User Guide of the Test Manager for the ATLAS DAQ Prototype -1

Authors : R.Hart
Keywords : DAQ, Test Manager, test, prototype -1, user guide

Abstract

This note represents the user guide for the Test Manager (TM) in its current state. It describes how to use the different possibilities the TM offers. Not only the usage of the TM is handled, also a description is given how to make a test and how to store it in the repository.

NoteNumber : 112
Version : 2.0
Date : October 13 1999
Reference : http://atddoc.cern.ch/Atlas/Notes/112/Note112-1.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] and is described in the implementation report [2]. It consists basically of two classes. The TM_Repository class which deals with the retrieval of the tests from the repository and the TM_Client class, which is the only object visible to the user. Therefore, this user guide will be concentrated on the TM_Client class. The base include file for the TM is `<tmgr/tmgr.h>`.

2 TM_Client

There are three aspects which are important if we use an instance of the TM_Client class:

1. **Partition:** Although the TM is partition independent an instance of a TM_Client class has to run in a partition. If not given, the default partition (`TMGR_DFLT`) is chosen.
2. **Repository:** The TM_Client instance needs a repository containing a set of tests. This repository argument is passed as a single file name, which has to comply to the rules of the federated database as described in [3]. If no data file is given, a default one is used, containing a set of tests known as the *demo test-suite*, which is meant to show the capabilities of the TM.
3. **Call-back**: The user may supply a call-back routine, which is invoked each time an asynchronously started test finishes. Only one call-back routine per TM_Client instance is possible. It is supplied at construction.

Only one instance of a TM_Client per process is allowed. Creating a second instance yields the following error message:

```
TM_Client: only ONE instance allowed
```

2.1 Constructors

The TM_Client class provides 2 constructors:

2.1.1 Basic constructor

Synopsis: `TM_Client();`

Creates an instance in the default partition, uses the *demo test-suite* repository and has no call-back routine.

2.1.2 General constructor

Synopsis:

```
TM_Client( const char *partition,
           const char repository,
           unsigned long callbackptr,
           void *callbackparam = 0);
```

Creates an instance in the given partition and uses the repository argument as file of a federated database. If `partition` is 0, the default is used. If `repository` is 0, the *demo test-suite* is used. If `callbackptr` is 0, no call-back is supported. The `callback-param` can be used freely by the user and is passed back as argument when the call-back is invoked. The syntax of the user supplied call-back routine is as follows:

```
void callBack(RWCString handle, TestResult result, void *param);
```

The `handle` argument belongs to the test which was previously started by the StartTestA method, which returned the same handle. The `result` argument contains the test result and the `param` argument is the user supplied call-back parameter.

2.2 Interface

This chapter gives an overview of the public methods of the TM_Client class. Several methods return in a way an error status. Most of them are quite obvious. If not, they are explained in more detail.

2.2.1 GetStatus

This method should be called after construction to see if something went wrong. A possible error could be an inconsistent repository. Returns **true** if ok, **false** otherwise. Synopsis:

```
RWBoolean GetStatus();
```

2.2.2 StartTestA

Start a test in asynchronous way, what means it uses the Process Manager (PMG) **[4]** to start the process which executes the test. Synopsis:

```
RWCString StartTestA(RWCString test,      // name of test in repository
                     RWCString host,      // host on which test runs
                     RWCString args,      // user supplied arguments
                     int &errstat);       // error status
```

If the `host` variable is not set, then the default host as set in the repository is taken. If both are not set an error occurs (TM_HOST_NOT_SET). The `args` variable is placed with the highest priority together with the default parameters found in the repository. If `errstat` is TM_SUCCESS, the returned RWCString contains the created process handle as supplied by the PMG system. This handle is passed as argument with the call-back routine on completion. On failure a NULL handle is returned and the reason can be found in the `errstat` variable. Possible reasons are:

```
TM_SUCCESS:              Start of test succeeded.
TM_TEST_NOT_FOUND:       Test not found in repository.
TM_MACHINE_NOT_FOUND:    Host not found in repository.
TM_NO_PROGRAM:           No binary for test on selected platform.
TM_HOST_NOT_SET:         Host and default host not set.
TM_RM_NOT_ALLOWED:       Refused by Resource Manager.
TM_PMG_ERROR:            PMG failed to start process.
```

The TM_NO_PROGRAM error means that no binary is available for the selected test on the selected machine. The TM_RM_NOT_ALLOWED error indicates that the test did not get authorization from the Resource Manager [5].

2.2.3 StartTestS

Start a test in synchronous way, what means it uses the Process Manager (PMG) [4] to start the process which executes the test. Synopsis:

```
TestResult StartTestS(RWCString test,     // name of test in repository
                      RWCString host,     // host on which test runs
                      RWCString args,     // user supplied arguments
                      int &errstat);      // error status
```

The semantics of the arguments is the same as for the StartTestA method. The list of possible failures, set in the `errstat` variable, is also the same. This call blocks until the test finishes. The result of the test is returned, provided that `errstat` is TM_SUCCESS.

2.2.4 StopTest

The StopTest method terminates a running test. Only tests which are started by the current TM_Client instance can be stopped. Tests initiated by other TM_Client objects cannot be stopped. The synopsis is:

```
int StopTest(RWCString handle);
```

The `handle` argument has to be one of the previously issued StartTestA calls. The result of the call is returned and can have the following reasons:

```
TM_SUCCESS:              Stop of test succeeded.
TM_PMG_ERROR:            PMG failed to stop process.
TM_INVALID_HANDLE:       Unknown handle.
```

2.2.5 ListAllTests

A list of available tests from the repository is obtained by the ListAllTests method. The synopsis is:

```
test_list ListAllTests();
```

The `test_list` type is defined as follows:

```
typedef RWTValSlist<RWCString> test_list;
```

The method returns a list of `RWCString`'s which contains the names of the tests.

2.2.6 ListAllMachines

A list of machines from the repository is obtained by the ListAllMachines method. Tests can only run on machines from this list. The synopsis is:

```
mach_list ListAllMachines();
```

The `mach_list` type is defined as follows:

```
typedef RWTValSlist<RWCString> mach_list;
```

The method returns a list of `RWCString`'s which contains the names of the machines.

2.2.7 GetTestInfo

The GetTestInfo method retrieves all static information of a particular test from the repository. The synopsis is:

```
int GetTestInfo(RWCString test, test_info &info);
```

The result of the call is returned and can have the following values:

```
TM_SUCCESS:              Information of test retrieved.
TM_TEST_NOT_FOUND:       Test not found in repository.
```

On success the `info` variable is filled. The `test_info` type is defined as follows:

```
typedef struct testinfo
{
     RWCString                  ti_test;          // name of test
     RWTValSlist<RWCString>     ti_authors;       // author list
     RWCString                  ti_helplink;      // link to help file
     RWCString                  ti_defhost;       // default host
     RWCString                  ti_description;   // description of test
     RWTValSlist<ConfdbComputer::OS>
                                ti_programs;      // possible platforms
} test_info;
```

The `ti_programs` variable is a list of operating systems for which a binary of the test exists. The meaning of the other elements are straightforward.

2.2.8 GetVarsInfo

The GetVarsInfo method obtains the variable information of a test to be run on a certain machine. The synopsis is:

```
int GetVarsInfo(RWCString test, RWCString machine, vars_info &info);
```

The result of the call is returned and can have the following values:

```
TM_SUCCESS:              Information of test and machine retrieved.
TM_TEST_NOT_FOUND:       Test not found in repository.
TM_MACHINE_NOT_FOUND:    Machine not found in repository.
TM_NO_PROGRAM:           No binary for test on selected platform.
```

The TM_NO_PROGRAM error means that no binary program is available for the selected test and computer.

On success the `info` variable is filled. The `vars_info` type is defined as follows:

```
typedef struct varsinfo
{
     RWCString            vi_executable;      // name of binary
     RWCString            vi_defparams;       // default parameters
     RWCString            vi_environment;     // environment string
     RWCString            vi_machine;         // DNS name of machine
     ConfdbComputer::OS   vi_os;              // platform
     ConfdbComputer::Type vi_type;            // type of machine
} vars_info;
```

2.2.9 GetHistory

Each TM_Client interface maintains a list called the *Legend*. It contains information of each test which was successfully launched. This implies that the list continues to grow. The list can be obtained by the `GetHistory` method. The synopsis is:

```
test_log GetHistory();
```

On construction the *Legend* is empty. Each successful call of StartTestA or StartTestS adds an element to the list. The contents of the elements are maintained by an internal call-back routine.

The `test_log` type is defined as follows:

```
typedef RWTValSlist<test_data> test_log;
```

The list contains items of type `test_data`, which is defined as follows:

```
typedef struct testdata
{
     unsigned long   td_tid;        // test-ID number
     RWCString       td_handle;     // handle of the started test
     RWCString       td_test;       // name of started test
     RWCString       td_host;       // host on which test runs
     RWCString       td_args;       // supplied arguments
     TestState       td_state;      // running, stopped, ready
     RWTime          td_start;      // start time of test
     RWTime          td_stop;       // stop time of test
     TestResult      td_result;     // POSIX 1003.3 result value
} test_data;
```

2.2.10 IsSyncBusy

This method returns **true** if the TM_Client object is performing an StartTestS call and waiting for the result of a test. Returns **false** otherwise. In principle it should be impossible to get out of the blocking StartTestS call. If however, a user manages to get out of it, for instance by means of an IPC alarm or input handler, then this method could be helpful to see if a StartTestS call is active. The synopsis is:

```
RWBoolean IsSyncBusy();
```

2.2.11 GetLastPmgStatus

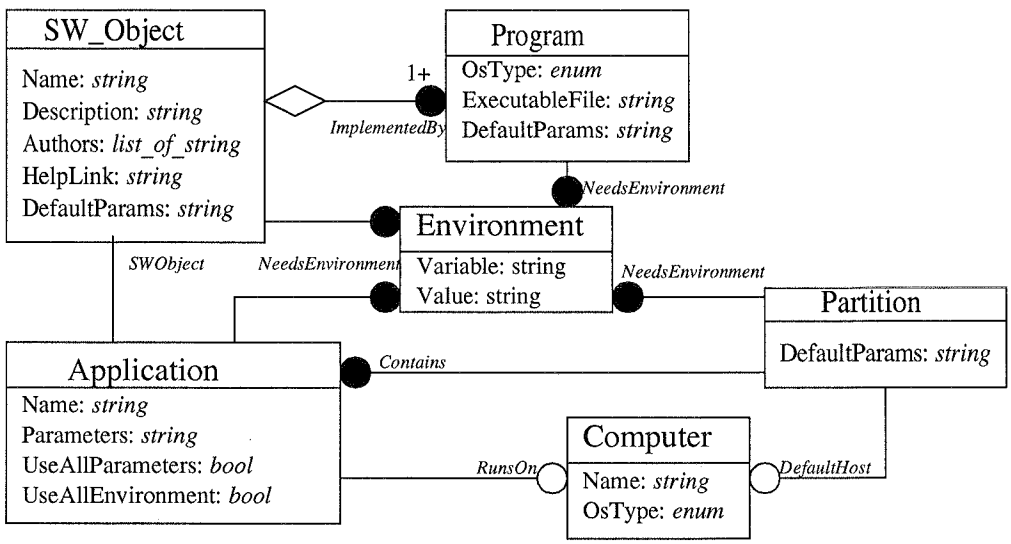
If a TM_Client method yields a TM_PMG_ERROR, then the actual value of the PMG error is returned by this method. The synopsis is:

```
int GetLastPmgStatus();
```

3 TM_Repository

The current implementation of the TM_Repository is based on the Software and Configuration view of the ConfigDB **[6]** as shown in figure 1. Only the classes and attributes relevant for the TM_Repository are visualized. The main idea of this scheme is that a test is a kind of application. The GetTestInfo and GetVarsInfo methods of the TM_Client class use the back-end Software Data Access Library (DAL) **[3]** to retrieve their information from the repository.

Figure 1: Current Object model of the TM_Repository.



A general comment about the name attribute of several classes; each object has an ID, implemented as string, which in some cases is used instead of the name attribute. The following list summarizes the meaning of the classes:

- **Partition**: Although this class is irrelevant for the TM_Repository, an instance of this class has to be the present, because it is the entry point of the DAL.
- **Computer**: A test can only run on a machine if the corresponding Computer object is present in the repository. The ID of the object is used as key and not the name attribute! The name attribute must contain the full DNS domain, which is used by the PMG to select the right agent. For instance: in the *demo test-suite* there is a computer called `sunatdaq01` with the following name attribute: `sunatdaq01.cern.ch`. The OsType (solaris, lynx, linux, etc.) is used to select the corresponding binary (Program object). The GetVarsInfo also returns the type (Workstation, CPU Board or I/O CPU) attribute, which is meaningless for the TM.

- **Application**: In fact the **test**! Like the Computer class the ID is used as key to select the test. The name attribute is not used. The *UseAllParameters* and *UseAllEnvironment* attributes are important for the aggregation of the parameter list and environment string.
- **SW_Object**: Each application (read test) is described by a single SW_Object, which delivers some static information like the list of authors, a helplink and a description. An important feature of this object that it points to a list of binaries (Programs). For each platform (OS type) at most one instance may exist.
- **Program**: This class contains the name of the binary belonging to the test. The name convention for the executable is that of the PMG. Thus, the *ExecutableFile* attribute starting with a '/' is considered to be absolute. If not, the current PMG will look in the "bin" directory of the proper platform and from the selected SRT release.
- **Environment**: The Partition, Application, SW_Object and Program class may have a set of environment variables. The *variable* and *value* attributes speak for themselves.

Most elements obtained by the GetTestInfo and GetVarsInfo methods are retrieved directly from the objects mentioned above. However, a couple of elements are obtained in a different way.

Default Host

To select the default computer the following algorithm is used. The *RunsOn* computer (pointed by the Application object) is selected first. If not set, the *DefaultHost* computer (pointed by the Partition object) is used. If this one is also not set, the default host is set to 0. In general, the *RunsOn* computer is the default for the test, the *DefaultHost* computer is the default for the entire repository.

Default Parameters

The default parameters are obtained from the repository according to the "Application command line" specification as described in the DAL User's Guide **[3]**. The *UseAllParameters* attribute of the Application class influences the result. Note that the user supplied arguments are placed at the end of the command line, thus having the highest priority.

Environment

The environment in which the process should run, is obtained according to the "Application Environment" description as described in the DAL User's Guide **[3]**. The *UseAllEnvironment* attribute of the Application class influences the result.

3.1 Demo test-suite

The *demo test-suite* contains a couple of tests, named after an exotic cocktail, which all executes the *tmgr_sleep* program. The synopsis of this program is as follows:

```
tmgr_sleep [-t delta] [-E exit]
```

Without arguments this program sleeps for 30 seconds and exits with value `TmPass`. Argument `-t` sets the duration of the sleep and the `-E` argument sets the exit (result) value.

4 Building and Running a client program

A user program, written in C++, which uses the TM has to include **<tmgr/tmgr.h>.** It contains all definitions, structures and type definitions mentioned in the previous chapters. In order to compile and link your source properly, look at the ATLAS DAQ Software Development Environment [7] on the web. To link your program, the following list of libraries has to be included:

```
-ltmgr -lpmg -lis -lipc -lilu-c++ -lilu -lswdal -lconfdb -loks -lrwtool
```

Furthermore, the `@socket-libs@` variable must be added to the previous line. It contains the platform dependant communication libraries.

In order to execute your program properly, you should modify your `LD_LIBRARY_PATH`. Before you can use your program, it needs some support from other components. First of all it needs a default *ipc_server* and an *ipc_server* running in the partition you would like to work in. Each test is executed by one process, which means that on the host where the process will be running, a *pmg_agent* has to run in the same partition. Using the PMG system requires also an *is_server*. The following commands have to be started first. Lets assume we use the default TM partition (`TMGR_DFLT`). The *ipc_server*'s and *is_server* have to run somewhere on an arbitrary host within the AFS framework. The *pmg_agent* has to run on each machine you want to execute a test.

```
bash# ipc_server &
bash# ipc_server -p TMGR_DFLT &
bash# is_server -p TMGR_DFLT -n PMG &
bash# pmg_agent -p TMGR_DFLT &
```

Some programs may yield a lot of warnings and/or error messages. The general *ipc_server* may already run. Ignore the warnings and continue to start the partition dependant *ipc_server*. The *pmg_agent* will complain about a non running MRS server, which can also be ignored. Other messages may indicate a serious error.

5 Utility Programs

The TM provides several utility programs, to make life a little easier. Each utility supports the `[-p partition]` argument. If not set, the default partition (TMGR_DFLT) is used. Furthermore, every utility supports the `[-d repository]` argument. If not set the *demo testsuite* is used. Test results and possible errors are printed in a readable form (not a cryptic number).

5.1 tmgr_list

Prints a list of available tests. If the `[-m]` flag is set, the list of available computers is printed. Synopsis:

```
tmgr_list [-m] [-p partition] [-d repository]
```

5.2 tmgr_info

Print static or variable information about a test. Synopsis:

```
tmgr_info -T test [-m machine] [-p partition] [-d repository]
```

If the machine argument is omitted, the static information of the test is printed. The retrieved `test_info` structure is printed in a readable form. If the machine argument is set, the variable information (`vars_info` structure) is obtained and printed in a readable form.

5.3 tmgr_exec

Execute a test synchronously. Synopsis:

```
tmgr_exec -T test [-m machine] [-A arguments] [-p partition] [-d repository]
```

If the machine argument is omitted, the default host is used. The user supplied arguments are placed behind the default parameters. This program blocks until the test finishes. On success it prints the result of the test.

5.4 tmgr_try

The *tmgr_try* program is an interactive program, which shows in a simple way the capabilities of the TM. It is by far the most comprehensive utility. Synopsis:

```
tmgr_try [-p partition] [-d repository]
```

After start-up the *tmgr_try* program turns into an interactive mode whereby the user can issue commands. Answers shown in parentheses are default answers and a simple <CR> is sufficient to select them. The program recognizes parts of input which makes the answer unique. The global command `quit` (or part of it) will stop the program. The program will prompt the following basic command line:

```
-<TM>: list, info, start, stop, legenda (list) ?
```

The default answer to this question is the *list* command. If selected it asks further whether you want a list of tests (executing the `ListAllTests` method) or a list of machines (executing the `ListAllMachines` method). The *tmgr_try* program gets the selected list and prints the available tests or machines on standard output. The *info* command asks whether you like the static information of a test or combined with a selected computer the variable information. The `GetTestInfo` or `GetVarsInfo` method is executed respectively. The program produces in a readable format the static or variable information of the test. The *start* command will ask you which test you want to execute, on which host and with which arguments. It also asks whether to start it asynchronously or synchronously. In case of starting a test asynchronously, it also asks how many times you want to start the test. The returned handle is shown after calling the `StartTestA` method successfully. An internal call-back routine of the *tmgr_try* program is invoked each time such a asynchronously started test finishes and the result of the test is printed. When a test is started in a synchronous way, using the `StartTestS` method, the flow of the program is blocked until the test

finishes. On completion the result of the test is printed. The *stop* command executes the `StopTest` method. It will ask for a valid handle. Finally the *legenda* command executes the `GetHistory` method. Several layouts are possible, such as: show me a list of successfully launched tests, a list of active tests, a list of finished tests or a detailed description of an individual test.

6 Making a test program

There a two distinct phases to create a test: the first one is to write and compile the program and the second to store the test in the repository. There are a couple of conditions required for a proper test program. The most important one is that it has to return a valid test result. This result is passed by means of the exit status of the program, what implies that a test program should always finish with a proper exit status. The result of the test has to comply with the POSIX 1003.3 definition **[8]** and should be of type `TestResult`, which is defined in the general include file of the TM `<tmgr/tmgr.h>`.

```
typedef enum tmResult
{
     TmPass          = TM_PASS,
     TmUndef         = TM_UNDEF,
     TmFail          = TM_FAIL,
     TmUnResolved    = TM_UNRESOLVED,
     TmUntested      = TM_UNTESTED,
     TmUnsupported   = TM_UNSUPPORTED
} TestResult;
```

6.1 Store a test in a repository

The primary link between a test program and repository is the `ExecutableFile` attribute of the Program class of figure 1. It should contain the name of the binary. To store a test or, even more important, to create a repository, requires some knowledge about the *oks_data_editor* **[9]**. The main scheme of figure 1 is found in the following schema file:

```
/afs/cern.ch/atlas/project/tdaq/databases/v1/schemes/DAQ-Confdb.schema
```

This schema should be loaded when using the *oks_data_editor*. This editor allows you to create instances of classes and their relationships. According to the DAL's User Guide **[3]**, the repository based on the model of figure 1 is split into three files:

1. **Hardware Description Data File** contains objects of configuration database class "Computer".
2. **Software Description Data File** contains objects of configuration database class "Program", "SW_Object" and "Environment".
3. **Configuration Description Data File** contains objects of configuration database class "Application", "Configuration", "DataFile", "SchemaFile" and their subclasses.

The "Configuration Description Data File" is used as the repository file, supplied as argument to the different utilities (`[-d repository]`). The other two files are loaded by means of the "Federated Database" concept. Two "DataFile" objects represent the other two files and are linked via the "Data" relationship to the "Partition" object (which is a kind of "Configuration").

7 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]R Hart, **Implementation of the Test Manager for the ATLAS DAQ Prototype-1**, Technical Note 111, V2.0, Oct. 11 1999 (URL:http//atddoc.cern.ch/Atlas/Notes/111/Note111-1.html).

[3]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).

[4]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).

[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]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).

[7]**ATLAS DAQ Software Development Environment** (URL: http://atddoc.cern.ch/Atlas/DaqSoft/sde/Welcome.html).

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

[9]I.Soloviev, **OKS - Object Kernel System**, Technical Note 033, V1.1, 13 Feb. 1998, (URL: http://atddoc.cern.ch/Atlas/Notes/033/Note033-1.html).