Modernized build and test infrastructure for control software at ESO: highly flexible building, testing, and automatic quality practices for telescope control software

F. Pellegrin, B. Jeram, J. Haucke, S. Feyrin
European Southern Observatory
Karl-Schwarzschild-Strasse 2, D-85748 Garching bei Muenchen, Germany

ABSTRACT

The paper describes the introduction of a new automatized build and test infrastructure, based on the open-source software Jenkins[1], into the ESO Very Large Telescope control software to replace the preexisting in-house solution. A brief introduction to software quality practices is given, a description of the previous solution, the limitations of it and new upcoming requirements. Modifications required to adapt the new system are described, how these were implemented to current software and the results obtained. An overview on how the new system may be used in future projects is also presented.

Keywords: automatic build and test, software quality, control software ESO, Jenkins

1. INTRODUCTION

Maximizing the observation time is definitely one of the most important quality goals at the ESO observing facilities. High uptime and fault-less operation of the software services related to the control of the telescopes is therefore an immediate consequence, leading to many activities pointed to assure quality in the development of software in the Control Software and Engineering department at ESO.

Introducing quality metrics and procedures from the early phases of software development is very important, as many studies analyze the exponential growth of the cost of fixing a bug in relation to the phase of lifecycle of the project[2]. Not just obvious malfunctions are important to catch, but also trends that may in the long time impair operations, such as computational resource usage, or maintenance, such as badly formatted code. At the same time, it is obvious that executing regularly manual quality controls can be a very expensive and tedious work. This work can be even more complicated when the team working on the project can be from different software cultures, as is the case of ESO where local developers, external contractors and Consortia teams are often tightly collaborating on the same project. Therefore, automatic quality checking procedures executed as early as possible into the software lifecycle are undoubtedly an important asset in modern software development.

Various operations can be, with a reasonable effort, automated in the software development process. First and most obvious is to maintain the code in a revision control system to keep code and track modifications over time. Checking if the modified code is still compiling is usually the natural next step. There are also many activities that can be done on the source code. These activities are usually classified under the umbrella of static checking, as the code is not really running. Part of this class of operations are for example coding conventions checking, such as a consistent code indentation scheme, analysis of source code for classic known problematic programming patterns, such as out of bounds array usage, and analysis of source code for complexity parameters, such as number of nested branches in a function.

Executing some tests on the compiled code is then on the line. In this case, a variety of tests families are executed: unit tests, to check the code on the modular level, regression tests, to check if the behavior changed from the previous run, integration tests, to check if the code works when put with other components, stress tests, if the code works in particular problematic conditions, and many others. Ideally, the tests should exercise as much as possible the code and we measure this with a metric usually named code coverage. While this dynamic inspection of code is done, resources can be monitored for possible leakage or excessive usage trends.

The timing of execution is also a very important decision. Ideally, an immediate execution, the so-called *continuous integration*, is the best road as an immediate feedback is given after a modification. But often the time to execute the operation needed, either a long build or a very comprehensive set of tests that require special hardware, can limit this possibility.

2. HISTORY ABOUT AUTOMATION IN CONTROL SOFTWARE AT ESO

The software team at ESO recognized the necessity for automating and regularly executing build, test and quality assurance tasks back in 2006. At that time, there were not many ready possibilities, either in the commercial or open-source world that could fit the necessities and peculiarities of the ESO software. Therefore, an in-house solution was studied and the NRI, Nightly Reporting Infrastructure, was born.

The NRI goal, as the name tells, was to execute all the quality related activities during the night and report anomalies to the developers at the beginning of the working day. The project was mainly written in Perl[3] and running on the GNU[4]/Linux[5] operating system distribution. It was using textual configuration files to determine and configure the activities to be executed and presented the output via a Web interface, while also sending error notifications immediately via email to actively notify the developers. One of the problems NRI had to tackle was the necessity to execute builds and tests on various platforms, as the Local Control Units (LCUs) present in the telescopes had for various reasons, like computational necessities and obsolescence, different architectures or architecture variants.

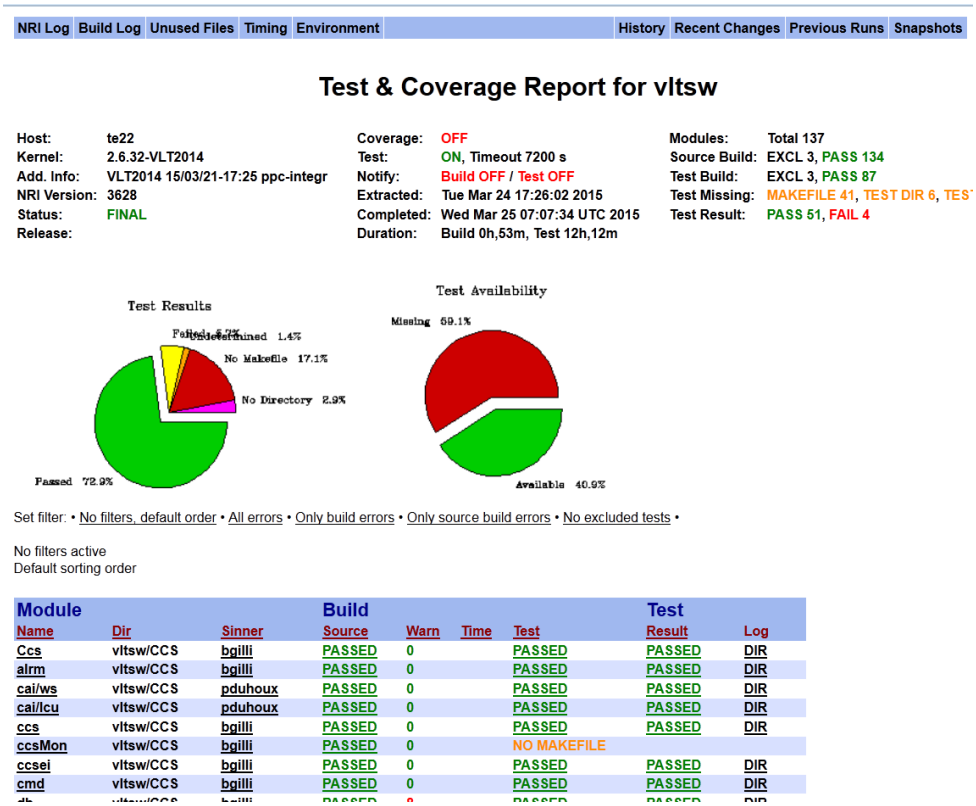


Figure 1: NRI Web Interface

After NRI introduction, many projects at ESO moved to this platform: the VLT Control Software, part of the SPARTA AO facilities, Dataflow systems and ALMA subsystems. Specific project needs and problems were then addressed by modifying the code and with time more and more new features were added, like C code static checking and test coverage statistics.

3. DEVELOPMENT ENVIRONMENT VLT CONTROL SOFTWARE

The automatic test and build infrastructure is just a part of the whole development environment for the VLT Control Software. The project uses as source code management software, or SCM, the software package Subversion[6], or SVN in short. The workstation software is running nowadays totally on GNU/Linux operating systems, where also most of the development is being done. Real-time components run on VXWorks[7] operating system by Wind River, running on various generation hardware executed with remote network execution techniques. Most of the source code is written in C/C++ and TCL/TK. The C/C++ code is compiled using the GNU C Compiler Collection[8], using either standard Linux compiler distributions or specific derived compilers supplied by Wind River for the VXWorks local control unit code. The core build system is the GNU Make[9] system, with an in-house developed tool named *pkgin* used to simplify and batch many operations. The ticketing system used by the project is Atlassian JIRA[10].

4. MODERNIZATION OF THE BUILD AND TEST INFRASTRUCTURE

After almost 10 years of successful service of NRI a few considerations were made on the software used, given also the new experience acquired and new trends with development environments and development tools. Three main concerns were raised with the current solution:

- Cost of adaptation to new tools was quite high, as new support required always work and coding from the NRI team. Especially with the number of new potentially interesting tools available in the community, it was hard to invest effort on their integration.
- Maintenance cost, as 10 years of continuous enrichment and adaptations for specific projects needs required a good amount of investment.
- Scalability: projects often requested the possibility to scale the execution to more parallelizable instances and the support of different operating systems platforms.

In an effort to modernize the automatic build and test infrastructure, a new solution was being searched for. The new solution had to be more easily portable to various platforms, offer easy parallelization of tasks, have a modular structure to introduce new tools as easily as possible and offer possibly a good support either, preferably, open-source or commercial. After a careful research, an open-source product was identified as the best candidate: Jenkins.

Jenkins

Jenkins[1], formerly known as Hudson, is one of the leading open-source automation servers. The software, written in Java, provides the possibility to automatize building, testing and any other task in the development and quality assurance process. It is based on a master-slave architecture where the master server is the one containing and managing the configuration and results of the defined activities, while the slaves are machines that execute some specific phase of the process, such a build or a code analysis. It is multiplatform as it can run on many platforms with the only requirements being mostly the availability of a fairly recent Java Runtime Environment, JRE, and the means to connect via network, such as a secure shell service enabled. It is highly expandable with the so-called plugins, pieces of code that expand both the frontend and the backend potentialities of the software. The plugins, which at the moment of writing count in the numbers of around 1100, bring support for all kind of standard development tools, source code management systems, network infrastructure and build customization. Specifically most of the source code management systems are supported, such as SVN, GIT[11], Bazaar[12] and Mercurial[13]. Should some special feature not be present Jenkins nor in any available plugin be needed, a creation of a new plugin could be done in Java. In recent versions of Jenkins a degree of extensibility can be also achieved using the native built-in support for Groovy scripting.

Jenkins focuses also on ease of use, providing a user-friendly Web interface that can be easily used by all the actors involved in software development process, from managers, to developers, to testers, to system administrators. Additional machines where to run operations can be easily added, as far as the basic JRE and network accessibility requirements are satisfied, as the client side software, a Java JAR archive, is automatically installed and run by the Jenkins master. Additional machines where to run operations can be either physical, virtual or cloud based machines, always connected to the Jenkins infrastructure or connected just when the master strictly needs them, opening also the possibility to use certain machines just partially for the Jenkins infrastructure. The system therefore offers easy scalability and possibility to parallelize tasks in the build chain.

Jenkins also offers a fine-grained authorization management, giving the possibility to clearly define roles and permissions on various levels. For example, some developer can access in read-only some project, while others can also request a build trigger, some can download the generated artefacts while others cannot even see a certain project and so on. Authentication can be made with either local users definition or using standard authentication services, such as NIS or Active Directory.

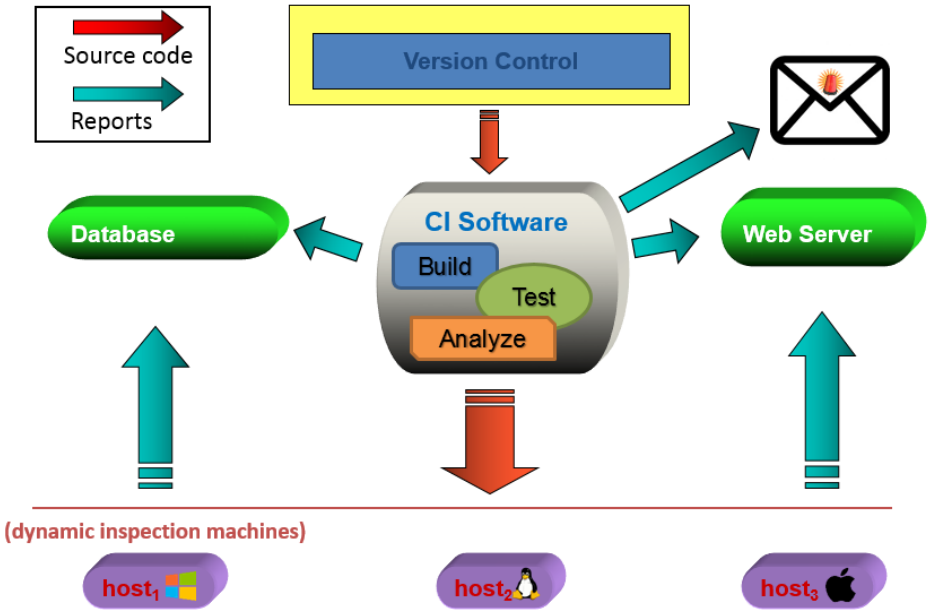


Figure 2: Graphical representation of the automatic build and test system

In general, an executed activity in Jenkins is named *job*. Each job is roughly defined by the method where to retrieve the source from, a SCM system, and the operations to perform for the execution, usually some shell script code. Additionally what to do afterwards can be defined, usually elaborate and publish the results on the Web page, send some email notifications, trigger the execution of another job or store some files, for example the generated binaries, for later retrieval by either another job or by a human actor. Of course, also timing constraints can be defined, defining both condition when the job has to run, either on each commit or on scheduled times, and when it should not, not to disturb some other activity. In the same way, dependencies or exclusions between jobs can be easily defined to create a chain of execution as desired.

5. CUSTOMIZATION OF JENKINS TO VLT CONTROL SOFTWARE NEEDS

A very important aspect to be kept in mind when changing the build and test automation system was to limit the impact on procedures and habits of the developers. Being the VLT control software project now over 15 years old and the NRI system about 10, it would be hard to ask developers to radically change their behavior. This is even more important if we consider that the developers are not just internal ESO staff but also external consultants and a big number of engineers and scientists from Consortia. Big changes in the workflow should therefore be avoided, both from low level build and test procedures and from high-level user interfaces and notification methods.

At the same time, a few new features should be added to improve the previous system:

- The most important of all was to try to shorten overall execution time, ideally by executing more tests in parallel, specifically for pieces of code that are compiled for more than one architecture, and present a single easily readable overview of the cumulative results. This was not possible with NRI, where a single architecture was built and tested per night and therefore multiple architectures could be only tested on different days of the week.

This should also be the first step toward having also a shorter total build and test of the basic VLT control software, which was currently running in about 21 hours, starting at the evening and giving the full results soon after noon of the following day.

- Introduce both in the report table and in the email notifications, additional important information such as the number of warnings encountered during compilation or time needed to build or execute a test
- Introduce test code coverage reports and additional static analysis activities
- Add the possibility to easily request execution, when special needs arise, also out of the common predefined schedules, to immediately have a feedback on some changes

Build level adaptations

The first step to integrate easily the VLT control software into Jenkins was to move from the current build system, comprising of an in-house made compilation procedure named *pkgin* that invokes the GNU Make program at a module level, to a system purely based on GNU Make configuration files for the building and testing phases.

- At a module level GNU Make was already used. Therefore, just the addition of a higher-level infrastructure of recursive Makefiles through the package structure was added, which calls at the end of the chain the pre-existing module ones. The impact on the module level for the developers would therefore be zero.
- The calls were encapsulated inside GNU Make macros, giving the possibility to easily act in the future on a single point, should changes be required.
- These macros simply chain the calls downwards to the module, by prior knowledge of standard directory structure, and create some very simple text files containing the results of the execution that is then used by the Jenkins infrastructure. These files contain information on the outcome of the build or test, separated if needed also by architecture, with timings and detailed error information.
- All the macros have standard names for each module, for example the *build* or *test* or *clean*, making the way of execution standard on both module and package level.

Recently important change that was done: fix in the base Makefile a few details, mostly dealing with correct dependencies generation and installation of files in parallel that permitted the execution of the build process using the GNU Make parallel execution support, the job server feature. The parallelization up to now works at a module level, meaning two modules cannot be compiled at the same time but more source files in a single module can. Going further by parallelizing the compilation of modules as well would be for sure possible but would require an explicit definition of module dependencies at a higher level. This is for sure on the list of things to do in the future.

On the opposite side, the possibility for local modules Makefiles to communicate with the higher-level ones, for example to specify what kind of architecture they support, was introduced with some classic GNU Make variables. By examining such variables, the top level Makefile knows exactly which operations the module requires. In detail, the variables were introduced to specify the architectures supported, currently Motorola 68000, PowerPC 604 and INTEL Nehalem, the support for multithreaded compilation, using pthreads, and the deprecation of parallel build on the current module, should the module compilation for some reason not be parallelizable.

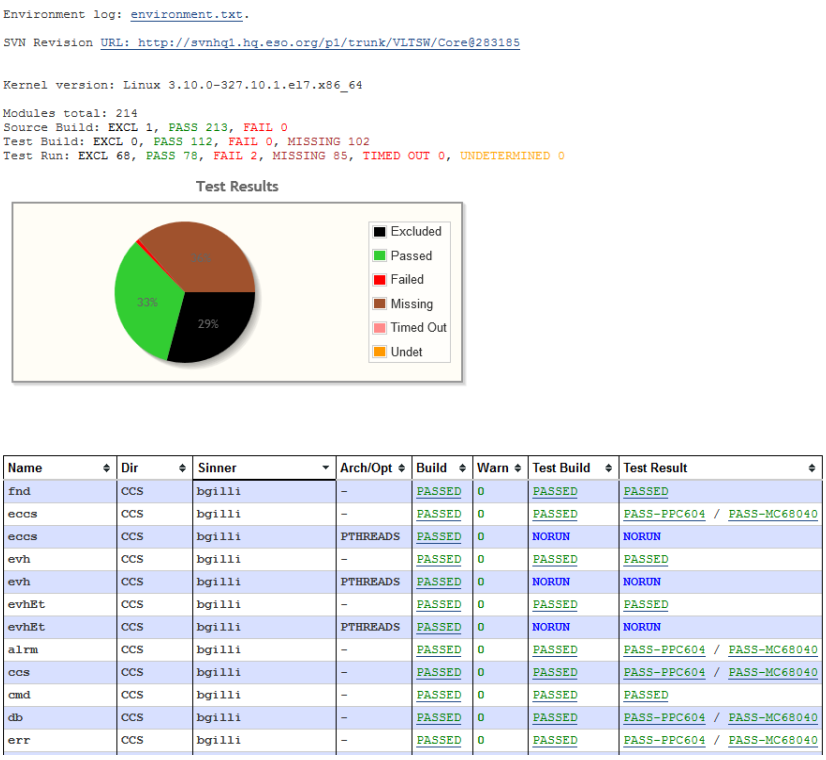
It is also important to say, that to additionally support and help developers, a small tool in Python was prepared to convert the old-style *pkgin* build execution into a Makefile template that can be used as a starting point with the standard GNU Make building. In some cases, where it is not possible to manage the Makefile in the repository, this tool is used even directly in the Jenkins activity to create on-the-fly the build configuration file.

Execution and results adaptations

The other important aspect with which users, especially the project manager, interacted a lot with the NRI system was the results user interface. The project manager could daily see the results and browse the detail of the operations, understanding at a glance for example which tests failed and then with a few clicks understand in detail what failed. The results were at a module level, but grouped by package for convenience.

In an ideal Jenkins setup, each module would be managed by a single *job*, an activity that can be run autonomously by Jenkins. Because modules in VLT control software do not have a full dependency information between themselves and due to the request to keep the output report table on the package level, it was decided to keep the job activities on a package level. This leads to the creation of a total of four main jobs in the build stage, representing the four main packages: the VLT core package, the telescope Interface, the detectors core package and the instrument core. Of course, this kind of division diminishes the final possible parallelization, but simplifies also the management over the one at module level that would imply the creation of around 150 jobs, of course possibly in an automatic way, each representing a module.

A consequence of the grouping of more modules in a single job meant also that an alternative way of displaying the result should be needed, as the concept is something not standard in Jenkins. To achieve this, after the invocation of the new GNU Make build script a *Python* script is invoked, which examines the results files generated by GNU Make, previously mentioned, and generates a highly customized, but template based, HTML table containing all the information needed in the desired format. This generated HTML table is then easily published using a Jenkins plugin. The nice thing about using HTML to convey the information is the possibility to introduce also elements such as a graphical representation, a pie chart, of the amount of builds or tests failed and the possibility to directly link relevant information, such as the low-level log of the build or test execution, from the stored artifacts to the table. The table is presented with appropriate colors and classic HTML table helpers such as column sorting and tooltips bringing additional information, such as for example the log of the last SVN commit in each specific module, that would result too heavy if directly inserted as a table column. The HTML is prepared using templates, which is filled with just the specifically generated data, giving therefore the possibility to easily change the look and feel in the future and reusing the work in other projects. The possibility to apply, as partially done, also dynamic Javascript code to the page makes the possible extensions of the reports potentially huge.



Name	Dir	Sinner	Arch/Opt	Build	Warn	Test Build	Test Result
fnd	CCS	bgilli	-	PASSED	0	PASSED	PASSED
eccs	CCS	bgilli	-	PASSED	0	PASSED	PASS-PPC604 / PASS-MC68040
eccs	CCS	bgilli	PTHREADS	PASSED	0	NORUN	NORUN
evh	CCS	bgilli	-	PASSED	0	PASSED	PASSED
evh	CCS	bgilli	PTHREADS	PASSED	0	NORUN	NORUN
evhEt	CCS	bgilli	-	PASSED	0	PASSED	PASSED
evhEt	CCS	bgilli	PTHREADS	PASSED	0	NORUN	NORUN
alrm	CCS	bgilli	-	PASSED	0	PASSED	PASS-PPC604 / PASS-MC68040
ccs	CCS	bgilli	-	PASSED	0	PASSED	PASS-PPC604 / PASS-MC68040
cmd	CCS	bgilli	-	PASSED	0	PASSED	PASSED
db	CCS	bgilli	-	PASSED	0	PASSED	PASS-PPC604 / PASS-MC68040
err	CCS	bgilli	-	PASSED	0	PASSED	PASS-PPC604 / PASS-MC68040

Figure 3: Jenkins results table example

Jenkins contains very flexible email notification capabilities, extensible also with additional plugins, to send customized email to predefined recipients or to recipients derived from the SCM changes. However, as it does not recognize the higher granularity of job splitting into modules, we implemented the email notification in the same Python script that generates the HTML output to be published on the web. The information about the recipient for each module, called in ESO VLT control software the *sinner*, was added as a SVN property not to create some text file or database containing this which is

hard to maintain. The Python script can therefore just request the SVN property *maintainer* for each module that needs reporting and use it to deliver the message. As an optimization, if one email address has more than one message to be delivered, that is a single maintainer has more than one module build or test failing, this will be united in a single email message. The email message content is highly customizable and in principle contains as the subject a clear indication on the package and operation that failed and as body the link to the specific job execution in the Jenkins HTML report and more details on the specific modules failure. Other helper functions, such as disabling notification per architecture, have been also implemented.

6. JENKINS INFRASTRUCTURE FOR THE VLT CONTROL SOFTWARE

At the end of the transition phase and after about 18 months of constant usage, a quite complete configuration is in place for the VLT control software. Currently the configuration is applied to the two software versions in development phase: the currently supported, which is receiving regular patch revisions, and the forthcoming major release. For both releases, a complete set of the VLT control software main packages is built, all the integrated tests executed and also documentation is generated, which can be also immediately accessed online via the Jenkins job page. This is executed at the end of the working day, slightly shifted per release not to create additional network transfer peaks, and is always completed early enough before the first developers arrive for the next working day. For both releases, the software is built and tested for the PowerPC 604 and Motorola 68040 supported architectures, with the newly supported INTEL architecture in addition for VLT release 2016. The results of all the different architectures are merged in an easy to read table. To reduce overall execution time and achieve the desired results, a complete run during the night, for each version we have a number of Jenkins slaves equal to one more than the supported architectures. In this way, we can run at once one test for each architecture and a test that does not require special LCU hardware attached. This calculation brings us, for example in the current situation, to a total of 3 machines for VLT patch release 2014 and 4 machines for VLT major release 2016, mostly very busy during the night execution. On each machine, one single test can run at once and in parallel with a build, therefore giving the possibility to execute also the same number of builds in parallel. Builds and tests are executed with different system users to isolate even more the possibility of conflicts between the two. For ease of management, the machines are all virtual machines running while the architecture specific LCU hardware is divided per type and per version into separate groups, named pools. The special hardware is monitored by specific scripts run at the beginning of the test phase, which can also perform a reset via hardware using remote power switches that can be controlled via SNMP packets.

Even if it is not an integral part of the VLT control software delivery, also an additional package which purpose is to heavily stimulate and stress test the central control software infrastructure, is built and tested in the process for both active versions.

Recently due to modifications needed to the build system scripts, a *continuous integration* system has been put into action: on every commit, with a fixed delay of 3 minutes that could be also eliminated, a build is done for the four main packages as in the normal nightly run. The compilation is done just on the package that received some modification and to the dependent packages, guaranteeing in this way a feedback that ranges from about 5 to 20 minutes. If the basic VLT Core software was modified then also an immediate test, based on the stress test module previously mentioned, will be executed to guarantee not just the correct compilation but also that at least a basic functionality is still intact. The tests indeed stimulates both workstation and LCU code and executes most of the general operations needed by the system, like environment setup, external hardware booting, system configuration, database usage and so on.

A few additional features came almost at no cost with the adaptation of the new system, for example the automatic generation of changelogs between executions and the possibility to link the Jenkins system with the ticketing system used for VLT control software - Jira. After configuring the specific plugin to point to the ticketing server and defining the pattern of the tickets, the tickets can be immediately accessed with one click from the changelog. The other way around, it is possible for Jenkins to automatically post the build result, containing a reference to a specific ticket, inside the comments of the ticket itself.

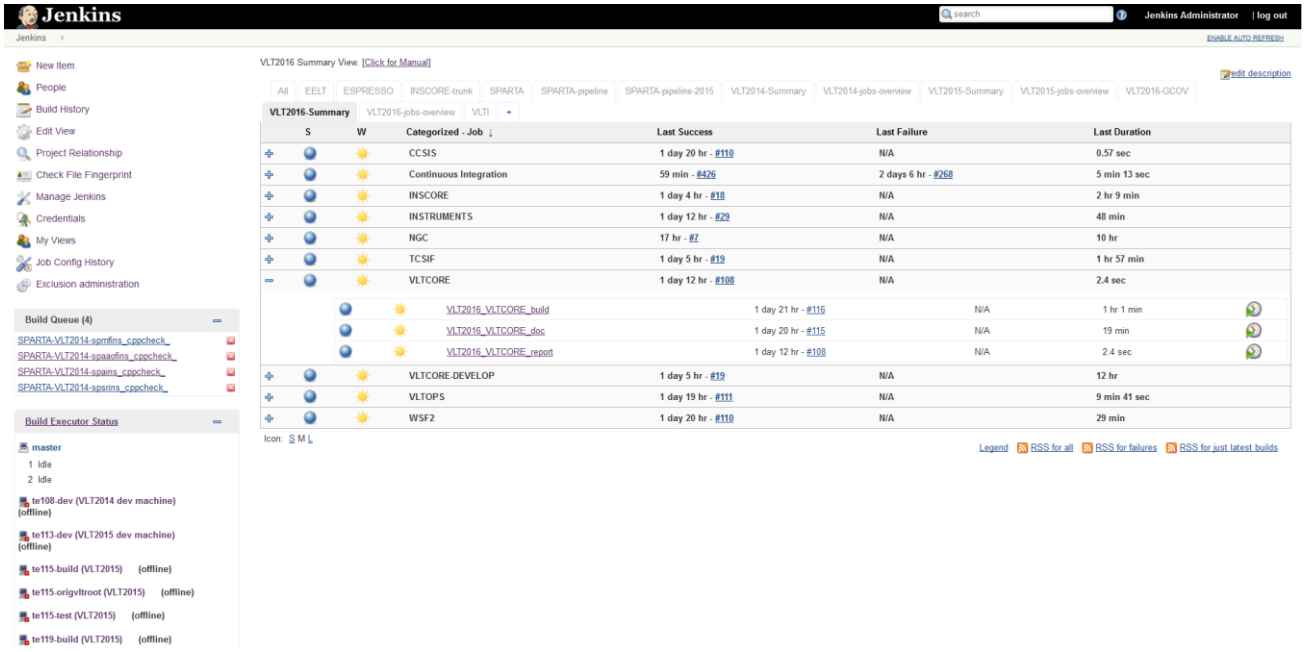


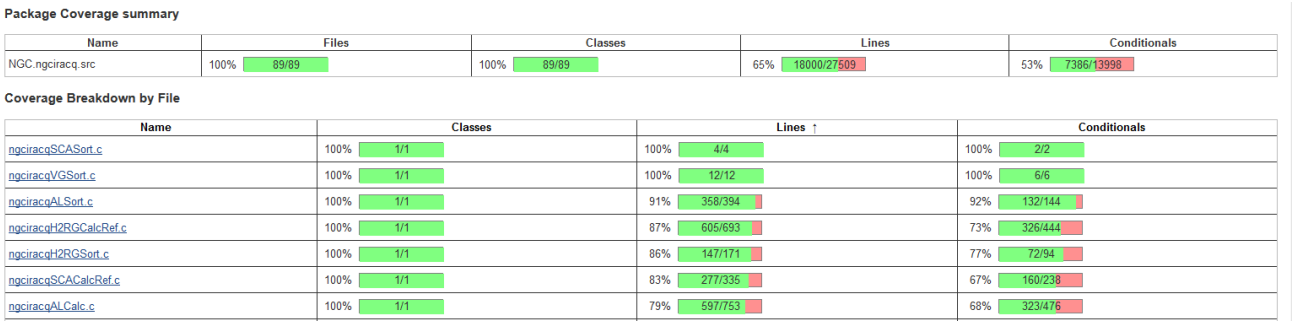
Figure 4: Jenkins home page with VLT2016 jobs overview

Quality assurance activities

Additional QA activities are executed on a slower pace basis as an execution on shorter intervals would probably bring little advantages. These activities use standard tools that have a dedicated plugin in the Jenkins system to ease the analysis and the display of the data on the Web interface.

On a biweekly basis, or more often on request during campaigns pointed to improvement of code quality, C and C++ static code analysis is performed using open-source product *cppcheck*[14], which output can be very neatly integrated into Jenkins via a plugin. The execution of this process is quite resource hungry, especially since care is taken to include all the specific header files to reduce possibly at a minimum the false positives. Therefore we dedicated a special 8 core virtual machine to this task, which is shared also with other activities that need temporary but very high computational power. The desired type of output, for example errors, warning, style incongruences, performance optimizations or portability issues, can be filtered at execution and then sorted in various ways on the user interface. From the user interface, it is very easy to immediately jump to the incriminated line of code, to understand better the context and the error, where also a summary explanation is given by the static code checker. Of course some false positives, correct code reported as an error, can sometimes occur. For those, the developer has the possibility to explicitly ignore a specific error by inserting a special comment tag in the source code.

Every Sunday during daytime, when no developers are present, all the C/C++ code for the workstation side, LCU side is currently into implementation phase, is rebuilt with coverage instrumentation, using the GCOV[15] libraries, and all the tests are run. This activity will give the possibility to the developer, and of course also the tester and project manager, to appreciate which amount of code is actually being touched by the test. What is even more interesting is that this activity will also clearly point out which lines were executed, and how many times, and of course, which were not. This gives the possibility to further try to improve the tests by studying the code lines that are obviously not stimulated by already existing tests. All the statistics and analysis can be applied by module, subdirectory or file, giving also a view on different levels on where more work is needed. All the statistics are collected in time so the overall improvements can be monitored in time.



Package Coverage summary

Name	Files	Classes	Lines	Conditionals
NGC.ngciracq.src	100% 89/89	100% 89/89	65% 18000/27509	53% 7386/13998

Coverage Breakdown by File

Name	Classes	Lines ↑	Conditionals
ngciracqSCASort.c	100% 1/1	100% 4/4	100% 2/2
ngciracqVGSort.c	100% 1/1	100% 12/12	100% 6/6
ngciracqALSort.c	100% 1/1	91% 358/394	92% 132/144
ngciracqH2RGCalcRef.c	100% 1/1	87% 605/693	73% 326/444
ngciracqH2RGSort.c	100% 1/1	86% 147/171	77% 72/94
ngciracqSCACalcRef.c	100% 1/1	83% 277/335	67% 160/238
ngciracqALCalc.c	100% 1/1	79% 597/753	68% 323/476

Figure 5: Code Coverage Output Example

Another important tool that was newly integrated into the automatized system is the memory leakage checking. This is done using the standard *valgrind*[16] tools suite and an integration plugin for Jenkins to manage its output. The execution of this activity is not automatic and planned, but is executed on request. Execution using the valgrind tools, on one side brings very important and useful information about possible resource leakage, on the other side poses a few problems in the context of the VLT control software. The execution, in fact, being done in a de-facto virtual machine is much slower, about 5 to 10 times, than the original one. Apart from making the overall test execution longer, which may not be a problem if properly planned, this brings also potential test failures when tests rely on specific timings. On the other side, VLT control software module tests are usually not pure unit tests, but higher-level tests, and require the setup of a working environment. This leads to the fact that the analysis of a test often comprises many components and therefore the output can become hard to read. Still the tool is available and has been used in cases when resource leakage was suspected and the tool brought up interesting information to the developer.

Recently, with VLT2016, also automatized configuration management of the test and development machines has been introduced. Using *Puppet*[17] as the configuration management tool and Jenkins as a frontend, a daily check of all the machines used by the Jenkins infrastructure and by the developers is done. The parameters checked are various and constantly growing to cover all the necessities, well separating in a modular structure the different necessities of different machines. If one single parameter for a machine is not aligned an email will be immediately sent to the machine administrators that can then decide how to proceed, either align the machine or recognize that the parameter need changed and act accordingly to the rules database, stored in SVN.

Other activities are also available to be run on demand: very useful examples are the execution of the tests with a special debug kernel that keeps accounting of resources usage on the LCU side or execution of tests that require special hardware that need to be also monitored by an operator. In this second case the level of automation is of course lower, but the very important aspect of running it into the Jenkins infrastructure is to easily keep track of all the executions, successes and failures in the long time and be able to correlate them with changes that happened in between.

Results and comparison of the renewed system

The whole execution of a night run with the NRI system lasted roughly 21 hours. Starting at 5 p.m. this would briefly give the last results, of the last package in line, by 2 p.m. of the next day, almost at the time to start with the new execution. With the parallelization introduced by Jenkins the total execution time dropped down to just over 8 hours. Starting still at 5 p.m. the full results run would be available at around 1 a.m., so even the start can be delayed to permit late modifications to developers and still remain largely on time for the early morning ones.

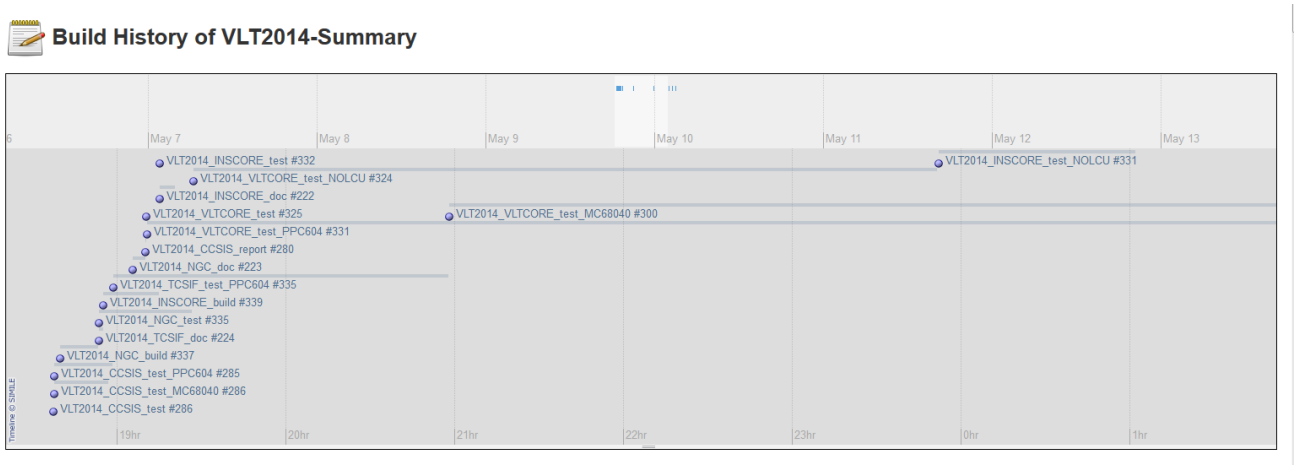


Figure 6: Timeline showing parallel execution of multiple build and test processes

What is very important to notice is that in the much shorter timespan not just one architecture is built and tested, as it was the case of NRI, but all the supported ones are. With the old system, different architectures could be tested just on different days, giving also the trouble of correlating slightly different source code given the time shift. Additionally in the timing also a few other activities are already counted here, as not in the past, such as documentation generation and some additional support packages.

Another very important thing to say is that the timing could be still much optimized if tests jobs were further divided and therefore executed in parallel. Of course, this would require a larger number of machines available and LCU pools, therefore a higher cost in hardware, but it would not really bring many advantages to the project since, as far as the test is shorter than the night stop, no extra time would be gained. The extra time remaining until the morning will be therefore instead used to run additional builds and jobs, such as build and test of the VLT operations software, the test instrument and other real instruments.

From the continuous integration point of view, after the modification to the build system to allow at least intra-module parallel building, the execution of the build on the four main modules has been reduced from approximately 1h40 minutes to approximately 25 minutes, roughly 25% of the time. These results are achieved using a 16-core machine and instructing GNU Make to execute up to a maximum of 16 parallel jobs. A higher number of cores available do not really change the results, as there are very few modules that contain more than 16 source files, for which additional parallelization would therefore bring an advantage. Optimizations on the memory, storage and network speeds could bring some additional improvements.

7. JENKINS INFRASTRUCTURE FOR THE SPARTA SYSTEM

Also the SPARTA project, the adaptive optics infrastructure, was inserted into the NRI execution loop. The SPARTA project is actually built over the VLT control software and therefore modifications done to the base VLT control software for the build automation can be almost automatically used within the SPARTA software infrastructure. Naturally, with the usage of Jenkins for the VLT control software also the SPARTA project migrated to the infrastructure.

It is important to keep in mind that the SPARTA project is used by a big number of different instruments and therefore a change in the basic platform can have theoretically different effects on them. In relation to this, in the Jenkins system a full chain has been setup so changes in the underlying levels are then propagated and tested on various instruments, at the moment of writing CIAO, GALACSI, GRAAL, NAOMI and SPHERE. For all the instruments a full build and a basic deployment and startup test is done. Then, specifically by instrument, many tests are executed focusing of course mostly on the parts that do not require special hardware, not available, to be executed. Compared to VLT control software the tests in the SPARTA case are of a higher level, trying to stimulate the system and the complex interaction in it as a whole, not at a modular level.

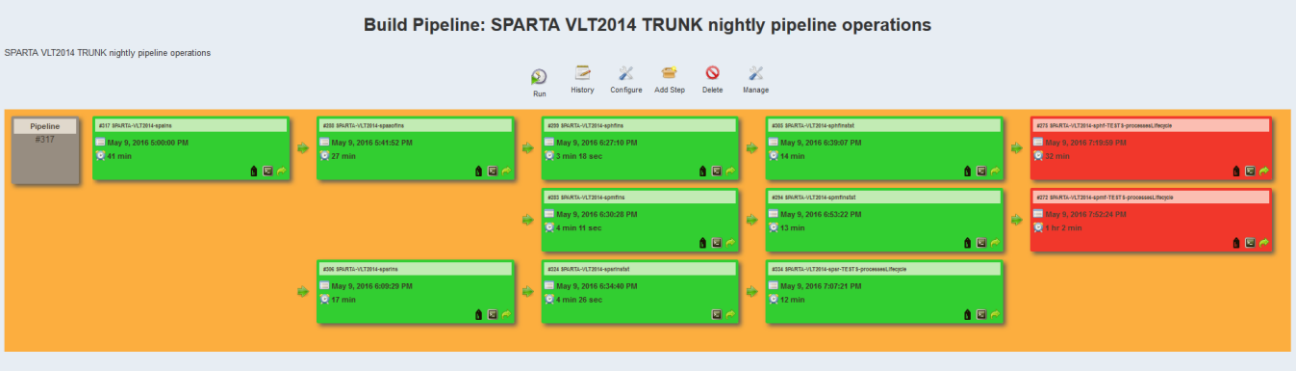


Figure 7: Dependencies and execution flow in the SPARTA system

An interesting aspect of the Jenkins infrastructure for SPARTA is the plan to execute also tests on the cluster with real hardware in specific days of the week when the cluster is available. To achieve this a helper application that manages the on-the-fly switching of the software versions on the cluster virtual machines has been implemented. This, coupled with a numerical test, should be the next natural step to be able to perform heavy-weight tests on a hardware configuration which is much similar to the one in operation.

Additionally, other quality assurance as for the VLT control software are executed: weekly coverage calculation in the weekend and monthly static code analysis of the source trees.

8. JENKINS INFRASTRUCTURE FOR THE E-ELT

The new ESO programme E-ELT will bring to the control software team very important challenges also on the testing automation and integration. At the moment, the idea to use the Jenkins infrastructure also for this project after the positive results with VLT for internal integration and testing is valid. Compared to VLT, the E-ELT project, from the software integration and testing point of view, brings many new challenges. First of all a new platform to support, in VLT everything is either Linux or VXWorks currently, while in E-ELT we will have the addition of the Microsoft Windows platform, and languages such as Java, Python, NI LabView[18] and PLC languages, all in the E-ELT standard. A bigger contribution from non-ESO resources is also expected and this means the necessity to be versatile to integrate different tools and procedures.

Many tests have been done in the prototyping of the Jenkins integration system on the various aspects. First and foremost, Jenkins has been successfully tested with slave machines running Windows 7 platform. Here some prototype jobs have been executed: PLC code compilation based on Visual Studio interface, LabView code analysis executed via LabView interface itself, prototypes of UI automation testing using the AutoIT[19] infrastructure and prototypes of system integration of interfaces using on the Robot Framework[20]. All this jobs, with required work to automate and control the execution under Windows environment, were successfully integrated into Jenkins and are run on a regular basis on in-house prototypes.

On the language support side, in addition to the C/C++ tools that are already integrated in Jenkins for the VLT control software, also Java tools were prototyped. Additionally to the build and test execution also code style checking was prototyped using the *Checkstyle*[21] open source tool, and its companion Jenkins plugin, and bug hunting was setup using the *FindBugs*[22] open source tool, again with an ready to use Jenkins extension plugin. Both tools, like the cppcheck for VLT, support easy to use graphical display of results found, with historical graphs and direct source code browsing.

Another interesting technological feature that has been tested is the usage of *Docker*[23] containers for test reproducibility. The Docker container technology give the possibility to recreate easily a well-known environment, from a filesystem but also networking point of view, in which tests can be executed. This methodology simplifies the management of the machine where the tests are executed, as by re-instantiating on each execution the container the state is very well known. All this is very well integrated in Jenkins as, again with a plugin, containers are dynamically instantiated and created when a job marked to be run on them is in the execution queue. There is interest to integrate this new way of reproducing test environments also inside the VLT control software testing environment.

9. CONCLUSIONS

The migration from the old in-house solution to the new Jenkins based solution has been for sure a very successful project. The two main goals, the possibility to scale and do it with little effort, were fully met and many other important secondary goals were reached. Now we can count on having an easily adaptable tool to new requests. No particular problems were encountered during the switch and, actually, the timing from experimentation to complete switching was quite short. Additionally, the tool quickly acquired confidence and sympathy from the software team with requests to add other projects, instruments and activities. This was also possible as from the very beginning a very generic and configurable philosophy was the base of the system.

10. REFERENCES

[1] https://www.jenkins.io
[2] J. Stecklein, "Error cost escalation through the project life cycle", http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20100036670.pdf, NASA, Tech. Rep., 2004.
[3] https://www.perl.org/
[4] https://www.gnu.org
[5] http://www.linuxfoundation.org/
[6] https://subversion.apache.org/
[7] http://windriver.com/products/vxworks/
[8] https://gcc.gnu.org/
[9] https://www.gnu.org/software/make/
[10] https://www.atlassian.com/software/jira
[11] https://git-scm.com/
[12] http://bazaar.canonical.com/en/
[13] https://www.mercurial-scm.org/
[14] http://cppcheck.sourceforge.net/
[15] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[16] http://valgrind.org/
[17] https://puppet.com/
[18] http://www.ni.com/labview/
[19] https://www.autoitscript.com/site/autoit/
[20] http://robotframework.org/
[21] http://checkstyle.sourceforge.net/
[22] http://findbugs.sourceforge.net/
[23] https://www.docker.com/