Overall principle

Testing covers a wide range of activities: from unit tests, to functional, stress, performance, all the way to full system verification. Some tests can be destructive (e.g. stress tests), others require to be non-intrusive in order to be able to run on production systems.

A wide and rich set of test tools and framework already exists, both in the commercial and open-source domain. We also recognise that the last thing we need to do is adding ETICS specific constraints to testers and their testsuites.

With these goals in mind, ETICS provides tools and techniques in order to ease the integration of new and old tests in order to be able to automate the execution of these tests, collect meaningful reports and perform trend analysis on important metrics.

On the topic of testing, one size fit all is unfortunately not realistic. In order to provide a maximum of flexibility, while maximising the quality of the reports ETICS can generate from tests, we have devised a simple process if test integration.

In order for ETICS to be able to execute a wide range of test types, we need to be able to standardise a few things:

- how to execute a test. This needs to include a single test or a complex orchestration of tests
- how to assess the results of a test
- how to bring the test outputs together into a coherent report for the user. This needs include as much information as possible on the context in which the tests have been performed, as well as the causes of failure, in the case of failures.

One could simply mandate an output *schema* for test developers to follow, but this bring problems. On the one hand, you wan the schema to contain potentially rich data, which can then produce rich reports. However, this means would mean a lot of work for test developers in terms of tooling. Remember, test developers should worry about tests, not tools. On the other hand, if the output schema is trivial, which means easy to generate, then the quality of the report might suffer, which in turn might mean that it is difficult to understand the context in which a test ran and the reasons of its failure.

Our solution is to provide different techniques to cater for all cases mentioned above, with the possibility to migrate from one test type to another. Further, with a few exceptions, this solution is based on established practices that have being used during the last 2 years by the testing team of JRA1 during EGEE-I. In otherwords, the idea here is to bring good ideas under a common roof and allow ETICS to remotely execute these tests and generate consistent reports.

Testsuites execution process

The ETICS solution in running tests is based on the process illustrated below. This process illustrates the different scenario for testsuites integration, as well as automatic execution and generation of test reports. This flow is implemented using a combination of the standard ETICS tooling and the *TestManager* module:

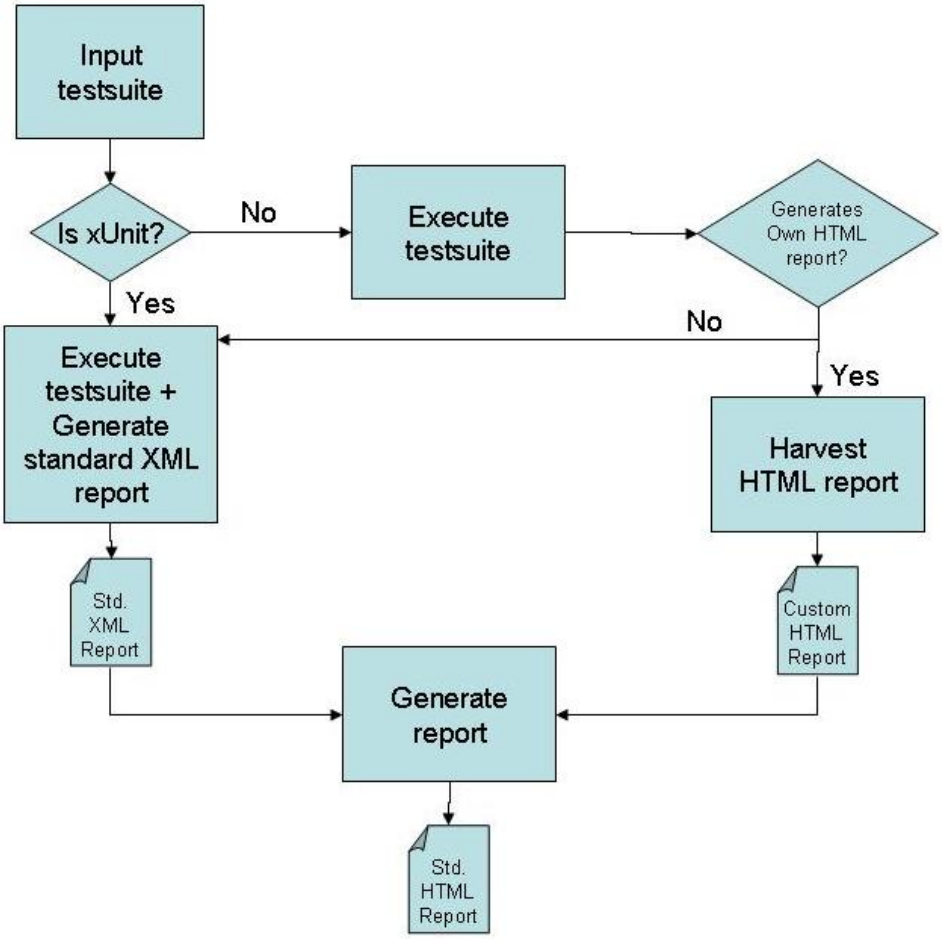

Figure 1: testsuite execution process diagrame

In this context, *xUnit* means all supported implementations of the original *junit* test framework. For example, we currently support:

- *PyUnit*: for Python (implemented in the standard Python module unittest.py)
- *junit*: the original Java implementation!!
- *CPPUNIT*: the C++ implementation. However, the newer *CxxTest* should replace it soon.

-- MebSter - 19 Jun 2006

Types of testsuites

With the workflow in mind (see above), we can support three types of testsuite:

1. a testsuite is implemented using a supported xUnit framework. In this case, the TestManager tool (see below) is able to execute the testsuite in such a way that guarantees the generation of a standard HTML report. Here you can see a sample of this report. This report in generic and hierarchical, which allows testsuites to be composed of other testuites, testcases and tests, with an explicit dump of the trace for each failed assertions. Using this test mode, users can also define dependencies between different testsuites and testcases, with conditional *fail on error* or *continue on error* semantic. An example of this test mode is the Data Management testsuite of gLite.
2. a testsuite is a simple script or executable, which returns '0' if the test(s) passed, and not '0' if an error or failure was detected. From this a standard XML report is generated and in turn an HTML report is as well generated. The standard report will include the pass/fail status, as well as the standard output and the standard error, as produced during the testsuite execution. Although the test report is standard,

it is very crued, since the only help as of the cause of the failure is in the standard output and error. An example of this test mode is the RGMA testsuite of gLite.

3. a testsuite takes the ownership of the generation of the entire HTML report. The tooling is then left with the job of wrapping this custom report into a top level report. An example of this test mode is the WMS and *Certification* testsuites, also referred sometimes as *Gilbert's testsuite*.

The choice of the testsuites probably depends on different factors, such as:

- type of test: unit, functional, stress, performance, system, etc
- language or technology of the item under test: command-line, web-service, API (C++, java, Python)
- complexity of testsuite
- specific requirements for custom report
- ...

Based on these, the appropriate type of testsuite should be selected. It would probably be a good idea to discuss these before launching into new development.

-- MebSter - 21 Jun 2006

Test Manager tool

TestManager is a simple tool that executes tests and generates an integrated HTML report. The tests can be written in an xUnit framework (i.e. currently supported CppUnit, JUnit and PyUnit) or in any other form (e.g. executable, script) where the return code defines whether the test passed or failed (0 = passed, >0 = error and <0 = failed). The TestManager ensures that the test results are formatted into an XML document in the format of the xUnit schema defined by EGEE. These reports are then transformed into HTML pages. The transformations are reused from the Apache Ant project.

The TestManager requires a simple configuration file, which define one or several fixtures (a term introduced by the original developers of junit which means a single or collection of test(s) packaged such that it can be executed) and there order of execution. Since a configuration file can point to another configuration file, and each fixture includes the possibility to `continueIfFail`, a complex orchestration of the tests is possible. At the end of the tests execution, all the tests outputs are merged together in order to provide a rich, hierarchical HTML report, with high-level test statistics, properties for each tests and a complete trace for failed tests when available.

Here's the format of the TestManager xml configuration file. Each file must contain a root element called '_tests_'. Each test fixture must be described by a 'test' element. To control the execution of the tests, the user must supply a test scenario file. The format of the scenario file is as follows: Attributes of element test:

Attribute name	Description	Type/Value
`fixture`	Name of the test to be executed	`string`
`type`	Type of the test	`string` with following possible values:
		`pyunit` for Python test using PyUnit
		`junit` for Java test using JUnit
		`cppunit` for C++ test using CppUnit
		`generic` for script or executable
		`subtest` for execution of another test scenario file. This is useful for building a hierarchy of tests and building dependencies between testsuites.
`continueIfFail`	(optional) Control whether to continue the testsuite following a failed test	boolean:
		`true` for continuing the tests following an error (default behaviour)

		`false` for stopping the execution of tests if this test fails.
`dir`	(optional) Set in which directory the fixture is, if not in the current directory	`string`
`shortDescription`	(optional) Short description of the test	`string`

To pass arguments to a test fixture (e.g. arguments to the test program), the user can define in an element `test` a nested element `arg`. This element accepts the following attributes:

Attribute name	Description	Type/Value
`name`	Argument	`string`

As defined by the xUnit format, the standard output and standard error are written into the test report, as well as the result for each test, and the reason for error or failure (if any) and the reason for the error (including a dump of the stack).

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<tests>
    <!-- 'generic' test called 'dir' (corresponding to the shell command) with arguments '*.py' -
    <test fixture="dir" type="generic">
        <arg name="*.py"/>
    </test>
    <!-- PyUnit test located in the current directory -->
    <test fixture="TestGenerateFailedReport" type="PyUnit"/>

    <!-- junit test in the local directory, and stop the tests if this test fails -->
    <test fixture="MyJUnitTest" type="junit" continueIfFail="false"/>

    <!-execute the tests defined in the test scenario file 'SubTests.xml', located in the relativ
    <test fixture="SubTests.xml" type="subtest" dir="data"/>

</tests>
```

You can find an example of generated report here.

Upon completion of all the tests, the TestManager creates a directory (if it doesn't exist) called 'report' and puts in it the HTML report. In order to allow tests to be replayed at a later stage, the TestManager also copies in this directory all the test configuration files. This directory can then be moved around with all the required information for re-running the same series of tests (assuming that the same tests are still available at the same place location).

The module also renames the report files to `TEST-< i >-< fixture >.xml`, where `< i >` is a counter of tests executed, starting from 1.

The module also fills the `properties` element of the test report with environment variables from the calling shell, the hostname, as well as the date/time at which the report was generated.

In order for the PyUnit package (unittest.py) to generate XML, xunittest.py is used. This Python module ships with TestManager.

TestManager standardises on the Ant/junit xml format. A schema for this format is available here.

-- MebSter - 21 Jun 2006

This topic: ETICS > TestingHowTo
Topic revision: r3 - 2006-12-13 - MebSter