SpaceOps-2023, ID # 239

Driving efficiencies in the maintenance of spacecraft operational simulators

D. Vicente[a]*, F. Croce[b]

[a] Satellite Ground Segment *Division, EUMETSAT, Eumetsat-Allee 1, Darmstadt, Hessen, 64295 Germany, diogo.vicente@eumetsat.int*
[b] Satellite Ground Segment *Division, EUMETSAT, Eumetsat-Allee 1, Darmstadt, Hessen, 64295 Germany, francesco.croce@eumetsat.int*
* Corresponding Author

Abstract

Operational spacecraft simulators are complex software systems essential for the effective preparation/execution of spacecraft operations or in support to the verification and validation of mission control ground segments.
As observed for other software systems, the maintenance costs span throughout the product's lifetime and tend to be much higher than the development costs.
Over the last decades, different authors have worked extensively to find efficient ways to develop these systems. Unfortunately, maintenance activities have not received the same attention and therefore represent an area where efficiencies can/should be explored and exploited. Domain literature notes that the software maintenance cost drivers often relate to product complexity, processes and people expertise. The first two categories are largely covered by an assortment of methodologies and tools typically introduced during the development phase. The availability of people's expertise and its impact on maintenance costs is arguably more challenging to address. This article looks at practical ways to improve communication, observability and autonomy during the maintenance phases. The work focuses on operational satellite simulator maintenance activities, but the concepts are transferable to other domains.

Keywords: software maintenance, spacecraft operational simulators, continuous improvement, change management

Acronyms/Abbreviations

ALM	Application Lifecycle Management
API	Application Programming Interface
EPS	EUMETSAT Polar System
EPS	Electric Power System
ESA	European Space Agency
ESOC	European Space Operations Centre
EU	European Union
EUMETSAT	European Organization for the Exploitation of Meteorological Satellites
FCT	Flight Control Team
FDIR	Failure Detection Isolation and Recovery
FOP	Flight Operations Procedure
GEO	Geostationary Earth Orbit
LEO	Low Earth Orbit
LEOP	Launch and Early Operations Phase
MetOp	Meteorological Operational satellite
MMI	Man Machine Interface
MSG	Meteosat Second Generation
MTG	Meteosat Third Generation
S3	Copernicus Sentinel-3
S6	Copernicus Sentinel-6
SATSIM	Satellite Simulator
SLES	SUSE Linux Enterprise Server
SSVT	Spacecraft System Validation Test

1. Introduction

Operational spacecraft simulators are complex software systems essential for the effective preparation/execution of spacecraft operations or in support to the verification and validation of mission control ground segments. These systems provide high-fidelity models of the spacecraft platform, payloads, and modelling of the space environment and ground stations. Operational spacecraft simulators typically emulate one or more onboard processors to enable the execution of the onboard software running on the flying asset. Over the last decades, there has been a solid push to reduce the development costs of these systems. In [3], [10] and [11], the authors focus on the usage of model-based development and agile-like practices in the development phases of operational satellite simulators as a way to focus on delivering value. In [4] and [5], the authors concentrate on ensuring reuse and portability. In [6], there is a focus on implementing tooling to ease the development of these systems. ESA and its industrial partners have successfully reduced the development costs over time [16] while creating an eco-system [17] and standard "state of practice". Domain experts are relatively efficient in developing a new system by reusing components from previous missions, even if some of the reused components were implemented by competing teams. The different references show that much emphasis has been given to achieving efficiencies and synergies in the development phase. Limited attention has been given to other software lifecycle stages, such as utilization/maintenance. This seems to align with the historical trend exposed in [22], where most organisations focus on finding efficient ways for their development phases and do not tend to consider maintenance. For earth observation missions, the development phase of spacecraft operational simulators represents a brief portion of the product lifetime (2-3 years). The maintenance phase starts on product acceptance and carries on until the mission's end, which might span over decades. In [2], Koskinen notes that software maintenance costs have increased over the last decades, e.g. by the year 2000, these represented about 90% of the total cost of product ownership. Dehaghani et al. [1] and Hajrahimi [12] provide compatible figures stressing that the maintenance costs throughout the product lifetime cannot be disregarded.

In [1], Dehaghani et al. identify 32 factors driving the cost of software maintenance. The publication demonstrates that maintenance costs are not only affected by inherent product characteristics such as the complexity or documentation availability but are often due to people (expertise of the teams and users), processes, infrastructure etc. In a narrower and more empirical study, Hajrahimi [12] showed that the factors impacting maintenance costs tend to vary across organisations but often relate to the team's expertise, processes and tooling support. The authors in [1] and [12] identify the list of cost drivers and leave the practical suggestions for implementation open. Often prescriptive methodologies addressing process improvement are compiled in bodies of knowledge such as [18], [19] or [20]. As raised by Dehaghani et al. l [1] and Hajrahimi [12], the processes represent a subset of the factors driving the maintenance cost.

This paper brings attention to the software maintenance area. It presents work carried out at EUMETSAT to optimise and streamline the maintenance of spacecraft operational simulators for different missions to reduce costs. The work is performed using lean principles such as the focus on waste reduction and aims at bringing the DevOps mindset to the maintenance state of practice (e.g. focus on increasing velocity). The learnings from this paper are considered generic enough to be transferable to other application domains, i.e. outside the satellite operational simulator area.

This article is organised into five sections. With section 1, the reader should be able to understand the objectives of the work done and the positioning of the paper concerning current literature. Section 2 provides the reader with context to understand the work described in later sections. Section 3 describes the work carried out. Section 4 analyses the results and discusses the findings. Finally, section 5 presents the high-level conclusions.

2. Operational Spacecraft Simulators

2.1 Purpose, features and functions

Operational spacecraft simulators are used at EUMETSAT throughout the mission lifetime and support three main use cases: training of flight control teams (e.g. SSVT preparation, rehearsals, contingency training), support to operational activities (e.g. procedure development, validation, anomaly investigation, rehearsals) and support to ground segment development and maintenance (e.g. use in support to end-to-end testing, performance testing, support subsystem delivery acceptance etc.).

These systems provide high-fidelity models of the spacecraft platform, payloads, modelling of the space environment and ground stations. The fidelity of the simulator should be such that operators should not be able to distinguish if a task is being carried out against the simulator or using the flying asset. This means that representativeness is one of the most relevant characteristics of these systems. Operators interact with the simulator using the mission control

system. As such, the telecommand and telemetry responses are highly scrutinised by users and therefore tend to have high fidelity. Typically, onboard processors are emulated and allow the execution of one or more onboard applications executing on the spacecraft (e.g. main computer and instruments). The operational spacecraft simulators in use at EUMETSAT provide the users with the ability to script scenarios which are often reused in support of recurring activities e.g. verifying a delivery and dry run contingency activities through the injection of failures. Figure 1 presents the typical simulator usage where flight control teams respond to simulation scenarios prepared by the simulations/training officer.

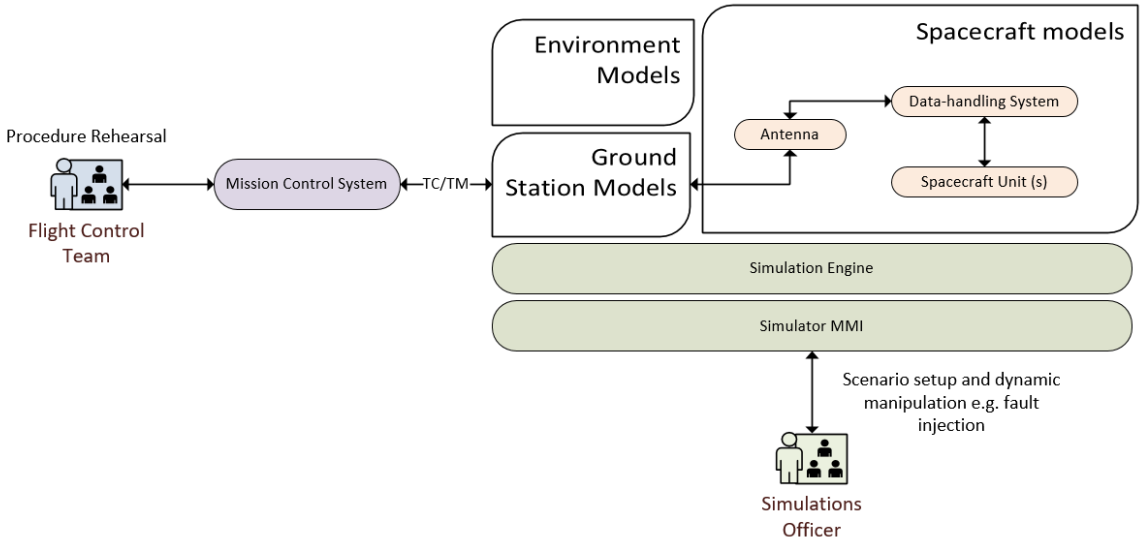

Figure 1: Operational spacecraft simulator usage example (several models and infrastructure components are not represented to increase readability)

2.2 Software Overview

Most of the current spacecraft operational simulators used at EUMETSAT are based on ESA's SIMULUS infrastructure. For this reason, the article uses SIMULUS-based simulators as an example. SIMULUS is ESOC's generic simulation infrastructure and provides a framework for developing and using operational spacecraft simulators. SIMULUS has matured over the last decades and provides developers with a set of products that allow bootstrapping the development of a new system while promoting/enabling reuse. The most relevant packages are listed below for context:

- GENM: Generic Models, a set of base models that can be (re)used to develop the different spacecraft models;
- REFA: Reference Architecture provides a reference implementation for the most common spacecraft subsystems;
- EMU: Software emulators, e.g. ERC32, Leon2,3
- UMF: Unified modelling framework an all-encompassing integrated development environment for ECSS-SMP [21] based simulators (UML integration, code generation, code merging, debugging, etc.);
- SIMSAT: Simulation engine and MMI (including script engine allowing dynamic manipulation of the simulation).

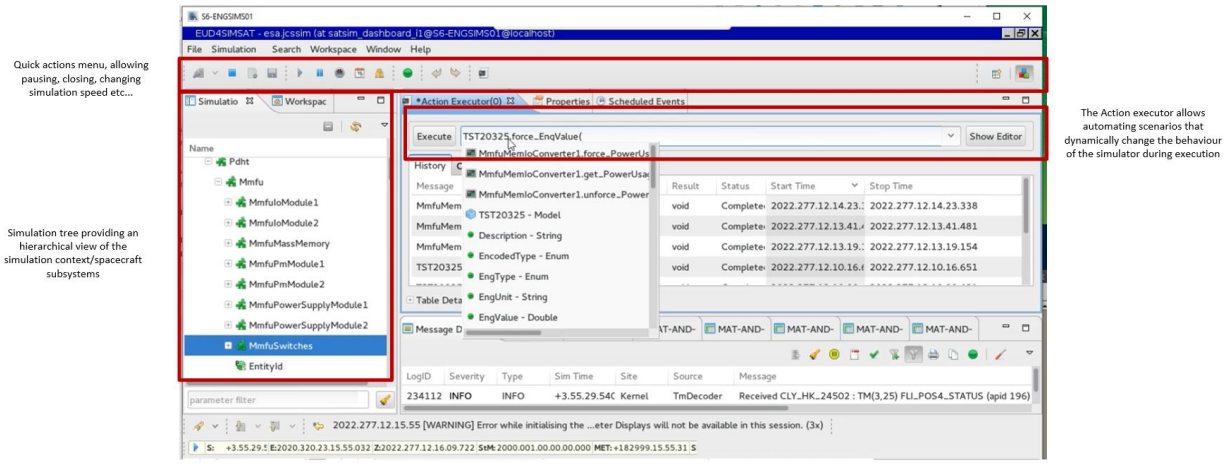

Figure 2: SIMULUS-based simulator - Sentinel-6 Satellite

As depicted in Figure 3, the mission-specific code of a SIMULUS-based simulator is mainly written in C/C++. JavaScript is used for testing and to implement scripts that control the simulator's behaviour during execution. The infrastructure has a much larger code base than the mission-specific code (over two million lines of code versus half a million lines of code, respectively). Java is used for the user interface. Although this is not depicted in Figure 3, a portion of the infrastructure code is written in Fortran, e.g. covering position and environment models. For both infrastructure and mission-specific code, there is a set of supporting scripts written in python and bash to carry out system-level tasks, e.g. automate some procedures such as trigger builds, clean logs, etc.

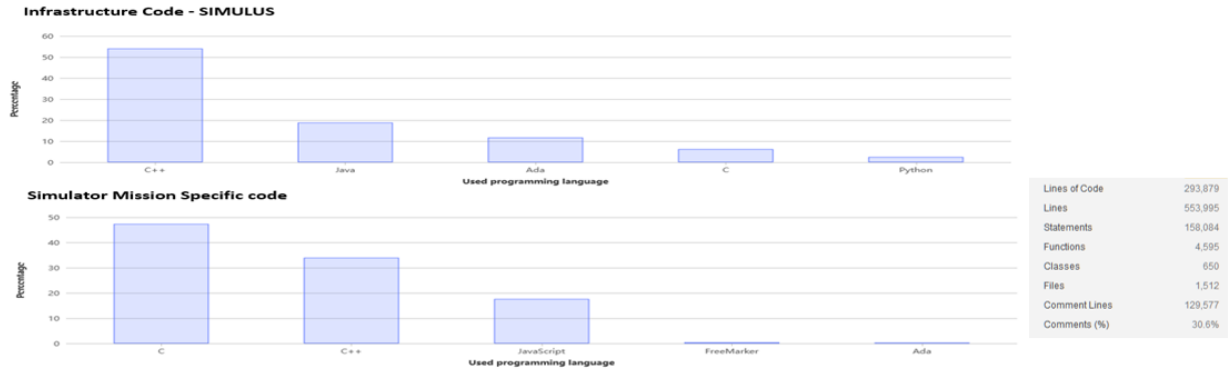

Figure 3: Typical programming languages on a SIMULUS-based operational simulator.

The SIMULUS infrastructure continues to evolve [17], and its currently moving to its next generation that aims at tackling simulation needs for missions launching after 2025. The work used simulators based on SIMULUS versions now considered legacy, e.g. SIMULUS 5.x, 6.x. These systems have a dependency on operating systems that can be considered obsolete (e.g. SLES 11 and 12). This fact, imposed constraints on the work developed where part of the efficiencies implemented relied on bringing novel tools and practices to legacy environments.

2.3 Infrastructure overview

Currently, at EUMETSAT, operational simulators are made available to different users in computing nodes that are dedicated to each mission. The different environments were implemented during the mission preparation phase as part of the ground segment implementation. The fact that the ground segments were implemented at different moments in time and sometimes by different teams has led to differences with respect to the implementation of the different production targets. Some simulators, such as for Sentinel-3 or Sentinel-6 missions, still run bare metal, while others such as the MSG simulator, run on type 1 hypervisors (bare-metal hypervisor).

The satellite simulator computing nodes sit within engineering environments dedicated to specific activities for each mission. For instance, Figure 4 depicts a SATSIM computing node dedicated to engineering purposes. This node is typically used to develop new simulation scenarios before rolling them out to other nodes with stricter configuration management policies. Figure 4 also depicts that multiple SATSIM instances are normally available within an environment as this is needed to support activities occurring in parallel or in simulation scenarios where multiple satellites need to be instantiated, e.g. for comparing behaviours across flying assets. This fact exposes the need to ensure good configuration management across simulator instances and across environments, e.g. scripts, simulator breakpoints (simulation state sets) and other configuration items that need to be managed and made available consistently across simulator instances, versions and environments.

Figure 4: Status of Sentinel-6 satellite computing node in the ENG environment

At any point, users can run previous versions of the satellite simulator to support ongoing activities. This is useful when one needs to replicate a previously used setup, e.g. anomaly investigation on a specific product version, or to compare the behaviour of a particular FOP across different simulator versions as part of SSVT preparatory activities.

2.4 Maintenance overview

The maintenance of satellite operational simulators at EUMETSAT is aligned with the definitions for software maintenance present in [22], [23], where maintenance is defined as having the objective of modifying an existing software product while preserving its integrity. Maintenance activities are framed around the concepts of corrective and non-corrective maintenance. Corrective maintenance ensures product integrity with respect to the applicable requirements baseline (i.e. fault correction/bug fixing). Non-corrective maintenance represents an umbrella term to capture three additional maintenance categories: preventive, perfective and adaptive. Preventive maintenance refers to a proactive response enabling capturing and pre-empting the occurrence of faults and failures during production/utilisation. Perfective maintenance responds to evolving requirements, i.e. extends or adapts the feature set available in the product. Like corrective maintenance, adaptive maintenance covers reactive activities ensuring the product can be used over time (e.g. upgrade to a new operating system).
This classification allows for a better understanding of the structure of the maintenance costs and informs decision-making while preparing the product roadmaps.
Maintenance requests are typically raised by the users while operating the applications in the production targets and are attended to based on their priority. Operational needs drive priorities, e.g. fixing a software problem blocking the validation of a procedure to be used on an upcoming SSVT tends to take priority over fixing other engineering/maintenance issues. The maintenance engineer responsible for a specific operational simulator is typically a domain specialist (e.g. SIMULUS specialist) that ideally should be able to capture the full depth of any issue raised. In practice, the technical depth required to understand most issues is often so significant that each request represents a massive learning opportunity. Apparent simple issues such as: "the application won't start" might require a deep understanding of the spacecraft architecture and behaviour, the FCT context, mission-specific and infrastructure design, configuration and code bases. For this reason, software maintenance requests are usually supported by multidisciplinary engineering teams.

3. Continuous improvement in the maintenance of operational spacecraft simulators

In [12] and [22], the authors concur that the factors driving the software maintenance costs are often intrinsic to each organisation. This can be easily accepted if one considers the vast literature correlating organisational structure with

organisational performance or the focus on processes as a way to achieve efficiencies in many publications [18], [19], [20] and [22].
Lodgaarda et al. [25] suggest that 2/3 of continuous improvement initiatives fail and that top managers, middle managers and workers tend to justify these results with different arguments. The article also suggests that continuous improvement initiatives achieve benefits in the short run but fade away as time passes. The challenge of ensuring that an initiative sticks is also captured in change models such as Kotter's 8-step model [24].
The continuous improvement initiatives carried out as part of this work followed Kolbs [26] experimental learning cycle and are expressed using the same approach, i.e. each improvement increment includes four components: concrete experience, observation, reflective analysis, abstract conceptualisation/experimentation.

3.1 Addressing context switching

Concrete experience/observation: EUMETSAT uses several SIMULUS-based operational simulators that are inherently similar. Unfortunately, a lot of effort is consumed in context recovery due to differences at the implementation level (not necessarily linked to mission-specific aspects). Typical examples are:

- Differences in the approach towards building the application: Although the simulators used in this work are all based in SIMULUS and consequently rely on a specific build system, i.e. maven, gnu auto-tools, there are substantial differences in the approach to building the application that imposes a technical depth that needs to be maintained throughout the project. The Sentinel-3 simulator built relies on about 70 bash scripts (and does not rely on maven) to build all applications, the sentinel-6 simulator uses four python scripts (and relies on maven) and the MSG simulator uses seven bash scripts (and does not rely on maven). It is easy to accept/recognise that the different scripts will enclose many options that deliver many features and, although undoubtedly useful, increase the maintainability effort.
- Differences in the approach towards the deployment/configuration of the application: As complex systems, operational simulators offer a lot of opportunities for customisation and extensions, the outcome is that the implementation diverges, and consequently, maintenance teams need to keep in mind the approach for configuring the different missions. A simple example would be the fact that both S6 and MSG satellite simulators rely on binary based deployments while S3 and MTG do not.
- Differences in the approach for testing the application: MSG and Sentinel-6 use cpp unit for unit and integration testing and javascript for system testing, while Sentinel-3 executes all tests using javascript (at system level). This imposes a very different level of complexity on the tests delivered, e.g. S3 tends to be much more complex. The test organisation is also typically different across missions. For example, S6 splits system tests up to unit level, e.g. battery, while MSG stays at the subsystem level, i.e. Electrical Power System. This creates some barriers to finding information due to the different approaches/complexity levels.

Reflective analysis: The discussion among the maintenance team identified knowledge management as the centre of the issues faced above. The differences between the systems can be easily identified and managed by a domain expert with many years of experience. Unfortunately, passing this tacit knowledge to other teams or even to engineers with different levels of experience becomes very challenging.

Abstract conceptualisation/Active experimentation: The analysis led to the implementation of a learning cycle focused on reducing the communication effort within the maintenance team on topics related to the build and deployment of the different simulators. The first suggestion proposed enforcing the pipeline stages for the different products. This was considered insufficient, as it would lead to the need to police the product pipelines to ensure that these were kept aligned over time and this was not considered a value-adding activity. A more audacious suggestion proposed defining a domain-specific language to enforce the business logic driving the pipelines. The idea aimed to drive the pipeline's behaviour through software. This way, the team could achieve scalability in this area because it would start managing all products at once instead of having mission-specific discussions. In practical terms, this would mean implementing a set of Jenkins libraries and defining the pipelines based on these Jenkins libraries. Each library would encapsulate the complexity of one tool used in the build system. For example, the GitLab Jenkins library would allow retrieving a product by tag, branch, commit, etc. The OpenStack Jenkins library would allow the spawning of a new virtual machine by specifying the resources needed and the base image/snapshot. The so-called ALM Jenkins library would use the lower-level libs, i.e. GitLab and OpenStack libs, to enforce the build steps independently from the mission specificities. The team agreed that the following pipelines were required:

- Product Continuous Integration (PCI): This represented the traditional continuous integration pipeline that builds a product from its sources.
- Product Delta Nightly Build (PDNB): This was a special pipeline for particular products where the changes on top of a specific product needed to be visible in all phases of the software application lifecycle.
- Facility Continuous Integration (FCI): An operational simulator comprises a set of products. This pipeline allowed building all products contributing from the operational simulator by triggering a single pipeline.
- Facility Release (FR): This allowed releasing all products in an operational simulator.

Figure 5 provides an example of the achieved outcome demonstrating the product continuous integration pipeline for the sentinel-6 simulator. In this case, the pipeline starts by spinning a new virtual machine with the base operating system applicable for this system and installing all packages required for build purposes. Once this is achieved, the products are built and the application is tested. Finally, the virtual machine is destroyed such that resources are released and can be used by other teams.
The approach discussed above transforms the implementation of a pipeline for a new product into a configuration job where one needs to define the appropriate parameters.

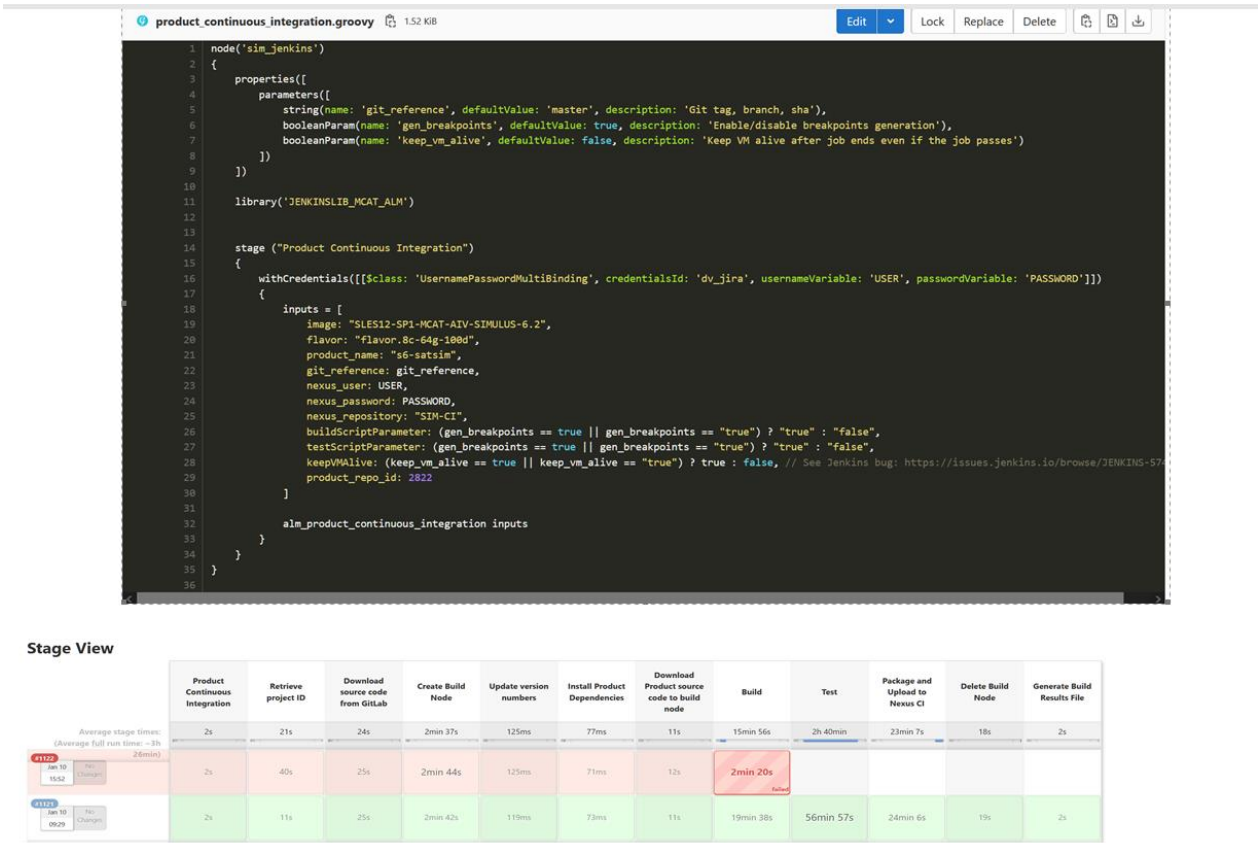

Figure 5: Pipelines are driven by software

Experience outcome/improvement for the next learning cycle: The experiment above took a leap of faith; it was not fully clear if the outcome would bring the desired value or if it was even fully feasible. The team was pleased to verify that it was possible and, more importantly, that it was much more useful than anticipated. For instance, extending the pipeline to include creating and deleting new virtual machines to verify the scratch installation process only required implementing a couple of calls in the applicable Jenkins library (in practice, these would be calls to the tool API). The following was observed:

- The build and deployment of all simulators were managed centrally using the Jenkins libraries;
- The semantics defined was adopted by the team i.e. engineers started to discuss maintenance activities using the taxonomy defined.

Since all systems were now being managed centrally, the following learning cycles focused on providing the team with observability of the activities' status. Figure 6 presents a daily email digest that the team started to receive with a summary of the latest execution for all builds. The generation of this digest was implemented within the Jenkins libraries and compiled information by querying the tools used by the build systems.

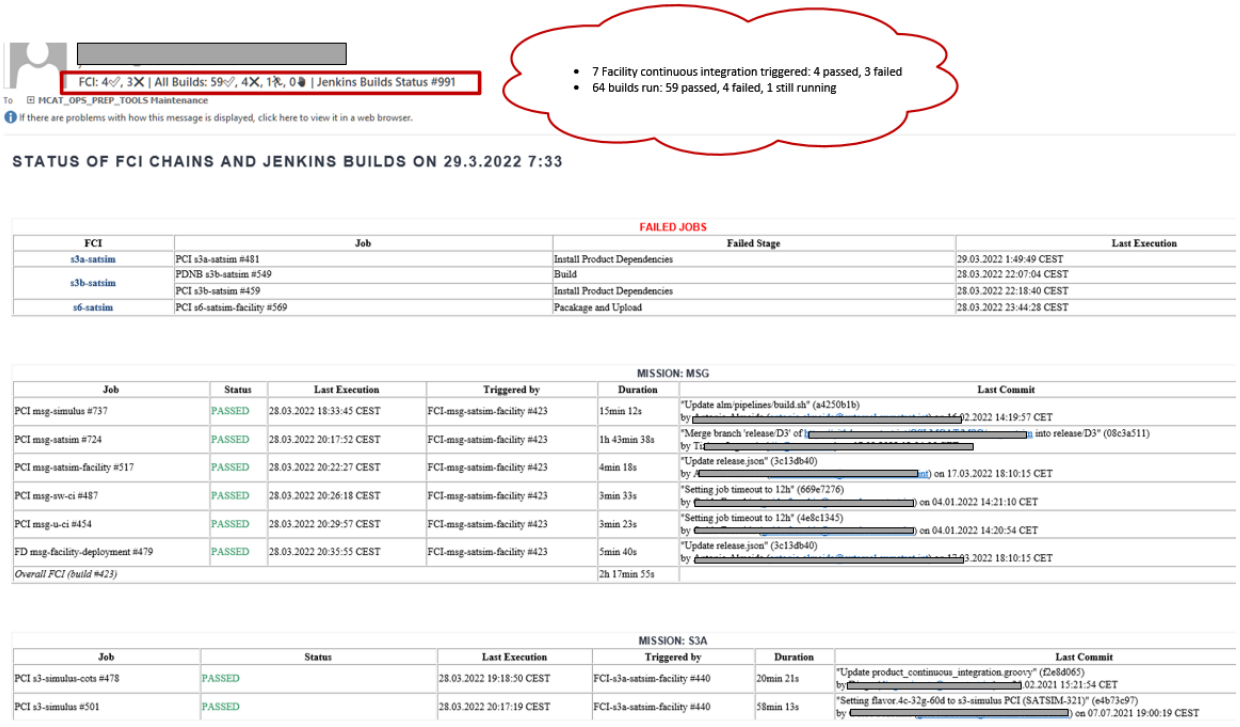

Figure 6: Continuous integration daily digest

This initiative provided the team with a lot of value and is still in place. One lesson the team could have anticipated was the technology lock-in trap. The Jenkins eco-system was capturing the team, which would mean that moving to a different continuous integration system would require rework. The solution was to migrate the libraries to a tool-independent implementation, e.g. python libs.

3.2 Reducing the number of requests for software support

Concrete experience/observation: As previously noted, operational satellite simulators are used for different purposes by diverse teams, which generate many user requests that could/should be avoided (see Table 1).

Reflective analysis: The discussion among the team identified the top 3 factors driving this type of request were:

- The fact that teams using these systems are often specialised in other engineering areas, e.g. spacecraft operations, verification and validation, etc. and hold uneven knowledge on the usage/configuration of operational simulators;

- The operational simulator's user interface is not very intuitive. It caters for an extensive set of user needs (developers, simulator officers, flight control teams) and, as a consequence, becomes somewhat convoluted;
- Most features the users need are provided but often require a complex setup, creating barriers to usage.

Table 1 lists user requests raised in the past that tend to re-occur with every new mission and therefore should be considered as hotspots for improvement at the applications level to reduce the impact on the maintenance costs in the form of user support requests.

Table 1: Examples of user requests

Issue raised by user	Actual issue
The ground stations do not start	The previous user forgot to close the ground station models
I cannot connect the mission control system to the simulator	There was another mission control system connected to the simulator
The MMI does not open	Instructions informed the user to connect to the simulator machine with "ssh –X user@ip"; the user used a lowercase "x" which prevented the X11 forwarding.
The "command" I wrote in the action executor gives an error	The step to load the simulator scripts was skipped leading to this outcome
How can I share my breakpoints	Simulator Infrastructure does not provide this feature
How can I start the simulator in permanent visibility with the spacecraft	Valid request, this feature is used so often that it should be more readily available to the users
How can I replay and orbit file produced by flight dynamics in support of an end-to-end test	Valid request. The setup is relatively complex and should be simplified.

Abstract conceptualisation/Active experimentation: The evidence collected identified the implementation of a learning cycle focused on enabling the users by empowering them with mechanisms that explained the behaviours experienced. This exercise started by identifying the areas where the users presented more difficulties. A common issue was "starting and configuring the simulator", which implies a set of steps:

1. Log in to the simulator server
2. Check if there are simulator instances free to be used;
3. Starting the Simulation Daemon and MMI;
4. Selecting the spacecraft to use;
5. Load the simulation breakpoint (state set defining the simulation starting point);
6. Load the scripts to control the simulation;
7. Execute command to start the ground stations.

Although a versed software engineer would grasp the steps and the rationale for most of these steps, engineers focused in other areas would likely expect that starting a simulator would be as simple as starting a computer game. To address this issue, the maintenance team implemented a draft version of the tool presented in Figure 6 that enabled users to start a simulation by selecting the simulator version and the target conditions (i.e. ground station setup, initial conditions – breakpoint). In addition, the tool provided the user with the status of a specific simulator server in terms of available simulator instances and users working on the server.

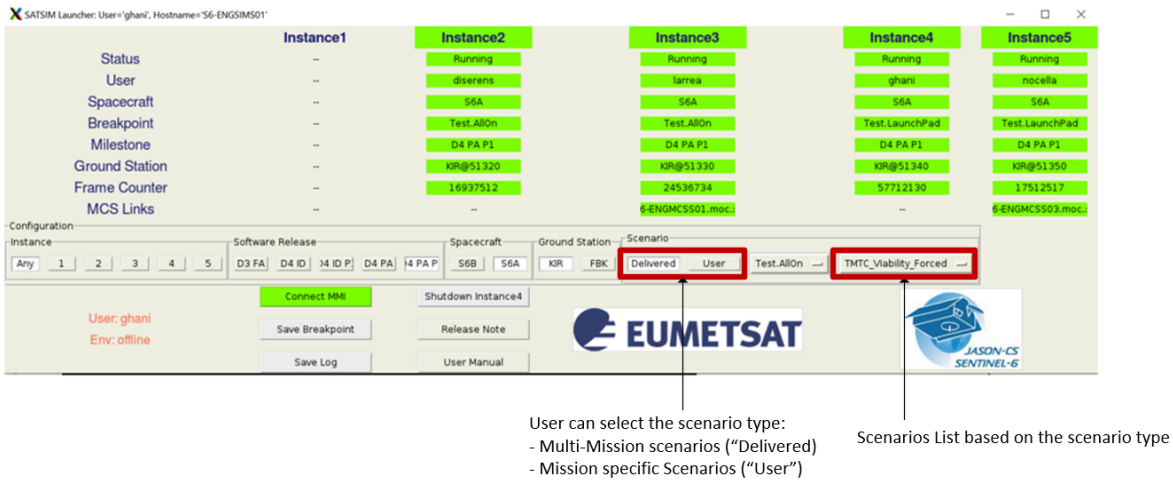

Figure 7: Earlier version of the SATSIM dashboard

Experience outcome/improvement for the next learning cycle: The result was highly encouraging as we could immediately perceive that the users were adopting and requesting additional features. Since the aim was to reduce the maintenance efforts, the focus was kept on simple features with visible value, i.e. no effort was injected into designing something very elaborate. This in practice meant using whatever software packages we had within the applicable baseline where the simulators were hosted (SLES11, SLES12). This was sufficient for resolving a large set of issues but soon imposed technical and user interface limitations (as demonstrated by the crowed MMI in Figure 7). As part of the follow-up cycles, a couple of features were very well received by the different teams:

- Delivering simulation scenarios on request "as-a-service": This concept focused allowing the teams to extract more value from the simulation while reducing the turn-around time from preparation to delivery to users. The preparation of scenarios is a complex activity because it requires a good understanding of the simulation infrastructure and the simulated context. Often this is addressed by having dedicated simulation officer that maintains a large set of scripts and configuration data. Some of the scenarios needed are mission specific others reflect needs observed across multiple missions. The feature introduced relied on the principle that users (FCT, Maintenance teams, IVV teams, etc.) would roughly specify the simulation needs in terms of conditions, dynamic behaviour and collaborate/iterate with simulator maintenance specialists in the implementation and fine tuning. The scenario could then easily be used by the different teams via the SATSIM dashboard (see Figure 7). This simple feature, led the teams to start communicating using simulation scenarios. Where in the past, one could find long written procedures to bring the simulator to a specific state, now, teams could simply state: "start simulator version 1.2.3 scenario X". In practice, this trivial approach enabled simplifying communication within and across teams.
  The feature also allowed the creation of a library of scenarios that could be used for the different missions and extend the default features delivered with the simulator as part of the development phase.
- Simplifying the sharing breakpoints (state sets) across the team: In the infrastructure used, the default approach was copying files around, which often led to additional calls for support due to the corruption of the default breakpoints. This was now made available in via the dashboard, where users could more easily share simulator outputs without dependencies on other teams or concerns about "breaking the simulator".
- Run simulation in recording mode: The steps carried out by the simulator user would be recorded for later replay. This feature allowed the maintenance team to collect all the information required to reproduce issues/bugs (version, environment, configuration loaded, breakpoint in use, commands executed) without the need for user support.

This set of learning cycles created a collaborative environment where features were delivered in a lean/agile fashion with a focus was reducing waste and providing value.

4. Results and discussion

This article aims at demonstrating strategies for finding efficiencies in the maintenance of software applications. The work carried out provides examples using the maintenance of a spacecraft operational simulator that can be transferred to the maintenance of other software applications.

4.1 Addressing context switching

This change initiative included a set of learning cycles/iterations focused on reducing the effort that the maintenance teams were wasting in context switching. The approach present in 3.1 led to the following outcomes:

- Reduced complexity in discussions around maintenance tasks: A taxonomy was formalised to define and categorise the different elements of software maintenance. This increased the team's ability to dive into the issues. For instance, if the sentinel-3 build was failing, it was necessary to pass the analysis of this issue to a newcomer. The lead engineer could simply state: "the Sentinel-3 PCI is failing. The issue is in the build.sh script located in the ALM folder of the product repo; please investigate. ". The previous approach would imply transferring a large portion of the mission's specific complexity to the assigned engineer.
- Increased collaboration within the team: Since the engineers were now sharing a framework, ideas for improvement started to emerge naturally as part of the ongoing work as the team took more ownership. For example, parameterising the Jenkins pipelines with a simple text field allowing the team to specify the Tag/Branch/Commit to be used when triggering jobs brought visible velocity gains that were initially not planned.
- Imposed harmonisation: Having a set of bespoke Jenkins libraries enforcing the semantics used by the team allowed:
    - Harmonising the build approach for all products;
    - Centralised the complexity through Jenkins libraries instead of having this scattered across the different projects.
    - Reduced the effort required to bring a new system to this approach (configuration instead of development activity);
    - Massively reduce the complexity of each pipeline by centralising the handling of tools in dedicated Jenkins libraries. With the new libraries, the systems followed a single approach, reducing the tacit knowledge being maintained. For instance, uploading binaries to the repository of artefacts (NEXUS) was previously achieved using curl or maven commands, and now a standard tool was used.
- Increase in team velocity: The creation of the Jenkins libraries has massively increased the velocity of the teams as it allowed bringing improvements to the build systems of all products at once instead of via dedicated initiatives (these would naturally lead to differences and therefore contribute to the effort wasted in context switching)
- Increased observability over the application lifecycle: Before the initiative, it wasn't easy to monitor the status of the different jobs. Simple questions like: "Were all jobs stable this week?" would imply linear effort where specific action had to be taken to review the outcome of the multiple builds. After the initiative (see Figure 6), all engineers had access to an informative daily digest that exposed the failing jobs, the location of the failures (stages) and pointed to the action that introduced the issue (e.g. commit).

Even when software systems have the same starting point, e.g. SIMULUS baseline, these will likely have substantial differences at the implementation level that often impose massive context switching effort during maintenance. These differences might reflect having different development teams or the fact that teams learned from the previous project and take different implementation decisions. The simplest and perhaps traditional way to prevent inefficiencies due to context switching during maintenance is to allocate dedicated specialists for the different systems. This might not be viable for economic reasons or simply because it is not possible to source sufficient experts in a specific area/domain. To address this, organisations tend to rationalise the available expertise across the different systems making context switching unavoidable. The article acknowledges context switching as a feature of software maintenance and stresses the relevance of managing its implications. The paper demonstrates how the effort linked to context switching can be minimised by framing the maintenance tasks using context-specific semantic constructs. Once this is achieved, relevant complexity fades away, and communication becomes more streamlined. If managers empower and incentivise teams to take ownership of these processes, continuous improvement becomes a feature of maintenance work and efficiencies start emerging more naturally.

4.2 Reducing the number of requests for software support

This change initiative included a set of learning cycles/iterations focused on reducing the effort that the maintenance teams were wasting on software support requests. The approach described in 3.2 led to the following outcomes:

- Increased observability of the system status during runtime: The SIMULUS infrastructure makes available an MMI that allows the user to configure, command and observe the dynamic behaviour of the simulator. Having SIMULUS based simulators in use at EUMETSAT in several missions has demonstrated that some features are perceived as not being there (not easily accessible) or missing for effective collaboration/communication within teams. This fact has led to the introduction of the "SATSIM dashboard", an additional MMI to provide the users with simple answers and automated actions not easily accessible while using the SIMULUS MMI. For example: who is using the system? What versions do we have available on this server? Is there an MCS connected to any of these simulators? Is telemetry arriving to the ground station? What breakpoint was loaded on this running simulator? The introduction of this component has contributed to an increase in the overall velocity of all teams using the simulators because it reduced the effort required to understand and perform simple actions.
- Lowered entry usage barriers: As detailed in Table 1, a lot of effort was previously consumed in training and follow-up support request activities to explain how certain actions can be achieved in the operational simulator. By automating and simplifying the most basic use cases, users became more independent and consequently released the maintenance team to other adding-value activities.
- Scenarios as-a-service: Previously, teams often took action to implement simulation scenarios. This is a realistic endeavour for simple cases but requires a good level of infrastructure (SIMULUS) knowledge for more elaborate setups and therefore becomes very inefficient. The introduction of a feature in the SATSIM dashboard enabling the users to start pre-defined scenarios led to a collaborative approach where users expressed needs, and this would be delivered as a scenario that could be started quickly.
- Shared scenarios across missions: The teams' simulation needs are often similar. For example, usually, all teams request a scenario where the spacecraft is in permanent ground visibility. This allows engineeris to focus on spacecraft functional aspects. Introducing the concept of scenarios as-a-service allowed the maintenance team to develop a multi-mission library of scenarios that is deployed for the different missions.
- Better characterisation of issues: The introduction of the SATSIM dashboard created a simplified route that users could use to manipulate and observe the status of the simulator. This allowed introducing a mechanism to better characterise the issues found. In the past, issues raised were described in anomaly reports, and these were often incomplete or difficult to reproduce without a lot of collaboration among the teams. The introduction of a recording feature in the dashboard allowed capturing all steps and system configuration when an issue occurred, and this allowed the maintenance teams to reproduce issues easily.

Well-established domains such as mission control applications tend to have well-established practices and systems. Software suites such as SIMULUS for satellite simulators or MICONYS for mission control systems are examples of packages that include a large set of features required by space missions. Unfortunately, these packages are not very intuitive for end users, specifically because expertise on these applications is very uneven (some highly specialised, others requiring a lot of support). The work above demonstrated the benefits of introducing components tailored to the way teams work can bring. In practice, the SATSIM dashboard was implemented as a simple set of scripts that plugged into the interfaces made available by SIMULUS. Benefits were experienced by all teams, e.g. users became more expedite in using the simulators and maintenance teams became less dependent in the users to reproduce some of the anomalies report raised.

5. Conclusions

This article reinforces a message that can be found in domain literature, i.e. the maintenance costs are typically much higher than the actual costs of developing a product. The work carried out used the maintenance of satellite operational simulators to demonstrate how relevant efficiencies can be achieved through empirical bottom-up approaches. In particular, the publication describes how the EUMETSAT mission control applications maintenance team implemented two distinct change initiatives that stood the test of time and led to tangible cost reductions on the effort required to maintain these systems.

The vast majority of the literature addressing software maintenance costs acknowledges that cost drivers relate to product complexity, inefficient processes and people expertise. The body of knowledge and tooling available to address issues related to complexity is vast and quite mature, e.g. there are multiple coding standards and metrics to characterise software complexity etc. There are also many tools to try and limit the software complexity and increase maintainability. This article cites several publications that prescribe processes and methodologies that drive development and maintenance activities efficiency. The novelty of this paper is linked to the fact that it addresses the area that is arguably more difficult to capture and codify, i.e. people's expertise. The article demonstrates that efficiencies related to people expertise can be addressed by empowering and motivating the teams to frame the maintenance tasks using semantic constructs that abstract the implementation complexity, i.e. reduce the barriers to communication. This approach reduces the overall load on specialists that can focus on providing more value in other areas.

The paper tries to stress a more general message that was somehow also captured by Hajrahimi [12], i.e. the factors driving maintenance costs are organisation specific and are, to a large extent, a consequence of how the organisation operates. Therefore, to find efficiencies, managers should enquire/challenge the state of practice and make teams accountable for finding efficiencies. Allowing teams to find their own ways of organising work is a pre-requisite to starting resolving sources of friction and shifting towards approaches that can bring visible efficiency improvements.

At a very high level, the article acknowledges the value of having organisation-wide end-to-end processes as these allow explaining how the organisation operates. In practice, these processes become very difficult to change due to the large number of stakeholders and derived implications. Therefore, it is probably beneficial to accept that providing teams with the ability to define their ways of organising and collaborating is perhaps a good strategy to in the pursuit for operational and organizational wide efficiencies.

Acknowledgements

This work would not have been possible without the contribution and openness of the EUMETSAT's flight operations teams contributing to the Sentinel-6, Sentinel-3 and MSG missions.

References

[1] Dehaghani SM, Hajrahimi N. Which factors affect software projects maintenance cost more? Acta Inform Med. 2013 Mar;21(1):63-6. doi: 10.5455/AIM.2012.21.63-66. PMID: 23572866; PMCID: PMC3610582.

[2] Koskinen, J., 2003. Software maintenance costs. Information Technology Research Institute, ELTIS-Project University of Jyväskylä, p.16.

[3] Walsh, A., Pecchioli, M., Reggestad, V. and Ellsiepen, P., 2014. ESA's model based approach for the development of operational spacecraft simulators. In SpaceOps 2014 Conference (p. 1866).

[4] Sebastiao, N., Reggestad, V., Spada, M., Williams, A., Pecchioli, M., Lindman, N. and Fritzen, P., 2008. A reference architecture for spacecraft simulators. In *SpaceOps 2008 Conference* (p. 3507). Ellsiepen, P., Fritzen, P.,

[5] Sebastiao, N. and di Nisio, N., 2008. SMP2 (E-40-07), a New Standard for Simulation Model's Portability and its Implementation in SIMULUS. *Netherlands, SESP/ESA*.

[6] Reggestad, V. and Walsh, A., 2012. UMF—a productive SMP2 modelling and development tool chain. *Proceedings of the Simulation and EGSE Facilities for Space Programes (SESP'12)*.

[7] Pantoquilho, M. and Margarido, P., 2013. Keeping ESA's Operational Spacecraft Simulators Young. In *AIAA SPACE 2013 Conference and Exposition* (p. 5360).

[8] Reggestad, V., Guerrucci, D., Emanuelli, P.P. and Verrier, D., 2004. Simulator Development: the flexible approach applied to Operational Spacecraft Simulators. *SpaceOps, May*.

[9] Guerrucci, D., Rothwell, D. and Metelo, F., 2008. Aeolus Spacecraft Simulator: A Smooth Transition to Linux. In *SpaceOps 2008 Conference* (p. 3460).

[10] Clerigo, I., Montagnon, E. and Segneri, D., 2014. A simulated journey to Mercury: The challenges of the BepiColombo simulator development for the flight control team. In *SpaceOps 2014 Conference* (p. 1844).

[11] Pignède, M., Morales, J., Fritzen, P. and Lewis, J., 2010. Swarm Constellation Simulator. In SpaceOps 2010 Conference Delivering on the Dream Hosted by NASA Marshall Space Flight Center and Organized by AIAA (p. 2323).

[12] Li, J., Stålhane, T., Kristiansen, J.M. and Conradi, R., 2010, September. Cost drivers of software corrective maintenance: An empirical study in two companies. In *2010 IEEE International Conference on Software Maintenance* (pp. 1-8). IEEE.

[13] Lwakatare, L.E., Kilamo, T., Karvonen, T., Sauvola, T., Heikkilä, V., Itkonen, J., Kuvaja, P., Mikkonen, T., Oivo, M. and Lassenius, C., 2019. DevOps in practice: A multiple case study of five companies. *Information and Software Technology*, *114*, pp.217-230.

[14] Floris, P. and Harald, H.V., 2014. How to save on software maintenance costs, omnext white paper. *SOURCE 2* VALUE
[15] European Cooperation for Space Standardization, ECSS Secretariat, ESA ESTEC, P.O. Box 299, 2200 AG Noordwijk, The Netherlands, ECSS-E-HB-40A Software Engineering Handbook—Software, March 2013.
[16] Merri, M., Ercolani, A., Guerrucci, D., Reggestad, V., Verrier, D., Emanuelli, P.P. and Ferri, P., 2007. Happy families-cutting the cost of ESA Mission Ground Software. *ESA bulletin*, *130*, pp.62-69.
[17] Steele, P., Reggestad V., 2017. Evolution of the Operation Simulator Infrastructure at ESOC: SIMULUS Next Generation. Workshop on Simulation and EGSE for Space Programmes (SESP)
[18] Chaudhary, M. and Chopra, A., 2016. CMMI for development: Implementation guide. Apress
[19] Van der Pols, R., 1970. ASL® 2-A Framework for Application Management. Van Haren.
[20] Sahibudin, S., Sharifi, M. and Ayat, M., 2008, May. Combining ITIL, COBIT and ISO/IEC 27002 in order to design a comprehensive IT framework in organisations. In *2008 Second Asia International Conference on Modelling & Simulation (AMS)* (pp. 749-753). IEEE.
[21] European Cooperation for Space Standardization, ECSS Secretariat, ESA ESTEC, P.O. Box 299, 2200 AG Noordwijk, The Netherlands, ECSS-E-ST-40-07C Space engineering – Simulation modelling platform
[22] SWEBOK® Guide V3.0 https://mireilleblayfornarino.i3s.unice.fr/lib/exe/fetch.php?media=teaching:reverse:swebokv3_-_chap5_-_code_maintenance.pdf (accessed 11.01.23).
[23] European Cooperation for Space Standardization, ECSS Secretariat, ESA ESTEC, P.O. Box 299, 2200 AG Noordwijk, The Netherlands, ECSS-E-ST-40C Space Engineering - Software, March 2009
[24] Kotter, J.P., 2012. *Leading change*. Harvard business press.
[25] Lodgaard, E., Ingvaldsen, J.A., Aschehoug, S. and Gamme, I., 2016. Barriers to continuous improvement: perceptions of top managers, middle managers and workers. *Procedia CIRP*, *41*, pp.1119-1124.
[26] Vince, R., 1998. Behind and beyond Kolb's learning cycle. *Journal of management education*, *22*(3), pp.304-319.