STRATEGY AND TOOLS TO TEST SOFTWARE IN THE SKA PROJECT: THE CSP.LMC CASE

G. Marotta[*], C. Baffa, E. Giani, INAF-OA Arcetri, Firenze, Italy
M. Colciago, I. Novak, Cosylab Switzerland, Brugg, Switzerland
G. Brajnik, University of Udine and IDS, Udine, Italy

Abstract

The Square Kilometre Array (SKA) Telescope will be one of the largest and most complex scientific instruments ever built. The development of a reliable software for monitoring and controlling its operations is critical to the success of the entire SKA project. The Local Monitoring and Control of the Central Signal Processor (CSP.LMC) is a software component responsible for controlling a key subsystem of the telescope, i.e. the Central Signal Processor (CSP). The software is implemented as a “device” within the TANGO framework, written in Python code.

In this paper we describe a testing strategy that addresses a series of problems typical of such a large and complex instrument. It is a multi-level strategy, based on a combination of automated tests (unit/component/integration), in the context of CI/CD practices. Software is also tested against possible errors and anomalous conditions that can occur while the CSP.LMC is interacting with external subsystems, which can be simulated.

The paper also discusses needs and solutions based on data mining test results. This allows us to obtain statistics of unexpected failures and to investigate their causes. Furthermore, a database containing test results over several weeks supports discovery of interesting and unexpected patterns of behaviors of the tests based on correlations about different test-related events and data. This helps us to develop a deeper understanding of the code’s functioning and to find suitable solutions to minimize unexpected behaviors. In addition it can be used also to support reliability testing.

THE SKA PROJECT AND THE CSP.LMC

The Square Kilometre Array (SKA) telescope is an international effort to build the world’s largest radio telescope. Construction is underway at two primary sites in South Africa and Australia. These sites will house unprecedentedly large interferometers designed to observe the sky across two distinct frequency ranges: Mid-range (350 MHz - 15.3 GHz) and Low-range (50 MHz - 350 MHz). SKA aims to address fundamental questions about the universe, including its evolution, the nature of gravity, and the search for extraterrestrial life. Consisting of thousands of antennas spread over long distances, the SKA will have a combined collecting area of approximately one square kilometer. Its unprecedented sensitivity and resolution will provide insights into cosmic phenomena, offering a deeper understanding of the early universe, black holes, and pulsars.

The Central Signal Processor (CSP) is a critical component of the Square Kilometre Array (SKA) telescope. It’s responsible for the real-time processing of the vast amount of data collected by the SKA antennas, in order to make it available for scientific data processing. Given the immense scale of SKA, the CSP is expected to handle unprecedented throughput of data. The estimated amount of data will be around 7.3 Tb/s for Low and 8.8 Tb/s for Mid [1].

CSP comprises three primary instruments (as shown in Fig. 1), each of which is a complex signal processing subsystem. These are:

- the Correlator and BeamFormer (CBF), that combines the signals from various antennas to create a unified and focused view of the sky.;
- the Pulsar Search (PSS), that identifies potential candidates for pulsar discovery;
- the Pulsar Timing (PST), that measures the frequency of the radiation emitted from pulsar candidates.

On top of these subsystems, the Local Monitoring and Control (CSP.LMC) ensures the seamless operation of the CSP by providing real-time health checks, performance metrics, and adaptive controls [2]. In other words, CSP.LMC is the software component that represents the interface of the entire CSP instrument. It interacts with the Telescope Monitoring and Control (TMC) system, serving as the bridge between the TMC and the individual components of the CSP. It communicates to the TMC all the required information to monitor the CSP’s subsystems and provides the interface to send all the commands required to perform an observation. Even though the concept of CSP is the same for Mid and Low telescopes, some differences can occur in the data reduction hardware for the two telescopes. Therefore, two instances of CSP.LMC are developed to address the differences between the two telescopes.

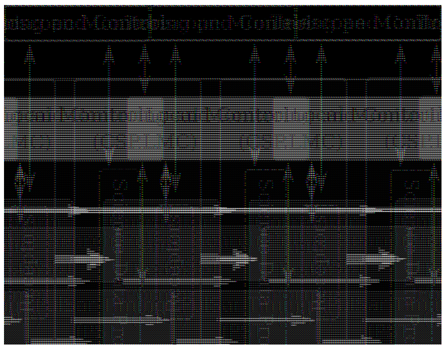

Figure 1: Simplified schema for CSP.

[*] gianluca.marotta@inaf.it

TESTING SKA SOFTWARE

Testing software components, especially as intricate and pivotal as the CSP.LMC in the context of large-scale projects like the SKA, is of paramount importance. Given its multifaceted responsibilities, even minor issues can result in significant operational disruptions. SKA Software needs to be reliable, ensuring that there's minimal downtime and that the system can recover gracefully from unforeseen issues. Furthermore, addressing software issues post-deployment, especially in a live environment like the SKA, can be incredibly costly in terms of both resources and time. For these reasons the testing strategy of software components is of utmost importance for the realization of such a big project.

The Software Engineering Group of SKA Observatory (SKAO) consists of more than 100 developers organized into different *Agile Teams*. It follows the Scaled Agile Framework (SAFe) approach, particularly suitable for big and complex software projects [3, 4]. Individual Agile Teams are usually responsible for a specific software subsystem, hence for its quality and its testing strategy.

In their usual workflow, developers use an *Continuous Integration & Delivery and/or Deployment (CI/CD)* approach [5], where changes in code are frequently integrated and tested in a shared environment. This approach promotes frequent and earlier detection of integration errors, resulting in higher code quality, increased development speed, and more reliable software releases.

Using this approach, code is collaboratively merged into a remote GitLab repository. Every time a developer pushes changes in code to this remote repository, a pipeline is automatically started, executing a sequence of operations —including running tests — to validate both the code's quality and the integrity of generated artifacts. Moreover, the testing process can be enhanced by scheduling recurring tests within the CI/CD pipeline. There's also the flexibility to initiate on-demand tests, for example, in specialized testing environments that might utilize distinct hardware.

Beyond the development teams and their CI/CD workflows, additional tests must be conducted by distinct stakeholders. This approach mitigates potential biases that emerge when the same individuals write both the software and its tests. To this end, specialized Assembly, Integration, and Verification (AIV) teams are tasked with crafting tests that directly verify formally agreed telescope requirements. However, the nuances of verification tests fall outside the scope of this article.

Spanning the efforts of individual teams, a *Testing Community of Practice* brings together developers to exchange knowledge and best practices about testing. This community provides an instant messaging channel and organizes monthly meetings where developers set the agenda based on topics they deem important. Typically, teams share tools and strategies, offering members insights on how to adapt these for their specific components to enhance quality and reliability.

CSP.LMC STRUCTURE

The global role of CSP.LMC is to coordinate the request received from TMC and provides responses on behalf of the entire CSP instrument. However, CSP.LMC is not just a singular software entity but a mosaic of different software components, developed as *Devices* within the TANGO Control System framework [6] and written in Python (*pytango*) [7]. The code utilizes an object-oriented approach, leveraging specialized classes to address the tasks assigned to it [8].

To maximize the efficiency of telescope resources, both Mid and Low telescopes permits users to partition the collecting area into as many as 16 *subarrays*. Each of these subarrays functions as a separate instrument, enabling independent scheduling, initiation, and termination of observations. The Local Monitoring and Control for each CSP subsystem, as well as the CSP itself, offer a software interface tailored for these subarrays. Meanwhile, the overarching management of the entire instrument is the responsibility of a component known as the controller.

Other CSP subsystems could have a similar structure, with subarray instances and a controller, plus a number of components devoted to directly control the hardware. Some of the subsystem's devices are directly monitored and controlled by CSP.LMC devices, as shown in Fig. 2. In fact, the CSP.LMC operates in a hierarchical fashion, with top-level components orchestrating and overseeing lower-level subsystems. This design ensures that while each component is specialized in its functions, there's a coherent, top-down approach to monitoring and control. Redundant connections between various devices guarantee the continuity of the system behaviour in the event of device failures.

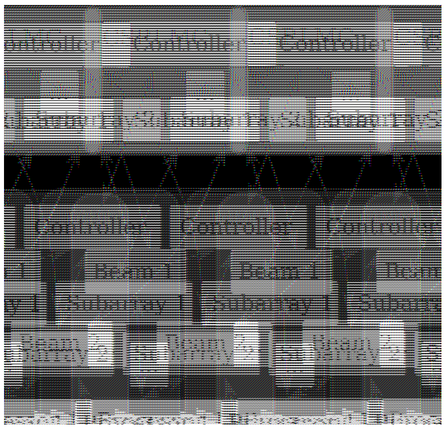

Figure 2: Simplified schema of interaction between CSP Software Components.

During runtime, all TANGO Devices are containerized and managed using Kubernetes (k8s) [9], which guarantees high availability—a critical feature for systems like SKA that need continuous operation. The fundamental unit in Kubernetes is the pod, representing the most basic deployable entity within the system. In the SKA software framework, the Kubernetes pod introduces an additional layer enveloping the Tango device. The business logic of the Tango device is implemented in a Python package (Fig. 3).

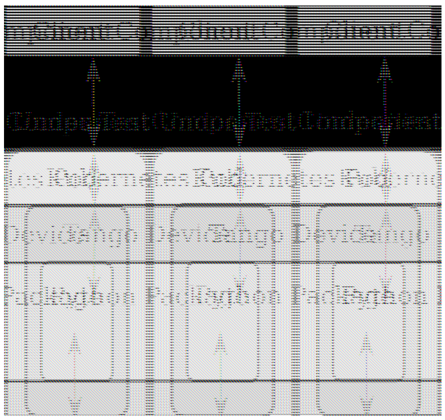

Figure 3: Simplified schema of the structure of Component Under Test and interactions.

The infrastructure that containerizes and deploys the Tango Device within k8s is not managed by the software component developers. Instead, it's developed and maintained by a dedicated agile team [10]. However, ensuring that the software effectively communicates with other devices within this framework falls under the purview of the software component developers.

UNIT, COMPONENT AND INTEGRATION TESTS

Given the complexity of the system, it's imperative to devise a multi-level testing strategy. A general categorization of the test levels — identifying unit, component, and integration tests — is based on the identification of boundaries of the *system under test* (SUT). For more detailed definitions and more details about the strategy, we reference the official SKAO Software Testing Policy and Strategy Document [11]. Many of the concepts and practices described in such a document are inspired by the ISTQB [12].

Unit Testing is defined as "the testing of individual software units (individual or clusters of functions, classes, methods, modules) that can be tested in isolation [...] the unit under test is run in an environment that is totally under control of the developer" [11].

In our implementation, given the object-oriented approach, unit tests are specifically designed to target the individual python classes within the codebase. By doing so, we aim to validate each object's behavior in isolation, ensuring its functionality is both correct and predictable. We employ `pytest` as our testing framework as it stands out for its simplicity in writing plain python tests with a rich set of supporting libraries and its capability to scale from simple unit tests to more complex functional tests.

Moreover, to ensure that unit tests exercise each SUT in isolation, mock objects are created to simulate the behavior of real, complex objects without invoking their associated side-effects or state. This is particularly beneficial as it means our tests can run without dependencies, ensuring that the object being tested is the sole focus.

Component testing is defined as it "aims at exposing defects of a particular component when run in an environment where other components or services are available, either production ones or test doubles" [11]. In our interpretation, the main purpose of Component Test is to test the component as a single entity that contains all the business logic. These tests are meant to verify the component in its environment and not the real interaction with other devices. Two different type of component tests are possible for CSP.LMC, depending on whether k8s environment is taken into consideration: *python-component* and *k8s-component* test.

In *python-component* the test script connects directly to the Tango device with the use of the *DeviceTestContext* classes of Pytango that allows to test a device without the need for Tango environment to be running. In the same context, server components are substituted with mocked objects, written in a way that simulates the interaction with the real devices.

In *k8s-component* tests, on the other hand, the entire kubernetes infrastructure is in place as well as the Tango environment. The test script is a Tango client that runs in a specific pod, driven by pytest as well. Instead of server components as above, *emulators* are in place. These fake devices use the same machinery as the real ones to be deployed in the k8s cluster. The emulators mimic the interface of the actual devices to support the needs of the specific tests. Emulators greatly simplify the internal implementation and eliminate downstream interactions. Additionally, they incorporate a range of methods and attributes, enabling external control over the device's behaviour and broadening the potential test cases.

Integration Testing is defined as "Testing performed to expose defects in the interfaces and in the interaction between components or subsystems. [...] The SUT consists of 2 or more components, and it is run in an environment where components or services external to the SUT include production ones and test doubles. [...] Integration testing may also include hardware-software integration testing" [11].

During integration tests, the test script operates within a pod, similar to a k8s-component tests. However, in this case, the server component represents the genuine software element, i.e. the subsystem's components developed and released by other teams. Downstream software and hardware can be either present or simulated. In fact, integration tests are not designed to be end-to-end tests, as their focus is solely on examining the adjacent interfaces of the Component Under Test. (i.e. CSP.LMC components). The presence of data processing hardware is a sensitive situation for a system like CSP.LMC, so the possibility of performing integration tests with hardware is critical for the validation of the functionalities. In this scenario, tests are initiated by on-demand CI/CD pipeline jobs, coordinated with the teams overseeing the facilities where the hardware is located.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BCO03

FAULT CONDITION ANALYSIS

Among all the test cases, a crucial testing area for a component like CSP.LMC, which is dedicated to monitoring and control, is the management of fault conditions that might arise in other CSP subsystems. To facilitate the creation of use cases and their associated tests, five primary areas have been identified, each containing several subareas. These are detailed below.

- **Networking**: error conditions that arise from problems related to networking connections. Related subareas are: *TangoDB connection*, *Lost connection with a running device, Lost connection with a stopped device, Event subscription, Disconnection during command execution, Connection timeouts*.
- **Configuration**: error conditions that arise during the configuration of an observation. Related subareas are: *invalid configuration; unavailable resources, unresponsive subsystems*
- **Command execution**: error condition related to the execution of commands on subsystems. Related subareas are: *wrong input, command not allowed, LMC device failure, subsystem device failure, slow execution*
- **Monitoring**: error condition that occurs during the process of monitoring the other subsystems. Related subareas are: *device failures, conflicting events, race conditions*
- **Infrastructure**: error conditions related to TANGO/k8s infrastructure. Related subareas are: *Failing/Restarting Pods, Tango DB wrong configuration, Tango DB unavailability*

To test the CSP.LMC behaviour in order to cover some of these tests conditions (e.g. device failures, slow execution), the behaviour of the subsystem device needs to be externally driven and errors should occur on-demand. This can occur only for both python and k8s component tests, because in these cases the test script can configure and inject appropriate mocks or emulators into the test environment in which the SUT is run. On the other hand, integration tests can verify almost only "*happy path*" scenarios, i.e. where the subsystems interacting with the SUT (the CSP.LMC) are assumed to perform as expected.

TESTING INFRASTRUCTURE

Except for unit tests, component and integration tests sometime exercise the same business logic of the Component Under Test. As stated before, the main difference between component and integration testing is the presence or not of the other subsystems' devices. The other main difference lies in the fact that for integration tests it is not possible to inject failures and undesired behaviour in the downstream subsystems. This means that there is a significant set of test cases that can be used in both component (python and k8s) and integration tests, another set of test cases that are common to k8s-component and integration (e.g Tango DB failures) and some test cases that can be only used in k8s-component, with emulated devices.

Given these overlaps, we strove to reduce redundancy as much as possible, and formulate test scripts that are parametric enough so that they can be applied to more than one environment and level. This makes it possible to use a testing infrastructure and testing code with as many shared elements as possible. This results in a more consistent and easier-to-maintain infrastructure. A simplified schema is provided in Fig. 4.

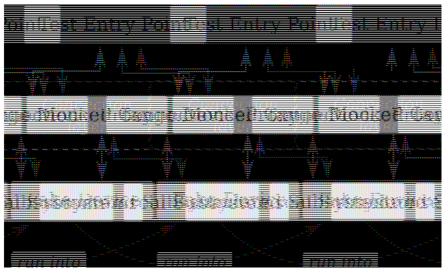

Figure 4: Simplified schema for the testing infrastructure.

Since all three types of tests use pytest as the underlying testing framework, in many cases the test scripts can be shared among them. Because for higher level tests (i.e. higher than unit tests) the potential set of stakeholders is larger and includes not only developers, but also product owners and managers, we decided to opt for using the Gherkin language [13] for writing executable test scripts that can be easily understood by anybody with sufficient knowledge of the CSP. Gherkin can be used to express human-readable descriptions of software behaviours without detailing how that functionality is implemented and independently from a programming language and testing framework. A specific plugin for the pytest framework handles processing of gherkin clauses. An example of a gherkin test case is presented below.

```
@python_component @k8s_component @integration
Scenario: Turn on CSP
  Given CSP Controller setup as off
  And all CSP Subarrays are setup as off
  When CSP Controller's OnCommand is triggered
  Then CSP Controller is eventually on
  And all CSP Subarrays are eventually on
```

Listing 1: a test written in Gherkin

DATA MINING ON TESTS RESULTS

Data mining aims at processing large and varied historical data sets to identify relationships that can provide answers to business problems [14]. In many cases, data mining is introduced in software testing for the purpose of using data to drive, or inform, the identification of suitable tests to write [15].

In our context, we want to use data mining techniques in another way. We intend to mine multiple test execution results because that presents an invaluable opportunity for extracting meaningful insights, optimizing testing processes, and improving product quality. When software or systems undergo multiple test cycles, the accumulated results are a rich source of data that can be mined for patterns, trends, and anomalies.

Source data sets consist of test results, i.e. the outcomes of each execution of any of the tests for the CSP.LMC (across test levels, test environments, and versions of the SUT and of the external emulators or subsystems). In addition, for higher level tests, data points include not only the test outcome (i.e. failed, passed, error), but also the outcome of each gherkin step, and the timestamp of each step (so that duration of the execution of a step can be computed).

The business problems to solve through data mining address complex analyses of test results that span two or more test executions and relationships between behaviours of two or more tests or steps of tests. The following kinds of analyses can be supported:

1. **Trend Analysis**: By mining data from multiple test executions, teams can identify trends in failures and successes. This is the case for longitudinal studies of the behaviour of individual tests (i.e. the time series of outcomes of a specific test, filtered according to different test environments, test levels and possibly versions of specific software components). For instance, if a specific module consistently fails across multiple test cycles, it may be indicative of a deeper, systemic issue that needs addressing.

2. **Optimizing Test Suites**: Mining can reveal which tests consistently pass and might be of lower value and which ones uncover defects, helping in refining and prioritizing the test suite. This optimization can lead to faster test cycles and resource savings.

3. **Predictive Analytics**: By analyzing patterns from past test results, predictive models can be built to forecast potential failures in future test cycles. This can guide teams to proactively focus on potential problem areas. For example, one can use statistical correlations between test outcomes or between timing information of different tests to identify clusters of data points that can reveal some underlying common and plausible cause.

4. **Root Cause Analysis**: Mining test data can help in pinpointing commonalities among failures, aiding in root cause analysis. This could be particularly useful in complex systems where failures might not be immediately apparent. We expect that in many cases, test incidents (i.e. the occurrence of a negative event, where a test fails or leads to an error) may be due to obscure bugs that require a substantial effort to be triaged and eventually diagnosed. For control systems this might be related to particular race conditions occurring within the SUT's parts or occurring in interactions of the SUT with the rest of the CSP. We expect these situations to be more frequent and important as the SUT includes more custom hardware devices in the test environments. Many of these test incidents cannot be solved simply by looking at a single test execution. History of the behaviour of tests and of the SUT is needed to provide insights.

5. **Flakiness Detection**: In the realm of automated testing, some tests might intermittently fail, because of issues like timing, order of execution, or external dependencies. Mining data from multiple executions can help identify such flaky tests. Analysis of correlation between test or step outcomes and their execution times might reveal patterns that suggest certain kinds of weaknesses in parts of the SUT.

6. **Environment Insights**: If tests are executed in various environments or configurations, mining can provide insights into environment-specific issues. For example, a feature might consistently fail on a particular environment, indicating compatibility issues.

7. **Resource Utilization**: Mining can also reveal insights about resource consumption during tests, like memory leaks or high CPU usage. These insights can be crucial for performance optimization.

8. **Feedback Loop Enhancement**: Continuous integration and continuous deployment (CI/CD) thrive on quick feedback loops. Mining data can provide rapid, actionable feedback to developers, reducing the defect lifecycle.

9. **Enhanced Reporting**: Data mining can lead to advanced visualization of test results, offering stakeholders a clearer picture of product health, risk areas, and quality metrics.

To implement this kind of infrastructure we are building upon what is already made available by the currently used machinery, namely CI/CD services and testing frameworks. To support data mining the typical Extract/Transform/Load activities need to be supported, to extract data from test reports, to transform the data to polish it and to support statistical processing, and finally store it into some sort of data warehouse.

For test data the challenges include data integrity, like consistently identifying test scripts and their steps across different versions of the SUT and of the tests. Another critical aspect is correct computation of durations of test or step executions.

Another challenge is the skills needed to perform these activities. A combination of data science skills and knowledge about the SUT being tested and the tests themselves are needed, and are not easy to acquire. For the moment this is confined to a few members of the team that is developing the CSP.LMC.

In conclusion, data mining on multiple test execution results can revolutionize the testing process, offering deeper insights, better resource utilization, and improved product quality. However, to harness its full potential, teams need to ensure data integrity, have the right tools in place, and possess the required skills.

CONCLUSIONS

In this paper we presented a comprehensive testing strategy for the CSP.LMC component, which plays a critical role in presenting the CSP as a unified instrument within the telescope and also to interact with the TMC.

The testing strategy adopts a multi-level approach. Throughout this paper we have demonstrated how different levels of testing (unit/components/integration) are strategically employed to evaluate our software within distinct contexts. With unit tests we scrutinize individual python classes in isolation, decoupling them from dependencies. Next, we progress to testing individual components, either through test scripts utilizing the DeviceTestContext without the Tango environment or by deploying the entire kubernetes infrastructure with the Tango environment in place using emulators as proxies for real devices. Integration tests extend the evaluation to the environment where Systems Under Test (SUT) coexists along with other subsystems components. Those tests are not devised as end-to-end, but they are examining real interfaces between two or more systems.

Important aspect of CSP.LMC component is the management of fault conditions. To test those thoroughly we have devised a systematic approach to the categorization of foreseeable failures and rely on emulators to drive the occurrence of those errors.

Our testing infrastructure has been consolidated to eliminate redundancy and duplication where possible with testing scripts shared across contexts. Readability of Gherkin test cases improves shared understanding of the tests among developers and other stakeholders.

Furthermore, we aim to utilize the data mining techniques to collect and analyze the results of multiple test runs. This will aid in addressing systemic issues, identifying elusive bugs, pinpointing flaky tests and assessing the issues with the environment. It will lead to optimizing testing cycles, improved resource utilization and provide valuable insight for developers while offering a clearer picture to a broader audience of stakeholders.

REFERENCES

[1] SKAO Multimedia Library, `https://skao.canto.global/v/SKAOLibrary/album/VH3Q8`

[2] C. Baffa, E. Giani, S. Vrcic, and M. Vela Nuñez, “SKA CSP controls: technological challenges”, in *Proc. SPIE 9913, Software Cyberinfrastructure Astron. IV*, Edinburgh, United Kingdom, 2016, p. 99132Z. doi:10.1117/12.2233325

[3] Scaled Agile, Inc., SAFe for Lean Enterprises 6.0, `https://www.scaledagileframework.com/`

[4] Valentina Alberti and Snehal Valame, “Inspecting and adapting via problem-solving workshops: the SKA experience”, in *Proc. SPIE 12189, Software Cyberinfrastructure Astron. VII*, Montréal, Québec, Canada, 2022, p. 1218901. doi:10.1117/12.2630038

[5] Jez Humble and David Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison Wesley Professional, 2010.

[6] Tango Controls, `https://www.tango-controls.org/`

[7] PyTango, `https://pytango.readthedocs.io/`

[8] G. Marotta, E. Giani, I. Novak, A. Söderqvist, and C. Baffa, “Software design for CSP.LMC in SKA”, in *Proc. SPIE 12189, Software Cyberinfrastructure Astron. VII*, Montréal, Québec, Canada, 2022, p. 121891A. doi:10.1117/12.2630140

[9] Kubernetes, `https://kubernetes.io/` (accessed 06/10/23)

[10] M. Di Carlo *et al.*, “CI-CD practices at SKA”, in *Proc. SPIE 12189, Software Cyberinfrastructure Astron. VII*, Montréal, Québec, Canada, 2022, p. 1218903. doi:10.1117/12.2620526

[11] SKAO Software Testing Policy and Strategy, `https://developer.skao.int/en/latest/policies/ska-testing-policy-and-strategy.html`

[12] ISTQB, `https://www.istqb.org/`

[13] Gherkin, `https://cucumber.io/docs/gherkin/`

[14] Jake VanderPlas, *Python Data Science Handbook*. O’Reilly Media, 2016. ISBN: 978-1491912068.

[15] Mark Last, Menahem Friedman, and Abraham Kandel, “Using data mining for automated software testing”, *Int. J. Software Eng. Knowl. Eng.*, vol. 14, no. 04, pp. 369-393, 2004. doi:10.1142/S0218194004001737