Challenges and strategies for the maintenance of the SKA Telescope Manager

Dolci, Mauro, Di Carlo, Matteo, Smareglia, Riccardo

Mauro Dolci, Matteo Di Carlo, Riccardo Smareglia, "Challenges and strategies for the maintenance of the SKA Telescope Manager," Proc. SPIE 9913, Software and Cyberinfrastructure for Astronomy IV, 99132J (8 August 2016); doi: 10.1117/12.2231642

Event: SPIE Astronomical Telescopes + Instrumentation, 2016, Edinburgh, United Kingdom

Challenges and strategies for the Maintenance of the SKA Telescope Manager

Mauro Dolci*[a], Matteo Di Carlo[a], Riccardo Smareglia[b]
[a]INAF – Osservatorio Astronomico di Teramo, Via Maggini snc, I-64100 Teramo, Italy; [b]INAF – Osservatorio Astronomico di Trieste, Via G. B. Tiepolo 11, I-34143 Trieste, Italy

ABSTRACT

The Square Kilometre Array (SKA) is an ambitious project aimed to build a radio telescope that will enable breakthrough science not possible with current facilities over the next 50 years. Because of this long expected operational period, the maintenance of Telescope Manager (TM), the SKA Element responsible for the coordination of all Elements composing the Telescope (e.g. Dishes for mid-frequency or Low-Frequency Aperture Arrays), plays a crucial role for the overall SKA operation. A challenge is represented by the technological evolution in hardware and software, which is rather fast nowadays: only in the last 10 years, for instance, new operating systems were born, as well as new technologies for data storage and for calculation. Dealing with such changing environment deserves therefore a deep analysis in terms of maintenance. In spite of the importance of hardware maintenance for TM, its software maintenance is actually the real challenge, given TM is a system almost entirely composed by software applications. In computer science, indeed, it is almost impossible to build a software which does not need to be changed over time: new requirements emerge, old requirements change during application lifetime, errors are discovered or performance must be improved. For all these reasons the management of software changes is critical to maintain the value of the software developed, especially for a complex system like SKA TM.
In this paper the maintenance for both SKA TM hardware and software is presented with respect to the Operational (i.e. related to Maintenance Process) and Organizational (i.e. related to Logistic Support) aspects.

Keywords: SKA, Telescope Manager, software development and testing, maintenance processes

1. INTRODUCTION

The Square Kilometre Array (SKA) is an ambitious project to build a radio telescope that will enable breakthrough science and discoveries not possible with the current facilities with a lifetime of around 50 years[1][1]. In the overall SKA architecture, each of the two telescopes (SKA MID and SKA LOW) is composed by several Elements covering all required functionalities: DISH and MFAA (Mid Frequency Aperture Array, for SKA MID) and LFAA (Low Frequency Aperture Array, for SKA LOW) are the front-end Elements for direct radiation detection, while elements such as CSP (Central Signal Processor), SDP (Science Data Processor), SAT (Synchronization And Timing), INFRA (Infrastructure) and SaDT (Signal and Data Transport) are devoted to all other operational and support functionalities. The global orchestration of this huge system is performed by a central element called Telescope Manager (TM).
The TM has three core responsibilities[1]:

1. management of astronomical observations;
2. management of the telescope hardware and software subsystems in order to perform those astronomical observations;
3. management of the engineering data to support operators, maintainers, engineers and science users in achieving operational, maintenance and engineering goals. Note that TM core responsibilities do not include management of science data products (visibilities, images, catalogues), which are the responsibility of Science Data Processor (SDP) Element.

To meet these responsibilities, TM performs high-level functions, namely: proposal handling, observation management, telescope management and engineering data management (including self-monitoring data).
To support proposal handling, the TM provides tools for proposal generation and submission by the science user, and for the evaluation, assessment and approval of and time allocation for the proposals by the proposal evaluation committee.
While performing observation management, the TM allows scientists or Principal Investigators (PIs) to generate Program Blocks, which consist of Scheduling Blocks (SBs). The SBs contain information needed to schedule and

Software and Cyberinfrastructure for Astronomy IV, edited by Gianluca Chiozzi, Juan C. Guzman, Proc. of SPIE Vol. 9913, 99132J · © 2016 SPIE
CCC code: 0277-786X/16/$18 · doi: 10.1117/12.2231642

execute observations for a Proposal. The SBs are then scheduled for execution. As observations are made, the TM provides the current telescope configuration and dynamic status to the SDP element for annotating the matching scientific data or visibilities.

While performing telescope management, the TM controls the appropriate elements, and collects monitor data that is used to track the entire status of the Telescope including element status, site security, weather monitoring, site power supply, etc. The TM manages engineering (monitoring and configuration) data as a system model that describes the status of the telescope at any one time. The TM continually collects telescope configuration, telescope dynamic status and environmental data. The data are time-stamped and stored. The TM provides the data to users as the current and historic state of the system to support operations and maintenance.

TM is a complex (and distributed) system, mostly composed by software packages (TELMGT, OBSMGT[4][5], LMC[6]), web applications (e.g. proposal submission tools designed in OBSMGT) and user interfaces running on a hardware & virtualization software platform provided and managed by LINFRA[7]. Each TM sub-element will be composed in turn by software (applications) and data (configuration data, initialization data, etc.).

Given the central role of TM, a proper and efficient maintenance of it plays a fundamental role for the overall SKA operation. In this paper the maintenance of the both the hardware and software of SKA TM is discussed. Since TM is composed mostly of software applications, special attention is devoted to software maintenance plan, from both the operational and organizational aspects.

2. TM DEPLOYMENT AND ARCHITECTURE

The Deployment Concept for TM Equipment is based on the following defined locations:

1. Central Processing Facility (CPF). This is near the core of the telescope and will contain the CSP. It will be located in the Karoo Central Astronomy Advantage Area (KCAAA, South Africa) for SKA1-Mid and in Boolardy (Australia) for SKA1-Low.
2. Engineering Operations Centre (EOC). This is envisaged as a centre where engineering operations can be coordinated. It will be located near (but outside) the RF restricted area of the telescope, i.e. in Klerefontein (South Africa) for SKA1-Mid and in Geraldton (Australia) for SKA1-Low.
3. Science Operations Centre (SOC). This is where the operational control of the Telescope will take place. It will be located inside the major city for each Telescope, i.e. Cape Town (South Africa) from SKA1-Mid and Perth (Australia) for SKA1-Low.
4. SKA Global Headquarters (SKA-GHQ). This is where the general management of the Telescope Scheduling and Operations will take place. The SKA GHQ are located in Jodrell Bank (UK).

The TM architecture concept is, in general, to have a central deployment in which the online equipment are all located at the same location. This will be the CPF.

If the facility provides enough infrastructure then the offline and archiving equipment can also be located there but the design can be flexible in that the archiving equipment can be remote from the CPF. Currently the proposal is to have the option to deploy offline equipment at the EOC but this can also be located anywhere else or even implemented by means of a cloud based storage service.

The display equipment that will contain the primary operator interfaces will be contained where the SOC is located. Auxiliary display components (for example engineering UI components) and possibly the Engineering Data Archive and Forensic Tool could be placed at the EOC.

The current physical architecture of TM can therefore be summarized into three deployment categories:

1. Online Systems: OBSMGT, LMC and TELMGT components that are directly involved in coordinating instrument functioning. Online systems must be situated at the CPF, for system robustness, so that failures in the long distance links to SOC and EOC do not leave the instrument in an uncontrolled state.
2. Offline Systems: OBSMGT, LMC and TELMGT components that provide offline support functions for SKA. This includes the proposal submission, scheduling, observation preparation, forensics, simulation, development and test environments, and any associated LMC components, as well as the data archives (EDA, proposal and project databases). Several of these components will be located at SKA GHQs. As to the deployment location for these systems in Australia and South Africa, instead, it is flexible: the current plan is to split them between the EOC and CPF depending on power availability.

3. Displays: containing all user interfacing functionality (may consist of a main part as well as an auxiliary part). Although display functionalities will have to be put at SKA GHQs, the current plan is to put primary operator displays at the SOC, while there may also be auxiliary displays at the EOC, so that the functionality is close to the user base.

3. HARDWARE MAINTENANCE

Hardware Maintenance is not necessarily based upon alarms or problems detection only. It could also be performed on a regular basis, related to the normal degradation of characteristics and performance of hardware devices, or on the basis of a trend, monitoring data-based prediction.
However, it appears unlikely that ILM and DLM will really be required for TM hardware components, as they will all be COTS such as computer desktop, servers and storage: in case of failure these COTS modules will be returned to the supplier (under warranty) and/or replaced by a new module.
Three types of Hardware Maintenance are identified:
1) (Hardware) Corrective Maintenance (CoM) is the set of tasks aimed at Failure Detection, Isolation and Recovery (FDIR) of a system. It corresponds to Operational State 3 in the SKA1 System Operational Model. CoM on TM hardware shall be generally accomplished through the replacement of Line Replaceable Units (LRU's) or Shop Replaceable Units (SRU's) at O-level. If planned, the replaced item shall undergo a repairing process at I-level. Such a process shall not be performed in-site, but moving the item to be repaired to the planned ILM location.
According to the severity of the failure (critical or non-critical), immediate or deferred CoM shall be performed, according to the response time and intervals defined in the SKA System Operational Model.
2) Preventive Maintenance (PrM) is the set of activities and procedures, including test, adjustments and parts replacement, performed specifically to correct incipient failures before they occur or before they develop into major defects, with the ultimate goal to prevent faults from occurring. It corresponds to Operational States 4 and 5 in the SKA1 System Operational Model. The importance of Preventive Maintenance even for TM hardware (which is expected to be entirely made of COTS components) lies in the fact that is does not impact the observation activities at all, since it is performed during devoted service periods, in comparison with the impact a Corrective Maintenance can definitely have in case of unexpected failure and need of hardware LRU replacement in a remote site.
The detailed definition of the PrM tasks, including procedures, required personnel and supply, is based on the results of RAM and FMECA analysis and shall be given at a later stage of the project.
PrM on TM hardware shall be generally accomplished through the replacement of Line Replaceable Units (LRU's or Shop Replaceable Units (SRU's) at O-level, taking into account the recommendations about limited human presence at sites (for RFI protection purposes).
3) Predictive Maintenance (PdM) represents a special sub-category of Preventive Maintenance, given the intrinsic nature of TM as a continuously monitored element. An advantage of PdM, with respect to PrM, is represented by the possibility to get up-to-date indications of the system behavior and performance trend and therefore to optimize the maintenance process, e.g. by reducing the frequency of "preventive" replacement of LRUs. However, the impact of such approach on supply support (spare availability, location, and so on) and operation schedule, will have to be analyzed in detail. On one hand, the improvement introduced by PdM with respect to PrM is probably not so relevant (RAM data are expected to comply with the observed failure occurrences so that trend indications from system monitoring will not substantially differ from RAM expectations); on the other hand, PdM implies a much higher risk for unscheduled interventions in remote locations (similar to CoM), which definitely impacts the operation schedule. PdM seems in conclusion a very interesting task, but not recommended for TM Hardware.

It must be noticed that, although dependability analysis for hardware could appear to provide indications on pure hardware failures, the consequences of hardware malfunctioning on software operations or performance have to be seriously taken into account. In general, hardware defects (even temporary, like for example a wrong memory addressing) are the source of many software stops or even crashes (Hardware-Software Interactions, HSI).
The importance of HSI has been addressed since much time[8] and led to the birth of Software Failure Modes Effect Analysis (SFMEA), to be performed in conjunction with the more widespread hardware FMECA. TM is mostly a software-based system and therefore SFMEA is expected to be as important as (or even more than) hardware FMECA.

3.1 Hardware Maintenance process

The process for hardware changes is shown in Figure 1.

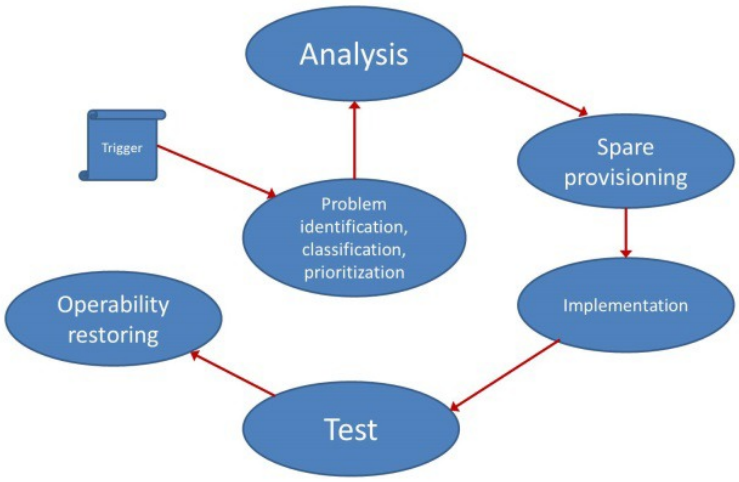

Figure 1 - Hardware Maintenance Process

The process starts with a trigger, usually composed by either a set of alarms (which will imply Corrective Maintenance) or warnings (suggesting Predictive Maintenance actions) or scheduled calls (Preventive Maintenance), and includes the following phases:

1) *Problem identification, classification, and prioritization*. The detected problem associated to the trigger is evaluated to determine its classification in terms of Corrective, Preventive or Predictive Maintenance, to establish its handling priority and to finally assign its solution to a maintainer. The result of this activity is stored into a repository which contains at least the following items: identification number, statement of the problem, type of maintenance required, initial priority, initial estimate of resources required to implement the change.

2) *Analysis*. The feasibility and scope of the modification is studied (*feasibility analysis*) and a plan for spare provision, implementation, test and final restoring of the system operability is created (*detailed analysis*). The feasibility analysis is aimed at generating a feasibility report that will contain the impact of the modification action (e.g. in terms of human footprint at the site), alternate solutions, safety and security implications and costs. The detailed analysis, on the other hand, generates an analysis report that will identify the faulty LRUs (if any) and the spares needed, the Level of Maintenance required, the team(s) needed and the safety and security issues. An implementation and test plan will also be made available at the end of this phase.

3) *Spare provisioning*. The spare LRUs are provisioned to the endorsed maintainers, according to the analysis report.

4) *Implementation*. Once the team(s) are defined, the spare are provisioned and the Level of Maintenance has been identified, the changes are implemented according to general prescriptions concerning system maintenance mode, possible operation stopping, RAM requirements and RFI protection of sites. It is important to notice that the implementation of a computer hardware change usually requires even software operations, like firmware check, software configuration, drivers loading, processes and service launching and so on.

5) *Testing*. At the same site where implementation takes place, final tests on the updated component operation have to be made. These can imply physical measurements (e.g. in case of replacement of an electrical component) or software-based interrogations of the system (for computer hardware changes). All testing equipment of electronic nature (e.g. oscilloscopes) will have to be compliant with the sites RFI requirements. At the end of the

Implementation and Testing phase, the reference documentation (hardware manuals, design, test and user documentation, training material) is updated and one can proceed at 6) *restoring the system to its full operability*.

A schematic time development of a full hardware maintenance process is shown in Figure 2.

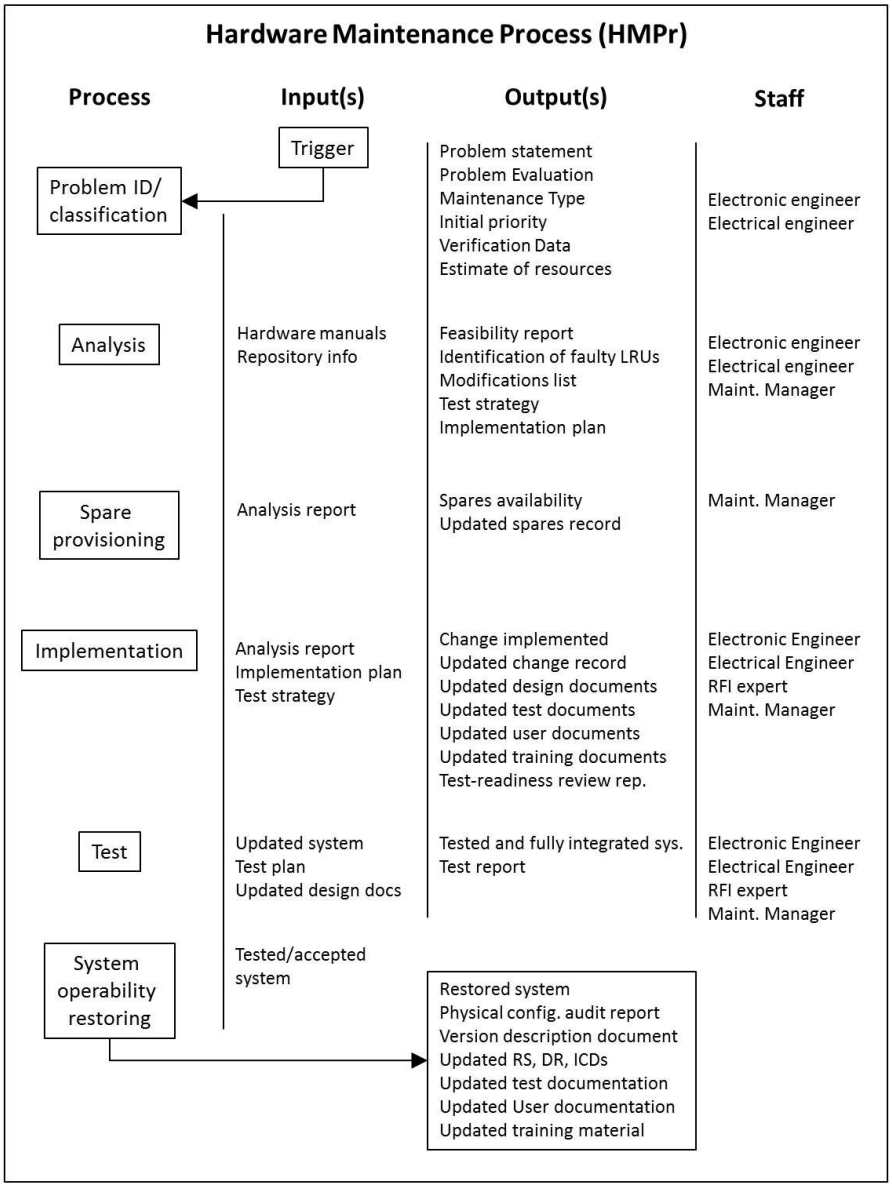

Figure 2 - Full Hardware Maintenance Process time development. Input and output data and reports, as well as staff profiles, are reported in 2nd, 3rd and 4th column respectively.

4. SOFTWARE MAINTENANCE

Unlike Hardware Maintenance, Software Maintenance is generally triggered by a specific request, aimed at solving a detected problem (e.g. bug-fixing) or at making a software package compliant with a changed (or changing)

environment (e.g. a change of a software platform or an hardware upgrade) or, finally, at improving the system performances (e.g. lowering execution latencies).
Three types of Software Maintenance, all based on scheduled operations, can be identified accordingly: (Software) Corrective Maintenance, Adaptive Maintenance and Perfective Maintenance. A forth type of maintenance, namely Software Emergency Maintenance, must be foreseen defined for unscheduled operations.

1) (Software) Corrective Maintenance (SCoM) is defined as the reactive modification of a software product, performed after its delivery, to correct discovered faults. Since it has to be scheduled, the fault cannot block the system. When such a catastrophic situation occurs, a Software Emergency Maintenance process has to be started (see below).
2) Adaptive Maintenance (AdM) is defined as the modification of a software product, performed after its delivery, to keep a computer program usable in a changed or changing environment.
3) Perfective Maintenance (PfM) is defined as the modification of a software product, performed after its delivery, to improve its performance or maintainability.
4) Software Emergency Maintenance (SEM) is defined as a set of corrective actions, put at the highest priority level in the maintenance process (see next sections), to be performed immediately in order to restore system operability. The nature of this type of maintenance is very close to hardware CoM.

4.1 Software maintenance process

The standard process[9] for changes in software, called Software Maintenance Process (SMPr), is shown in Figure 3.

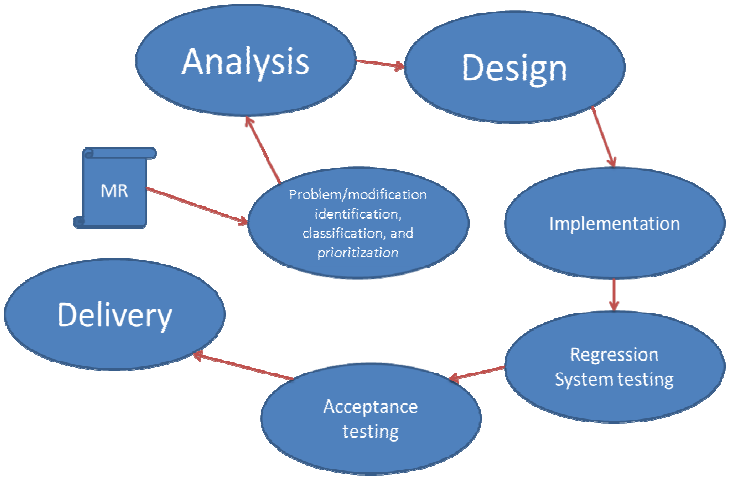

Figure 3 - Software Maintenance Process

It includes the following phases:

1) *Problem/modification identification, classification, and prioritization*. The SMPr starts with a Modification Request (MR). This is a generic term that includes the forms associated with the various trouble/problem-reporting documents (e.g. incident report, trouble report) and the configuration change control documents. In this phase, each MR shall be evaluated to determine its classification and handling priority. An MR can be classified into the already-mentioned software maintenance types: *Corrective, Adaptive, Perfective* and *Emergency*. At the end of this phase, the MR is assigned to a maintainer and all the information produced (e.g. identification number, statement of the problem, type of maintenance required, initial priority, initial estimate of resources required to modify the system) is stored into a repository, to be used as input for the next phase.

2) *Analysis*. The main purpose of this step is to study the feasibility and scope of the modification and to create a preliminary plan for design, implementation, test and delivery. Interaction with the user who created the request could be needed. As for the hardware process, the software maintenance analysis phase is split into *feasibility analysis*, aimed at evaluating impact of the modification, alternate solutions (including prototyping), safety and security implications, costs, final benefit of the modification, and *detailed analysis*, whose goal is to define requirements for the modification, identify the elements of modification, identify safety and security issues, devise a test strategy and develop an implementation plan.

3) *Design*. In this step the software modules affected by the modification are identified, the software module documentation is modified, test cases for the new design (including safety and security issues) are created, regression tests are identified and documentation (system/user) requirements are updated. It is important to note that the possibility of a major software change could arise in this phase (for example the need for re-engineering a component or something even bigger). In these cases, an authorization request has to be submitted to an Authority Team (see Section…).

4) *Implementation*. Modification is implemented in this phase, basing upon the results of the previous ones and the current source code. As the change is implemented into the code, unit testing and other software quality assurance process have to be performed. The final result of this phase should include up-to-date software, design documentation, test documentation, user documentation and training material.

5) *Regression/system testing*. The main goal of this phase is to ensure that software does not crash and that it meets the documented requirements. System testing should be performed, already during this phase, on the fully integrated and modified system. Typically tests are prepared by developers in conjunction with software analysts starting from the input received by the MR. System functional test, Interface testing, Regression testing and Test-readiness review (to assess preparedness for acceptance testing) are some of the basic tests that must be performed.

6) *Acceptance testing*. Its goal is to verify that the solution proposed for the specific MR is good for the final user, i.e. to determine if the requirements are met. It must be performed on the fully integrated system, possibly by either the final user (User Acceptance Testing, UAT), the specific user of the modification package, or a third party designated by the stakeholder. Usually the tests are performed within test scenarios which reproduce the typical usage scenario for the specific user. Another kind of acceptance test is the Operational Acceptance Test (OAT). It is a non-functional system testing used to check the operational readiness of a software product, service or system. Examples of this test are backup/restore, disaster recovery, maintenance tasks and periodic check of security vulnerabilities and so on. Typically these tests are performed by system administrators.
Once the acceptance test is passed, the modification package can go to the final phase of 7) *Delivery*.

A schematic time development of a full software maintenance process is shown in Figure 4.

4.2 Software Maintenance Techniques

A software maintainer can use different techniques, like Re-engineering, Reverse engineering and Holistic reusing.

1) *Re-engineering*. During the lifetime of the maintenance of a software application, it is possible that some of the modifications of the source code are treated as a minor part of the development process and delegated to less experienced programmers. This happens because most of these changes are less important than others or because it is assumed that they have a short life span. The mass of changings can become significant, however. When this happens, the re-engineering technique can become an important part of the software maintenance. The re-development of key systems can be very expensive not only economically but also in terms of development time but, at the same time, can bring old systems up to current standards and supporting new technologies. When re-engineering a software or a system, it is also important to think in terms of providing reusable material for future systems. Generally the re-engineering technique is composed by a reverse engineering (see below) and forward engineering which is the process of system building.
2) *Reverse engineering*. This is a technique for documenting a project which has source code as the only reliable representation. This generally happens when dealing with a long-lifetime software application because often changes occurred during this time did not bring to a documentation update. With this technique, it is possible to retrieve an alternate view of the system with schematics, structure chart, flow diagrams, etc. to assist in

understanding the logic of the code. Additionally this process offers opportunities for measurement, problem identification and formulation of corrective procedures.

3) *Holistic reusing*. It consists in delivering a new system from a functioning system. This can happen when a parent system during its lifetime gives birth to a new stand-alone system that has its own individuality. One reason can be the importance of not interrupting the online system so engineers think to separate the initial system into more components that have their own life; in this sense updating a component does not need to stop the application as a whole but only the component to be updated while the system continues its work.

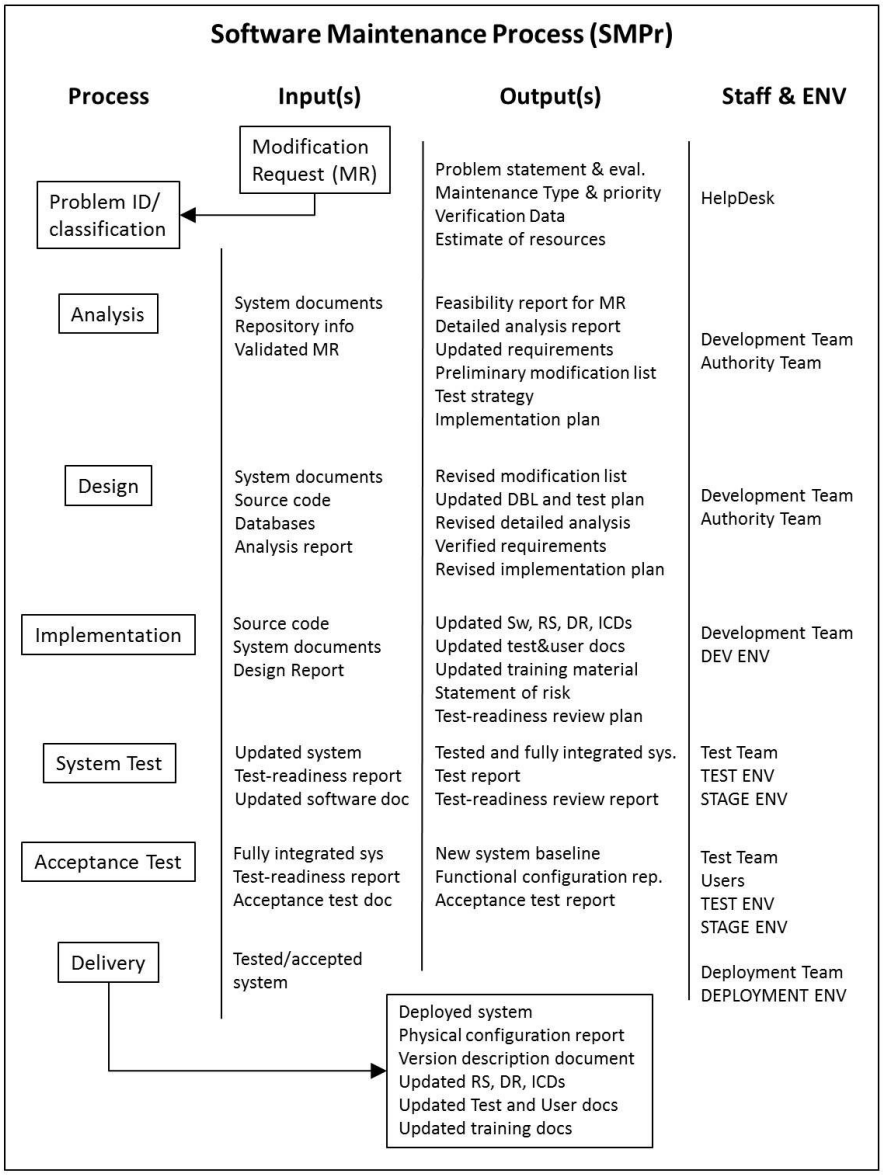

Figure 4 - Full Software Maintenance Process time development. Inputs, outputs, required teams and environments are reported for each stage.

5. ORGANIZATIONAL ASPECTS

5.1 Maintenance Teams

The organization of both hardware and software maintenance must include devoted teams working at different levels of maintenance and, in case, at different geographical locations. The teams should have nearly the same duties and composition, being replicated and made available for each of the sites where O-level, I-level and D-level maintenance is expected to be performed (S-level maintenance is outside this aspect of the organization).

Given the prescriptions for limited human footprint presence at observatory sites (primarily for RFI-related reasons), each *hardware maintenance team* working at O-level should be composed by a minim number of people (at most three) covering the following competences: 1) electronic engineering for LRUs treatment; 2) electronic/telecommunication engineering for RFI treatment on site; 3) electrical engineering for electrical safety-aspects.
Maintainers should be experienced people with a good knowledge and understanding of the system hardware. This means that when a trigger arrives they are prepared to understand the type of problem and to define quickly the corrective actions to be adopted accordingly. To be able to do that, a regular training program will be adopted, making extensive use of complete and exhaustive hardware documentation composed by manuals, schemes, photos, diagrams, maps, and so on, as well as local simulators (e.g. to experience dismounting and mounting detailed operations).

The organization of *software maintenance teams* appears more complex (see Figure 5). The process must be based on the following teams:

1) *Help Desk Team*. It is the first one to react to a MR as it is issued. It carries a very preliminary analysis of the MR and provides a quick reply to whom has issued the MR.
2) *Development Team*. It carries out the activities foreseen for the phases of Analysis, Design and Implementation. In every phase it will be possible to have interaction with the person who raised the MR.
3) *Test Team*. It works on tests coming from different MRs. At least two test levels will be foreseen (system test and acceptance test), possibly carried by a different person of the team.
4) *Authority Team*. It must approve (or reject) the modification proposed after the analysis of the MR before being done. Similarly, at the end of the process, system test results must be evaluated by the Authority Team before the final acceptance test which should be performed by the same person who raised the MR.

These teams, not necessary geographically distributed, should work at different levels of knowledge. The maintainers should be experienced people with a good understanding of the software architecture from every point of view: when a request arrives they have to be prepared to understand the type of request and which are the corrective actions to be adopted accordingly. The ability to identify the request, as well as a good analysis capacity, are of the utmost importance because relate to the most time consuming activities for a SMPr – issue identification and analysis.
To own such a quality, usually a maintainer should have good experience with the development of the software being maintained, or at least be in connection with the software developer. An history of the project maintenance plays therefore a crucial role, and maintainers could be asked to know in details only the system versions starting from some epoch (e.g. over the last 2 years).

In general, training of younger maintainers by senior ones will be of fundamental importance. Given the very long lifetime expected for the project (50 years), risks associated with turnover (retirements and resigns) will have to be taken into careful account.

5.2 Hardware Asset Management System

Every hardware change/modification/replacement performed during the project lifetime has to be tracked and controlled. An *Asset Management System* (AMS) will be implemented for managing the logistics of maintenance, the spare provisioning and related contracts, as well as for tracking the system documentation and supplying real-time information (concerning installed parts, fault conditions and changes in system documentation) to the System Model.
In particular, a Change record will allow to keep up-to-date documentation about the hardware. It will essentially list the technical and manufacturer data of the replaced components and the replacing ones: manufacturer name, product number, serial number, date of supply, release notes, etc. for the hardware, as well as data (e.g. version number) concerning the firmware installed on-board, and so on. The change record will list also the expected RAM data for the

replaced component and the replacing one (typically Mean Time to Fail), as well as the recorded life time (from installation to failure) and the fraction of usage, and a series of detailed information about the failure that has occurred. The data will help to update both the maintenance plan and the RAM data for the system under examination.

5.3 Software Configuration Management System

Tracking and controlling changes in source code, configuration items, documentation, etc. as a consequence of not only corrective maintenance (e.g. bug fixing) but also perfective maintenance (e.g. following a change in requirements) is of fundamental importance in software maintenance. A Configuration Management System (CMS) will be implemented in order to help the software administrators. It includes configuration identification (for instance identification of baselines), configuration control (which is the ability to accept or reject a MR of a baseline), configuration reporting (records all the information about the development process), Environment management (both software and hardware), Build management (managing the tools to make a build of the application), Teamwork (tools and practices for the team to work together) and defect tracking (every MR should be traceable back to the source). Examples of tools available for this purpose are represented by the open source packages Puppet[10] or Chef [11].
Other fundamental tools that a CMS must include are the Issue Tracking System (ITS), which manages and maintains lists of issues (or MRs) usually through a ticketing system (e.g. Jira[12]), and the Version Control System (VCS), used by developers to keep track of all the modifications made in their source code (e.g. Apache subversion[13] or Git[14]).

5.4 Deployment Environment

In software deployment, an environment (or tier) is a hardware computer system where a computer program or software component is deployed and executed. In industrial use, it is quite common to separate development environment (where changes are originally made) from production environment (where computer programs are used by end-users), often with different stages in between so that it is possible a phased deployment, testing and rollback in case of problems. Common tiers are development, testing, staging, production (Figure 5).
Development Environment (DevEnv) is where changes are originally made by developers who receive the MRs and are in the implementation phase of the SMPr. The most simple situation is when developers can make changes in their local workstation. In this case, however, as soon as the need for a simulator (or other shared resources locally unavailable) is raised, or if there are multiple developers, difficulties arise in keeping the latest version of the software in the developer's workstation. In a distributed software application, like SKA TM, it is recommended to avoid the complications and risks of always keeping the latest version in a local computer.
Testing Environment (TestEnv) has the goal to replicate the condition that brought to the discovery of a specific bug or anomaly. Test environments are continuously evolving and it is very important to own the test environment configuration and its usage details, making it challenging for maintainers. New possibilities came up with the birth of Cloud Computing and the Platform as a Service (PaaS): developers can ask for a specific test tier in the form of a virtual machine and having always an up-to-date platform. Note that since the database configuration in a particular moment can affect the test results, the maintenance of databases and configuration (database tables, stored procedures and models that can change during the lifetime of the software) will have to be done in turn.
Quality Assurance (QA) is another type of environment especially suitable for the acceptance test phase of the maintenance process.
Staging Environment (StagEnv) is an exact copy of the production environment (see below) used to test and review a newer version of the software before moving it to the production environment. It can even connect to (but not modify) the data stored in production environment and usually only a restricted number of users can use it.
Production Environment (ProdEnv) is the last one and allows the final users to use the software produced. Only software compiled in release can be in this tier and usually it is associated with a production support. The delivery phase of the SMPr is very sensitive in this environment because if hot swapping is not possible, deploying a new release generally requires a restart and therefore an interruption in service. If there is a redundancy mechanism, it is possible to start a new server and, when started, redirect all the traffic to the new server with the new software.

.

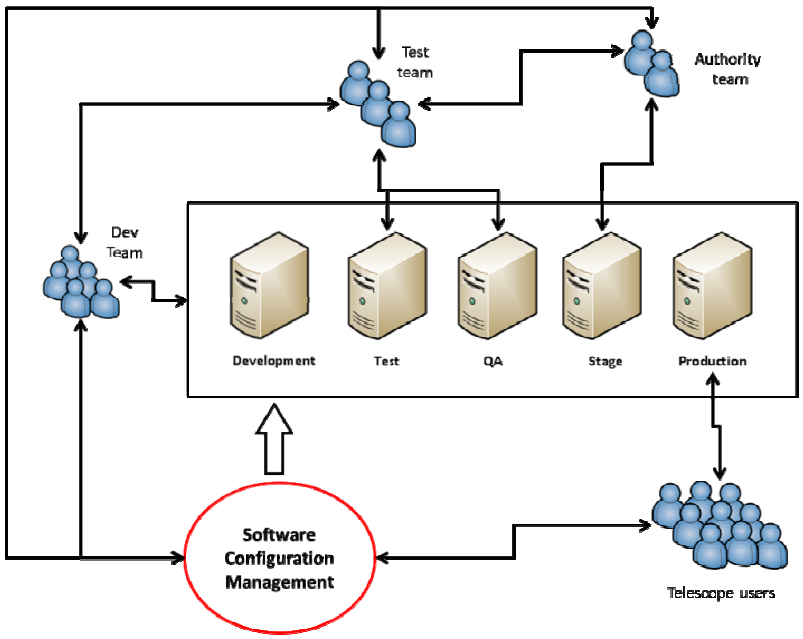

Figure 5: Software Maintenance organization model

6. PERFORMANCE METRICS FOR SOFTWARE MAINTENANCE

Although it is possible to define a performance metrics for each phase of the standard SMPr, the different maintenance phases are not completely separated each other, and the performance of each phase is somewhat depending on other phases, for example through the output of a phase taken as input by the next one, or because of the concurrent use of the same test environment for QA and Acceptance.

Defining a performance metrics for the overall SMPr could therefore be a better practice. There are several SMPr performance metrics that can work alone or be considered as factors for an overall, unique merit figure Q. Three of these metrics are related to the main working-quality factors of the SMPr shown in Figure 6 and described as follows.

For the a given software application/component, let us suppose that N_{MR} modification requests (MRs) are issued for maintenance. Once assigned to a maintainer, the *i-th* MR (MR_i) will undergo the various phases of the SMPr up to its closure, acceptance by user and final deployment, that will occur after a time τ_i from the initial issuing. This time will be the final budget of the time contributions by the various phases: among them, it is worth of notice to mention the number $N_A^{(i)}$ of iterations/interactions between the analyst(s) and the user(s) during the analysis phase, which depends not only on the maintenance, but also on the component properties.

Notice that the time to solve the issue, τ_i , is the equivalent of the hardware-related concept of Mean Time to Repair (MTTR), even if for Emergency and Corrective Maintenance only.

According to this description, and considering that among the outputs of the analysis phase a minimum expected time to close the issue $\tau_i^{(exp)}$ will have to be provided, a merit figure for the SMPr of the *i-th* MR of C component can be simply defined as

$$Q_i = \frac{\tau_i^{(exp)}}{\tau_i}$$

and will be a scalar number ranging from 0 to 1.

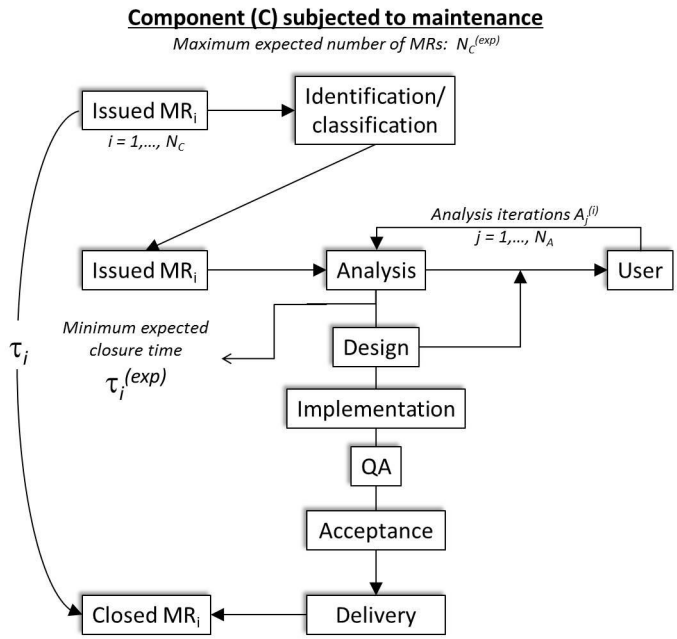

Figure 6: Example of working quality factors for SMPr

At overall component level, a merit figure for the SMPr of component C can be defined as the average of Q_i over the MRs globally issued for that component:

$$Q_C = \frac{1}{N_C}\sum_{i=1}^{N_C} Q_i$$

This definition of Q_C , however, does not allow to compare the maintenance process performance for components that differ as to software complexity and criticality (with respect to the rest of the system, a reliability-related concept), as well as to usage statistics.
It can help however to generate a metrics for getting indications about the quality of the component. Let us suppose indeed that during the design of component "C" a maximum number of MRs that can be issued, $N_C^{(max)}$, can be defined (for example, equal to the number of physical or logical code lines). The quality of the component C will then be proportional to

$$1 - \frac{N_C}{N_C^{(max)}}$$

i.e. it will be 1 (highest level) in case there are indeed no MRs issued, and will be 0 (lowest level) if the number of MRs issued is equal to the maximum (worst) expected number.
On the other hand, the number of MRs issued could depend on the fraction of usage time u_C (which can be known basing upon software monitoring data). Finally, the reliability of the component could be expressed by a weight function w_C describing the impact that component's problems should have on the rest of the system (w_C=1 for maximum impact and 0 for no impact), computed basing on the criticality analysis process (SFMEA) performed on the overall system.

In conclusion, a possible merit figure for the quality level SQ_C of a software component "C" could be derived from the maintenance performance figure, the monitoring data and the design aspects in the following way:

$$SQ_C = Q_C u_C (1 - w_C)\left(1 - \frac{N_C}{N_C^{(max)}}\right)$$

It is easy to verify that also SQ_C will have values in the range from 0 to 1.
The evaluation of the quality level of a software component, as a result of a Software Maintenance Process, can be of great importance in case of software re-engineering.

7. CONCLUSIONS AND FUTURE WORK

In this paper the operational and organizational aspects of the hardware and software maintenance of the SKA Telescope Manager have been described. From the operational point of view, the challenging aspects are related to the distributed nature of TM, as well as to its mostly software-based nature, which has consequences on the maintenance process, the maintenance time and the logistical support aspects (teams, environments, hardware spares availability and so on). On the other hand, the challenging organizational aspects are related to the need to ensure an efficient and fast maintenance process, so that the system possible downtimes are extremely reduced or, ideally, cancelled. This objective translates in proper strategies that must be adopted for the final design of TM, in such a way both maintenance and deployment could be performed without stopping the system operation. Currently the detailed design of TM is still ongoing and evolution and refinements are expected to some extent. This paper should therefore be seen as an overview of the current status of the maintenance strategy for TM, which will have to be refined, together with the TM design itself, in the future stages of the project.

Acknowledgements. The authors are grateful to the Italian Ministero dell'Istruzione, dell'Università e della Ricerca (MIUR) for the financial support to this work.

REFERENCES

[1] Alistair M. McPherson 2016: SKA Telescope update through re-baselining and preliminary design phase**,** *Proc. of the SPIE Astronomical Telescopes and Instrumentation 2016, paper no. 9906-75 (this conference)*

[2] Gary R. Davis, Douglas C. Bock, Antonio C. Chrysostomou, Cornelius Taljaard 2016: Operations concept for the Square Kilometre Array, *Proc. of the SPIE Astronomical Telescopes and Instrumentation 2016, paper no. 9910-17 (this conference)*

[3] Swaminathan Natarajan, Yashwant Gupta, Paul Swart, Gerhard LeRoux, Alan Bridger, Subhrojyoti Roy Choudhuri, Mauro Dolci, Domingos Barbosa, Lize Van den Heever, Juan Guzman, Sonja Vrcic 2016: SKA Telescope Manager (TM): Status and Architecture Overview, *Proc. of the SPIE Astronomical Telescopes and Instrumentation 2016, paper no. 9913-1 (this conference)*

[4] Stewart J. Williams, Alan Bridger, Subhrojyoti R. Choudhury 2016: The SKA observation control system, *Proc. of the SPIE Astronomical Telescopes and Instrumentation 2016, paper no. 9913-98 (this conference)*

[5] Alan Bridger, Stewart J. Williams, Mark Nicol, Pamela Klaassen, Roger S. Thompson, Cristina Knapic, Giovanna Jerse, Andrea Orlati, Marco Messina, Snehal Valame 2016: Observation management challenges of the Square Kilometre Array, *Proc. of the SPIE Astronomical Telescopes and Instrumentation 2016, paper no. 9913-35 (this conference)*

[6] Di Carlo, Matteo, Dolci, Mauro, Smareglia, Riccardo, Canzari, Matteo, Riggi, Simone: "Monitoring and controlling the SKA telescope manager: a peculiar LMC system in the framework of the SKA LMCs", *Proc. of the SPIE Astronomical Telescopes and Instrumentation 2016, paper no. 9913-117 (this conference)*

[7] Domingos Barbosa, João Paulo Barraca, Dalmiro Maia, Antonio Cruz, Gerhard Le Roux, Swaminathan Natarajan, Paul Swart, Yashwant Gupta 2016: A cyber infrastructure for the SKA Telescope Manager, *Proc. of the SPIE Astronomical Telescopes and Instrumentation 2016, paper no. 9913-20 (this conference)*

[8] R. K. Iyer and P. Velardi 1985: Hardware-related software errors: measurement and analysis, *IEEE Transactions on Software Engineering*, Vol. SE-11, issue 2, pp. 223 (1985)

[9] IEEE Std 1219-1998 Standard for software maintenance
[10] *Puppet* Configuration management: https://puppetlabs.com/solutions/configuration-management
[11] *Chef* configuration management: https://www.chef.io/solutions/configuration-management/
[12] *Jira* ticketing system: www.atlassian.com/software/jira
[13] *Apache subversion* version control software: subversion.apache.org
[14] *Git* version control software: git-scm.com