Test Manager design for the ATLAS DAQ Prototype -1

Authors : R.Hart, H.Boterenbrood, W.Heubers

Keywords : DAQ, Test Manager, test, prototype -1, POSIX 1003.3, design

Abstract

The Test Manager (TM) provides a means of organizing individual tests for hardware and software components within the ATLAS DAQ system. A high-level design for the Test Manager in the ATLAS DAQ Prototype -1 system is presented.

NoteNumber : 066
Version : 3.0
Date : May 27 1998
Reference : http://atddoc.cern.ch/Atlas/Notes/066/Note066-1.html

1 Introduction

The purpose of the Test Manager (TM) is to provide a means of organising tests of hardware and software components within the ATLAS DAQ system. It is the responsibility of the TM to ensure the execution of tests and verification of their output. The TM is not responsible for the tests themselves. They should be created by the experts of the components. However, for implementing tests in a uniform way, templates, guidelines and a API library will be provided.

The user requirements for the TM are described in the back-end DAQ URD **[1]** and are again listed below:

1.1 Capability requirements

- **UR TMGR-1** The TM shall have the ability to execute individual tests.

1.2 Constraint requirements

- **UR TMGR-2** The TM shall run on all the platforms supported by the DAQ back-end software.
- **UR TMGR-3** Output from the individual tests shall comply to POSIX standard 1003.3.
- **UR TMGR-4** Apart from POSIX conforming output the TM shall not impose any other constraints on the individual tests.
- **UR TMGR-5** The TM shall only be used to execute tests when the DAQ is not taking data.
- **UR TMGR-6** The TM package shall provide tools and facilities to create tests in a uniform way and to add them to the global set of tests.

2 General considerations

2.1 What is a test?

A **test** exercises a software or a hardware component or a combination of both, resulting in a single result. It consists of one or more processes running on one or more computers.

Tests are intended to verify the functionality of a component and are not intended to modify the state of a component or to retrieve status information about a component. A test is not necessarily optimised for speed or use of resources. A test can only be run when the tested component is not taking part in data taking or otherwise is in use.

The TM is not meant to be used during the development of software components or to debug a piece of hardware. Nor will it be used to evaluate the quality of the software component or the efficiency of the used algorithm.

The outcome of a test will comply to POSIX 1003.3 and therefore should be one of: PASS, FAIL, UNRESOLVED or UNTESTED (for more information on POSIX 1003.3 see section A). However, there should also be a mechanism to convey more elaborate and detailed output of a test to the initiator of a test. This detailed output of a test have to be written into a logfile, so the launcher of the test (and NOT the TM) can investigate the results. The Message Reporting System [8] will be used to send information about the progress and to announce the completion (including its result) of the test to the launcher.

A log of all executed tests and their results will be kept in a central logfile.

2.2 Who runs tests?

The Run Control system [2] may test hardware and software components when necessary before starting data taking using the TM.

The Diagnostics Package uses the TM to perform logical sequences of tests to diagnose problems with the DAQ system or to confirm its functionality.

A DAQ or detector expert may perform individual tests or test sequences, through an interactive application with a graphical user interface; this application may be part of the Diagnostics Package or of the TM itself.

2.3 When to run tests?

There are two situations where we want to run tests:

1. On the basis of the detection of an error; the aim is to identify the problem and localise the error.
2. To check the functionality of a component; this can be done:
   - (automatically) on a regular basis (in between 'physics runs')
   - (automatically) on the basis of the state of the system:
     at start-up, at start-of-run, at a change of configuration, etc.
   - at a user's request

2.4 Getting Authorisation

The TM uses the Process Manager [3] to start, stop and monitor a test, which consists of one or more processes. A test can NOT be run when the hardware or software component to test is in use, by the DAQ for example (except possibly as part of the DAQ procedure itself). This requirement is implicitly fulfilled by the Process Manager, who consults the Resource Manager [7], before executing a process which is part of the test. Nevertheless, it could be helpful to know in advance whether it is permitted to execute a test or not. Therefore, the TM consults all active Run Controls and Resource Manager to check if it has permission to execute a particular process belonging to the test. The TM is not aware of partitioning and therefore partition independent.

2.5 Databases

Several databases are needed to store information related to the TM:

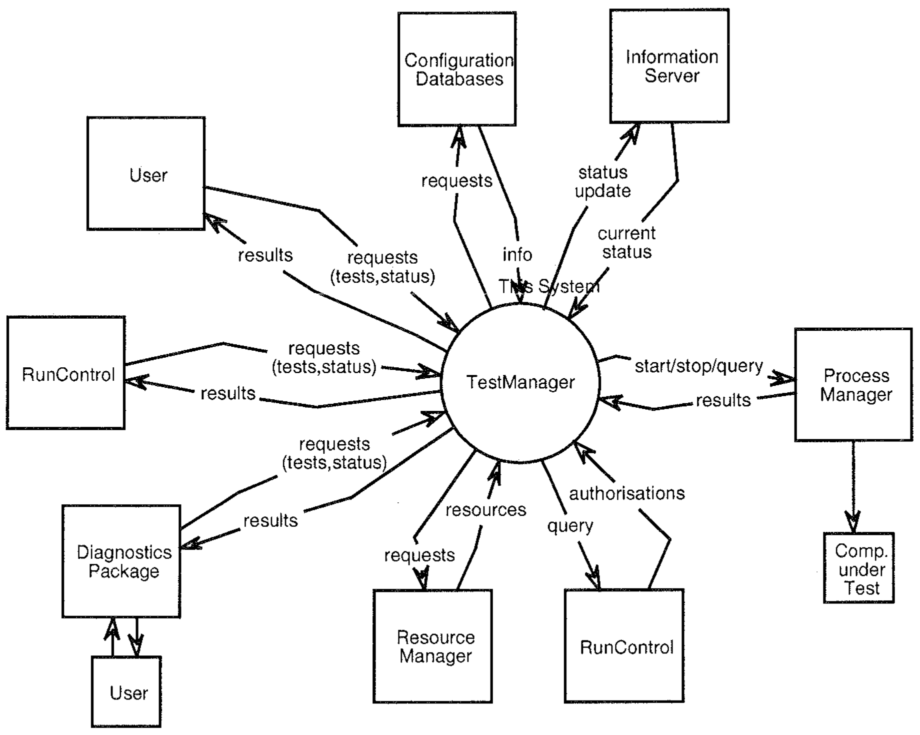
- A database containing descriptions of the tests; this will be part of the Configuration Database [4].

- A dynamic database containing a list of ongoing tests; the services provided by the Information Service [5] could be used for this.
- A database containing logical sequences of tests; this could be part of the future Diagnostics Package.

3 System Context

A detailed context diagram of the TM is shown in figure 1.

Figure 1: Detailed context diagram of the Test Manager in the ATLAS Back-end DAQ system.



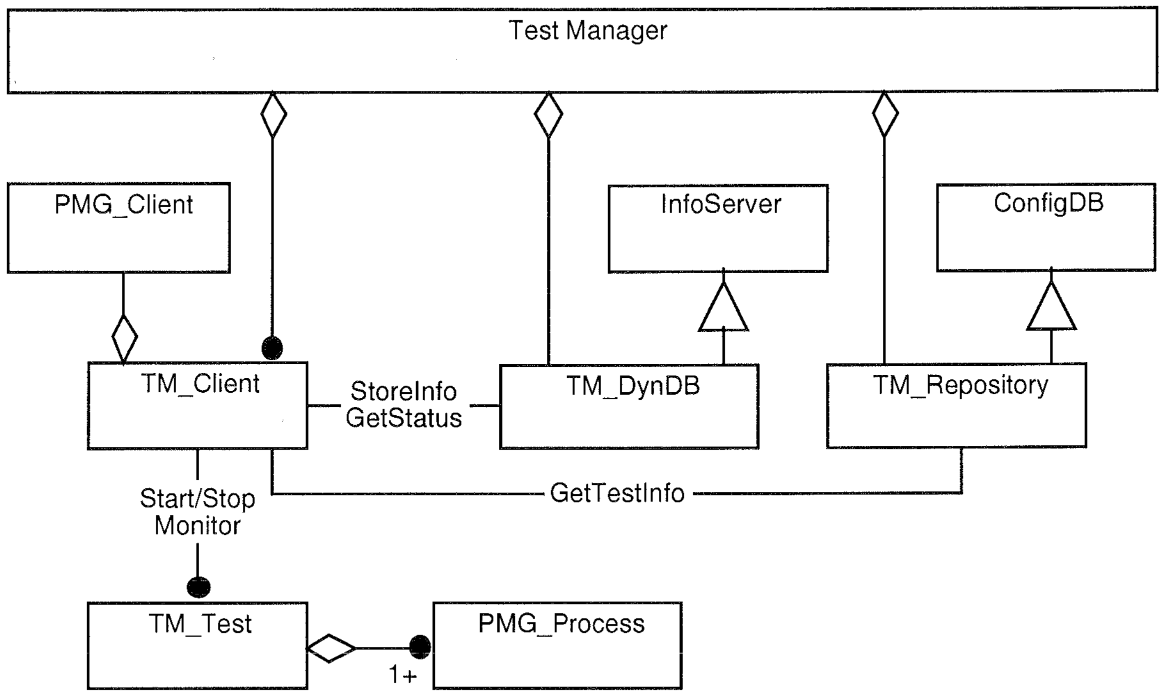
4 Object Model

We distinguish 4 types of objects in the model of the TM:

1. Client objects that will allow outside objects ('users') access to the TM services. They are able to start, stop and monitor tests.
2. The actual running tests.
3. A static database containing all information necessary to execute a particular test.
4. A dynamic database object to store which tests are active.

The Object Model of the TM thus obtained is shown in figure 2.

Figure 2: Object Model of the Test Manager.



The model presented here derives to a significant degree from other components developed within the back-end daq framework, like the Process Manager, Information Service and the Configuration Databases. The global class on top represents the Test Manager service within the back-end DAQ environment. The following paragraphs describe for the other TM classes their functionality.

4.1 TM_Client

This class implements the client part of the TM. It is the only object visible to the user and it is the interface between user programs and the TM. The TM_Client interface is shown in table 1. The *test* argument is a reference to a particular test inside the TM_Repository. The *handle* argument is used to identify an active test.

The most important task of the TM_Client class is to launch a test. A test consists of one or more processes. Therefore, the implementation of this class depends heavily on the Process Manager (PMG). It uses an instance of PMG_Client to start and stop processes. Only the initiator of a test may stop the test.

A TM_Client object obtains information about the available tests and which tests are active, by browsing respectively through the TM_Repository and TM_DynDB. The default attributes for the configuration and arguments for the test are retrieved from the TM_Repository.

The TM_Client will request permission to run a test from all running Run Control and possibly obtain the necessary system resources (tokens) from the Resource Manager. The absence of Run Control is no obstacle for the TM to continue. If permission is granted the

Process Manager is requested to start the processes belonging to the test through the instance of a PMG_Client; through the same interface requests for status and termination of the test (if required) is done.

Table 1: TM_Client Interface

TM_Client		
Attribute	**Type**	**Default Value**
identification	client identification	
Operation	**Arguments**	**Return Type**
StartTest	test, parameters	handle
StopTest	handle	
StopAllMyTests		
ListMyActiveTests		handle list
ListAllActiveTests		handle list
ListAllTests		test list
GetTestInfo	test	attributes

4.2 TM_Test

The TM_Test object represents an active test. So it consists essentially of one or more PMG_Processes started through a PMG_Client interface in the TM_Client object. The creator of the test has to take several constraints into account. At start-up the test has to inform the launcher in which logfile the specific output will be placed. During the lifetime of the test, the launcher has to be notified about the progress of the test. Finally, on completion the test has to notify the launcher the result of the test and place it in the central logfile. All information is exchanged using the MRS [8].

4.3 TM_Repository

The TM_Repository database relies on the Configuration Database (see [4]) for its implementation. It contains for every test:

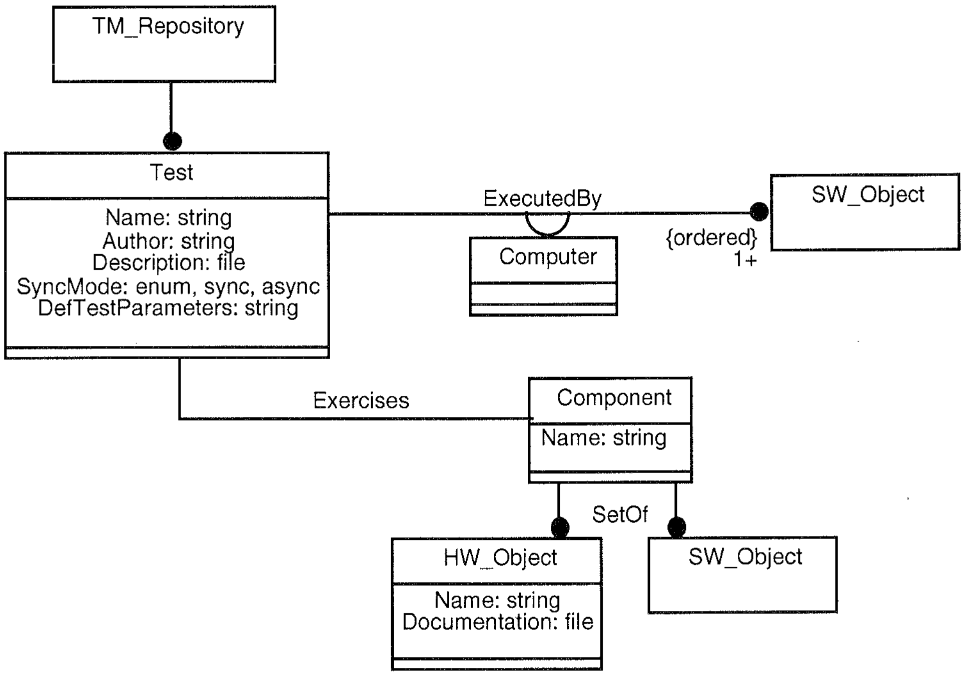
- name of test
- set of processes implementing the test
- component under test
- default test configuration and parameters
- description and author

Figure 3 shows the object model of the TM_Repository. It consists of a set of Test objects. This object is the entry point in the repository describing a particular test. The *test* argument used in the TM_Client interface is a reference to this object.

An active test consists of one or more processes, which are represented by the SW_Object. This object is described in the design of the Configuration Database [4]. The link between Test and SW_Object is ordered, because in case the test is composed of more than one process, the order in which the processes are started is significant. The *SyncMode* attribute denotes whether the processes have to be started synchronously or asynchronously. The SW_Object is platform independent. Therefore the Computer link attribute is necessary to select the right binary.

A test exercises a component. This object is a set of SW_Objects and HW_Objects. The HW_Object should point to some object in the hardware configuration database. Such an object is not available yet.

Figure 3: Object Model of the TM_Repository.



4.4 TM_DynDB

The TM_DynDB contains a list of all active tests. It will rely upon the InfoServer [5] for its implementation.

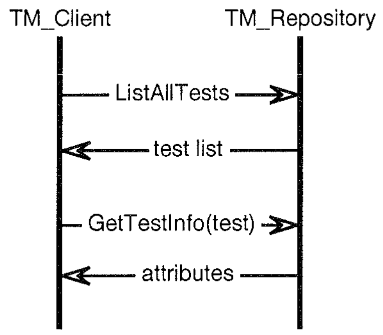
5 Object Interactions

In the following paragraphs, the TM object interactions in the form of event trace diagrams show in more detail the message sequences between the various objects for several possible scenarios.

5.1 Selecting a test from the repository

1. The TM_Client requests the TM_Repository for a list of all tests. The list is returned to the client.
2. From this list a test is selected. Additional information of this test is retrieved by the GetTestInfo message. The repository returns the attributes belonging to the selected test.

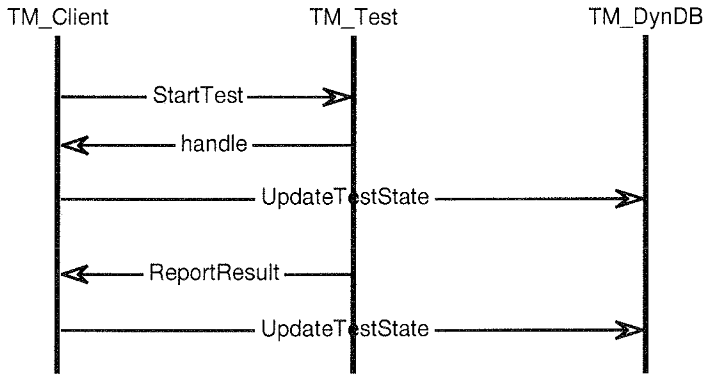
Figure 4: Selecting a test from the repository.



5.2 Executing a test

1. The TM_Client object launches a test
2. The TM_DynDB is notified this particular test is active.
3. The result of the test is reported to TM_Client, when the test has been completed.
4. TM_Client informs TM_DynDB the test has ended.

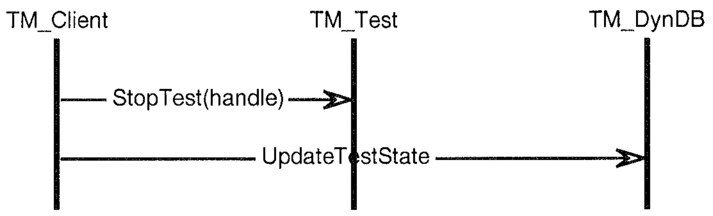
Figure 5: Executing a test.



5.3 Stopping a test

1. The TM_Client object stops the test. This test must have been started earlier by the TM_Client itself.
2. The TM_DynDB will be notified that the test has been interrupted by TM_Client.

Figure 6: Stopping a test.



6 References

[1]**ATLAS DAQ Back-end software,** User Requirements Document, Draft Revision 3, PSS05-ATLAS-DAQ-SW-URD, 30 May 1996.

[2]P.Duval, R.Jones, S.Kolos, S.Wheeler, **ATLAS -1 Run Control System design**, Technical Note 29 V1.3, 26 June 1997 (URL: http://atddoc.cern.ch/Atlas/Notes/029/Note029-1.html).

[3]P.Duval, A.Levansuu, Z.Qian, **ATLAS -1 Process Manager Design**, Technical Note 28 V1.0, 26 June 1997 (URL: http://atddoc.cern.ch/Atlas/Notes/028/Note028-1.html).

[4]R.Jones, M,Michelotto, A.Patel, I.Soloviev, S.Wheeler, **Design of the Configuration Databases for ATLAS DAQ Prototype -1**, Technical Note 30 V1.0, 3 Feb. 1997 (URL: http://atddoc.cern.ch/Atlas/Notes/030/Note030-1.html).

[5]M.Caprini, P.Duval, R.Jones, S.Kolos, **Information service for ATLAS DAQ Prototype - 1**, Technical Note 31 V1.0, 3 Feb. 1997 (URL: http://atddoc.cern.ch/Atlas/Notes/031/ Note031-1.html).

[6]Rob Savoye, **The DejaGnu Testing Framework** for DejaGnu Version 1.3, Jan. 1996 (URL: http://phantom.iweb.net:80/docs/gnu/dejagnu).

[7]I.Alexandrov, V.Iambourenko, R.Jones, V.Kotov, V.Roumiantsev, **High-level design of the Resource Manager**, Technical Note 052 V1.2, 2 Feb 1998 (URL: http://atd-doc.cern.ch/Atlas/Notes/052/Note052-1.html).

[8]D.Burckhart, M. Caprini, S.Kolos, Z.Qian, **Design of the Message Reporting System for the ATLAS DAQ prototype-1**, Technical Note 032 V1.0, 30 Jan. 1997 (URL: http://atd-doc.cern.ch/Atlas/Notes/032/Note032-1.html).

APPENDIX

A On POSIX 1003.3

The following description is based on the DejaGnu documentation **[6]**.

DejaGnu is a software testing utility, a framework for testing other programs. DejaGnu is written in `expect`, a tool to control and automate interactive application, which in turn uses `tcl` (Tool Command Language).

POSIX standard 1003.3 defines what a testing framework needs to provide, in order to permit the creation of POSIX conformance test suites. This standard is primarily oriented to running POSIX conformance tests, but its requirements also support testing of features not related to POSIX conformance. POSIX 1003.3 does not specify a particular testing framework.

The POSIX documentation refers to assertions. An assertion is a description of behaviour. For example, if a standard says "The sun shall shine", a corresponding assertion might be "The sun is shining." A test based on this assertion would pass or fail depending on whether it is daytime or night-time. It is important to note that the standard being tested is never 1003.3; the standard being tested is some other standard, for which the assertions were written.

As there is no test suite to test testing frameworks for POSIX 1003.3 conformance, verifying conformance to this standard is done by repeatedly reading the standard and experimenting. One of the main things 1003.3 does specify is the set of allowed output messages, and their definitions. Four messages are supported for a required feature of POSIX conforming systems, and a fifth for a conditional feature.

These definitions specify the output of a test case:

- **PASS**

  A test has succeeded. That is, it demonstrated that the assertion is true.

  POSIX 1003.3 does not incorporate the notion of expected failures, so a success when a failure was expected should return PASS.

- **FAIL**

  A test has detected a problem it was intended to capture. That is, it has demonstrated that the assertion is false. The FAIL message is based on the test case only. Other messages are used to indicate a failure of the framework. As with PASS, POSIX tests must return FAIL even when a failure was expected.

- **UNRESOLVED**

  A test produced indeterminate results. Usually, this means the test executed in an unexpected fashion; this outcome requires that a human being go over results, to determine if the test should have passed or failed. This message is also used for any test that requires human intervention because it is beyond the abilities of the testing framework. Any unresolved test should resolved to PASS or FAIL before a test run can be considered finished. Note that for

POSIX, each assertion must produce a test result code. If the test isn't actually run, it must produce UNRESOLVED rather than just leaving that test out of the output. Here are some of the ways a test may wind up UNRESOLVED:

- A test's execution is interrupted.
- A test does not produce a clear result. This is usually because there was an ERROR while processing the test, or because there were three or more WARNING messages (this is the case in DejaGNU). Any WARNING or ERROR messages can invalidate the output of the test. This usually requires a human being to examine the output to determine what really happened--and to improve the test case.
- A test depends on a previous test, which fails.
- The test was set up incorrectly.

- **UNTESTED**

  A test was not run. This is a placeholder, used when there is no real test case yet.

The only remaining output message left is intended to test features that are specified by the applicable POSIX standard as conditional:

- **UNSUPPORTED**

  There is no support for the tested case. This may mean that a conditional feature is not implemented.