Implementation of the Test Manager for the ATLAS DAQ Prototype -1

Authors : R.Hart

Keywords : DAQ, Test Manager, test, prototype -1, implementation

Abstract

This note describes the current implementation of the Test Manager (TM). The TM provides a means to start and stop tests in a predefined manner. It consists of two main parts: a client class, which is the interface to the user and a repository class, which contains a set of tests. The TM uses the Process Manager (PMG) to start and stop a test and the DAQ Configuration Database to store the tests.

NoteNumber : 111
Version : 2.0
Date : October 11 1999
Reference : http://atddoc.cern.ch/Atlas/Notes/111/Note111-1.html

1 Introduction

The TM is an ATLAS back-end service to use tests in an organized way. The tests themselves have to be written by the hardware/software experts and are not the responsibility of the TM. The implementation of the TM is based on the high-level design **[1]**. However, the current implementation deviates from the design for the following subjects:

1. Each test consists of **one** process. Later versions should allow multiple processes on different hosts. The process is started by the PMG **[2]**.
2. Instead of the TM_Repository object model of the high-level design, the Software model of the Configuration Database **[3]** is used. A test is considered as a kind of application. The back-end Software Data Access Library (DAL) **[4]** is used to retrieve information from the database.
3. The TM_DynDB is not implemented yet. Instead each TM_Client instance maintains its own list of running tests. Hence, they are invisible for the outside world.
4. The Resource Manager **[5]** is not used yet. This component distributes tokens, which should be allocated before a test is allowed to start. This issue is also valid for the PMG.
5. The *component* class, as part of the TM_Repository object model, is not supported. The TM implementation can do without, because it is a passive object only describing a hard- or software object to be tested.

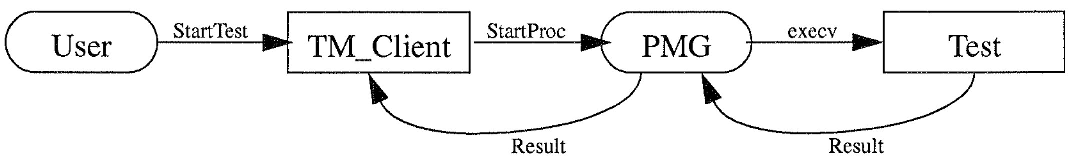
2 Test Manager architecture and implementation

The high-level design showed two object models: a global one and a refined one of the TM_Repository. With the premises of the previous chapter in mind, the implementation is basically limited to the TM_Client class and TM_Repository class.

2.1 TM_Client

The TM_Client class, as described in the high-level design, implements the client part of the TM and is the only object visible to the user. Its main task is to launch tests. Figure 1 shows the typical flow of a test in relationship with the PMG. The exit status of a test is used to pass the result of the test.

Figure 1: Typical flow of a test.



In order to use the PMG each TM_Client creates at construction an instance of the PMG_CLIENT class. Tests are started by the *StartProcS* method of the PMG_CLIENT object. On completion, the PMG notifies the TM_Client object by calling a call-back routine. Each TM_Client object has a dedicated call-back routine for this purpose, known as *TmPmgCallback*. This routine is known to the TM_Client only and is defined as *friend*, which enables it to access private methods. The reason for this is that in C++ it is not allowed to use methods as call-back routine. Apart from the user supplied arguments, two important arguments for the *StartProcS* method are supplied by the TM_Client object. The first one is a pointer to the *TmPmgCallback* routine and the second one is a pointer to the object of TM_Client itself (also known as *this*). When the test finishes, this pointer is together with the result of the test passed as argument to the *TmPmgCallback* routine. This mechanism re-establishes the connection between PMG and TM_Client.

As mentioned before the exit status is used to pass the result of a test. The PMG system catches the result and passes it to the TM_Client by means as described above. The result has to comply with the POSIX 1003.3 definition **[6]**. The following results are valid and defined in the general include file of the TM `<tmgr/tmgr.h>`:

```
enum tmResult
{
     TmPass          = 0;
     TmUndef         = 182;
     TmFail          = 183;
     TmUnresolved    = 184;
     TmUntested      = 185;
     TmUnsupported   = 186;
};
```

A successful test will return `TmPass,` which has the value 0. According to the exit status convention [7] this is correct. The other values are more or less arbitrary and in terms of the TM correct (the test returns a proper exit value), but for the convention they indicate an error. An exit status not equal to zero means an error. This situation remains for the time being unsolved.

There are two methods to start a test (asynchronous and synchronous), called *StartTestA* and *StartTestS*. The asynchronous method is non-blocking. On completion of a test the user is notified by a user supplied call-back routine (not to be confused with the internal call-back routine mentioned above). This routine has to be supplied at construction, which implies that for each TM_Client instance only one call-back routine should be kept. The synchronous method is of course blocking. On completion this method returns the result of the test. The implementation of this method is done by means of an *ipc_server* **[8]**. The *run* method of this server blocks the flow of the program, but handles all incoming call-back routines, including the internal *TmPmgCallback* routine handling the PMG calls. If this routine detects the completion of the synchronously started test, it calls the *stop* method of the *ipc_server*, in this way resuming the flow of the program.

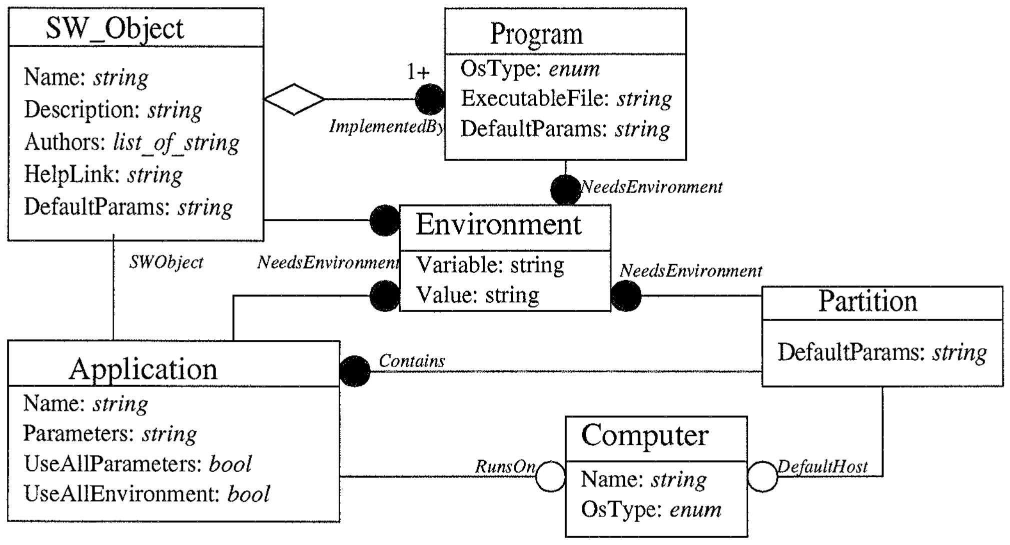
A TM_Client instance maintains a list of started tests, called the *Legend.* It contains dynamic information of a test such as current state, host, start and stop time, result, etc. The TM_Client class provides a public method to obtain this *Legend.* After a test has been launched successfully, a list item is added. The *TmPmgCallback* will on termination complete the list item. To ensure the integrity of the list a simple mutual exclusion algorithm (based on Peterson's solution) is used.

No more than one instance of a TM_Client per process is allowed.

2.2 TM_Repository

The TM_Repository is part of the ConfigDB and based on the Software and Configuration view. The tests are stored in this database. A test is considered as a kind of application. Not all objects and attributes of this scheme are used. The scheme used by the TM is shown in figure 2.

Figure 2: Current Object model of the TM_Repository.



Although the Partition object is not necessary for the TM, an instance of it is required. It is used as entry point for the DAL. The TM follows the rules for the parameters and environment as described by the DAL Users guide. The *UseAllParameters* and *UseAllEnvironment* attributes of the Application class switches this behaviour on or off. If the computer argument of a *StartTest* method is not given, then it will use the *RunsOn* computer. If this computer is not set, the *DefaultHost* of the Partition class is used. An error is returned if this one is also not set. In general the *RunsOn* computer is the default for the test and the *DefaultHost* computer is the default for the entire repository. The repository has to contain a list of computers on which the test can run. If a given computer is not present in the list, an error is returned.

The TM_Client class provides several methods to retrieve all kind of information from the repository. The *ExecutableFile* attribute from the Program class denotes the binary of the test and is passed as argument to the PMG, which has its own strategy to select the proper binary.

The TM_Repository class is virtual. There is no implementation (source-code) of the TM_Repository class. Instead the functionality of it is completely covered by the back-end Software DAL and TM_Client.

3 References

[1]R.Hart, H.Boterenbrood, W.Heubers, **Test Manager design for the ATLAS DAQ Prototype-1,** Technical Note 066, V3.1, June 19 1998 (URL: http://atddoc.cern.ch/Atlas/Notes/066/Note066-1.html).

[2]P.Duval, L.Cohen, **Users guide for the Process Manager**, Technical Note 081, V1.0, 2 Feb. 1998 (URL: http://atddoc.cern.ch/Atlas/Notes/081/Note081-1.html).

[3]R.Jones, M,Michelotto, A.Patel, I.Soloviev, S.Wheeler, **Design of the Configuration Databases for ATLAS DAQ Prototype -1**, Technical Note 030, V1.1, 26 Jun. 1997 (URL: http://atddoc.cern.ch/Atlas/Notes/030/Note030-1.html).

[4]I.Soloviev, **Configuration Databases User's Guide**, Technical Note 135, V1.0, 6 Oct. 1999, (URL: http://atddoc.cern.ch/Atlas/Notes/135/Note135-1.html).

[5]I.Alexandrov, V.Iambourenko, R.Jones, V.Kotov, V.Roumiantsev, **High-level design of the Resource Manager**, Technical Note 052, V1.2, 12 Feb. 1998 (URL: http://atddoc.cern.ch/Atlas/Notes/052/Note052-1.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]D.Burckhart, **Atlas DAQ Conventions - Proposal**, 29 Jul. 1999 (URL: http//atddoc.cern.ch/Atlas/DaqSoft/sde/inspect/ATLAS_DAQ_Conventions.html).

[8]S.Kolos, **Inter Process Communication package**, V1.5, 15 Dec. 1998, (URL: http://atddoc.cern.ch/Atlas/Notes/075/Note075-1.html).