understanding all the review item discrepancies (RIDs) in each report. Each RID was then classified using the taxonomy described in Section III. It is important to state that the specialists (co-authors of this paper) that analyzed the error reports and proposed the classification never had any contact with the members of the development or ISVV teams. In other words, the classification of the requirement errors presented in the paper is solely based on the analysis of the verification reports.

The seven reports analyzed in this work belong to the three satellite projects as follows:

- Project 1: one report
- Project 2: two reports
- Project 3: four reports

The following nomenclature is used in this paper to refer to each report: “report 1.1” refers to the report of project 1; “report 2.1” and “report 2.2” correspond to the two reports of project 2; and “report 3.1”, “report 3.2”, “report 3.3” and “report 3.4” refer to the reports of the third project. There is no special reason for having used these documents and not others. Actually, the access to this kind of document is very difficult due to their confidential nature. In this way, these reports were the ones available for analysis that contained relevant information.

B. Overall Results

Table II presents, for each report analyzed, the number of requirements, the number of RIDs, and the percentage of defective requirements. The most right column shows the percentage of defective requirements found in each document. For the 2188 requirements analyzed by the verification team, 209 RIDs were found, which gives an average of almost one RID per 10 requirements. This figure is surprising, as we are talking about software requirements for critical systems, developed according to a formal process. This also demonstrates the relevance of using requirement reviews in the development of critical embedded software.

TABLE II. NUMBER OF REQUIREMENTS AND RIDS ANALYZED.

Report	# of reqs.	# of RIDs	% of defective reqs.
Report 1.1	691	22	3.18
Report 2.1	370	12	3.24
Report 2.2	430	28	6.51
Report 3.1	46	19	41.30
Report 3.2	53	11	20.75
Report 3.3	134	34	25.37
Report 3.4	464	83	17.89
Total	**2188**	**209**	**9.55**

As can be seen, the percentage of defective requirements varies in a wide range, from 3.18% in report 1.1 to 41.30% in report 3.1. If we analyze this ratio across the different projects, we can see that project 3 presents much higher values than the other projects. The discrepancy in the number of RIDs found for the different projects may be explained by the fact that the projects were executed by different teams

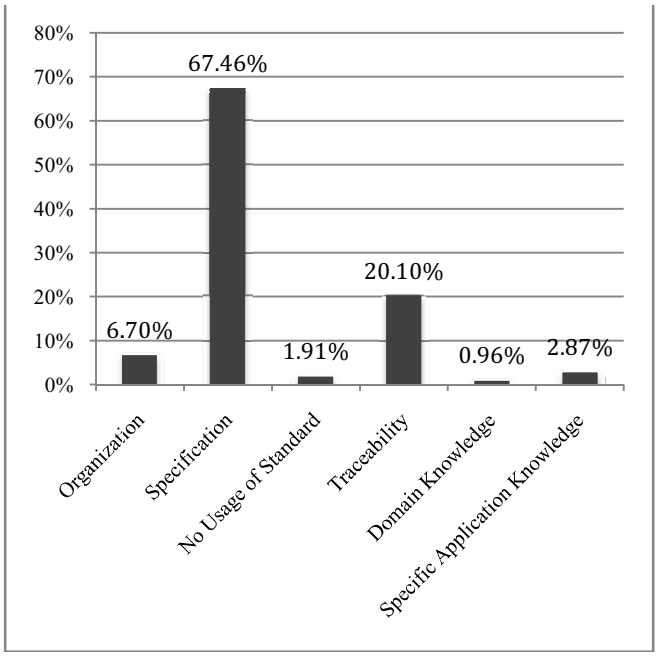

Figure 1. Requirement errors classification average.

from different companies. Probably, these teams may have different capabilities and use slightly different processes and development methodologies.

Fig. 1 presents the result of the classification of the RIDs according to the taxonomy described in Section III. As shown, the majority of the errors were classified as *specification* errors (almost 68% of the total of errors analyzed). Next, we have errors related to *traceability* (20%), *organization* (6.7%), and *specific application knowledge* (2.9%). The least common classes found are *no usage of standard* (1.9%) and *domain knowledge* (almost 1%).

TABLE III. EXTENDED CLASSIFICATION FOR SPECIFICATION ERRORS.

Error Subclass	***Description***
1. External Conflict/Consistency	Errors related to requirements that are in conflict with any information (requirements, figures, tables) in other documents
2. External Completeness	Requirements that are not fully complete when analyzed in the scope of requirements of other documents (in general, the system requirements document).
3. Internal Conflict/Consistency	Conflicting requirements or requirements that are inconsistent with other requirements or information (such as a figure) of the same requirements document.
4. Requirement Completeness	Requirements that lack some information to be completely understood. This includes requirements that have TBDs – To Be Defined, TBCs – To Be Confirmed, TBWritten – To Be Written, and lack some numeric value.
5. Document Completeness	Lack of information in the document as a whole.
6. Correctness	Requirements containing wrong information (related to the requirement itself).
7. Readability	Typos and word or sentence malformation and repetitions

C. Extended Classification and Detailed Analysis

As discussed before, the *specification* error class includes most of the errors found in the requirements review reports used in our study. This was expected given the large scope of this class, but it limits the analysis. This way, based on a detailed analysis of the errors is the *specification* error class and on the well-known attributes of software requirements quality (e.g., completeness, correctness and consistency) we defined a sub-classification for the errors, as described in Table III.

The next step was to reclassify all the errors according to the new classification, which includes the following classes:

1 - Organization
2 - Document Completeness
3 - External Conflict/Consistency
4 - External Completeness
5 - Internal Conflict/Consistency
6 - Requirement Completeness
7 - Correctness
8 - Readability
9 - No Usage of Standard
10 - Traceability
11 - Domain Knowledge
12 - Specific Application Knowledge

In addition to the extension of the classification proposed by Walia and Carver, we also separated the error classes that are related to the software requirements document as a whole (the first two error classes in the list above, *organization* and *document completeness*) and the error classes that are related to individual requirements (the remaining classes of the list). Table IV shows the results of the division of the RIDs in

these two groups. As we can see, the majority of the errors are related to individual requirements, with almost 90% of the total of RIDs. Although this was expected, it shows that there is a considerable number of problems that affect the entire document.

TABLE IV. DOCUMENT ERRORS VS. INDIVIDUAL REQUIREMENT ERRORS.

Error Class	# of RIDs	% of RIDs
Whole document errors	22	10.53
Individual requirement errors	187	89.47
Total RIDs	**209**	**100**

Table V shows the breakdown of the results for the errors related to the document as a whole (two classes of RIDs). Results show that organization problems are more frequent than document completeness. However, this result should be analyzed taking into account that it is in general much easier to find out what is wrong in the requirements than what is missing. In any case, the field data shows 14 RIDs related to organization problems against only 8 RIDs related to document completeness.

TABLE V. DOCUMENT ERRORS CLASSIFICATION.

Error Type	# of RIDs	% of RIDs
Organization	14	63.64
Document Completeness	8	36.36
Total RIDs	**22**	**100**

Fig. 2 presents the result of the classification of the remaining RIDs (187 RIDs related to individual requirements). As shown, the most common error class found is *external conflict/consistency*, corresponding to 24.6% of the individual requirement errors found. Next, we have errors related to *traceability* (22.5%), *external completeness* (17.1%), and *requirement completeness* (15.5%). These four error classes comprise almost 80% of the individual requirement errors found in the software requirement documents. The least common error classes are *domain knowledge* (1.1%), *non-usage of standard* (2,1%), *readability* (2.7%), *Specific application knowledge* (3.2%), and *correctness* (6.4%). In general, individuals that work with safety-critical and high cost space systems, are well trained and have good knowledge about the system. This can explain why the problems related to knowledge have the minor percentages of error types found.

Table VI presents the breakdown of the results per report. The percentages presented are related to each report individually and do not consider the errors related to the requirements document as a whole. The values marked with a dark gray background highlight the most common error class in each report, and the values marked with a light gray background highlight the second most common class.

As we can observe, the traceability error class is the most common in 3 of the 7 reports (R2.2, R3.1, and R3.3). Additionally, external conflict/consistency is the most common class in 2 reports (R3.2 and R3.4) and the second most common in 2 other reports (R2.2 and R3.1). An important aspect is that there is no clear pattern among the reports, which is a clear sign that the generalization of these results should be taken very carefully.

V. FAULT OPERATORS FOR SPACE SOFTWARE REQUIREMENTS

Although great care should be taken in the generalization of the error distribution presented in the previous section (even for the specific context of embedded space software), in the absence of better evidences from other field studies, we should consider this distribution (as a best effort) for the emulation of realistic requirement errors.

Defining an approach to inject realistic requirement

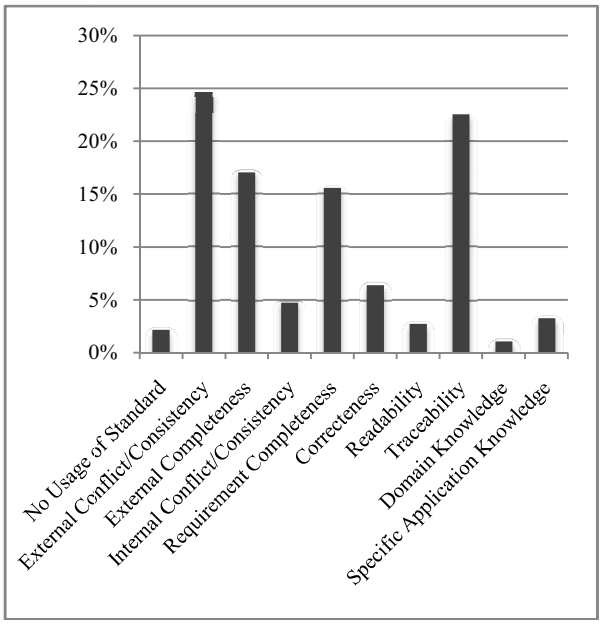

Figure 2. Individual requirement error classification average.

TABLE VI. INDIVIDUAL REQUIREMENT ERRORS CLASSIFICATION RESULT PER REPORT.

Error class	R1.1	R2.1	R2.2	R3.1	R3.2	R3.3	R3.4
No usage of standard	5.26%	0.00%	8.00%	0.00%	0.00%	0.00%	1.25%
External conflict/consistency	15.79%	11.11%	20.00%	16.67%	37.50%	14.29%	33.75%
External completeness	26.32%	11.11%	4.00%	11.11%	0.00%	0.00%	28.75%
Internal conflict/consistency	5.26%	33.33%	4.00%	5.56%	12.50%	3.57%	1.25%
Requirement completeness	15.79%	44.44%	8.00%	16.67%	12.50%	7.14%	17.50%
Correctness	0.00%	0.00%	4.00%	0.00%	12.50%	25.00%	3.75%
Readability	0.00%	0.00%	12.00%	0.00%	0.00%	0.00%	2.50%
Traceability	21.05%	0.00%	28.00%	44.44%	12.50%	46.43%	11.25%
Domain knowledge	5.26%	0.00%	0.00%	0.00%	0.00%	3.57%	0.00%
Specific application knowledge	5.26%	0.00%	12.00%	5.56%	12.50%	0.00%	0.00%
Total	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**

errors requires understanding the most common requirement error classes (as presented in our field study). Based on these classes we need then to define fault injection operators, which represent a structured description of the concrete changes that should be introduced in a requirements document in order to emulate a specific error class (or subclass, as we will see further on). The concept of fault operator has already been used successfully in traditional software fault injection [20].

It is worth noting that the error distributions observed in the field (discussed in Section IV) does not affect the definition of the fault operators needed to emulate realistic requirement errors. In fact, even if additional field studies show that the distribution of the most common types of errors is different, the new error distribution will not change the fault operators themselves, but will simply change the mixture of fault operators used to compose a faultload. In the limit, new studies may identify new fault operators that should be added to the set already identified in the present field study.

The proposed fault types (corresponding to the classification classes proposed in Section IV.C) and their definitions are:

1. *Problem in the Organization*. Introduction of two requirements with the same title, a requirement with a title that does not represent its specification, or a duplicated requirement (that is covered by another requirement).
2. *Document Incompleteness*. Removal of a complete requirement.
3. *Missing Usage of Standard*. Introduction of non-standard references to some variables or other requirements or documents, or use of different formats for documenting requirements.
4. *External Inconsistency*. Introduction of some information in conflict with the information of any other applicable document or using a nomenclature or term that is different from the one used in other applicable documents.
5. *External Incompleteness*. Removal of some information that is as complete as the information contained in a high level document (such as the system requirements), in a way that the software requirement is not omitted, however it is no longer as complete as before.
6. *Internal Inconsistency*. Inclusion of some information in conflict (or with some inconsistency) with another requirement or information (such as a figure or table) in the same document.
7. *Requirement Incompleteness*. Some important information necessary to the requirement be completely understood is omitted (such as the numeric value of some variable or the inclusion of some "TBC" or "TBD" (to be confirmed or to be defined).
8. *Incorrectness*. Some wrong information related to the requirement itself is included (or some correct information is changed).
9. *Low Readability*. Inclusion of some typo error or word or sentence repetition in a requirement.
10. *Missing of Traceability*. Some traceability information is omitted or changed to wrong information.
11. *Lack of Domain Knowledge*. Some wrong information related to the domain or system knowledge is included or changed in some requirement.
12. *Lack of Specific Application Knowledge*. Introduction of wrong information related to the software interface with the rest of the system (e.g., timing, data dependency and event constraints, mistakes in expressing the end state or output expected of some process).

From these fault types, we created fault operators to describe how to inject each fault in a concrete software requirements document. Ideally, the process of injecting requirement faults (actually, requirement errors) should be fully automatic. However, a possible tool to inject requirement faults in a complete automatic fashion is very complex, as it should be able to deal with the natural language used to specify requirements. This would require the use of advanced artificial intelligence techniques, which are beyond the aim of this work.

A more practical approach is to use a simple tool in the form of a text editor add-on, to assist the user in the task of manually inserting the changes in the requirements as specified by the fault operators. A tool to provide a semi-automatic (i.e., user assisted) injection of realistic errors in

requirement documents is currently being developed as a follow up of the research work presented in this paper.

In the following paragraphs we describe the fault operators for the three most common fault types.

***Fault type*:** External Inconsistency (ES)
- **ES-01**: Change the value of some variable that is also defined in another document (such as the initial, maximum or minimum value of some variable).
- **ES-02**: Change the size of some memory region defined in another document.
- **ES-03**: Change the type of some variable in order to differ from its real type defined in another document.
- **ES-04**: Change the behavior of some variable when it reaches its maximum value in a way that it becomes inconsistent with the information in other document (e.g., stop incrementing or restart to zero).
- **ES-05**: Omit some input or output parameter of some function described in the requirements that is also described in another document.

Fault type: Missing of Traceability (MT)
- **MT-01**: Delete some references from software requirements to system requirements from the traceability matrix.
- **MT-02**: Delete some references from software requirements to interface requirements from the traceability matrix.
- **MT-03**: Change, in the traceability matrix, the system requirement to which some software requirement is traced.
- **MT-04**: Change the interface requirement to which some software requirement is traced.
- **MT-05**: Change the reference to some system requirement or interface requirement or function in the software requirement text.

Fault type: Requirement Incompleteness (RI)
- **RI-01**: Omit a numeric value defined in a software requirement.
- **RI-02**: Insert a "TBC" near a numeric value in a software requirement.
- **RI-03**: Replace a numeric value by a "TBD" in a software requirement.
- **RI-04**: Omit some of the events in a software requirement that describes the behavior of the software in response to various events.

The use of some of these fault operators has constraints that depend on the kind of fault operator. For example, the fault operator ES-01 can be used only in a requirement that defines the initial, maximum and/or minimum value of some variable that is also defined in another document. The fault operator MT-05 can be used only in a requirement that refers to some system requirement, interface requirement or function in its text. Obviously, the results presented in Section IV can be easily used to maintain the number of injected faults proportional to the importance of each error class.

In order to give the reader an idea of how these operators can be used in practice, we show below some simple examples of requirements with concrete faults injected.

Original requirements:
Software requirement: When the counter C0 reaches its maximum value, it shall restart from zero.
System requirement: The counter C0 shall restart from zero if it reaches its maximum value.

Requirement with fault:
Software requirement: When the counter C0 reaches its maximum value, it shall stop incrementing.

Error: There is an inconsistency between the software and system requirements about the action of the counter C0 when it reaches its maximum value.

Type: External Inconsistency.

Fault Operator: ES-01

Original requirement:
The counter C0 shall restart from zero when it reaches its maximum value defined in Sys-Req-00110.

Requirement with fault:
The counter C0 shall restart from zero when it reaches its maximum value defined in Sys-Req-00120.

Error: Actually, the maximum value of the counter C0 is defined in Sys-Req-00110.

Type: Missing of Traceability.

Fault Operator: MT-05.

Original Requirement:
The data memory shall be divided into 8 virtual pages to be addressed by the user, each one with 32 kilobytes of size.

Requirement with fault:
The data memory shall be divided into 8 virtual pages to be addressed by the user.

Error: The requirement does not define the size of each page (and it is not defined wherever in the document).

Type: Requirement Incompleteness.

Fault Operator: RI-01.

Note that faults must be injected in the software requirements in a manner that the sentence remains grammatically correct (incorrect sentences are easier to identify).

VI. Application Scenarios

This section proposes some application scenarios for the field study results and for the fault operators for the injection of realistic requirement errors. These application scenarios are presented here to motivate further research on this topic

and the development of tools that facilitate the concrete use of the results.

Evaluating and training requirement reviewers

Faults can be injected in requirement documents using the requirement fault operators and submitted to the inspection of the reviewers under training/evaluation. The number and kind of requirement errors detected give an indication of the expertise of the reviewers and points to possible gaps or biases in their review analysis. This information is an important feedback for the reviewers' training activity. A similar process for the security area has already been proposed in [21].

Estimating the number of requirement errors in real requirement specifications

This is similar to the traditional defect seeding [6] approach applied to the requirements specifications. The proposed application scenario has two main differences when compared to traditional defect seeding: 1) uses realistic defects (instead of simple mutations), as the injection is guided by the proposed fault operators and 2) is applied to the requirements documents while traditional defect seeding is normally applied to program code.

The idea consists of injecting a given number of faults in the requirements document and submitting it (i.e., the document that contains the seeded faults and possibly some real unknown faults) to the review process. The ratio between the number of real faults detected and the percentage of injected faults detected provides a good prediction on the total number of real faults still existing in the requirements document. In fact, if the fault injection technique is realistic and representative, the ratio between the detected and the total number of faults should be similar for seeded and real faults. This highlights the relevance of a consistent field study for the proposal of fault injection operators for realistic seeding.

Defining checklists for quick requirements verification

The fault operators can be used to elaborate a checklist to verify the requirement documents in a systematic, cheap and quick way. A checklist can be used with different purposes. For example, a reviewer can use a checklist during the review process (following a checklist based on the most common types of requirement errors found in the field will improve the effectiveness of the reviews). A checklist can also be used by the development team to indicate the degree of maturity of the requirements document, avoiding premature reviews and saving money and time for the project. Finally, development teams can use a checklist to generate guidelines to improve the process of developing requirement documents.

Benchmarking requirements specifications

Another application of fault operators for software requirements is to provide the basis for the definition of a benchmark for the requirement specification documents. The motivation for such benchmark is to make the requirement review process faster and cheaper (a key aspect is also to have a review process that is less dependent on reviewers' judgments and expertise). As a starting point, the fault operators will allow the definition of a faultload for the benchmark. Many possibilities have to be explored and this idea of applying the concept of benchmarking to the software development process basically represents future work. In any case, as the concept of dependability benchmarking implies a faultload (quite often mixed up with the workload), we believe that the definition of fault operators for software requirements is a first and mandatory step to investigate the feasibility of this idea. Naturally, such benchmark must be regarded as part of the requirement review and not as a replacement for it.

VII. Conclusion and Future Work

This paper proposes a set of fault operators to support the process of injecting faults in software requirements. The definition of these fault operators is based on a field study on software requirement errors of space systems. Results suggest that the most frequent types of errors are: *External Conflict/Consistency*, *Traceability*, *External Completeness* and *Requirement Completeness*.

The paper also puts forward a list of concrete application scenarios for the proposed fault operators: evaluating and training requirement reviewers, estimating the number of errors in real requirement specifications, elaborating checklists for quick requirement evaluation, and benchmarking for requirement specifications documents.

The next steps of this research is the definition of a benchmark for space software requirements using a faultload based on this field study, as well as the development of a toolset to facilitate the use of the benchmark for software requirements in concrete projects.

Acknowledgment

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

References

[1] ECSS Space Engineering: Software – Part1: Principles and requirements, ECSS-E-40 Part 1B Standard, 2003.

[2] F. P. Brooks, "No silver bullet: essence and accidents of software engineering", IEEE Computer, vol. 20, n. 4, 1987, pp. 10-19.

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

[4] 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, July 1996, pp. 231-261

[5] N. G. Leveson, "The role of software in spacecraft accidents", AIAA Journal of Spacecraft and Rockets, vol. 41, n. 4, 2004, pp. 564-575.

[6] K. Y. Cai, "Software defect and operational profile modeling", Kluwer Academic Publishers, Boston, 1998.

[7] K. Kanoun and L. Spainhower (editors), "Dependability Benchmarking for Computer Systems", Wiley IEEE Computer Society Press, 2008.

[8] P. Koopman, and H. Madeira, "Dependability Benchmarking & Prediction: A Grand Challenge Technology Problem", 1st IEEE Int. Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems, Phoenix, Arizona, USA, November 30, 1999.

[9] 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, July 2009, pp. 1087-1109.

[10] R. Lewis, "IV&V: a life cycle engineering process for quality software", Wiley, 1992.

[11] R. J. Halligan, "Requirements metrics: the basis of informed requirements engineering management", Complex Systems Engineering Synthesis and Assessment Technology Workshop, 1993.

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

[13] 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, June 2007, pp. 417-432.

[14] F. T. Sheldon, H. Y. Kim, and Z. Zhou, "A case study: validation of guidance control software requirements for completeness, consistency and fault tolerance", Eighth Pacific Rim International Symposium on Dependable Computing (PRDC'01), 17th-19th December 2001, IEEE Computer Society, Seoul, Korea, 2001, pp. 311-318.

[15] 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, June 2009, pp. 28-42.

[16] H. F. Hofmann, and F. Lehner, "Requirements engineering as a success factor in software projects", IEEE Software, July/August 2001.

[17] B. Bernárdez, M. Genero, A. Durán, and M. Toro, "A controlled experiment for evaluating a metric-based reading technique for requirements inspection", Proceedings of the 10th International Symposium on Software Metrics (METRICS'04), 11th-17th Sept. 2004, Washington, DC; USA, 2004, pp. 257-268.

[18] M. Hammil, and K. Goseva-Popstojanova, "Common Trends in Software Fault and Failure Data", IEEE Transactions on Software Engineering, vol. 35, n. 4, 2009, pp. 484-496.

[19] ESA Guide for Independent Software Verification and Validation, Issue 2.0, European Space Agency, ESTEC, The Netherlands, Dec. 2008.

[20] J. Duraes, and H. Madeira, "Emulation of Software Faults: A Field Data Study and a Practical Approach", IEEE Transactions on Software Engineering, vol. 32, n. 11, 2006, pp.849-867.

[21] J. Fonseca, M. Vieira, and H. Madeira, "Training Security Assurance Teams using Vulnerability Injection", 14th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'08), Taipei, Taiwan, Dec. 2008, pp. 297-304.