SOFTWARE QUALITY ASSURANCE FOR THE DANIEL K. INOUYE SOLAR TELESCOPE CONTROL SOFTWARE

A. Greer, A. Yoshimura, Observatory Sciences Ltd, Cambridge, UK
B. Goodrich, S. Guzzo, C. Mayer, DKIST, Tucson, AZ 85719, USA

Abstract

The Daniel K. Inouye Solar Telescope (DKIST) is currently under construction in Hawaii. The telescope control system comprises a significant number of subsystems to coordinate the operation of the telescope and its instruments. Integrating delivered subsystems into the control framework and managing existing subsystem versions requires careful management, including processes that provide confidence in the current operational state of the whole control system. Continuous software Quality Assurance provides test metrics on these systems using a Testing Automation Framework (TAF), which provides system and assembly test capabilities to ensure that software and control requirements are met. This paper discusses the requirements for a Quality Assurance program and the implementation of the TAF to execute it.

INTRODUCTION

During the software conceptual design phase DKIST elected to use a Common Services model as a basis for the standard distributed software infrastructure used to build the control subsystems. The Common Services Framework (CSF) was developed as a result of this decision, providing a standard framework supported in three programming languages (Java, C++ and Python) [1]. The framework offers many features including deployment support, communications support, persistence support, as well as application support and a broad library of additional tools. All of the DKIST control software subsystems are built using the CSF. Figure 1 shows a block-representation of the layout of the CSF. This infrastructure software, and the software required to control a large telescope such as DKIST presents developers and maintainers with a substantial level of testing needed to ensure the quality of the software remains at a satisfactory level throughout the lifetime of the project.
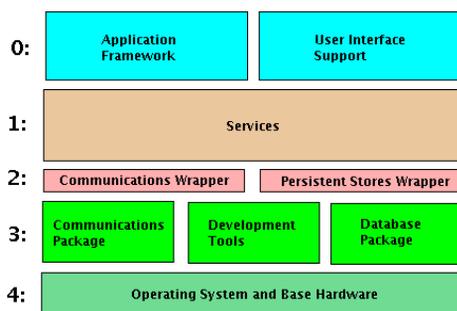
DEVELOPMENT AND TEST SET-UP

The DKIST project offices in Tucson, Arizona and Boulder, Colorado have been equipped with the necessary control hardware to be capable of running the entire control software stack. This includes the required network infrastructure, Data Handling System (DHS) servers, CSF servers, an operator's station, 4k monitors and network hub [2]. The hub is provided for future expansion. This hardware installation is called the End To End (E2E) simulator. Every subsystem accepted by DKIST must be delivered with the ability to simulate all hardware at the interface level. This requirement allows the development team to run the simulated subsystem within the E2E environment. It is possible to run all of the subsystems together natively on the hardware, or spawn virtual machines to execute a subsystem in isolation. A network of virtual machines can be spawned to verify messaging and database logging across operating system and software versions. The virtual machines are created and executed using the VMWare commercial product VMWare Workstation [3]. VMWare has been developed for more than 15 years and aims to provide the most stable and secure local desktop virtualization platform in the industry. The E2E rack and operator's basic layout is shown in Figure 2. The rack shows the required network infrastructure, DHS, and CSF servers. The operator's station is shown with its two 4k monitors, desktop, and network hub. The instrument operator's position is shown with its Quality Assurance computer and 2 HD monitors. Figure 3 is a screen shot taken from the two 4k monitors with the subsystems and simulators running, and has a photo of the server rack overlaid in the top right hand corner.


Figure 1: Layout of Common Services Framework.
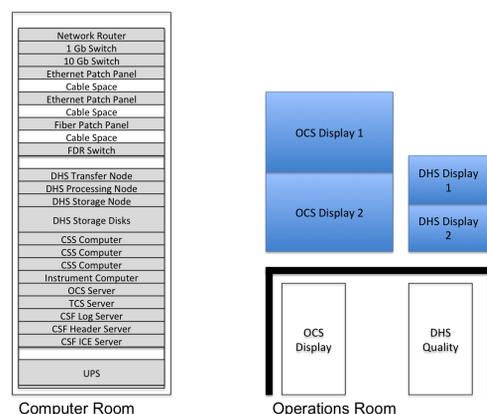
Figure 2: End to End hardware schematic.
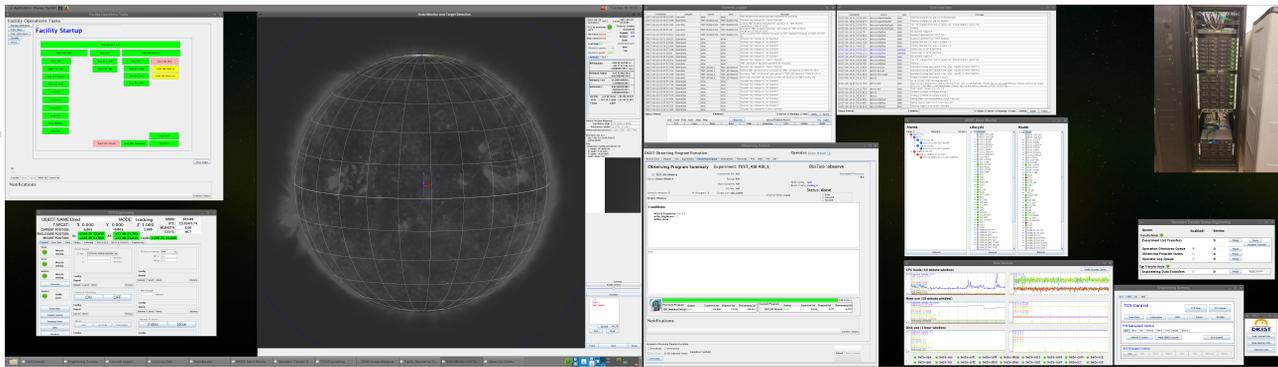
Figure 3: Screen shot taken of the E2E with a photo of the hardware in the top right corner.

SOFTWARE QUALITY REQUIREMENTS

The DKIST control software is supported in three languages, with differing levels of support for each. The communications and middleware independent layers have been developed in-house by DKIST staff, but many individual telescope subsystem software modules have been developed by external commercial companies and delivered as part of the whole subsystem. Upon delivery these modules must be integrated into the overall control software system and then maintained by DKIST staff. Each module has passed the factory acceptance tests (FAT) prior to delivery but for every new release of the infrastructure software the tests must be executed and the results verified once more. Add to this the need to regression test for any bugs and subsequent bug-fixes found in either the subsystems or the infrastructure software itself, and the matrix of tests becomes large. For this reason investment was made to produce a software quality assurance (QA) program which allows all tests to be grouped, executed, monitored and logged in a consistent manner, with continuous integration and automated notification of failures. Unit, regression and system tests can be scheduled to execute at predetermined times, with specially marked benchmarking tests executed multiple times and then statistical analysis of the results performed. Test reports should contain a concise summary, with detailed report information available if required. Below is a summary of the requirements identified:

- Consistent reports generated for all tests.
- Unit test framework required for each supported language.
- Use existing tools and frameworks where possible.
- Ability to group different test types and languages.
- Ability to install relevant software subsystems for tests.
- Continuous execution of tests.
- Notifications of test results.

Existing Tools

For the development of test frameworks it was clear that making use of existing test frameworks would reduce costs and provide well supported libraries to use. For the continuous execution of tests there were also products available but the decision was made to develop a thin Python layer that made use of the existing VMWare virtual machines that were already in use within the DKIST project. VMWare Workstation provides an API for interacting with the virtual machines which made it easy to control them, as well as interact with the sessions running on the virtual machines from the host. The benefit of this is to be able to coordinate test execution and result collection across multiple virtual machines.

TEST REPORTING FORMAT

To ensure that test reports are easily compiled and processed from multiple test frameworks and languages it was necessary to settle on an agreed standard report format for recording the results of all tests. This test reporting format was agreed upon before any development of testing frameworks had begun. To make the reports easily parsed by automated metric generating software, but also relatively human readable it was agreed that an XML format should be used. This format provided scalability for storage of test reports; a file could contain a single report or many hundreds of reports and each report could contain every type of attribute available or just a subset. Each report contains information that can be used to identify which test the report relates to, where and when the test was executed, the version control tag of the test script (to ensure exactly the same version of tests can be re-run if required), and the results of the test execution. Other metadata associated with a test can be added to the report as required. Figure 4 shows an example test report of a test that has failed. The example contains additional information describing which versions of software packages were checked out for the test execution, along with details of the specific failure.

TESTING FRAMEWORK FOR MULTIPLE LANGUAGES

All languages have existing testing frameworks available and for the QA programme DKIST wanted to make use of these where possible. When selecting a framework there was a requirement to customise the generated output and so this was one of the major factors that contributed to the decision. The overall QA infrastructure and tools were developed using

```
<?xml version="1.0" ?>
<TEST>
   <ID>
      SYS_REQ_4.1_1110
   </ID>
   <CVSID>
      $Id: SYS_REQ_4.1_1110.py,v 1.2
      2015/09/10 08:27:35 Exp $
   </CVSID>
   <DESCR>
      Disconnection - Invocation of target
      public interface method shall no
      longer be possible after
      disconnection.
   </DESCR>
   <TYPE>system</TYPE>
   <TIME>2017-05-17 07:31:25.662</TIME>
   <MACHINE>localhost.localdomain</MACHINE>
   <VERSION name="CSF">HEAD</VERSION>
   <VERSION name="Base">HEAD</VERSION>
   <RESULT>
      FAIL
   </RESULT>
   <FAILURES>
      <FAILURE>
         <STEP>
            18
         </STEP>
         <REASON>
            qas.test.41.1110.java
            .comp.client.javacomp
            shouldn't have been added
            by 'get'.
         </REASON>
         <LOCATION>
            StepFailGetTest
         </LOCATION>
      </FAILURE>
   </FAILURES>
</TEST>
```

Figure 4: Test report example.

python, as the language suited the requirement to coordinate multiple applications, and as a mature scripting language has built in support for many of the tasks that were required to operate the QA programme.

C++ Testing

The testing framework that was selected for executing C++ tests was CxxTest [4]. CxxTest has a good set of assertion macros, is distributed as header files and has no other dependencies. CxxTest also allows for the creation of a custom test runner that can be compiled into the test application and also the registration of a test listener, used to catch failed tests and generate reports of the correct format. A custom executor and listener were constructed for the DKIST QA framework, providing the C++ unit tests with the capability of generating the expected XML report format. This method of building the custom test runner into the framework means that the end user who is constructing tests does not need to write any specific boiler plate code for the tests to be compatible with the DKIST QA framework. The resulting executable test applications generated by CxxTest can be executed from the command line and as part of a larger suite of tests. TinyXML-2 [5] was selected as the library for the production of XML reports. TinyXML-2 is a small and efficient C++ XML parser, it was easy to use and consists of a single header and single cpp file which could be compiled into the test applications.

Java Testing

For Java unit testing JUnit [6] was used, which is a unit testing framework for the Java programming language. JUnit provides an interface for creating and executing custom test runners providing the necessary hooks to be able to customise the tests. A custom runner was produced that could create test reports with the same format as those produced by C++ tests. Once again JUnit tests can be executed from the command line. Java provides native support for generating XML documents.

Python System Testing

There was no requirement to provide unit tests for Python as the CSF main classes provide only Java and C++ support. Python is used by DKIST for scripting at the system level, with a Jython interpreter providing the interface between the python scripts and the CSF framework. Therefore for the python tests it was only necessary to create the relevant test classes to assist with testing standard system level operations. This included testing methods for loading and initialising subsystems, connecting to the subsystems and verifying the health, submitting command configurations and testing success and failure of those configurations, setting and getting data items, verification of expected log messages and other permanent storage interactions, publishing and receiving events. Custom Python classes were created to generate the required XML report files and these were kept consistent with the C++ and Java generated reports.

TESTING AUTOMATION FRAMEWORK

Having developed testing capabilities for all three languages supported by the DKIST CSF control software, the next step was to support the execution of suites of tests. Each of the test scripts can be executed from the command line and so the decision was made to create a framework that can execute tests of any language. The Testing Automation Framework (TAF) is responsible for coordinating a large set of tests. It consists of a set of Python scripts, Java unit tests and CPP unit tests that are executed on the (virtual) machine where the tests are to be run. The TAF has developed into a

tool that can perform installation of any system or subsystem of the DKIST software including the installation of the CSF software itself. The TAF runs under the assumption that tests are to be executed from a clean system. The TAF checks its configuration file to find out which version or versions of the DKIST CSF and subsystem code should be installed and tested. The TAF will then check out the required software versions from the DKIST software repositories into the designated directory and perform all installation steps required to have a fully operational CSF installation. Once the installation is complete the TAF runs in sequence all of the pre-selected system tests, followed by the unit tests. Tests that execute on a particular instance of the TAF are entirely configurable. Multiple TAF instances can run simultaneously within separate virtual machines so that several versions of CSF can be regression tested at the same time (subject to the physical memory constraints of the underlying hardware). It is possible to configure an instance of the TAF to not execute any tests at all, instead checking out the required versions of the DKIST software and performing the installation only. This is useful for a situation where tests executing under another TAF instance on a separate virtual machine expect to connect to remote application instances. A set of virtual machines can be configured to mimic a fully networked system ready for network tests. This is not suitable for testing the network architecture, only for testing the software behaviour within a distributed deployment. The TAF is responsible for collecting and processing all of the individual reports from the tests which are saved to disk at a specified location. The TAF is not responsible for compiling the final report or sending the emails containing the reports; that is the responsibility of the End To End Test Executor described below.

END TO END TEST EXECUTOR

The final component required to complete the QA software infrastructure was a tool that could coordinate the execution of TAF instances, spin up and down virtual machines, parse the test reports and send out notifications of test summaries. The E2E Test Executor (ETE) was developed to perform this list of tasks, and is responsible for all of the tasks mentioned above. The ETE comprises a set of Python scripts and configuration files that are executed as part of a cron job and can be easily configured from a command line tool. The ETE can be configured to start up a test run at any pre-defined time on any available virtual machines. When the specified time is reached the ETE will wake up, start the necessary virtual machines and then copy the TAF scripts and configuration files to the virtual machines. After the copies have completed the ETE will execute TAF scripts on the virtual machines and wait for the completion of all tests. Once the tests have been completed the ETE will shut down the virtual machines and then perform the appropriate notifications of the test results. The ETE will be responsible for ensuring only one test run is performed at any one time on one virtual machine, but will not stop multiple virtual machines from being active and running separate test runs at any one time. From the command line it is possible for an operator to invoke the ETE to start a test run immediately. For situations where real hardware might be used for testing the ETE would not be required, instead configured instances of the TAF would be manually executed on physical machines. The result is a full set of tests run with real hardware and reports stored in the configured location.

ETE AND TAF WORKING TOGETHER

In summary the following QA test runs are performed each day as follows:

- The ETE checks its configuration files and starts any virtual machines as appropriate.
- The ETE copies the TAF source distribution and the specified TAF configuration file onto the virtual machine.
- The ETE executes a script on the virtual machine which runs up the TAF.
- The ETE instance then waits for a signal from the virtual machine that marks completion of the TAF execution.
- The TAF instance executing on the virtual machine reads the configuration file which was transfered to the machine by the ETE.
- The TAF checks out the software modules specified in the configuration file (unless specified not to).
- The TAF updates any configuration parameters required to run the software on the specified virtual machine (such as location of event servers, database IP address).
- The TAF builds the DKIST control software (unless specified not to).
- The TAF initiates tools required to run the DKIST control software (starts database and event servers).
- The TAF runs system tests and unit tests, generating the reports and storing them on the host machine.
- If there are more virtual machines to run, the ETE executes them as before.
- When there are no more virtual machines to run, the ETE shuts down all the virtual machines, writes the final report and, optionally, emails the report.

BENCHMARK TESTS

After the initial QA software infrastructure release had been operating for some time a request was made to be able to add benchmarking tests. The testing code was updated to allow the notification of a benchmark value from a test and the TAF updated to allow configuration of a benchmarked test. Essentially the TAF configuration can specify how many times to execute a benchmark test, and the results are not treated as separate test results; instead the benchmark statistics are calculated and added to the report.

FUTURE ENHANCEMENTS

The following enhancements are on the roadmap for the DKIST QA software:

- It would be useful to be able to configure the TAF to ignore a specific test failure, if the test always fails and it is understood why the test fails. This would allow the known test failure to be ignored in the final test report so that any new test failures are prominent and not masked by existing failures.

- Disk space monitoring will be added to the TAF to stop disk full errors if logs have been filled. During previous test runs a particular error that occurred would result in a tight loop writing errors to a log file. This eventually resulted in the disk filling up and locking up of the virtual machine. The TAF could monitor the available disk space and safely shut down a virtual machine prior to the disk becoming full.

- The TAF will have permanent storage of results history added so that any significant changes to benchmark results can be recorded and appropriate notifications sent.

- The implementation of a standard set of system test methods and associated templates to simplify the writing of subsystem tests is currently ongoing. The templates can be used by test developers and will ensure that the same kind of tests are carried out in exactly the same way for each subsystem. Examples of the sort of tests include checking health transitions and associated log messages, starting and stopping subsystems and ensuring correctly formatted command configurations are accepted by subsystems within allowed time budgets.

CONCLUSION

DKIST have fully invested in the software QA programme for project control software. As a result of this investment, a QA infrastructure tailored to the needs of the DKIST project has been developed, leveraging on the power and flexibility of existing virtualisation and testing tools. Many hundreds of tests are automatically executed on the DKIST servers throughout every day, with key project members receiving detailed email reports of the results. DKIST control software developers can commit new features with the confidence that all existing requirements will be re-verified and that any failures will be reported. Additionally new unit or system tests can be integrated into the test suite and either executed in complete isolation or as part of the full E2E control software deployment.

REFERENCES

[1] S. Guzzo, S. Wampler, B. Goodrich, "Common Services Framework Design Requirements", August 5, 2016.

[2] B. Goodrich, "Boulder End-to-End Test Bed Design", November 18, 2016.

[3] VMWare, https://www.vmware.com/.

[4] CxxTest, http://cxxtest.com/.

[5] TinyXML-2, http://www.grinninglizard.com/tinyxml2/.

[6] JUnit, http://junit.org/.