Is 100% Test Coverage a Reasonable Requirement? Lessons Learned from a Space Software Project

Christian R. Prause[1], Jürgen Werner[2], Kay Hornig[2], Sascha Bosecker[2], and Marco Kuhrmann[3]

[1] German Aerospace Center, Bonn, Germany
christian.prause@dlr.de
[2] Test Spacecom GmbH, Backnang, Germany
Juergen.Werner@tesat.de, Kay.Hornig@tesat.de, Sascha.Bosecker@tesat.de
[3] Clausthal University of Technology, Institute for Applied Software Systems Engineering, Goslar, Germany
kuhrmann@acm.org

Abstract. To ensure the dependability and safety of spaceflight devices, rigorous standards are defined. Among others, one requirement from the European Cooperation for Space Standardization (ECSS) standards is 100% test coverage at software unit level. Different stakeholders need to have a good knowledge of the implications of such a requirement to avoid risks for the project that this requirement might entail. In this paper, we study if such a 100% test coverage requirement is a reasonable one. For this, we interviewed the industrial developers who ran a project that had the sole goal of achieving 100% unit test coverage in a spaceflight software. We discuss costs, benefits, risks, effects on quality, interplay with surrounding conditions, and project management implications. We distill lessons learned with which we hope to support other developers and decision makers when considering a 100% unit test coverage requirement.

Keywords: Validation and verification, Software quality, Unit testing, Test coverage, Expert interviews, Spaceflight, Software criticality, Process requirements

1 Introduction

Software has become key to spacecrafts. It is the devices' brain that, among other things, maintains altitude and orbit, reads and analyzes sensor data, and controls the hardware. In particular, software is key to detect, isolate, and recover from unexpected situations and failures and, eventually, software ensures communication with ground stations. Due to the special environment, once deployed, a spacecraft has to 'survive' autonomously. Maintenance of its hardware is—if at all possible—impractical.

More fatal than crashing software is software that performs in the wrong way, as it may give commands that destroy a device or the whole spacecraft. For instance, recently, the *Hitomi* telescope was erroneously commanded by its software to start spinning faster and faster until it disintegrated [22]. A software problem caused the recent crash of the *Schiaparelli* lander, which prematurely released its parachute several hundred kilometers above ground: Contradicting calculations of sensor data made the navigation software erroneously assume the lander had already touched Mars' surface [20]. Another software problem hit the Mars rover *Spirit* 18 Sols[1] after landing. The rover was caught in the rebooting cycle as it could not read a full fixed-memory block. The rover successfully passed a 10-Sol test concerning exactly this kind of problem prior to landing, yet, the memory bank was full at Sol 18, and the rover could only be put back into operation by using some 'backdoors' in the system [1]. Hence, software failures in space devices can be costly. Malfunctioning spacecrafts, moreover, may seriously threaten human life or the environment, e.g., remnants of space probes orbiting Earth endanger other satellites[2], uncontrolled reentry might endanger whole regions, e.g., ROSAT's uncontrolled reentry [18], and so forth. Software of a space device has to be dependable (i.e., reliable, available, and maintainable) and safe. To ensure high dependability and safety, space software and systems are developed under a strict quality and product assurance regime according to an extensive system of standards [19].

Context The standards of the *European Cooperation for Space Standardization* (ECSS) are a coherent and comprehensive collection of standards addressing all areas of spaceflight. At the highest level, the ECSS standards are divided into management, engineering, and product assurance (quality) branches, and further subdivided into so-called areas. Each standard comprises a large number of requirements prescribing what is to be achieved. The standards ECSS-E-ST-40C (Software Engineering; [6]) and ECSS-Q-ST-80C (Software Product Assurance; [8]) address the development and product assurance of software for space applications. One of the standards' requirements for highly-critical software is that "100% code branch coverage at unit testing level" must be achieved. However, ECSS prescribes 100% for classes[3] A and B only [6], but leaves coverage for classes C and D open to negotiation. Achieving a high—or even a full—coverage is demanding, though. Adopting Tom Cargill's 90/90 rule [4] to unit testing, one could state: *The first 90% of the unit test code accounts for the first 90% of the development time. The remaining 10% of the unit test code accounts for the other 90% of the development time.* So, what happens, if contracting bodies ask for a 100% unit test coverage?

Contribution In this paper, we report lessons learned from a space software project that had the goal of reaching a test coverage of 100% using unit tests for the flight software of a laser communication device (LCT). While the project very closely reached the goal

[1] A Sol is a day on Mars, which is 24h 37m, while a day on Earth is 23h 56m. The time unit Sol is used to run Mars operations and to not have the demand of continuously converting time.

[2] Estimates mention more than 500,000 pieces of junk, so-called 'space debris', orbiting Earth at high speeds of dozens of km/s [17]. Due to their extreme speeds, the kinetic energy of even small particles of only a few millimeters can cause impact craters of several dozen centimeters on the spacecraft, and lead to fatal and catastrophic effects like disintegration of the target.

[3] The ECSS standards define four levels for criticality from A to D (ECSS-Q-ST-30C [7]). For instance, criticality class A comprises catastrophic events, e.g., loss of life, launch site facilities, or the entire spacecraft. Class B is for the risk of losing the ability to perform the mission (loss of mission), and Class C for a major mission degradation. The LCT system, which is the subject of this paper (see Section 2) is classified as B (system), and its software as C.

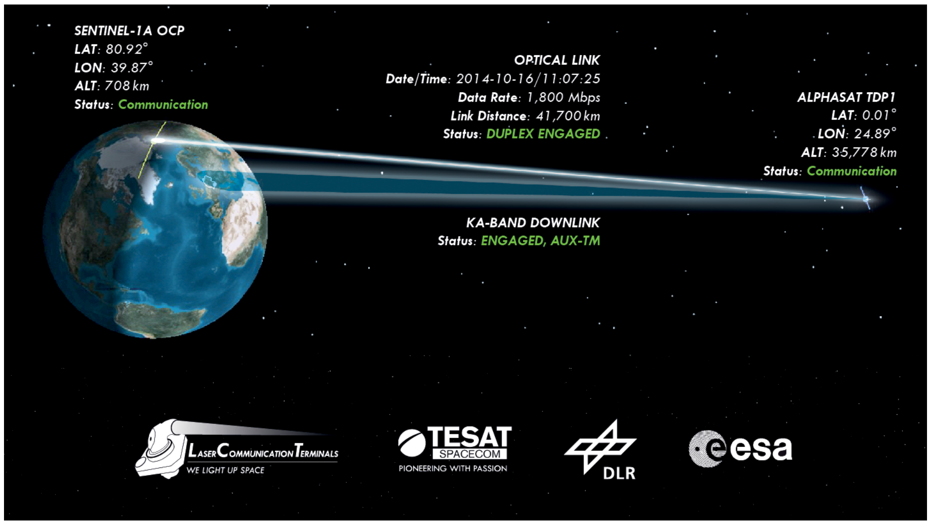

Fig. 1. Sentinel and AlphaSat satellite link, and relay to Earth (source: Tesat Spacecom GmbH).

of 100% test coverage ($> 99.5\%$), the effort turned out to be tremendous. Two developers spent two years developing the unit tests for a software of about 25,000 lines of code. Using semi-structured expert interviews, we studied how the project incrementally increased the test coverage to achieve the 100%-goal. We present lessons learned to stimulate a critical discussion about cost, benefits, and reasonableness of the 100% test coverage requirement.

Outline The remainder of the paper is organized as follows: Section 2 provides the background of the project reviewed. Section 3 presents the research design, before we present our findings in Section 4. Section 5 discusses related work. Finally, Section 6 concludes the paper.

2 Background

The 'information society' relies on data and data exchange; and the amount is increasing year-by-year. More than 100 communication satellites in service build the communication backbone providing communication, bringing internet to remote locations, and broadcasting tens of thousands of television and radio programs worldwide.

The *Copernicus* program of the European Commission aims to establish a European capacity for earth observation by providing atmosphere, maritime, land, climate, emergency and security services. Several *Sentinel* satellites are the program's heart and produce large amounts of data. For instance, *Sentinel-2A* orbits Earth at an altitude of 786 km, delivering optical images on 13 spectral channels at a depth of 12 Bit per channel at resolutions of up to 10 m. A typical image is a tile of 100 km^2, or approx. 500 MB. A setup of two *Sentinel* satellites generates up to 1.6 TB of compressed raw

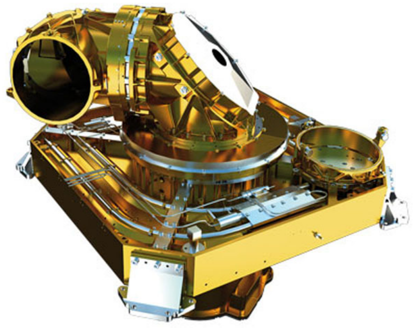

Fig. 2. The Laser Communication Terminal (source: Tesat Spacecom GmbH).

image data per day, or 160 MBit/s continuously. Having access to imagery as quickly as possible is crucial for a number of *Copernicus* applications. However, earth's curvature prevents continuous radio communication with ground stations in Europe (broken line-of-sight) [9].

Laser Communication To overcome this limitation, *Sentinel* satellites use the *European Data Relay System* (EDRS). EDRS features geostationary satellites at 36,000 km altitude that have a permanent link to European ground stations (Fig. 1). EDRS and *Sentinel* satellites carry novel *Laser Communication Terminal* (LCT) devices to establish laser links among one another to overcome bandwidth limitations and to reduce off-line windows (Fig. 2). LCT devices allow for data transfer rates of up to 1.8 GBit/s. For this, the LCT laser has to hit a target of 200 m in diameter from a distance of 45,000 km, which corresponds to a moving 2-Euro coin from a 6.8 km distance. Besides 'housekeeping' activities, the LCT software is primarily responsible for laser-targeting and controlling the hardware, e.g., power management or controlling the coolant system for the laser. Using software allows for precise targeting of the laser and, moreover, the software allows for compensating degrading hardware, and failure detection, isolation and recovery (FDIR).

The LCT Device LCT plays a key role for the *Sentinel* and EDRS satellites, and for the *Copernicus* program as a whole. Due to the criticality, software quality is crucial and, therefore, quality assurance is a vital part of the system's development. Development of the LCT device itself stretched over several projects and lasted longer than a decade, resulting in several changes of key personnel. Furthermore, the LCT project involves several stakeholders: The LCT devices are developed by Tesat Spacecom GmbH, which was contracted by the national space agency, the German Aerospace Center (DLR) that also defined the quality requirements. Those requirements are, basically, grounded in ECSS standards, yet differ in some aspects, and, in particular, are tailored to project characteristics according to different technical, programmatic and risk criteria [19]. The

Sentinel 2 and *EDRS-A* satellites that host the LCTs are manufactured by Airbus DS on behalf of the European Space Agency (ESA), which applies mostly unmodified ECSS.

Within the conglomerate of partners involved and standards to implement, the ECSS standards received a major revision (*Issue B* to *Issue C*) while the LCT devices were produced. The new revision makes test coverage a first-class-citizen. Even though LCT was rigorously quality assured[4], the manufacturer did not yet collect test coverage data. This led to a situation in which test coverage was unknown while contracting agencies insisted on the new 100% test coverage requirement and proving its fulfillment. To overcome this situation, an agreement among the involved parties was made to initiate a separate project, which had the goal of increasing test coverage to 100% before the launch of the satellite.

3 Research Design

This section describes our research design, starting with describing the research objective and the research questions in Section 3.1. Section 3.2 describes the data collection procedures, including the interview instrument and the subjects selection. The analysis procedures are described in Section 3.3, and we discuss threats to validity in Section 3.4.

3.1 Research Objective and Questions

The overall objective of this study is to shed light on what a 100% coverage-requirement entails. We aim to study whether 100% test coverage is a reasonable requirement, what experienced practitioners "normally" consider good/high quality, and what benefits practitioners see in going beyond "normal" coverage and towards 100%. We collect information about the practitioners' perception of the requirement's effects on costs, benefits, risks, its interplay with surrounding conditions, and project management implications. Hence, the overall research question investigates:

RQ: *Is 100% test coverage a reasonable requirement?*

3.2 Implementation

The study was conducted as semi-structured interview with experts in a 2-day workshop. We talked to all interviewees separately.

[4] The product assurance process performed so far includes several parties and procedures. The device manufacturer's product assurance reports to and is supervised by the customer's product assurance (cf. [19]). Further involved on satellite-level are the customer's and the prime contractor's product assurance. At the technical level V&V activities include, inter alia, static analyses, verification controls' and reviews. At device level, separate test teams carry out software tests in isolation and as part of the integrated device prior to shipment for full integration and system testing.

Interview Instrument Table 1 shows the guideline of the semi-structured interview. The table shows the eight top-level 'entry' questions and (selected) detailed questions. In total, the guideline comprises a maximum of 66 questions in eight categories to ensure all relevant topics are addressed in every interview. In the interview, the participants were asked the entry question of the respective category to start the conversation. The interviewers traced the guideline and only asked follow-up questions from the question pools if information was not provided or if responses required clarification.

Interview Subjects This paper reports on the interviews with the project's core personnel, i.e., the two developers and the project manager. For the developers, it was their first space project. The project manager already had a few years of experience in the space domain. All interviewees previously worked in other embedded software domains, mostly automotive software. They all look back on an industrial development experience of 10–25 years, working primarily with the languages C and C++ (C is the predominant language in space projects).

Interview Procedure Before the interviews, we informed interviewees about the interview and its purpose, and asked them to prepare themselves. Participation was voluntary; no test development team member opted out. The interviews were conducted individually and face-to-face at the company's site and took between 60-90 minutes. One researcher preceded the interview using the guideline. The second one made short notes and only asked clarifying or follow-up questions. Each interview was audio recorded. Finally, all participants were summoned for a wrap-up session to clarify possibly remaining open points, ask things we might have missed, and to provide room for further discussion.

Project Performance Data Complementing the qualitative data collected in the interview, we had access to project performance data, notably, the test coverage statistics. These data sets were included in our analysis to complement and to help interpret the qualitative data.

3.3 Analysis Procedures

To qualitatively analyze the interview data, both researchers performed an initial review to plan the transcription and to revise the data analysis plan. A secretary was appointed to transcribe the interviews, which was performed interactively with regular consultations and quality assurance on (tentative) results. Based on the transcripts, we qualitatively analyzed the data to extract the required information and to answer the research questions. Project performance data amended the analysis[5].

3.4 Validity Considerations

The lessons learned are based on the experts' opinions expressed during the interviews. Although the experts are experienced industrial developers, we still convey opinions

[5] Due to the sensitivity of the data, we only present excerpts and anonymized results.

Table 1. Summary of the interview guideline, including (selected) detailed questions used to drive the interview.

No.	Top-level Question/Question Category	#Q
1.	Demographics *Questions: Role in the organization? Role in the project? Years of experience in Space projects? Years of experience in using the development environment? Did you do something like this before?*	5
2.	Is achieving 100% test coverage a reasonable goal? *Selected Detailed Questions: What level of test coverage do you consider 'normal'? What level of test coverage do you consider efficient from an economical perspective? Which minimum level of test coverage do your clients usually expect you to deliver? For what kind of critical software do you consider a 100% test coverage always necessary? Which methods and techniques do you apply to maximize test coverage? Related to your personal experience, do you expect positive impact on the overall system quality?*	13
3.	Have you achieved the project goals? *Selected Detailed Questions: Have you reached the 100% test coverage? What were the biggest organizational challenges? What were the biggest technical challenges? Are there further factors that positively/negatively affect the reachability of the 100% goal? Did the quality of the system improve (due to the 100% goal)?*	10
4.	Development tools and methods *Selected Detailed Questions: Which positive/negative impact did the programming language(s) have on the 100% goal? Did you have to adhere to an external standard (multiple standards)? If yes, were these standards supportive for the management, developers, or quality assurance? If yes, how did these standards influence your selected approach? If yes, does your software fulfill all requirements set by the standards completely? Which tool to measure test coverage was used? Would you rate your chosen approach more 'agile' or more 'traditional'?*	20
5.	Extra (general) questions *Questions: How many errors were found through unit testing? When were the most errors found? (time, at coverage rate of x%, distribution function) How big is the overall software system? How big is the test system?*	4
6.	Static Verification *Questions: Which tools and methods were used in the project to support the static verification? Was there a considerable flow of information to and from static verification? If yes, how was the information flow implemented, and what did the information exchange comprise? Were synergy effects between both approaches observed? Did the testing make the static verification dispensable (or vice versa)?*	5
7.	Software Metrics *Questions: Why did you collect KPIs in your project? Was collecting KPIs beneficial to the project (KPIs, e.g., coverage, test lines, etc.)? Would you have liked to collect even more or more sophisticated KPIs? What other KPIs would you have liked to collect?*	4
8.	Lessons Learned *Questions: What should others know, who face the same situation? What should be the learning for the space agency as client? What is a general take-away for clients concerning the 100% goal? What are your own lessons learned from this project? From your experience, what are the biggest risks in this kind of project?*	5

#Q: Number of questions per top-level question/question category related to one particular project only. The interviews were conducted during the final week of the project. One of the interviewers was also the customer's appointed software quality manager during the project. While this situation possibly affected interviewees' responses (see also disadvantages of interviews as stated in [23]), it has to be noted that the project was conducted primarily on demand from the prime/satellite customer. To improve the objectivity of the interview, an external researcher, who was not involved in the project, was called in. Due to this interview setup, the participants could speak rather freely. The interviews were conducted in the participants' mother tongue (German), and quotes presented in this paper were translated to English from the German interview transcripts afterward. We tried to preserve as many intricacies of the responses as possible, yet, there is the risk that a few subtleties have been lost during translation. The interviewees were given the opportunity to review the completed paper, and encouraged to provide clarifications and comments.

4 Results

In this section, we analyze quantitative project performance data (Section 4.1) and condense lessons learned based on qualitative findings from interviews in Section 4.2.

4.1 Quantitative Data

The project tracked test-coverage progress on a daily basis. This resulted in approx. 400 data points covering approx. 700 days. The actual project duration was longer than two years because of the necessary management activities (ramp-up times, creation of documents, reviews, delivery and acceptance, etc.). Figure 3 plots the percentage of code covered. The curve is quite linear for the most part of the project. However, it bends when it reaches approximately 90% of coverage. This indicates that the last few percent of coverage require significantly more effort.

Figure 3 also compares statement coverage to branch coverage. Branch coverage was not monitored in the first place. The data shows that—by its nature—it tends to be a bit lower than statement coverage if not monitored. At the same time, however, it is not far out. Once monitored, branch coverage can be improved in conjunction with statement coverage without much additional effort. This changed upon reaching approx. 90% of coverage, when branch coverage started to fall behind, until it again catches up when getting closer to 100%. As stated in the interviews: *"In the beginning, coverage increased quite linearly. Of course, there were some disturbances* [e.g., Christmas]. *But the last few percent were really difficult."* The unit test development project found between 20 to 30 issues that could have been interpreted as actual errors. Less than three of these errors detected were considered having a potentially serious impact on the device's funtionality. The project caused a development effort of four person years plus support staff for about 25,000 lines of code (LoC).

4.2 Lessons Learned

This study aimed at collecting experience from a project in which a test coverage of 100% should have been achieved in order to meet requirements defined by an exter-

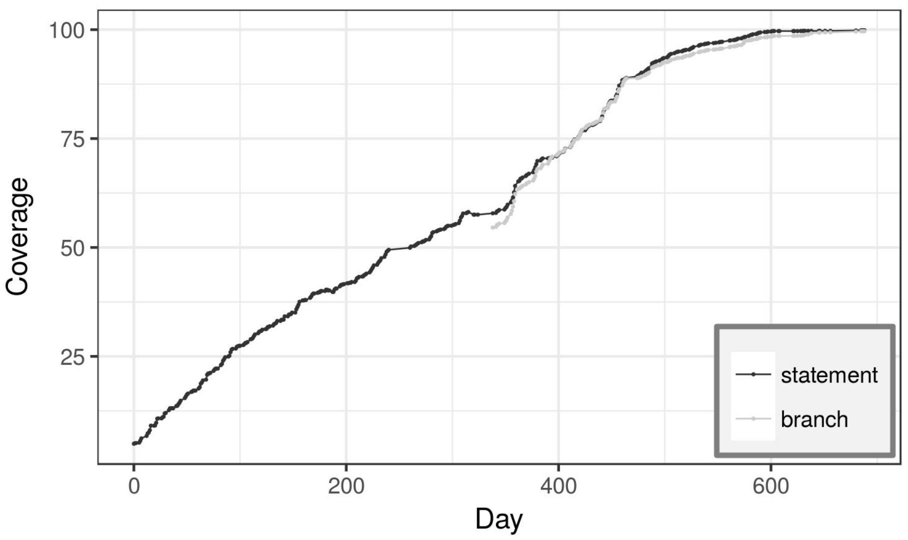

Fig. 3. Statement and branch coverage over time.

nal standard. From the qualitative analysis of the interviews conducted, we extract the following lessons learned.

100% coverage is unusual but achievable To start the interviews, we asked participants if they ever faced a similar requirement before. Prior to this project, all participants worked for different companies. Yet, all of them faced *"such a requirement for the first time, and for the first time it was stated that explicitly."* Furthermore, neither have they faced *"such a high coverage ratio"* before nor did they have to realize it *"in this way, as a follow-up project."* In their previous experience, coverage *"was not directly being looked at"*, and *"was an issue only in the area of [complex electronics, i.e.,] ASIC and FPGA"*.

Asked what they consider a 'normal' coverage, the participants had difficulties in naming a precise number, as *"it depends on what one wants to achieve"*. As a general reference, the participants mentioned it *"may be around 80%* [... because] *the effort per percentage point of coverage, typically increases dramatically towards the end of a project."* Yet, one participant stated that, in general, referring to all static and dynamic verification techniques available, *"when you have reached more than 90%, you are doing well."* This raises the discussion, what 'good' coverage is after all; because over the last decades, the threshold for what constitutes a 'good' coverage ratio may have risen: *"With reasonable effort, I would say 90% is a lot. In this project here, everything went smooth until we reached 95% and then it became difficult, because you start to deal with the code that is difficult to reach.* [...] *My experience in early years was that 85% was excellent."*

Nevertheless, participants basically agreed that a 100% coverage can be achieved. Nitpicking, the project reached "only" a coverage of 99.9% and one participant stated: *"100%—you can say good-bye to that—but 100 minus epsilon is probably possible."*, but another one disagreed: *"100%, and I really mean 100 dot zero zero zero percent, is definitely achievable."* In fact, true 100% or 100%-epsilon, is probably an academic question, as from a practical perspective other problems are more relevant.

100% coverage is sometimes necessary One participant considers 100% as *"a necessary, but not a sufficient condition for quality"*. He explained that coverage *"should be 100%"* because otherwise there is the *"risk of fair-weather tests* [. . . , i.e.,] *that potentially the most complex, hardest to understand, or most difficult to reach functionality is left untested."* If less was the target, developers might *"pick the 80% most beautiful tests that they can wangle most easily,* [. . . and think that as they] *satisfied the metric, now everything is fine."* The remaining 20%, *"what harm can it do?"*. But if *"the remaining 20% are full of bugs"*, then the other 80% are useless.

However, all participants were in agreement that *"for criticality class A, i.e., loss of human live or catastrophic consequences, I would demand 100%"*. Also *"a high financial loss"* was seen as justification. But for criticality class B—*"i.e., loss of mission, that is an economic loss"* or for *"a smaller satellite that just orbits some place where nobody cares"*, participants agreed that *"less may be potentially fine."* One participant, however, mentioned that *"one should always target 100%. The reason is: if I aim at less than 100%, what do I leave out? How do I justify not testing something?"*

100% coverage brings in new risks All participants agree that the 100% coverage requirement introduces risks to a project. In particular, the participants saw a risk to the schedule, i.e., that they might *"lock jaws in some problem and let the project slip out of control already at its beginning."* A development team needs to be aware of and have a strategy to cope with this risk. Moreover, they all agreed that a fixed and high coverage ratio imposes a financial risk as it is difficult to say in advance what and how many *"hard nuts"* are in the project. If developers *"postponed the difficult things"*, the real difficulties will start at some point. This point may be somewhere between 80% and 95% (so-called Pareto Principle[6]) and the 100% coverage requirement is likely beyond this point. In the studied project, two developers required two years to achieve 100% coverage for about 25,000 lines of code. A customer demanding this should know that *"it will cost a lot. It is going to be expensive"*.

Don't optimize for the 100%-metric On the one hand, a clear point was made: *"100%: it sounds really good. But I think those who demand it, do not know what they are asking for."* On the other hand, participants mentioned several risks of setting a 100%-requirement. Therefore, it is necessary that all stakeholders understand the implications of such a requirement. As mentioned before, the higher the coverage, the more

[6] This is also called the "Pareto principle"; according to Joseph M. Juran who proposed the 80/20-rule, which roughly says that the first 80% are easy to achieve while the remaining 20% are not.

expensive. The question is, however, whether this relationship is linear. The project curve in Figure 3, which is extracted from the project performance data, is fairly linear and just bends at about 90%. The remaining 10%, however, do not account for about 80% of project effort. Consequently, the *Pareto principle* only applies very roughly here. This is in contrast to a quick analysis we did in preparation of the test coverage project. On the basis of test coverage data from a randomly chosen open source project, we found that to 'organically grown' unit tests (i.e., without using coverage metrics) the Pareto principle seems to fit better. While more rigorous verification of this observation is needed, it seems that the use of metrics effected the relationship between effort for developing tests and coverage. The use of metrics seems to be responsible for the linear relationship for most of the project duration. While using coverage measurements during the project made good (linear) progress possible, optimizing for a metric might have hidden downsides (see Section 5).

In the same vain as 'standard' discussions on metrics, 100% coverage is also just a metric, and focusing too much on it could mean that one *"loses track of the actual goal; which is to increase the quality."* Instead, testers might know best where to find bugs and how to use their *"available resources so that he will find all errors."* A misunderstood metric can create a sense of false security: *"I just want to say that I think it is bad to say: Now we have 80% unit tests, now it's fine."* A high coverage is not a guarantee of good quality.

Develop a proper strategy to maximize coverage Monitoring using metrics is important and a prerequisite to achieve a high coverage. However, aiming at high coverage also requires an appropriate development approach. When production code is developed, testing must already be planned to avoid problems that the participants faced in the project: *"Testability is a goal that one actually has to code into the code. It does not come automatically, along the way, or for free. It is a goal that one must prescribe."* Furthermore, when setting a test coverage requirement, customers should be careful that a plain 100% coverage may be too undifferentiated: *"What can be tested very well with unit tests is business logic* [e.g., a PI controller[7]] *because it abstracts from the hardware and the operating system, and because it can be reused. Here unit tests make a lot of sense"* and are *"economically reasonable"*. *"The hardware* [...] *and the operating system, and all the things at those lower layers are hard to test, and require a lot of effort, and they are not really what unit tests are intended for. You leave these things out, and the resulting percentage is what is economically reasonable."* So, if there is *"10% hardware-specific stuff,* [...] *90% are good tests."*

This includes that a test strategy has to pay attention to the different system parts. Hence, the participants also argued for considering a combination of different V&V techniques: unit tests are not an end in itself, but should be considered in the scope of the whole V&V ecosystem, where they complement each other. For example, *"reading from or writing to a register,* [...] *these are things* [...] *on a different level. They will*

[7] A proportional-integral (PI) controller is a control loop feedback mechanism that continuously computes the difference between an expected and an actual value for a variable (e.g., temperature, electrical current flow, angles, ...) and applies a correction based on proportional, integral, and derivative terms.

certainly be caught by integration tests," and *"there were integration tests that covered large areas"*, so unit tests do not find many errors (see Section 4.1). Instead, *"unit tests were done to ensure certification of the software."* One may also consider the metric results of other static and dynamic *"verification techniques that have a notion of coverage"*. In this regard, the role of unit tests was also critically discussed by the participants: test-driven development leads to a high coverage (*"we certainly would have had 90%"*) but does not lead to 100%. It is *"not relevant whether or not you really reached 100%, but that interfaces were covered"*.

Eventually, even though a strategy needs to be in place to achieve a high coverage, our participants ended up with a fairly pragmatic approach: *Do easy things first*. One participant noted that *"it is normal: first one does the things that can be easily done."* It allows the team *"to get into a decent flow,* [...] *to carve out some lead initially, to be able to crack the hard nuts at the end."* If, at some later point, the project should *"slip out of control"*, it could be easier for negotiations if good progress has been made so far. Nonetheless—also regarding the 'special' setup of the studied project—the findings from the interview, again, confirm the saying: *You can't test quality into a product*. It has to be built in right from the start. The participants noted that unit testing cannot *"be done after the code freeze* [...when] *not a single bit is allowed to be changed."* Testing *"has to happen in parallel, or* [... as] *test after coding.* [...] *It all depends on when one starts with the tests."* An extreme target value of 100% must be set early on and reflected in the quality assurance approach, or, otherwise, it may cause serious trouble.

100% coverage is not a sufficient condition for good quality The participants concluded that *"100% coverage is not a sufficient condition for good quality."* In fact, it might *"have a slight impact on quality"* because in the *"extreme case, one can achieve 100% coverage by just 'running all code' but without doing a single test."* One just *"claims that one tested something"* but only shows that *"the functions did not crash"*. It does not necessarily mean that *"the software/the functions really do what they are supposed to."* Hence, coverage is *"a start,* [...but] *one may not forget, there is also test depth. Test depth is difficult to measure."*

5 Related Work

According to Bennet and Wennberg [3], bug-fixing cost increases by magnitudes in later system lifecycle stages. In particular for space systems, however, bug-fixing cost could mean the system's cost in total, as a software failure might cause a complete system loss, e.g., as recently happened to ESA's *Schiaparelli* lander [20].

Therefore, rigorous software quality assurance as part of the overall product assurance is crucial. Hence, and as also found in our interviews, a 100%-coverage can be a reasonable requirement. However, the implications need to be considered as well, especially concerning the efficiency and effectiveness of the instrument (i.e., unit test) on the project's operation. For instance, Gokhale and Mullen empirically investigated the marginal value of increased testing [12]. In their tests, they observed an asymptotic convergence of test coverage towards 100%. The marginal coverage as a function of the number of tests decreases logarithmically, reaching almost zero at about 1,000 tests.

Approximately linear growth of coverage ends between 50% and 80% of coverage. However, Arthur Lowell stated ironically that *"20% of the code has 80% of the errors. Find them, fix them!"* [2]. If he is right, then just a few percent of uncovered code might still contain many (critical) errors (see also [5]). And, eventually, Mockus et al. [16] found that, on the one hand, cost increases dramatically if achieving higher coverage rates, but on the other hand, reduction of field issues increases linearly. They conclude that, for most projects, (economically) optimal coverage rates are below 100%. However, it has to be mentioned that, to the best of our knowledge, test coverage and its economic implications to the space domain has not yet been investigated in detail as most of the papers listed above are concerned with 'normal' software-intensive systems. For instance, although Mockus et al. [16] might be right, in the space domain, even one 'field issue' might lead to a complete system loss. Referring to the *Schiaparelli* lander [20], there is no bug-fixing strategy; the probe is just gone.

Practitioners also have to be careful to not be trapped in 'chasing the rabbit'. In particular, Marick [14] describes the misuse of coverage metrics, e.g., in the problematic different perception of developers, managers, and product testers, and their respective constraints and requirements. He makes a clear statement that coverage (tools) should enhance thought, not replace it. On the other hand, Martin [15] demands a high coverage to be a goal of any professional development team. Yet, he is often criticized for this opinion, since people argue that a high coverage does not necessarily lead to meaningful tests. In this regard, the *2016 Software Testing Technology Report* by Vector Software [21] makes a strong statement that one of the most misunderstood issues with code coverage is its relevance to software quality. Authors conclude that a 100% code coverage should not be the goal of software testing, rather than the result of complete testing—a statement that we also found in our interviews.

Regarding the strategy to achieve a high coverage, our study revealed a fairly pragmatic approach. This comes as no surprise, as recent research illustrated a significantly different perspective on software testing [11]. That is, even though using the same terminology, industry and academia quite often put emphasis on different 'things'. On the other hand, empirical evidence on particular methods/approaches is rare. For example, Fucci et al. [10] found no difference in applying test-first or test-last approaches. Only thing that counts is the granularity (and quality) of the work packages and requirements specifications. Furthermore, even though driven by standards, quite often, safety-related requirements are implemented and assured in a mixed approach. For instance, Ingibergsson et al. [13] found a discrepancy between method- and development-level implementation of standards to adhere to quality requirements in the field of autonomous robotics—providing further support for [11]. Also, our interview participants emphasized the importance of combining different testing techniques, and that an improved combination of different verification and validation approaches (including e.g., static analyses), would be wiser than a fairly 'academic' (not to say 'bureaucratic') 100% coverage requirement; maybe even more efficient.

The paper at hand thus adds to the body of knowledge by studying high test coverage ratios in the domain of software and system development for space systems. This paper adds an experience report and lessons learned from a space project and shows a still present need to study (economic) reasonableness of a 100% coverage goal.

6 Conclusion

Space systems are critical systems that require substantial quality assurance during development. If errors occur, such systems might be completely lost. However, what is substantial quality assurance? According to the ECSS standards, software for space systems shall have 100% test coverage (for criticality classes A and B). Is this a reasonable goal? In order to answer this question, we studied a project performed in which a software system's test suite was to be improved towards meeting a 100% unit test coverage goal. Eventually, the team managed to achieve >99.5% test coverage (statement and branch coverage), yet, it became obvious that the effort required to implement such a comprehensive test suite was tremendous. Therefore, we wanted to reflect on the project and we wanted to study if the 100%-goal is a reasonable one.

This paper presents the findings of an interview study performed at *Tesat Spacecom* in the final phase of the *LCT* project (Section 2). Our leading question, *"Is 100% test coverage a reasonable requirement?"* was studied from different perspectives, e.g., need for 100% coverage, break-even points, and strategies to achieve this goal. The interviews provided numerous of valuable insights from which we condensed a set of lessons learned. There is some justification for setting 100% coverage as a requirement. However, a plain 100% requirement may be too undifferentiated, and one should really understand the effects and possible alternatives (which might find possible errors more cost-efficiently). In a nutshell, our interviews resulted in the following key lessons:

- 100% coverage is unusual but achievable
- 100% coverage is sometimes necessary
- 100% coverage brings in new risks
- Don't optimize for the 100%-metric
- Develop a proper strategy to maximize coverage
- 100% coverage is not a sufficient condition for good quality

Our findings include that there seems to be a break-even point between 80% and 95%, and everything beyond this points is increasingly costly and could introduce new project risks—which confirms findings reported so far in literature (Section 5). However, the interview revealed that, still, 100% coverage can be a reasonable quality requirement; even though a 100% requirement is not a good indicator for the software quality as such. Especially for dependable systems, the decision to test less also includes a decision of what *not* to test, i.e., which parts of the system to exclude from the tests. Feedback from an author of the test coverage requirements in ECSS standard was: Only the idea that some statements may never have been exercised at all by any test should be a source of anxiety. Yet, such a rule should not be taken and applied too literally, and be discussed carefully.

Furthermore, we found the participants arguing that 100% should not become a 'formal' goal only, which leads to a situation in which just a metric is optimized. 100% coverage should always be the result of good testing but it makes few sense as a goal in itself. So how should the issue be treated on the contractual and standards level? As a customer, one wants to have 100% unit test coverage but achieving it by a formal demand (requirement) does not guarantee quality.

Moreover, the test depth and applying different V&V techniques should be considered. Nonetheless, all participants agreed that lifting the unit test coverage to 100% ex-post has to be criticized (time, effort, no options to change the software due to already performed certification). If a high coverage must be achieved in a project, the respective approach needs to be defined upfront and implemented continuously.

Finally, regarding the question whether or not the ECSS goal of 100% test coverage is reasonable: if there is only a small chance to avoid an extreme risk from materializing, it should be seized. However, when resources are limited, one has to make the decision whether effort should be spent on increasing the unit test coverage ratio, or to better put emphasis on other V&V activities. Hence, the answer to the question whether 100% is a reasonable requirement still is: “It depends”.

Limitations As stated in Section 3.4, our interview only covers one particular project, which was in the special situation that the high degree of test coverage had to be achieved ex-post. Furthermore, we only interviewed one project team. Hence, our findings are grounded in few developers’ opinions and, therefore, are hard to generalize. Also, in the project studied, a 100% coverage was not required from the beginning. That is, it remains unclear if the lessons learned would be the same if a project starts with such a requirement right from the beginning. Finally, further implications on the system as such were not in scope of this study.

Future Work As part of the future work, we plan to include the remaining interviews conducted with project support personnel into the evaluation. Furthermore, since the interviews revealed numerous interesting findings not directly aligned with the major research question, future work will put more emphasis on the other parts of the interviews. We also want to investigate links between techniques like “defensive programming” and their effect on coverage, and as a justification for not achieving 100% coverage. Finally, even though we already collected and presented some qualitative data (Section 4.1), we also plan to include more quantitative data into the study to gather further insights. We might still be able to obtain and to take into consideration some data that has high correlation with hard-to-test modules, like complexity, nesting depth, fan-in, fan-out, etc.

Acknowledgements

We thank our colleagues Karin Schmitz for transcribing the several hours of recorded interviews, and Björn Gütlich and Sabine Philipp-May for supporting our undertaking.

References

1. M. Adler. Spirit Sol 18 Anomaly. Online: `http://web.archive.org/web/20110605095126/http://www.planetary.org/blog/article/00000702`, September 2006.
2. L. J. Arthur. Quantum improvements in software system quality. *Communications of the ACM*, 40(6):46–52, June 1997.

3. Bennett, T. and Wennberg, P. Eliminating embedded software defects prior to integration test. *Quality Assurance Institute Journal*, 2006.
4. J. Bentley. Programming pearls. *Communications of the ACM*, 28(9):896–901, 1985.
5. B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, Jan. 2001.
6. ECSS-E-ST-40 Working Group. ECSS-E-ST-40C: Space engineering – Software. Standard, ECSS Secretariat, March 2009.
7. ECSS-Q-ST-30 Working Group. ECSS-Q-ST-30C: Space product assurance – Dependability. Standard, ECSS Secretariat, March 2009.
8. ECSS-Q-ST-80C Working Group. ECSS-Q-ST-80C: Space product assurance – Software product assurance. Standard, ECSS Secretariat, March 2009.
9. ESA. Sentinel online. Online: `https://sentinel.esa.int`, 2017.
10. D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo. A dissection of test-driven development: Does it really matter to test-first or to test-last? *IEEE Transactions on Software Engineering*, (in Press) 2017.
11. V. Garousi and M. Felderer. Worlds apart: a comparison of industry and academic focus areas in software testing. *IEEE Software*, (in press) 2017.
12. S. S. Gokhale and R. E. Mullen. The marginal value of increased testing: An empirical analysis using four code coverage measures. *J. Braz. Comp. Soc.*, 12(3):13–30, 2006.
13. J. T. M. Ingibergsson, U. P. Schultz, and M. Kuhrmann. On the use of safety certification practices in autonomous field robot software development: A systematic mapping study. *Lecture Notes in Comuter Science*, 9459:335–352, Dec 2015.
14. B. Marick. How to misuse code coverage. In *Proceedings of the 16th International Conference on Testing Computer Software*, pages 16–18, 1999.
15. R. C. Martin. *The Clean Coder: A Code of Conduct for Professional Programmers*. Pearson Education, 2011.
16. A. Mockus, N. Nagappan, and T. T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 291–301, Oct 2009.
17. NASA. NASA Missions: Space Station. Online: `https://www.nasa.gov/mission_pages/station/news/orbital_debris.html`, Sept 2013.
18. NBC News. German satellite crashed over asia's bay of bengal. Online: `http://www.nbcnews.com/id/45032034/ns/technology_and_science-space`, Oct 2011.
19. C. R. Prause, M. Bibus, C. Dietrich, and W. Jobi. *Managing Software Process Evolution for Spacecraft from a Customer's Perspective*, pages 137–163. Springer, 2016.
20. T. Tolker-Nielsen. EXOMARS 2016 - Schiaparelli Anomaly Inquiry. Report DG-I/2017/546/TTN, European Space Agency (ESA), May 2017.
21. Vector Software, Inc. 2016 Software Testing Technology Report. Technical report, Vector Software, Sept 2016.
22. A. Witze. Software error doomed japanese hitomi spacecraft. *Nature*, 533:18,19, May 2016.
23. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer, 2012.