Software testing for the CTA observation execution system

Thomas Murach, Anze Zagar, Urban Leben, Igor Oya, Matthias Füßling, et al.

Thomas Murach, Anze Zagar, Urban Leben, Igor Oya, Matthias Füßling, Dejan Dezman, Vito Conforti, Fabian Krack, Etienne Lyard, David Melkumyan, Klemens Mosshammer, Iftach Sadeh, Torsten Schmidt, Ullrich Schwanke, Joseph Schwarz, Stephan Wiesand, "Software testing for the CTA observation execution system," Proc. SPIE 10707, Software and Cyberinfrastructure for Astronomy V, 107070D (6 July 2018); doi: 10.1117/12.2313437

Event: SPIE Astronomical Telescopes + Instrumentation, 2018, Austin, Texas, United States

Software Testing for the CTA Observation Execution System

Thomas Murach[a], Anze Zagar[b], Urban Leben[b], Igor Oya[a], Matthias Füßling[a], Dejan Dezman[b], Vito Conforti[c], Fabian Krack[a], Etienne Lyard[d], David Melkumyan[a], Klemens Mosshammer[a], Iftach Sadeh[a], Torsten Schmidt[a], Ullrich Schwanke[e], Joseph Schwarz[f], Stephan Wiesand[a], and for the CTA Consortium

[a]DESY, D-15738 Zeuthen, Germany
[b]Cosylab, Ljubljana, Slovenia
[c]INAF - Istituto di Astrofisica Spaziale e Fisica Cosmica di Bologna, Via Piero Gobetti 101, 40129 Bologna, Italy
[d]ISDC Data Centre for Astrophysics, Observatory of Geneva, University of Geneva, Chemin d'Ecogia 16, CH-1290 Versoix, Switzerland
[e]Institut für Physik, Humboldt-Universität zu Berlin, Newtonstraße 15, 12459 Berlin, Germany
[f]INAF - Osservatorio Astronomico di Brera, Italy

ABSTRACT

The Cherenkov Telescope Array (CTA) will be the next-generation ground-based instrument for detecting very-high energy gamma rays. It will consist of roughly 100 telescopes of different sizes and designs. In addition, a variety of auxiliary instrumentation will be part of the array. The Observation Execution System (OES) is the software system in charge of operating and monitoring all telescopes and devices, applying short-term observation schedules depending on the hardware status and environmental conditions and handling the data. Motivated by the wealth of tasks to accomplish and requirements to fulfil, a software development procedure is conceived for the development of OES. Part of this development process is the application of software testing procedures. These procedures range from unit tests up to system tests and stress tests. In this contribution, the software development process and the application of static and dynamic code analysis tools are described.

Keywords: CTA, Observation Execution System (OES), ACS, Software Testing, Software Development Life Cycle, Continuous Integration, Static Code Analysis, SonarQube, Jenkins

1. INTRODUCTION

The Cherenkov Telescope Array (CTA*) will be the next-generation array of imaging atmospheric Cherenkov telescopes (IACTs). While current-generation IACT arrays consist of up to five telescopes at most, the CTA design foresees the installation of more than 100 telescopes in two different locations. The northern array, located at the Roque de los Muchachos Observatory on the island of La Palma, Canary Islands, Spain, will consist of 4 large-sized telescopes and 15 medium-sized telescopes. The southern array is foreseen to comprise 4 large-sized telescopes, 25 medium-sized telescopes and 70 small-sized telescopes. This setup provides unprecedented opportunities for the study of violent, non-thermal processes both inside and outside the Milky Way. Very-high energy (VHE) gamma rays with energies covering four orders of magnitude between approximately 30 GeV and 300 TeV will be detectable by CTA telescopes. At energies of a few TeV, the sensitivity of either of the arrays will be a factor of approximately ten better than the sensitivity of any of the current-generation IACT experiments.

The instrumental setup of CTA will comprise telescopes and array-level instrumentation like weather stations or LIDARs. In the software architecture of CTA it is foreseen that the Observation Execution System (OES[1]) controls the involved hardware centrally and at high level, i.e. firmware or software close to the hardware is not part of OES. This high-level control software is the tool with which all array operations from preparing and starting operations at the beginning of a night to shutting down devices, e.g. at the end of a night, are

*https://www.cta-observatory.org/ (2018)

performed. These operations include scientific observations of dedicated areas of the sky as well as calibration data-taking procedures.

The scientific observation schedule is prepared in advance by a separate program called Scheduler.[2] Based on this schedule, the telescopes are grouped into so-called subarrays dynamically and instructed to operate according to plan. OES provides a software interface for the array scheduler.

OES also provides an interface for human operators.[3] This interface will display status information about the array. Furthermore, this interface will be used whenever manual intervention is required. Other tasks of OES comprise the storage of the raw data stream received from each Cherenkov camera on hard disk and the storage of monitoring data and logging information from all devices deployed on a CTA site. Several of the aforementioned tasks are time-critical, so the system needs to be designed with performance in mind. There are strict requirements on the performance and reliability of this software system, since any potential error or delay caused by the system might lead to the loss of observation time. There are (at the time of writing) more than 140 requirements governing the behaviour of OES to ensure that a reliable and performant system is developed.

The size and complexity of the task of designing and implementing such a system requires the combination of efforts from contributors hosted at different institutes or companies, or in the case of OES also located in different countries. To ensure that the individual subsystems converge into the desired software system that fulfils all requirements, a software development life cycle (SDLC) plan was created. This plan includes procedures for testing the developed software products at various levels of complexity, starting from unit tests up to system tests and verification and validation procedures.

In section 2 of this paper, the OES SDLC plan will be presented in more detail. In section 3, the implementation of a first software testing setup in the context of OES is presented. In section 4, plans for future, more sophisticated software testing procedures are described briefly.

2. SOFTWARE DEVELOPMENT LIFE CYCLE FOR OES

The SDLC plan[4] created for the OES software development process is designed to be embedded into a future CTA-level SDLC plan. It is tailored for the particular needs of the OES development process, keeping in mind the current status of the project and the necessity to progress fast despite the fact that several requirements to be fulfilled by OES may be updated in the future. Thus a process that allows for flexibility in the software development process is targeted, which at the same time causes little overhead.

Several SDLC models like the waterfall model[5] or the V-model[6] exist, each providing different levels of freedom to the development process and, therefore, different amounts of effort put into initial planning. Given the possibility that requirements may be updated in the future and that the OES development may proceed in stages, an SDLC model that provides relatively large flexibility is targeted. The model that was chosen is a hybrid model incorporating aspects of the *iterative* and the *incremental* SDLC models.

2.1 SDLC Model

In the iterative model, initial focus is put on a simplified product implementation, based on only a subset of the requirements. After this first implementation has been completed, the software system is allowed to progressively evolve into the final, feature-complete software system. In general, the iterative model prescribes a cyclical development process, in which, after an initial planning phase, a sequence of actions and stages is repeated. In each cycle, also requirements, the software design and functionality are allowed to be subject to modifications.

The other SDLC model of which aspects are incorporated into the model used for OES development is the incremental model. In this model, the software system is decomposed into a number of components or subsystems which are then implemented each at an individual pace and at suitable points in time. When the development of a subsystem or component has come to an end, the respective software product is added to the product, which thereby grows with time. With each release of the product, new parts can be added, which is a repetitive process until all parts have been implemented.

In the combined iterative and incremental model,[7] the system is decomposed into smaller parts like subsystems or components. These are developed individually and added to the system when ready. Still it is possible to update requirements or to define new features. Whenever such an update is performed, a complete iteration is executed, including the design updates, implementation and testing of deliverables. The combined model provides greater flexibility than the individual models mentioned above since it is not limited to one iteration at a time. At the end of any iteration, changes can be integrated into the final product, yielding functional and verified deliverables that can be integrated into following product releases. Furthermore, in this model it is also possible to include potential user feedback as input to future iterations.

2.2 SDLC Phases

The workflow described in the SDLC model is grouped in phases. For each phase of the SDLC, several key aspects are identified and highlighted. In general, these aspects are the answers to the questions

- What are the deliverables?
- How will these deliverables be produced?
- Which roles will be involved?
- When will deliverables be ready to pass on to the next stage?
- Where will results be deposited?

A list of roles identified to be required in the presented SDLC model is shown in Tab. 2 together with a brief explanation of the respective tasks of each role. Individual people may have several roles, and several people may have similar roles. In Tab. 3, key word definitions are given for reference, and in Fig. 1 the general workflow is represented schematically. In the following paragraphs, the phases will be summarised comprehensively, with a focus on software testing aspects.

A crucial part of a successful software system development is the *Requirements Analysis*, which represents the first phase of the presented SDLC model. After collecting requirements they need to be checked for necessity, consistency, completeness and feasibility. CTA requirements are grouped in two categories called "level A" and "level B". Level A requirements are observatory-level requirements. From these, level B requirements concerning e.g. OES are derived. Once finalised, all requirements are collected, grouped, documented and discussed on a Jama[†] web platform. This phase is the main activity of the requirements analyst. The testing lead assists by ensuring that the requirements are verifiable. The latter role also produces draft specifications for qualitication testing. Later and more detailed test plans are expected to be aligned with this high-level test plan.

The development of the software system *Architecture* based on the requirements and use cases is subject of the second phase. A hierarchy of interacting subsystems and components is developed by the software architect. Common quality attributes like performance, security or manageability are optimised. Using the UML and SysML notation standards, the architecture[8] is manifested using the Sparx Enterprise Architect[‡] tool. Functional, logical, technical and conceptual data models are created in the architecture. Input requirements and output architecture specifications are synchronised between Jama and Enterprise Architect. In this way, a complete requirements coverage can be assured. The testing lead creates draft specifications for integration testing based on the software architecture. These testing specifications are entered into Jama and linked to requirements and use cases.

In *Detailed Design* specification phase, the architecture from the previous phase is refined to much finer levels of detail by the software designer(s). Apart from external interfaces of subsystems and components, also their internal structure, algorithms and data structures are defined. In case of subsystems, the design process is repeated until the level of components is reached. In addition to source code specifications, also database schemas are produced at this stage. The output specifications of this phase provide the basis for direct implementation.

[†]https://www.jamasoftware.com/ (2018)
[‡]https://www.sparxsystems.eu/start/home/ (2018)

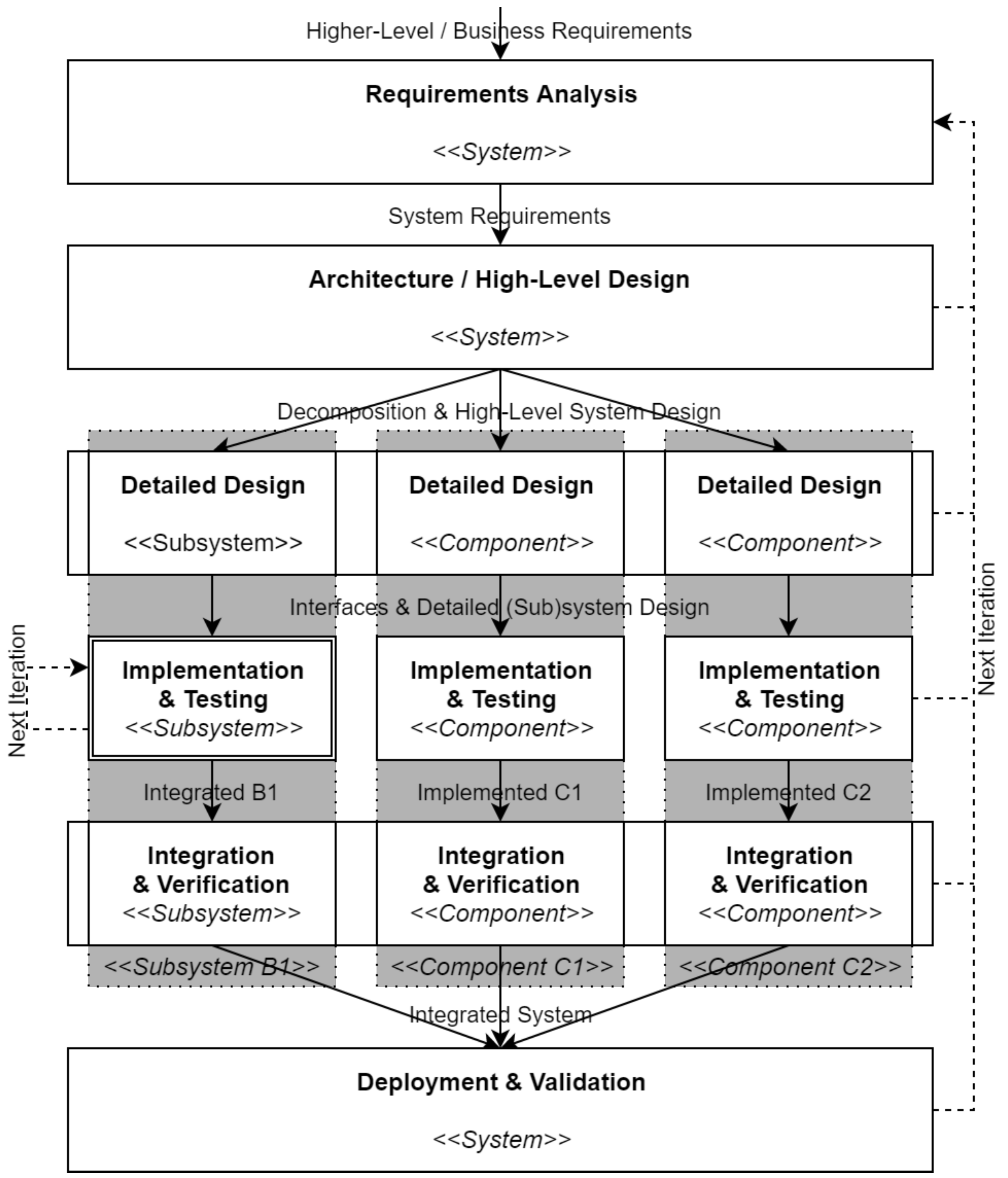


Figure 1: Schematic representation of the iterative and incremental development model.[4] Starting from higher-level requirements, the entire SDLC model described in the main text, down to the deployment and validation phase, can be traced in this flow chart. Grey boxes represent increments. Both the procedures to be followed for subsystems and components are shown. The implementation and testing phase of the subsystem increment is shown with a double border, suggesting that the same development model as for the system should also be applied here.

Testability is scrutinised by the testing lead, who at the same time further refines test cases. Complete test coverage of the design specifications is required by specifying unit tests.

In the next phase, *Implementation and Testing* according to specfications takes place. This phase is applied individually to each subsystem or component. In case of subsystems, the implementation phase implies the reapplication of the entire SDLC process. In case of components, software and database schemas are implemented, configuration files and all necessary software artifacts are created. The compliance of delivered products with specifications and quality requirements is verified through static code analysis, code reviews and automated testing. All described tasks are mostly the responsibility of the software developer. The implementation of the aforementioned testing aspects is described in the next section. Once the component is functionally complete and tested, extensive code reviews are to be performed.

After the implementation of a compontent or functionality of the software system, the *Integration and Verification* phase follows. Here, the individual software parts are integrated to act as a single system. To complete this phase, the integrated system needs to be verified by strictly following the integration test plan. Mainly the software system integrator and the software tester are involved in this step.

The final phase of the SDLC process is the *Deployment and Validation* phase where the integrated and verified system is deployed in a production environment. This system is then validated according to the qualification test plan, which is based on the qualification test specfications and created at the beginning of this phase. The successful validation results in a system that is ready for provisioning. The software system integrator is responsible for the deployment and provisioning of the system. Stakeholders participate in the validation and provisioning process and approve deliverables.

3. SOFTWARE TESTING FOR OES

The OES software system can be subdivided into the following blocks:[1]

- Short Term and Target of Opportunity Scheduler
- Manager and Central control
- Data Handling System
- Operator Interface (graphical user interface, GUI)
- Monitoring System
- Reporting and Diagnosis.

A wealth of tasks is implied, ranging from e.g. scientific alert handling, coordination of telescope activities, execution of schedules to error handling. For a more complete overview, the reader is referred to reference 1. For all of the abovementioned building blocks of OES, prototype code exists. The software is built according to a container-component model based on the ACS framework,[9] a design choice taken at the CTA level. ACS itself is a distributed software system based on CORBA[10] that simplifies the distributed deployment of software components on a computing cluster, abstracts the communication of pieces of software over a network, and provides a multitude of generic functionality like the regular publishing of monitoring values. In the context of CTA, the design choice was taken to support the Java, C++ and Python programming languages, which are natively supported by ACS.

As described in Sec. 2, software testing happens on several levels. In the following sections, the tests performed in the implementation and testing phase are described. Currently available to the group of OES developers is a setup consisting of a continuous integration (CI) system that provides automated software builds, execution of unit tests, code branch coverage analysis and static code analysis. In the presented software development model, sufficient test coverage is requested at any time, which implies that the unit tests must be provided by the developer either before or at the time of implementation, but not afterwards. Furthermore, all code needs to be integrated into a version control system. Currently, Subversion (SVN) is used for this purpose, but in the

Property	**Worker Node**	**Storage Node**
Number of machines	10	5
Brand	Dell PowerEdge R630	Dell PowerEdge R730xd
Processors	2x Intel Xeon 2640v3 (2.6 GHz, 8 cores/socket, 2 threads/core)	1x Intel Xeon E5-2630Lv3 (1.8 GHz, 8 cores, 2 threads/core)
RAM	64 GB	32 GB
Storage	2x 600 GB SAS disks	2x 300 GB SAS disks for operating system + 12x 6 GB NL SAS disks for data
Network interface cards	2x 10 Gb/s + 2x 1 Gb/s	2x 10 Gb/s + 2x 1 Gb/s
Operating system	Scientific Linux 6/7	Scientific Linux 6/7

Table 1: Outfitting of the computer cluster used for OES software testing. The labelling of nodes as *worker nodes* or *storage node* only describes the use case the respective machines were designed for. This labelling does not exclude that e.g. storage nodes perform computationally intense tasks, and vice versa.

near future a transition towards a git infrastructure is planned. All code is supposed to be checked by the CI system at all times to provide continuous evaluation of build successes and code quality.

Since a large code basis has been created before the SDLC model was established and also before the software testing infrastructure existed, additional effort may be required to adapt already existing code. Before an initial release of the OES software system, these older parts of the code will be updated to fulfil current code quality requirements.

3.1 Computer Cluster for OES Software Testing

For the CI system mentioned above, a dedicated computer cluster is available. This cluster is part of the IT infrastructure at DESY, Zeuthen, and central services like shielding by a central firewall, DNS or SMTP services are provided by the central IT department.

This OES software testing cluster consists of fifteen dedicated machines. Ten of them are optimised for computing-intense tasks, the remaining five are configured to best act as storage nodes. The hardware specifications of all machines are listed in Tab. 1. A central Arista DCS 7304 switch provides network connections between the machines. All devices including the switch are capable of providing network communication speeds up to 10 Gb/s. The operating system installation and initial setup is largely automated, such that simple re-installation or factory resets can be performed any time with minimal time investment.

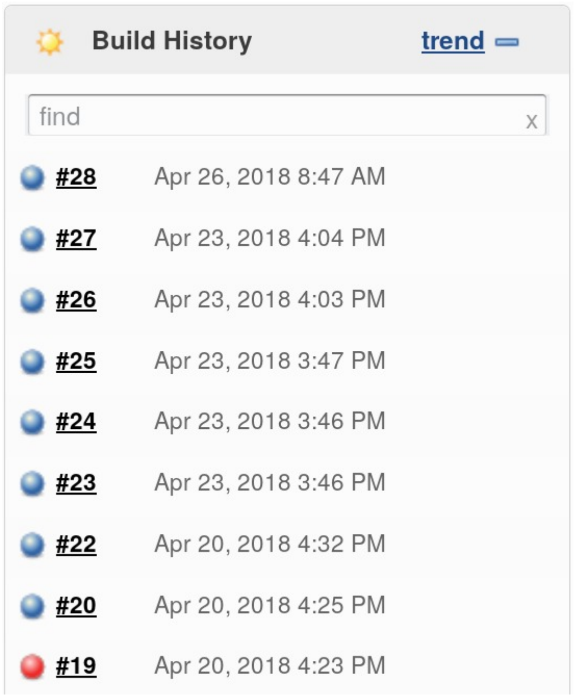
Access to the testing tools is granted from white-listed IPs only. For OES users, the usual way of interacting with the system is the access through a web browser. All services are accessible only after providing credentials. Authentication is done via an LDAP[§] query from the central CTA LDAP service, hosted in Heidelberg, Germany. This centralised user management reduces overhead whenever developers join or leave the OES team.

In the mid-term future, advanced performance testing of software components, databases and integrated software systems will be needed. For these tests it may be more convenient to have command-line access via SSH. The formal procedure how to motivate, request and perform such tests is yet to be defined.

3.2 Software Testing in Practice

The entry point for the CI system is provided by the Jenkins CI platform[¶]. The base functionality offered by Jenkins is the execution of software builds based on certain trigger criteria, which could be the passing of a

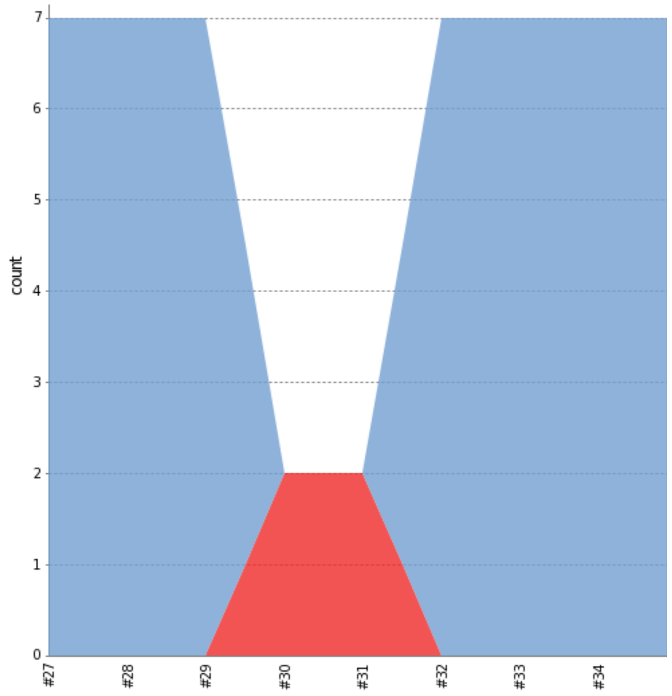
[§]https://tools.ietf.org/html/rfc4511 (2018)
[¶]https://jenkins.io/ (2018)


Build History trend

find x

#28	Apr 26, 2018 8:47 AM
#27	Apr 23, 2018 4:04 PM
#26	Apr 23, 2018 4:03 PM
#25	Apr 23, 2018 3:47 PM
#24	Apr 23, 2018 3:46 PM
#23	Apr 23, 2018 3:46 PM
#22	Apr 20, 2018 4:32 PM
#20	Apr 20, 2018 4:25 PM
#19	Apr 20, 2018 4:23 PM

(a)



(b)

Figure 2: *(a)*: Example picture of the Jenkins widget showing the build success as a function of time and the number of the build attempt. Red bullets indicate build errors, blue bullets indicate build successes. *(b)*: Example picture of the diagram showing the number of successful (blue) and non-successful (red) unit test evaluations as a function of the evaluation attempt number.

user-defined period of time, the commit of source code to a repository or a manual request. Apart from this, Jenkins is very flexible due to the large number of available plug-ins. In case of the OES software testing setup, the relevant plug-ins that are enabled are the LDAP integration plug-in, the unit testing plug-in and the version control plug-in, in this case configured for SVN repositories.

The infrastructure is set up such that each developer can create their own projects. Depending on the source code and dependencies, certain pre-configuration steps are necessary. In case of software components based on the ACS framework, a corresponding environment needs to be set up. A peculiar issue arises in case of unit tests or component tests that require running ACS instances. When an ACS instance is started, it is attempted to reserve several local ports on the host machine. The port numbers can be influenced by means of setting an environment variable accordingly. To avoid port number collisions between ACS instances started when testing different code modules simultaneously, the check for other running ACS instances needs to be performed when the test evaluation of a respective Jenkins project is started. A shell script included in a Jenkins project template was developed and provided to the OES developers.

Feedback provided to the developer includes the success of the software build and a detailed report about the outcomes of all involved unit test executions. Both quantities are also visualised graphically and displayed as functions of time as shown in Fig. 2. Detailed log outputs are available as well.

It is possible to configure Jenkins projects such that notification e-mails are sent, either to a static e-mail address or to the person who committed the respective patch, in case an error occurred either during the build process or during the unit test evaluation. This notification functionality is not used yet, but the activation of this feature is planned for the near future.

At the end of the build and unit test analysis step performed by Jenkins, a static code analysis and coverage

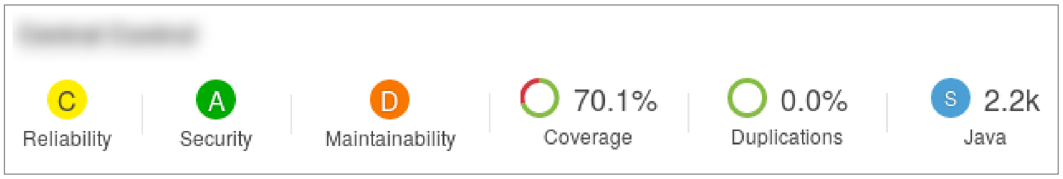


Figure 3: SonarQube summary for an example project (name blurred). Summary ratings for the quantities reliability, security and maintainability are shown, as well as the code coverage fraction introduced in the main text and the percentage of duplicated source code lines. Also the total number of lines of code is presented. Based on these measures, a simple and fast evaluation of the status of a source code component is possible.

calculation tool called SonarQube[‖] is activated. This is an open source tool that provides the aforementioned functionality for a large variety of programming languages, most notably the three languages targeted for usage in CTA software projects (Java, Python and C++).

In case of the setup chosen for OES development, SonarQube projects correspond to source code modules as well as to Jenkins projects. The basis for the static code analysis is a set of rules against which the source code of the project as such and also every commit individually is checked. These rules are grouped by criticality. The most critical class of rules is able to detect software errors (”bugs”), the second-most critical class detects more subtle vulnerabilities, the third class of rules detects so-called ”code smells”, which are a set of coding style conventions and low-impact flaws in the usage of the respective programming languages.

In principle, SonarQube rules can be altered or defined arbitrarily by e.g. the testing lead, but a large fraction of the rules predefined in the standard SonarQube installation are applicable to most software projects. The revision of rules for the three aforementioned programming languages is currently ongoing with the aim of keeping all rules that concern either bugs or vulnerabilities and the most important coding style or convention rules while disregarding rules that express more subjective style conventions in order to lower the number of reported issues to a reasonable minimum. Also CTA-level conventions[11] yet need to be implemented by means of SonarQube rules.

For any SonarQube project, the maximum number of bugs, vulnerabilities and code smells that is considered acceptable for a project can be defined. If a project violates such a threshold, the code would be rejected for a release unless it is revised.

As mentioned earlier, a large prototype code base exists for the OES software system. For these projects it is required that newly added code fulfils quality requirements, while the historically grown code base needs to be modified only at a later stage. The latest possible moment in time is before an initial release of the software system is scheduled.

A second measure reported by SonarQube for every project is the code coverage through unit tests, measured as a percentage of code branches. The underlying calculations are performed by independent tools like *Coverage* in case of Python or *JUnit* in case of Java. In case of Java projects, a Java module based on JUnit was created with the purpose to create test reports in XML format for a given Java source code module. This utility is executed by the Jenkins setup. The reports created by this tool can be imported and represented graphically by SonarQube. Based on the coverage measure, a software component can be accepted or rejected for a release. The threshold percentages for such a decision on OES components are yet to be defined. At the time of writing, the default value of 80 % is used. In Fig. 3, the SonarQube summary line of an example project is shown. The calculation of the individual ratings shown on this image will not be discussed here. The reader is referred to the SonarQube documentation referenced above instead.

4. SUMMARY & OUTLOOK

An SDLC process applicable for the development of the OES software system has been introduced. The purpose of this process is to guarantee that a good-quality software system is produced that fulfils all requirements.

[‖]https://www.sonarqube.org/ (2018)

For the newly introduced iterative and incremental development process, the involved roles are identified and assigned tasks. A sequence of phases is defined, with associated entry and exit criteria, deliverables and work assignment for individual roles. Software testing is highlighted in this process. Testing occurs in several phases. In this document, the deployment of the continuous integration tools Jenkins and SonarQube on a dedicated OES software testing cluster and the configuration for the needs of this system is described.

The adoption of the entire CI infrastructure and automated feedback for OES developers has begun only very recently. After an initial testing phase, which is currently ongoing, a wider usage by OES developers and potentially by developers from other CTA work packages is foreseen. Results of the static code analysis, code coverage calculations and unit test evaluations will be a crucial element in the acceptance procedure conducted when the final OES software system is handed over to the CTA Observatory.

Apart from unit tests, also the development of the operator GUI mentioned above needs to be quality controlled. In the coming months, options for automatic GUI testing will be explored and integrated into the OES test cluster environment.

All software testing approaches introduced above correspond to the implementation and testing phase introduced in Sec. 2. As mentioned earlier, also higher-level tests, like integration tests, system tests, performance tests or stress tests, need to be executed to ensure that a system that conforms requirements is produced and delivered. Such advanced software tests are currently prepared and will be presented as results become available.

APPENDIX A. DEFINITION OF SDLC TERMS

In the main text, several terms that have a well-defined meaning in the context of this work and in software engineering in general are defined. In Tab. 2, the roles identified necessary in the OES SDLC plan are specified and described briefly. In Tab. 3, terms referring to software products or SDLC phases are described.

ACKNOWLEDGMENTS

This work was conducted in the context of the CTA OES Work Package. We gratefully acknowledge financial support from the agencies and organizations listed here: http://www.cta-observatory.org/consortium_acknowledgments

REFERENCES

[1] Oya, I. et al., "Designing and prototyping the control system for the Cherenkov Telescope Array," (2017). ACAT 2017, Seattle.

[2] Colomé, J., Colomer, P., Campreciós, J., Coiffard, T., de Oña, E., Pedaletti, G., Torres, D. F., and Garcia-Piquer, A., "Artificial intelligence for the CTA Observatory scheduler," in [*Observatory Operations: Strategies, Processes, and Systems V*], *Proc. SPIE* **9149**, 91490H (Aug. 2014).

[3] Sadeh, I., Oya, I., Schwarz, J., Pietriga, E., and Dezman, D., "The Graphical User Interface of the Operator of the Cherenkov Telescope Array," *ArXiv e-prints* (Oct. 2017).

[4] Žagar, A., Murach, T., Leben, U., Dežman, D., et al., "CTA OES Software Development Life-Cycle," (2018).

[5] Royce, W., "Managing the Development of Large Software Systems," *Proceedings of IEEE WESCON* , 1–9 (August 1970).

[6] Forsberg, K. and Mooz, H., "The Relationship of System Engineering to the Project Cycle," *Proceedings of the First Annual Symposium of National Council on System Engineering* , 57–65 (October 1991).

[7] Larman, C. and Basili, V. R., "Iterative and incremental developments. a brief history," *Computer* **36**, 47–56 (June 2003).

[8] Oya, I., Füßling, M., Oliveira Antonino, P., Conforti, V., Hagge, L., Melkumyan, D., Morgenstern, A., Tosti, G., Schwanke, U., Schwarz, J., Wegner, P., Colomé, J., and Lyard, E., "The software architecture to control the cherenkov telescope array," *Proc. SPIE* **9913**, 9913 – 9913 – 15 (2016).

[9] Chiozzi, G. and Šekoranja, M., "ACS: ALMA Common Software." Astrophysics Source Code Library (Feb. 2013).

Role	**Description**
Auditor	Conducts a systematic examination of products for compliance with criteria (requirements, . . .) required for passing a certain validation process
Domain expert	Has in-depth knowledge of the software system's domain of use
Project manager	Plans and coordinates work, ensures the availability of required resources, and manages project constraints (cost, time, scope and quality)
Release manager	Plans and manages the software system release schedule and scope through all phases of the software development life cycle
Requirement analyst	Gathers, analyses and documents the needs and expectations of the software system stakeholders
Risk analyst	Identifies potential adverse events (HW/SW crashes, accidents, malicious use, etc.)
Software architect	Specifies the high-level system design based on requirements: Decomposition into simpler subsystems and components, selecting technologies, . . .
Software designer	Further refinement of the system architecture into the software system design that is detailed enough to permit implementation
Software developer	Implements software components based on detailed design. Provides unit tests
Software system integrator	Integrates software components and subsystems into higher-level subsystems or systems and performs integration testing
Stakeholder	Entity that may affect or be affected by the delivered software system
Tester	Executes well-specified tests and test plans
Testing lead	Responsible for specification and implementation of software tests. Ensures testability of requirements and specifications, specifies test cases at the software architecture and design level, . . .

Table 2: List and description of roles identified to be necessary in the SDLC plan produced for OES. These roles do not translate directly to distinct, individual persons. In fact, an individual may represent several roles, and several people with similar roles may cooperate.

[10] Lo, S.-L. and Pope, S., "The Implementation of a High Performance ORB over Multiple Network Transports," in [*Middleware'98*], Davies, N., Jochen, S., and Raymond, K., eds., 157–172, Springer London (1998).

[11] Dazzi, F., Füßling, M., Kosack, K., et al., "Programming standards," (2016). SYS-STAND/161012.

Term	Description
Architecture	Defines the system structure according to the requirements
Component	Unit with contractually specified interfaces and explicit context dependencies only
Design	Software design refines the architecture by defining interfaces between subsystems and components and their interaction. Defines data structures, algorithms, ...
Evaluation	Systematic determination of the compliance of an entity with its specified criteria
Qualification testing	Performed to demonstrate that a software system meets specifications
Review	Assessment of activities and verification of their outcomes against requirements
Risk register	Risk register is a record of information about identified risks. Defines risk categories, impact, probability, mitigation plan, ...
Subsystem	Unit of composition that is a system itself, while being a part of a larger system
System	Composition of interacting or independent subsystems and components
Test case	Set of test inputs, execution conditions and expected results suited for a particular objective
Test specification	Description how the system will be tested. Ensures a complete coverage of requirements and specifications. Consists of a number of test cases
Test plan	Further refinement of test cases described in the test specification, explaining step-by-step procedures and pass/fail criteria
Verification	Ensures that the system was built according to requirements and specifications
Validation	Ensures that the system meets user's needs and expectations

Table 3: Description of terms used in the main text. This is an excerpt from the OES SDLC plan.[4]