Published in final edited form as:
IEEE Int Conf Softw Qual Reliab Secur. 2016 ; 3: 386–397. doi:10.1109/QRS.2016.50.

Software Reliability Analysis of NASA Space Flight Software: A Practical Experience

Harish Sukhwani[1], **Javier Alonso**[1,2], **Kishor S. Trivedi**[1], and **Issac Mcginnis**[3]

[1]Department of Electrical & Computer Engineering, Duke University, Durham, USA

[2]Research Institute of Advanced Studies on Cybersecurity, University of León, León, Spain

[3]NASA Goddard Space Flight Center, Greenbelt, MD, USA

Abstract

In this paper, we present the software reliability analysis of the flight software of a recently launched space mission. For our analysis, we use the defect reports collected during the flight software development. We find that this software was developed in multiple releases, each release spanning across all software life-cycle phases. We also find that the software releases were developed and tested for four different hardware platforms, spanning from off-the-shelf or emulation hardware to actual flight hardware. For releases that exhibit reliability growth or decay, we fit Software Reliability Growth Models (SRGM); otherwise we fit a distribution function. We find that most releases exhibit reliability growth, with Log-Logistic (NHPP) and S-Shaped (NHPP) as the best-fit SRGMs. For the releases that experience reliability decay, we investigate the causes for the same. We find that such releases were the first software releases to be tested on a new hardware platform, and hence they encountered major hardware integration issues. Also such releases seem to have been developed under time pressure in order to start testing on the new hardware platform sooner. Such releases exhibit poor reliability growth, and hence exhibit high predicted failure rate. Other problems include hardware specification changes and delivery delays from vendors. Thus, our analysis provides critical insights and inputs to the management to improve the software development process. As NASA has moved towards a product line engineering for its flight software development, software for future space missions will be developed in a similar manner and hence the analysis results for this mission can be considered as a baseline for future flight software missions.

Index Terms

Defect Reports; Flight Software; Incremental Development; Software Reliability; Software Reliability Growth Models

I. Introduction

Today’s mission-critical systems are becoming increasingly dependent on software controls for performing critical functions. One prominent example is space missions [1], [2]. Unfortunately, software failures are one of the most dominant causes of failures in today’s mission-critical systems [3], [4]. Software failures in such missions can cause mission performance degradation or even complete mission failure, incurring a heavy scientific and economical penalty.

Flight software is the software that executes on-board a spacecraft. It is an embedded real-time software system, a domain that has experienced an exponential growth, where the software size grows by an order of magnitude every ten years [1]. This has been particularly challenging in the areas of spacecraft, aircraft and automobiles. The main source of growth is the increasingly ambitious requirements and the advantages of situating new functionality in software rather than hardware [1], [5]. Another area of concern is the increasing complexity of the flight system architecture, which gives rise to increasingly complex interdependent variables that could cause a failure, thus making the testing of such systems extremely challenging.

Software faults (“bugs”) are the underlying causes of software failures. These defects can be inserted at any software life-cycle phase. In order to maximize the likelihood of mission success, the quality of software needs to be assessed and evaluated at each phase of the software life-cycle. This can prevent the propagation of defects from one phase to another. Furthermore, measuring software quality in terms of software reliability at any time of the software life-cycle will become a powerful tool for project managers to decide if it is worth moving to next phase or spend extra time and resources on current phase to guarantee specific levels of software quality.

One of the most valuable repository of information captured during the software life-cycle phases are defect (bug) reports. These defect reports are used to track defects found in any phase of the software life-cycle, including development, testing as well as mission phase. Although most organizations use defect reports solely for bug-tracking purposes [6], analysis of these reports provide insights into the software development process [7], and hence frequently used to infer the quality of software. Note that in the software life-cycle context, “defects” in a broad sense include faults, errors or any other anomaly that surfaced in the software that may or may not cause a failure. Another practical way to understand is that “defect” is a necessary change to the software [5].

Software Reliability Growth Models (SRGM) provide a set of well-developed techniques to assess reliability growth, using data collected during the testing phase. It has been used successfully to count number of faults remaining, estimate and predict reliability during the test and operational phase [5]. Although SRGMs are black-box technique, they have been successfully applied in projects at IBM, NASA, JPL, Hewlett-Packard, CISCO, US Air Force and many such organizations [8]. Since there are challenges involved in applying SRGMs to datasets collected from real-world systems, we attempt to provide a clear approach to apply SRGMs to defect reports from development stage, and also share our experience and challenges encountered.

In collaboration with NASA Goddard Space Flight Center (GSFC), we analyzed the defect reports for the on-board software of a recently launched space mission. These reports were generated during the development and testing phases (pre-launch phase) of the mission. The software development for this project spanned several years, and the software was developed in multiple releases. Each release spanned across several software life-cycle phases such as requirements analysis, design, code, test, integration and verification. We also received a document that was used as a running document for the team to share updates across the team and project managers, which was updated throughout the mission development. We were particularly interested in the details regarding the development goals of each build, the timeline for delivery of the releases, timeline for receiving the various hardware setups, and log of problems experienced during the development. Overall, this document helped us understand the defect reports better, and we attempt to use this information to interpret our analysis results.

The major contributions of this paper are as follows. First, we correlate the software reliability analysis of major releases with the activities performed and problems encountered during the development of those releases. We find that some of the major problems encountered include integration issues with new hardware platforms, issues with new releases of COTS software and problems encountered in hardware test environment. We find these reasons consistent with known observations in the flight-software development domain [1], [9]. Second, we identify important software releases to perform our analysis. Since there were 35 different releases, we first identify the major and minor releases by reviewing the development goals and timeline of each release. For the set of releases addressing the same set of requirements, we find that such releases were developed and tested in a sequence. Hence from the software reliability analysis perspective, we analyze such releases together as a set. Third, from the software reliability analysis of the major software releases, we find that the Log-Logistic (NHPP) and S-Shaped (NHPP) are the best-fit models across most of the releases. Thus, in the future, it may be sufficient to consider only a subset of SRGMs.

To the best of our knowledge, this is the first paper analyzing defects reports from a flight software during the pre-launch phase. We feel that there would be a peculiar pattern to the development of such mission-critical software for flight domain, and hence these results can be considered as a baseline for similar analysis of the data from other flight software missions. Overall, software reliability analysis provides useful insights and inputs to the management, and hence improve the process.

The paper is structured as follows. First, in Section II, we provide a summary of the defect reports, including the description of important fields, field frequency analysis of the relevant fields, and plots to aid visualize the data. Next, in Section III, we provide software engineering details specific to this mission. In Section IV, we provide our analysis approach, followed by detailed results in Section V. We discuss our results in Section VI, followed by related work in Section VII, and conclude our paper in Section VIII.

II. Summary of the Defect Reports

The software engineering team at GSFC uses a custom-made defect tracking tool (let us call it Defect Tracking Tool (DTT)). This tool is used to track defects and change requests across all the software life-cycle phases and releases. We obtained the reports for one of their recently launched mission. The reports span across more than five years, covering multiple releases of software. In this section, we provide a summary of the important fields of the dataset, along with the field frequency analysis of some of the critical fields.

The dataset includes reports filed against the flight software as well as simulator developed for the same mission. Since we are focusing on flight software development, we subset and analyze reports related to the flight software only.

We summarize the important fields that are present in the reports along with a brief description in Table I. For fields that were not relevant for this paper, we skipped them for brevity. Also for the field values that were missing, we have replaced them with the tag (`missing`). In the following figures, we provide field frequency analysis of some of the relevant fields. Such analysis is useful to obtain a high-level view of the dataset, before delving into software reliability analysis.

A. DTT Type v/s Severity

From Figure 1, we see that 63% of reports are “Change Requests”, and 33% of reports are “Defects” reports. External DTT refers to reports against external products or subsystems, and they are a tiny share of defect reports.

B. DTT Type v/s Build Found

Since these reports were collected during the pre-launch phase of the mission, the software builds signify major or a minor milestone in the development, and it is safe to assume that only the final release (build 4.7.2) of this software (and patches thereafter) was deployed in the mission. However, we can still refer to some of the major builds as “releases”, since they signify a major milestone of the project. In this paper, we use the terms “build” and “release” interchangeably.

We found that reports span across 35 different builds. Since this number is large, in Figure 2 we choose to display only the builds against which more than 5 reports were filed. We provide more details about the major software releases in Section III-A.

From Figure 2, observing the count of reports against each build, it seems that builds 1.0.0, 2.0.0a, 2.0.0b, 3.0.0, 4.0.0 and 4.1.1 correspond to major software releases. We also observe that builds up to 4.1.1 have a large share of “Change Requests”, with following builds having increasing share of “Defects”. Note that builds developed late in the mission development, such as 4.2.x and later, seem to have an unreasonably large number of “Change Requests”.

C. DTT Type v/s Phase Found

We notice that very few reports are filed for phases like “Preliminary Design”, “Unit Test”, and “System Acceptance”. Hence we merged the reports in these phases as follows: “Preliminary Design” and “Detailed Design” are merged into “Design”; “Code” and “Unit Test” are merged into “Code&Test”; “System Acceptance” and “System Validation” are merged into “System Validation”; “Spacecraft I&T”, “Customer” and “Operations & Maintenance” are merged into “Customer”. Thus all the phases are as shown in Figure 3. Note that the development process in each release passes through all the above-mentioned phases. From Figure 3, observing across all the releases, we notice that more “Change Request” reports are filed in the earlier phases (up to Build Integration) and more “Defect” reports in the later phases of software-life-cycle. This confirms the observations made in literature [1], [10] that more defects emerge as the subsystems are integrated and verified (Build Integration and Verification phases) and latter phases.

D. Timeline plot

Since the reports span across multiple releases and phases, it is useful to visualize the development time spent in each phase of each release as a two-dimensional timeline. The information regarding the calendar dates signifying the start and end dates for each release was not readily available, we decided to infer this information based on the information captured in the defect reports. In Fig. 4, we plot the *Date Created* of the defects along with the *Build Found* and *Phase Found*. Also since there are many releases, we clubbed the releases that would be considered as a logical entity together (discussed further in Section III-A), which also makes it easier to visualize. We plot only the defect reports, since we plan to use this plot in our analysis in the later sections. Since the development spans across several years, we divided the plot into three parts by the releases.

From the timeline plot, we can see the multiple phases in each build overlap each other. This means that activities corresponding to different phases like “Build Verification” and “Build Integration” are conducted in parallel, as opposed to in sequence. Also we see an overlap in activities across the consecutive software releases, but only by a few weeks. Also from the density of the defects, it seems that the development activity was busier during the builds 3.x and 4.0–4.1.x (Part II of the timeline) as compared to other releases.

Since the software is developed in multiple releases, and since the phases within each release seem to overlap in time, we can ascertain that the software is developed in an incremental life-cycle (as opposed to waterfall software life-cycle model). We are not sure if each increment followed an iterative life-cycle.

III. Mission-Specific Software Engineering Details

Flight software provides mission-level capabilities such as guidance, navigation and control (GN&C); command and data handling (C&DH); entry, descent, and landing (EDL); and instruments for science and environmental observation [1]. Flight software also consists of “infrastructure” or “platform” software that provides important services and is customized and reused across space missions. For the mission we are analyzing, flight software uses Core Flight System (cFS) system, which is a mission-independent, platform-independent flight software environment [11] developed at NASA-GSFC. It mainly consists of Core Flight Executive (cFE), which is the main foundation of the framework, along with OS abstraction layer. This allows platform-independent software to be developed for the mission and then ported to mission hardware when it is ready. Readers are welcome to review further details about cFS in [11].

Prior to performing our analysis, it is important to understand the development activities performed in each release. Also within each release, the development is divided into several phases, such as those listed in the description of the field *Phase Found* in Table I. This section summarizes our understanding.

A. Releases (Builds)

The document provided by NASA contained a summary of each build, which listed the subsystem modules that were developed, the hardware target, and the purpose of that build. Our goal of reviewing this information was to identify the major and minor builds, and also the relationship between the minor and major builds (whether the minor builds were planned or unplanned, and the reason for the same). Unfortunately much of the details provided were confidential and cannot be reproduced here. However, in Table II, we provide a summary that would be useful for the analysis conducted in this paper. Since there are many builds, we provide summary only for builds with more than 5 defects.

We observe that the hardware target evolves during the development. The first few releases are developed for the *commercial off-the-shelf* (referred to as *COTS*) hardware, usually consisting of the main platform with no peripheral hardware. In this case, it was provided by the vendor supplying the hardware platform for the main flight. The next set of releases are built for an *emulation* platform (referred to as *EM*) where new flight hardware peripheral are added as they are received from the suppliers. This platform converges closer to the final on-board hardware. Then later releases are developed for the *FlatSat* platform, which is useful for performing functional / interface tests, system-level tests and procedure validation (referred to as closed-loop testing) [12]. The last few releases are tested on the actual flight hardware. We also notice that the software release before the new build target (say 2.0.0a before EM target) consists of code to integrate the new hardware platform. Overall, our takeaway is that we need to be aware of the build target and take into account the major goals of each build, before comparing the software reliability analysis results across builds.

We also observe that in many cases, the minor builds (like 3.1.0, 3.2.0) are branched out during the development of the major build (like 3.0.0) to add features that were not completed before handing over the major build to the test team. We assume that while the testing of the major build was in progress, defects found were corrected in the minor build as well. In most of such scenarios (examples in Section V-B), we see that in releases following the release of the major build (like 3.0.0), most of the defects are filed against the minor build (like 3.1.0). For the exceptional cases, we feel that this could be a clerical error. In summary, in cases where the requirements for minor builds were subset of the requirements for the major build, we consider merging the reports for minor builds and major builds, and analyze them together as a set. Thus, the builds separated by double lines in Table II are analyzed as a set. Note that we skip the analysis for releases 2.1.0 and 2.2.0, since they had very less defects and those builds were created for an external test team.

B. Phases

Phase Found field in our dataset refers to the phase in which the defect is introduced. We attempted to find the set of activities that are defined for each of the phases, unfortunately no clear definitions were available either in the documents provided to us or in the NASA Software Assurance Standards document [13]. However, software assurance standards does refer to the ISO/IEC 12207-1995 standards for defining the software life-cycle processes. Hence, we referred to the latest ISO/IEC 12207-2008 standards [14] to seek a better understanding. We skip the details for brevity.

IV. Software Reliability Analysis: Our Approach

In this section, we present our approach for software reliability analysis. Our objective is to assess and evaluate the software reliability growth achieved in each of the software releases, and select the set of growth models that fits the best, from which we can estimate the number of defects remaining at the time of release, and the failure intensity curve over the entire observation period.

We now present our analysis approach. Further details about the techniques are provided in the subsequent sections.

1. Select all the defects belonging to the specific release. Follow additional guidelines from Section III-A related to merging of defects from different releases.
2. From the collection, remove the defects that were found before the official delivery date of the release to the testing team.
3. From the collection, remove the defects that were filed by an independent test team (like Independent Verification & Validation (IV&V)). Since such teams are assigned to investigate specific kind of defects [13] (e.g., defects found while performing static analysis) and since they work independently, these defects should be analyzed separately if needed.
4. Assuming time between defects as time-between-failures, extract it in units of days, and perform Trend Analysis (see Section IV–A).
   - **a.** If a significant growth or decay trend is detected, we determine the best Software Reliability Growth Model (SRGM) candidate and discuss the results (see Section IV–B).
   - **b.** If no trend is detected, it is possible to assume that the time-between-failures are independent and identical distributed. Hence, distributional analysis allows us to determine the underlying time distribution between defects determining the best distribution candidate (see Section IV–C).

A. Trend Analysis

From the software reliability perspective, trend analysis provides relevant insights about the evolution of the reliability during the software life-cycle. Note that it is necessary to establish a growth or decay trend before attempting to fit software reliability growth models [15]. We perform trend analysis using Laplace Trend Test [5], [15] since it is the most widely used of all the other trend estimation techniques [15], [16]. Moreover, the trend evolution also provides hints about the most suitable software reliability growth model candidates [5]. We perform two-sided Laplace trend test at 5% level of significance.

B. Software Reliability Growth Models

For the software releases that exhibit reliability growth, SRGMs help us understand the underlying defect correction process. We would like to see if the underlying defect correction process changes from release to release. From the best-fit SRGM model, we can estimate the number of faults remaining, and estimate and predict the reliability during the test and (with some assumptions) during operational phase. Moreover, from NASA GSFC's perspective, we would like to establish the baseline failure intensity characteristics that have been observed over several releases of this mission. This baseline information can then be used to plan resources for their future missions, and help manage their project more effectively.

We use the SMERFSˆ3 [17] tool to fit the following SRGM models: Geometric Model (GEO), Jelinski-Moranda (JM), Littlewood-Verrall linear (LVL), Musa Basic (MB), Musa Logarithmic (ML). We use SREPT [18] to fit the following finite failure NHPP SRGM models with the time to detect an individual defect having the following distributions: Weibull (WB), S-shaped (S), Log-logistic (LogL). We skipped the Littlewood-Verrall quadratic (LVQ) model from SMERFSˆ3, since it had a valid fit on very few of our datasets. Refer to [5] for the formulas of mean-value and failure-intensity functions of the SRGMs.

Note that the test intensity is assumed to be constant for the entire duration of the testing period of each release. We also assume that the operations performed during testing are representative of operations performed during the flight.

We outline the steps specific for this analysis.

1. Extract the time-between-failures datasets (same as outlined in Section IV).
2. Input this data into the SRGM tools (via text file) and obtain the parameter value point estimates for all the models. We choose Maximum Likelihood Estimate (MLE) technique for parameter estimation, since it is considered to be more robust and yields estimators with good statistical properties [19].
3. Compute the estimated mean-value function and failure-intensity function for all the models.
4. Evaluate goodness-of-fit using Mean Square Error (MSE), Mean Absolute Scaled Error (MASE), and Mean Absolute Percentage Error (MAPE) criteria.
5. Compute failure intensity at the end of the observation period $T_n(\hat{\lambda}(T_n))$.

The performance of an SRGM is judged by its ability to fit the observed fault removal data (referred as "Historical predictive validity" in IEEE Std. 2008 [20]) as well as predict satisfactorily the future behavior of the software fault detection / correction process from observed data (referred as "Future predictive accuracy" in IEEE Std. 2008) [8]. We would have preferred to perform future predictive accuracy using techniques like u-plot and prequential likelihood [5], however, for most of the builds considered in our dataset, there are no data points after the test phase of the same build. Hence, we perform goodness-of-fit tests using historical predictive validity approach.

Using the historical predictive validity approach, we perform goodness-of-fit using three techniques: *mean squared error (MSE)*, *Mean Absolute Percentage Error (MAPE)* and *Mean Absolute Scaled Error (MASE)*. Since MSE is sensitive to outliers, it is useful to consider the other two criteria [21], [22]. We rank the models for every dataset using these three criteria, and use the median rank to determine the best model. In cases where median-ranks clash, we give preference to models that are observed to fit well on our datasets, viz. S-Shaped (NHPP), Log-Logistic (NHPP) and Musa-Basic.

As an example, we provide analysis results for Build 4.0.× – 4.1.× in Table III. Thus we see that the top three best-fit models are S-Shaped (NHPP), Log-Logistic (NHPP) and Weibull (NHPP). We provide the best-fit models for all releases in Table V.

C. Distributional Analysis

For the datasets for which we cannot establish a trend, we assume all the samples of “Time between Failure (TBF)” are from independent and identically distributed (i.i.d.) random variables, in order to identify an underlying distribution. We consider the three popular distributions, viz. Exponential, Weibull and Gamma, since they are widely used and applicable to time-between-failure datasets in software reliability applications. We again use Maximum Likelihood Estimation (MLE) technique to determine the point-estimate of the parameter values. The density function for the distributions that we used for our analysis are as follows:

$$\text{Exponential Distn.: } f_X(t)=\lambda e^{-\lambda t}, \;\; t \geq 0$$

$$\text{Weibull Distn.: } f_X(t)=\lambda \alpha t^{\alpha-1} e^{-\lambda t^{\alpha}}, \;\; \alpha>0, \; t \geq 0$$

$$\text{Gamma Distn.: } f_X(t)=\frac{\lambda^{\alpha} t^{\alpha-1} e^{-\lambda t}}{\Gamma(\alpha)}, \;\; \alpha>0, \; t \geq 0$$

where α is a shape parameter and λ is the scale parameter.

We use Kolmogorov-Smirnov (KS) test as a goodness-of-fit test. It is distribution free, in the sense that the critical values do not depend on the specific distribution being tested [23]. Also KS test is recommended for continuous distributions. We perform KS test at 5% level of significance.

Based on the KS test criteria, if multiple distributions are fitting the random sample, we use Akaike information criterion (AIC) to determine the best candidate among distributions [24]. AIC balances the value attained by the max. log-likelihood value with number of model parameters. The model with lowest AIC can be chosen as the best model. However, since the AIC values are too close, it is wiser to choose a simpler distribution with lesser number of parameters.

$$AIC = 2 \cdot ln\,(\text{maximum likelihood}) + 2 \cdot (\text{number of model parameters})\,;$$

V. Software Reliability Analysis : Results

From the timeline plot (Figure 4), we notice that the defects found in the same release overlap in time across life-cycle phases such as “Code&Test”, “Build Integration”. From the discussion in Section III, although we would expect that the different phases of each release work in a water-fall like development model, that does not seem to be the case. Hence, we analyze defects across all the life-cycle phases in a single release. Also, as discussed in Section III-A, we have grouped the releases that had the same requirements to start with. In order to satisfy a critical assumption that software reliability growth models are applied only during phases where defects are found and no new substantial code is added, we consider only the defects filed after the date the build is released to the testing team.

In this section, we present the results obtained from the software reliability analysis. First, we summarize our results (Subsection V-A). Then, we peek into the analysis of the specific releases to seek a deeper assessment of our results (Subsection V-B).

A. Summary of the Results

Following the approach presented in Section IV, we first perform trend analysis on the time-between-failures on the major releases and present our results in Table IV. We notice that most releases exhibit reliability “Growth” trend. Some releases exhibit “Decay” trend and we provide insights into the possible reasons / causes in Section V-B.

Next, for the releases which exhibit reliability growth or delay, we fit software reliability growth models. Otherwise we fit distribution (renewal) models. We summarize our findings in Table V. We present the top three best-fit models along with the estimated failure intensity (FI) at the end of the observation period T_n. $(\text{denoted as } \hat{\lambda}(T_n))$ The model parameters for the SRGM models are consistent with the expressions found in [5]. We also plot the estimated failure intensity curves for the best-fit model for all the builds in Figure 5.

Thus we notice that among the SRGMs, Log-Logistic (NHPP) and S-Shaped (NHPP) feature among the best-fit models across most of the releases. Also since the estimated FI values are fairly consistent across the top three models, we could consider fitting only a subset of models for future analysis.

B. Analysis of specific releases

In this section, we provide the trend plot and timeline plot for the subset of releases that did not achieve a “growth” trend and summarize our findings from the document provided by NASA. Note that the vertical dotted line in the timeline plots indicate the official start of the testing period for the respective build.

1) Release 1.0.0—The timeline and trend plot are provided in Figure 6. We see that a lot of build integration activity was performed at the end of cycle and hardly any build verification activity was performed after that, thus this release achieved no growth. This correlates with a steady climb in the trend plot. From the records, it seems that around the time of mass defects, the test team was rushing towards the deadline for handing over Build 2.0.0a to the test team (see Figure 4a).

2) Release 2.3.x—The timeline and trend plot are provided in Figure 7. From the records, Build 2.3.0 was missing one critical software component, which was essential for the FlatSat hardware testing for Release 3.0 onwards. Once the component was ready, it was integrated in Build 2.3.1 along with the fixes from Build 2.3.0, and rest of testing proceeds. Also around mid-November, the team received a new revision of the Interface Control Document (ICD) for one of the critical subsystem, which needed to be accommodated before the Build 3.0.0 development. Such modifications delayed the testing of Build 2.3.1, thus resulting in burst of defects starting December. Similar to Release 1.0.0, this release involved a lot of Build Integration activity, presumably for the upcoming FlatSat testing. Thus this release fails to achieve reliability growth as well.

3) Release 3.x—The timeline and trend plot are provided in Figure 8. We note that Build 3.0 is a large build, covering most of the cFS application. However, it runs into issues early on, mainly due to a missed cFE release deadline (cFE is developed by a separate team). It was ready to be incorporated in Build 3.1, however due to major changes involved, it was eventually integrated for Build 4.0.

Several technical issues encountered during this release. The first issue is a software defect in one of the components, causing the watchdog timer to perform power-on reset instead of processor reset. This component was sourced from an external vendor. So a code work-around was applied, till the vendor released a new version of the code, which was integrated in Build 4.0. Another issue was a hardware problem in the FPGA of one of the communication devices on the FlatSat. This was also fixed before releasing Build 4.0 to the test team.

We find that the Builds 3.1, 3.2 and 3.3 were created along the way, to address missing requirements and testing from the previous releases. We see a progressively decreasing defect count with each build. Also from the document and the defect reports, we find that large number of components are addressed in this release. Although the document lists several issues, we found it hard to correlate with the trend plot, and thus we are unable to explain the peaks observed in the trend plot. We also tried excluding the first one or two months of data points and then perform trend analysis, but we still find that the release exhibits no growth trend. Overall, due to constant changes in the build throughout the testing, the build failed to achieve growth. Also since many critical pieces (particularly the new cFE release) were pushed to Release 4.0, the testing team switched to the new release right away.

4) Release 2.0.0a—The timeline and trend plot are provided in Figure 9. This build mainly focuses on the integration of interfaces for C&DH, GN&C, to prepare for the next release that will be tested on the emulation hardware. We are unable to explain the spike of defects in around mid-March. From the timeline and trend plot, the activity seems well spread throughout the phases. It is trending towards reliability growth eventually, but the testing activity switched to a new release right after.

VI. Discussion

In this section, we discuss implications of performing software reliability analysis on flight software systems. We also discuss some of the problems specific to flight software development that affected the software reliability growth, and whether software reliability analysis helped spot such problems.

A. Results obtained from Software Reliability analysis

We summarize some of the pertinent results obtained from performing the software reliability analysis. From the trend analysis results, we obtain a first-hand idea of whether the build achieved a growth. In addition, the Laplace trend plot helps us spot any potential issues occurring during the development (see examples in Section V-B). Then we attempt to fit SRGMs, which provides us with the parameters for estimated failure intensity $(\hat{\lambda}(t))$ and mean value $(\hat{\mu}(t))$ functions. We then plot the failure intensity trend experienced across the testing phase, which provides insights into the testing process. If a similar failure intensity trend is expected in a future project, this plot can help us estimate the staffing requirements across the life-cycle [25]. From our interaction with NASA-GSFC and from some of the staffing related issues mentioned in the records, performing such analysis on a baseline mission could be a valuable asset for the future missions. This could help correlate activities performed in each release with the Laplace trend observed and the nature of the best-fit growth models. Unfortunately, we could not obtain any records regarding the staffing during the testing process, so we could not perform any further analysis. SRGM model parameters for finite-failure models also provide the estimated number of defects remaining at the end of testing period.

Overall, since different releases have different goals, it is hard to assess the reliability of the overall mission by modeling reliability growth in each release. Nonetheless, software reliability analysis for each release provides pertinent insights that could be useful for managing the mission development.

B. Problems encountered during flight software development

While investigating reasons for lack of growth achieved in specific software releases, we came across variety of qualitative issues that caused problems during the software development. Our observations are consistent with those highlighted in the literature [1], [26], [9], [2], [10]. In this subsection, we summarize some of our observations and discuss whether software reliability analysis techniques were helpful to spot such issues.

1) Integration Issues—One of the most common integration issue was missing or delayed hardware components on a test setup or final flight hardware. Since these hardware components are usually procured from external agencies, delays occur in selecting the right vendor and/or agreeing upon the requirements. This delays the contractors' delivery schedule. Also if there is a mismatch between the expected hardware specification and the actual hardware delivered, then flight software code needs to be reworked to accommodate this. Another form of integration issue was shifting the test process across different hardware setups over the course of the development. In order to support the new hardware platform, new integration code was added in the releases prior to moving onto new hardware platform (e.g., Releases 2.0.0a, 2.3.x in our case). We find that such releases usually fail to achieve reliability growth.

2) Hardware test environment issues—From the event logs and list of resources allocated to each build development, it seems that the team ran into a few issues related to managing the test environment. One issue was the delay in procuring an EM setup from another mission project, thus resulting in testing delays. Another issue was a limited number of hardware setups available for COTS testing. In one case, the test setup's computer started malfunctioning, resulting in teammates queuing up on other setups.

3) External Software—This mission was built on top of a flight software platform (called Core Flight Executive (cFE)). Note that three different releases of cFE were used during the mission development. One of the issues encountered was a change in the code directory structure of a new cFE release (coinciding with Release 2.0.0a), resulting in restructuring the flight software code and thus delaying the new release. Another issue was the delay in an awaited cFE release, resulting in postponing the testing with new cFE release for the next flight software release. Overall, issues related to using COTS software for flight software has been highlighted in the literature [9], [10] as well. However, the efforts of developing a reusable flight software system like cFE has been regarded a step in the right direction [1], [27], and we need to evolve our development process to accommodate this better.

C. Threats to validity

For our research, we were provided the defect reports for the mission and provided a document to aid our understanding of the reports. The results of our analysis are subjected to the validity of the information collected in the defect reports, whether the fields have been populated properly by the team. One of the most important fields for our analysis is the *Build Found* field. As a sanity check, we verified that the points close-by are also classified with the same Build Found. Also, we found only one defect report that had *Created Date* way beyond the defects from the same Build, and hence we discarded it as a typo error. Another important field was the *Created Date*. We did not find any reports missing this field. Another important field was the *Title*, which carried a keyword indicating the defects filed by a separate test team (like IV&V, V&V). We exclude such defects from our analysis. The information from rest of the fields would not influence the analysis results directly. For the fields that were missing any value, we replaced them with (`missing`). Regarding our analysis, the steps and results were first cross-verified by authors based at Duke University, and then the results were shared with collaborators at NASA-Goddard for final validation.

External validity of our results are limited by the domain of interest (flight software) and by the organization (NASA-GSFC) [26]. Just like flight software, software for other mission-critical systems like aircraft or even drones run on complex embedded systems and are developed in an incremental manner, and hence we conjecture that the future studies will help generalize the conclusions to similar systems. One of the best examples would be the future flight software missions developed at NASA-GSFC using the cFE/cFS system.

D. Product line approach in Flight Software

Every space mission is unique in terms of mission requirements. Traditionally flight software has been developed in a “clone and own” manner, where new mission flight software was ported from a previous successful mission’s flight software, and adopted with new hardware platform. However, this resulted in laborious, error prone and excessive software requirements and design changes. Since 2005, the Flight Software systems branch at NASA Goddard took a product line approach to FSW development and developed a Core Flight Software (cFS) System [11]. This integrates with Core Flight Executive (cFE) and OS abstraction layer and enables teams to write software independent of the underlying system hardware and OS. This platform has already been reused in several flight software missions supported by GSFC, and has significantly reduced the cost and effort of developing such missions. This is the future of flight software development for years to come.

Although the mission we analyzed is developed using cFS system, we still find issues as listed in Section VI-B. Thus mission management should continue improving the processes to ensure good software reliability characteristics across all the releases.

VII. Related Work

Although flight software domain has been at the forefront of the software reliability research, there are few empirical studies performing software reliability analysis for flight software defects found during the development phase.

Recent empirical studies in the area of flight software anomalies has been conducted by Duke University in collaboration with NASA - Jet Propulsion Laboratory (JPL) [28], [22]. In this study, flight software defects from 18 space missions has been analyzed for their fault type characteristics. The first paper [28] identifies the relationships between fault type and characteristics like failure effects, and failure risk. It also studies the fault type proportions across the mission duration. For a subset of 8 space missions, the second paper [22] analyzes the time-between-failure characteristics of the failures observed, by first performing trend analysis and then fitting either SRGMs or distribution fn. depending on the trend analysis results. Also the analysis was conducted separately for different fault types, and results were interpreted accordingly. The analytical techniques developed and used in [22] form the seeds of the analysis of our paper. However, note that the above two papers analyze software defects that have been found during the operational phase of the mission, whereas in our paper, we analyze reports from the development phase of a mission. Also we analyze software developed in multiple increments, which means we perform reliability analysis at every major milestone of the development.

Anomalies found during operational phase of NASA-JPL missions are analyzed by two other research groups as well. The first group [26] analyzed safety-critical anomalies from seven deep-space missions using Orthogonal Defect Classification (ODC) [7] techniques. The second group [9] analyzes anomalies found in three long-life spacecraft missions using timeline of anomalies observed and frequency of various fields. In addition to flight software, these two papers analyzed anomalies across ground software, flight procedures, and ground procedures as well, thus providing a wider view of the challenges encountered during the mission operation from software reliability perspective. However, these papers provide limited view of the nature of defects and challenges encountered during the development phase, and also they do not perform any statistical defect modeling analysis.

Authors in [29] analyze anomalies found in four mission-critical software systems used by European Space projects. In this research, 240 anomaly reports found during both development and operation phase were analyzed using ODC. However, the authors found the projects were developed in waterfall manner, hence was not developed in multiple releases. Also they do not provide any relevant details, such as the hardware platforms used for testing.

VIII. Conclusion and Future Work

We performed software reliability analysis of the defect reports from the development life-cycle of a flight software. We observed that software for this mission has been developed across multiple releases, spanning 35 build versions. After detailed investigation, we found that a set of major and minor builds that were created to address same set of requirements can be analyzed as a single software release, and thus we analyzed 9 software releases of this mission. We find that many of the releases exhibit either growth or no trend. For the releases that experience decay, we perform causal analysis and find that the main cause of problems were integration issues, issues with COTS and hardware test environment. These observations are consistent with those in the literature. Since the flight software development has taken a product line approach, it is clear that the future GSFC missions will be developed in a similar pattern, and hence we feel that these results can be used as a baseline for future flight software missions. We also provide clear guidelines on performing such analysis on similar mission-critical software application, and also share the challenges we experienced. Overall, software reliability analysis provides techniques to model the defect trend and estimate the number of defects remaining. However, it provides limited insights into the day-to-day challenges encountered during testing. As an ongoing investigation, we are currently classifying defect reports using Orthogonal Defect classification (ODC) [7], [26] and fault triggers classification [30]. Also, there have been attempts in the recent literature [31] to design software reliability modeling techniques for software developed in an incremental and iterative life-cycle. We understand that it is possible that such models would provide better fit for our dataset. As a future work, we plan to compare their performance with the traditional SRGMs.

Acknowledgments

This research was supported by the NASA NSSC grant no. NNX14AL90G. This research was also supported in part by IBM under a faculty award and an IBM student fellowship. This work was supported by the Spanish National Institute of Cybersecurity (INCIBE) according to the Rule 19 of the Spanish Digital Trust plan (Spanish Digital Agenda) and University of León under contract X43.

References

1. "NASA Study on Flight Software Complexity", NASA office of Chief Engineer. Tech Rep. 2009. [Online]. Available: http://www.nasa.gov/pdf/418878main_FSWC_Final_Report.pdf
2. Madden WA, Rone KY. Design, Development, Integration: Space Shuttle Primary Flight Software System. ACM Comm. Sep.1984 27(9):914–925.
3. Leveson NG. Role of Software in Spacecraft Accidents. Journal of Spacecraft and Rockets. 2004; 41(4):564–575.
4. Foreman V, Favaro F, Saleh J. Analysis of Software Contributions to Military Aviation and Drone Mishaps. Reliability and Maintainability Symp (RAMS). Jan.2014 :1–6.
5. Lyu M. Handbook of Software Reliability Engineering IEEE Computer Society Press McGraw Hill. 1996
6. Li J, Stålhane T, Conradi R, Kristiansen J. Enhancing Defect Tracking Systems to Facilitate Software Quality Improvement. IEEE Software. Mar; 2012 29(2):59–66.
7. Chillarege R, Bhandari I, Chaar J, Halliday M, Moebus D, Ray B, Wong M-Y. Orthogonal Defect Classification - A Concept for In-Process Measurements. IEEE Trans on Soft Eng. Nov; 1992 18(11):943–956.
8. Musa, J. Software Reliability Engineering : More Reliable Software, Faster and Cheaper. 2nd. AuthorHouse; 2004.
9. Green NW, Hoffman AR, Garrett HB. Anomaly Trends for Long-Life Robotic Spacecraft. Journal of Spacecraft and Rockets. Jan-Feb;2006 43(1):218–224.
10. Malcom H, Utterback HK. Flight Software in the Space Department: A Look at the Past and a View Toward the Future. John Hopkins APL Technical Digest. 1999; 20(4):522–532.
11. Aeronautics, N., S. A. (NASA). Core Flight System. 2015. [Online] Available: https://cfs.gsfc.nasa.gov/
12. Michael Wright, NA., S. A. (NASA). NASA FlatSat. 2008. [Online] Available: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20080040717.pdf
13. NASA. May Software Assurance Standard 8739.8. 2005. [Online]. Available: http://www.hq.nasa.gov/office/codeq/doctree/87398.pdf
14. Systems and software engineering – Software life cycle processes. ISO; Geneva, Switzerland: 2008. no. ISO/IEC 12207:2008
15. Goel AL, Yang K. Software Reliability and Readiness Assessment Based on the Non-Homogeneous Poisson Process. Advances in Computers. 1997; 45:197–267.
16. Gaudoin O. Optimal Properties of the Laplace trend test for Software Reliability models. IEEE Trans on Rel. Dec; 1992 41(4):525–532.
17. Farr, W. SMERFS Cubed. http://www.slingcode.com/smerfs/downloads/, accessed: 2016-04-01
18. Ramani S, Gokhale SS, Trivedi KS. SREPT: Software Reliability Estimation and Prediction Tool. Performance Evaluation. 2000; 39(14):37–60.
19. Pham H. Handbook of Reliability Engineering Springer. 2003
20. IEEE Recommended Practice on Software Reliability. IEEE STD 1633-2008. Jun.2008 :c1–72.
21. Hyndman RJ, Koehler AB. Another Look at Measures of Forecast Accuracy. Int Journal of Forecasting. 2006; 22(4):679–688.
22. Alonso J, Grottke M, Nikora A, Trivedi K. The Nature of the Times to Flight Software Failure during Space Missions. IEEE Int Symp on Soft Rel Eng (ISSRE). Nov.2012 :331–340.
23. NIST. Kolmogorov-Smirnov Goodness-of-fit test. http://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm, accessed: 2016-04-01
24. Akaike H. A New Look at the Statistical Model Identification. IEEE Trans on Auto Control. Dec; 1974 19(6):716–723.
25. Kan, S. Metrics and Models in Software Quality Engineering. 2nd. Addison-Wesley; 2002.
26. Lutz R, Mikulski I. Empirical Analysis of Safety-Critical Anomalies during Operations. IEEE Trans on Soft Eng. Mar; 2004 30(3):172–180.

27. Wilmot J. A Core Plug and Play Architecture for Reusable Flight Software Systems. IEEE Int Conf on Space Mission Challenges for IT. 2006

28. Grottke M, Nikora A, Trivedi K. An Empirical Investigation of Fault Types in Space Mission System Software. IEEE/IFIP Int Conf on Dependable Systems and Networks (DSN). Jun.2010 : 447–456.

29. Silva N, Vieira M. Experience Report: Orthogonal Classification of Safety Critical Issues. IEEE Int Symp on Soft Rel Eng (ISSRE). Nov.2014 :156–166.

30. Cotroneo D, Grottke M, Natella R, Pietrantuono R, Trivedi K. Fault Triggers in Open-Source Software: An Experience Report. IEEE Int Symp on Soft Rel Eng (ISSRE). Nov.2013 :178–187.

31. Fujii T, Dohi T, Fujiwara T. Towards Quantitative Software Reliability Assessment in Incremental Development Processes. 2011 33rd Int Conf on Soft Eng (ICSE). May.2011 :41–50.

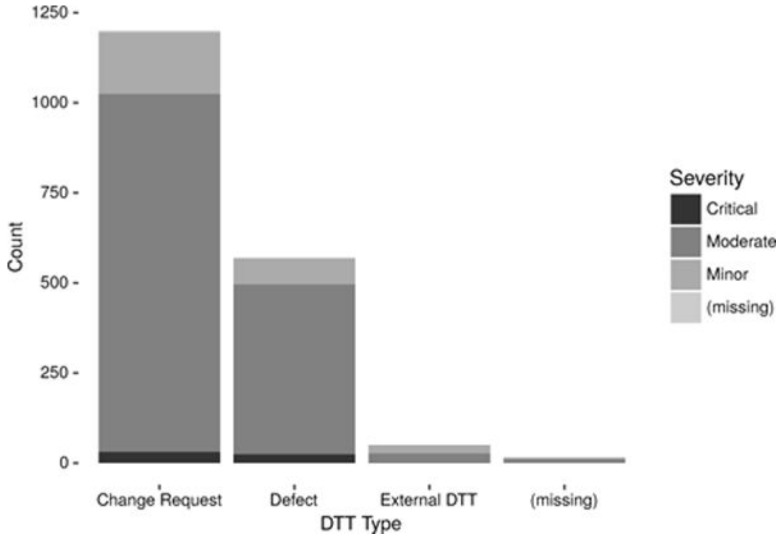


Fig. 1.

Frequencies by DTT Type and Severity

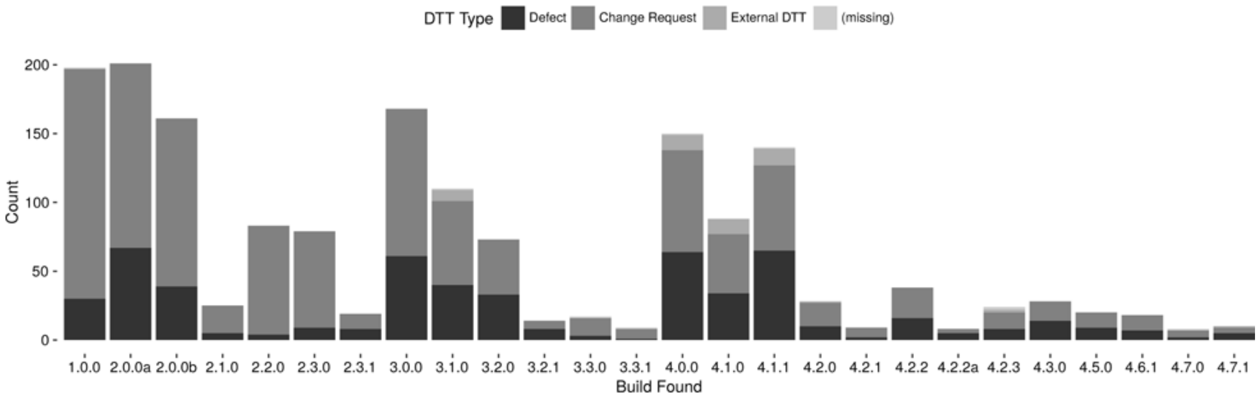


Fig. 2.

Frequencies by DTT Type and Build Found

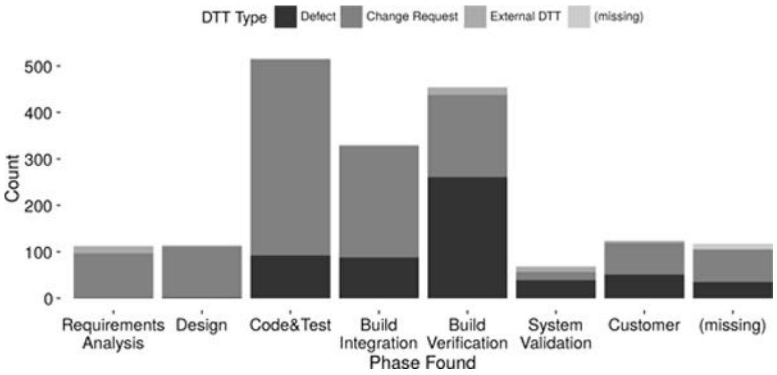
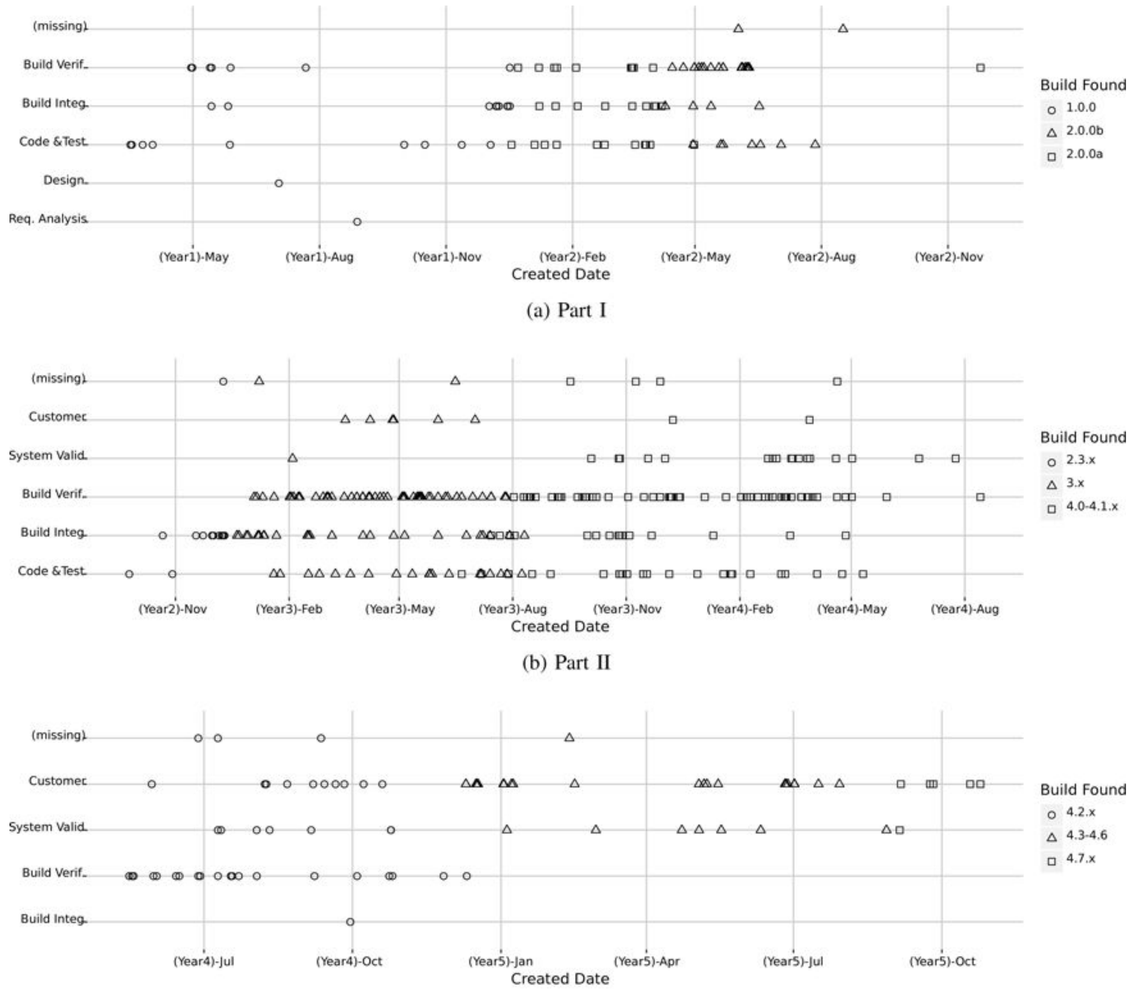


Fig. 3.

Frequencies by DTT Type and Phase Found field



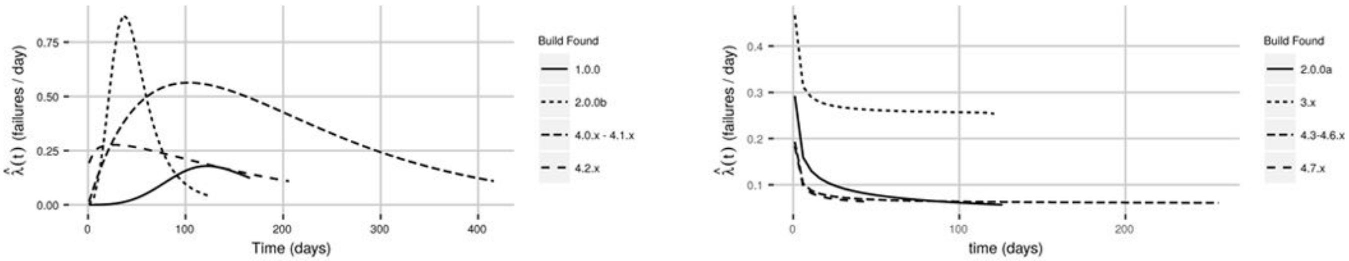
(a) Part I

(b) Part II

(c) Part III

Fig. 4.

Timeline of defects reports founds across various Builds and Phases in the dataset

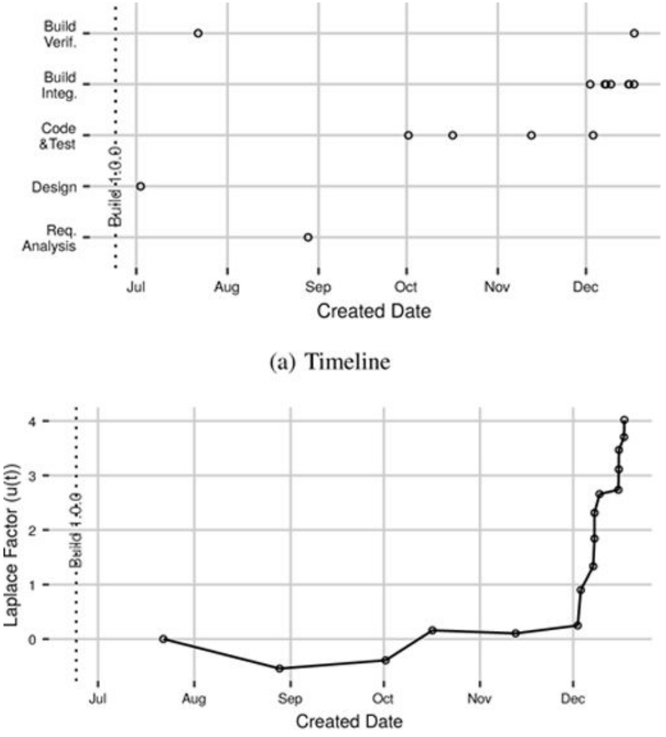


(a) Builds with SRGM fit

(b) Builds with Distribution Fn. fit

Fig. 5.

Plot of Estimated Failure Intensity (FI) for all Builds

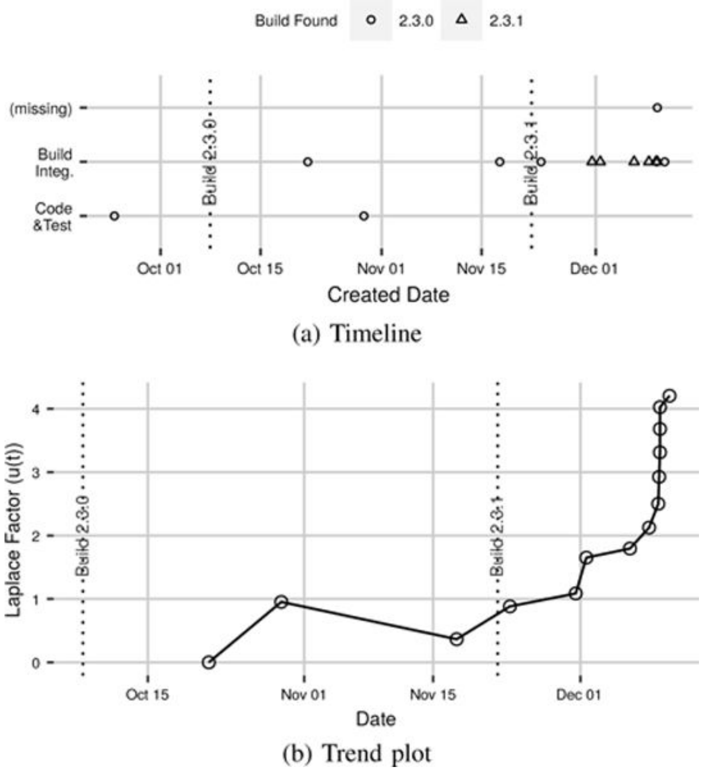


(a) Timeline

(b) Trend plot

Fig. 6.

Release 1.0.0

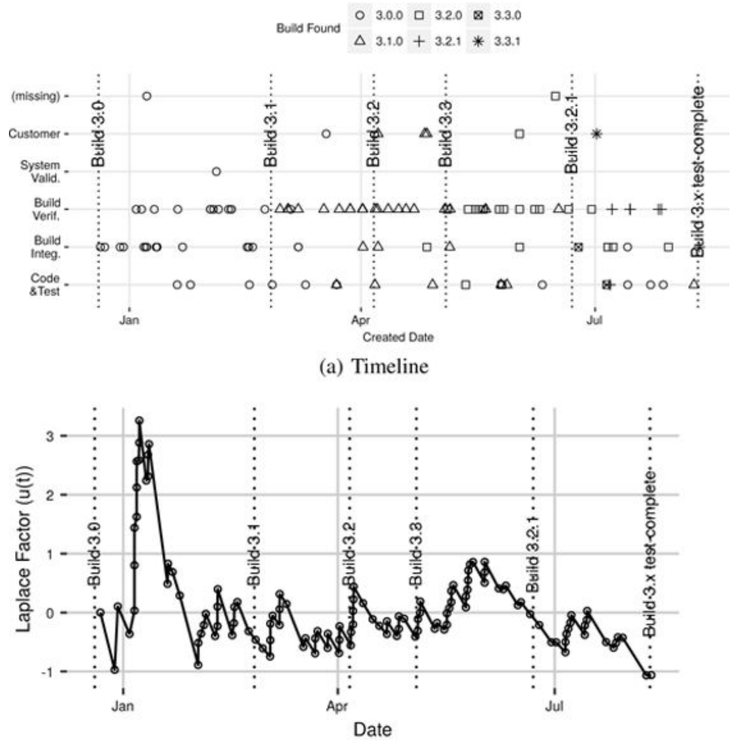


(a) Timeline

(b) Trend plot

Fig. 7.

Release 2.3.x

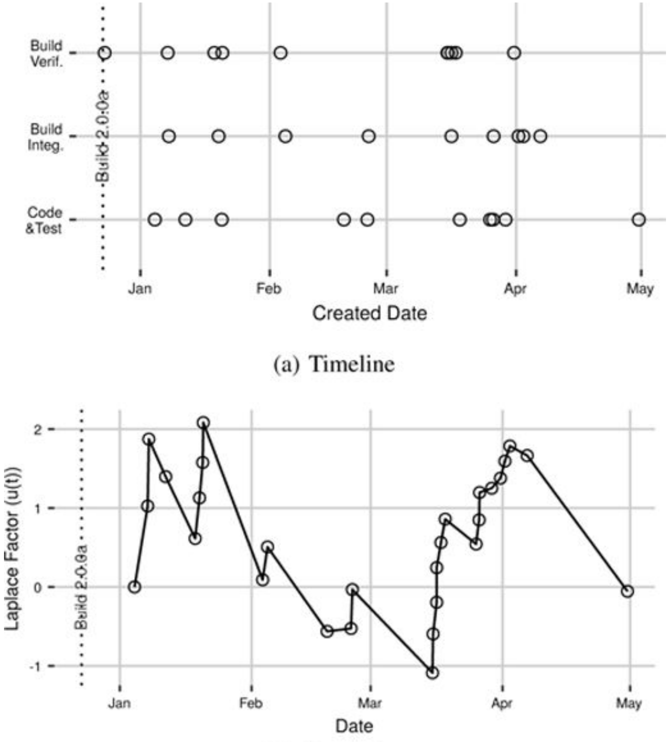


(a) Timeline

(b) Trend plot

Fig. 8.

Release 3.x



(a) Timeline

(b) Trend plot

Fig. 9.
Release 2.0.0a

TABLE I

Description of Important Fields In The DTT

Field Name	Description
DTT ID	A sequential identification no. that uniquely identifies a DTT report
Type	Type of report, whether it is defect report or change request. A defect report is used to formally document an error in configuration-controlled product. A change request report is used to formally document a request to change a configuration-controlled product.
Title & Description	Explanation of defect or change request report
Status	Current status of the report. Possible options are “Submitted”, “Assigned”, “In Work”, “Work Completed”, “Build Integration”, “In Test”, “Test Completed” “Ready for Closure”, “Closed”, “Closed with Defect, “On Hold”, or “Rejected”.
Subsystem	Subsystem corresponding to the report
Phase Found	Software life-cycle phase in which the defect is found / change report is filed. Possible options are “Requirements Analysis”, “Preliminary Design”, “Detailed Design”, “Code”, “Unit Test”, “Build Integration”, “Build Verification”, “System Acceptance”, “System Validation”, “Spacecraft I&T”, “Customer”, “Operations & Maintenance”. Collated into “Requirements Analysis”, “Design”, “Code&Test”, “Build Integration”, “Build Verification”, “System Validation”, “Customer”
Build Found	The software build in which the defect is found / change report is filed
Severity	Severity based on the originator’s assessment of the impact on the FSW-related activities. Possible options are “Critical”, “Moderate”, “Minor”.
Date Created	Date when the report is submitted
Date Assigned	Date when a software developer is assigned responsibility to investigate and resolve the report.
Date Work Completed	Date when the DTT solution and related work is completed by the assigned.
Date Closed	Date when the DTT is closed.

TABLE II

Summary of Builds Strategy

Build (Re-lease) Name	Major Goals	Build Target (Hardware)
1.0.0	Infrastructure build to setup the build process and code repository. Inherit flight software (FSW) infrastructure code from a previous mission and integrate with the flight software applications currently developed	COTS
2.0.0a	Implement / test most GN&C interfaces, also functional C&DH interfaces for smooth integration of FSW for Emulation (EM) boards	COTS
2.0.0b	Further development of C&DH, GN&C modules	EM
2.1.0	Build for Engineering Test Unit (ETU) team. Fix defects and missing requirements from 2.0.0b	EM
2.2.0	Further development of C&DH. Fix defects and missing requirements from 2.1.0	EM
2.3.0	Build for FlatSat testing. Fix defects and missing requirements from 2.x builds	EM
3.0.0	Complete all cFS applications	FlatSat
3.1.0, 3.2.0	Fix defects in release 3.0.0	FlatSat
4.0.0	Meet all flight software requirements and start Integration & Test. Deliver final cFS applications and configurations	FlatSat
4.1.0, 4.1.1	Complete cFS applications, and other tasks that were not completed by release 4.0.0	FlatSat
4.2.x	Release for regression testing. Also cleanup before end-to-end testing	FlatSat
4.3.0 – 4.6.x	Set of cleanup builds during end-to-end testing	FlatSat, Flight Hardware
4.7.0, 4.7.1	Set of cleanup builds during pre-ship	Flight Hardware

TABLE III

Summary of Srgm Analysis For Build 4.0.X – 4.1.X

Model	Criteria Value			Criteria Rank			Median Rank
	RMSE	MASE	MAPE	RMSE	MASE	MAPE	
GEO	11.47	9.90	0.40	4	4	5	4
JM	14.74	12.86	0.61	5	5	6	5
LVL	288.16	287.40	10.02	8	8	8	8
MB	16.49	13.94	0.39	6	6	4	6
ML	20.05	16.89	0.38	7	7	3	7
WB	9.25	7.56	0.37	3	3	7	3
S	4.80	3.92	0.123	1	1	2	1
LogL	7.99	5.77	0.11	2	2	1	2

TABLE IV

Laplace Trend Test Results for Software Releases

Release	Sample Size	Laplace factor	Reliability Trend
1.0.0	16	4.0202	Decay
2.0.0a	28	−0.0551	No Trend
2.0.0b	43	−3.103	Growth
2.3.x	14	4.204	Decay
3.x	120	−1.0613	No Trend
4.0.x – 4.1.x	144	−3.6373	Growth
4.2.x	42	−1.9817	Growth
4.3-4.6.x	34	−1.6726	No Trend
4.7.x	6	0.2799	No Trend

TABLE V

Summary of Best-Fitting Models for Software Releases

		Model Rank			$\hat{\lambda}(T_n)$ (failures/day)			Model Parameters (round to 3 digits)		
Build Analysis	**Analysis**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**
1.0.0	SRGM	LogL (NHPP)	Weibull (NHPP)	–	0.1205	0.1042	–	α=23.485, λ=0.007, k = 3.969	N = 168.838, α = 1.15, β = 3e-4	–
2.0.0a	Distn.	Weibull	Gamma	–	0.0564	0.1214	–	λ=0.442, α=0.661	λ=0.118, α=0.54	–
2.0.0b	SRGM	LogL (NHPP)	S (NHPP)	Geo	0.0347	0.0674	0.2203	α=44.431, λ=0.022, k=3.24	α=45.21, β=0.037	β=0.979, D=0.554
2.3.x	SRGM	(no fit)	–	–	–	–	–	–	–	–
3.x	Distn.	Gamma	Weibull	–	0.0624	Inf	–	λ=0.252, α=0.488	λ=0.793, α=0.621	–
4.0.x –4.1.x	SRGM	S (NHPP)	LogL (NHPP)	Weibull (NHPP)	0.1073	0.0561	0.1631	α=157.85, β=0.010	α=152.91, λ=0.006, k=2.805	N=182.01, α=1.15, β=0.0015
4.2.x	SRGM	ML	S (NHPP)	Weibull (NHPP)	0.1395	0.0683	0.1064	b_0=52.422, b_1=0.006	α=46.59, β=0.018	N=55.889, α=1.15, β=0.003
4.3-4.6.x	Distn.	Gamma	Weibull	–	0.022	0.061	–	λ=0.059, α=0.454	λ=0.387, α=0.582	–
4.7.x	Distn.	Gamma	Weibull	Exp.	0.0404	0.0626	0.12	λ=0.054, α=0.453	λ=0.352, α=0.584	λ=0.12