Testerman Documentation

Release 1.4.0

Sebastien Lefevre

August 27, 2017

CHAPTER

ONE

INTRODUCTION

Welcome to Testerman

Testerman is an attempt to produce a TTCN-3 inspired test automation system without the strict typing model of the TTCN-3 test control notation that can be too heavy to be practical with non-stricly defined protocols. This is achieved by bringing TTCN-3 primitives and concepts to the Python programming language.

It provides a complete environment to design, manage, execute and analyze automated tests, and can be used as a platform to develop test simulators (drivers and stubs) and prototypes of network applications.

Unlike existing tools specialized in a single or a single kind of protocol support (such as "web services", "GUI testing", "GSM", etc), Testerman focuses on providing all the necessary base to perform end-to-end, system testing of network-based applications mixing multiple protocols. In particular, it was designed with telecom platforms in mind, such as VoIP/IMS based ones.

Testerman also turns out to be a way to develop equipment simulators that could be used to automate interop testing. For instance, we may imagine some endpoint manufacturers developing simulators in Testerman, and making them available to test them with another vendor's call control platform.

Testerman is Open Source/Free Software, and is freely available under the GNU General Public License, version 2[1].

About

Testerman was initially developed by Sebastien Lefevre, a software engineer employed at the time by a telecommunications equipment provider.

Fed up with the over-priced, proprietary or never-as-integrated-as-needed end-to-end testing tools, he decided to develop his own and give it back to the community he learned so much from.

Let's hope you'll enjoy using Testerman as much as he did while developing it !

Contact

Main developer/maintainer: seb.lefevre (at) gmail.com.

Getting Testerman

Testerman is made of several components that are designed to work together:

- server: the Testerman core server
- tacs: the Testerman Agent Controller Server,

[1] http://www.gnu.org/licenses/gpl-2.0.html

- cliclient: a command-line interface Testerman client,
- qtesterman: a multi-platform rich client, used as an IDE to develop, run and monitor tests
- pyagent: a python-based remote agent used to run probes on the SUT machines
- webclientserver: a web front-end that can be used to monitor and trigger test suites and can be deployed in a DMZ.

While each component evolves at its own rate, baseline kits, containing a reference version for all of them, are released from time to time.

Releases

Testerman releases can be downloaded from http://testerman.fr.

Development versions

If you want the latest probes or component enhancement, you may download the current Testerman version using anonymous git access on github:

```
git clone https://github.com/seblefevre/testerman.git
```

The anonymous git access is read-only. If you want to contribute and get a read-write access, feel free to contact me.

CHAPTER

TWO

USING TESTERMAN

Testerman Overview

Testerman is a project that brings most of the powerful capabilities and concepts of TTCN-3 to the Python programming language, both providing a set of Python libraries and a complete, multi-user, distributed test execution environment.

TTCN-3 is a test description language standardized by the ETSI as ES 201 873[2]. Testerman implements most language primitives described in its part 1 (ES201-873-1: TTCN-3 Core Part) as Python functions, however it drops all the strict typing model of TTCN-3 and privileges the weak typing of Python, enabling to write test cases faster than in TTCN-3.

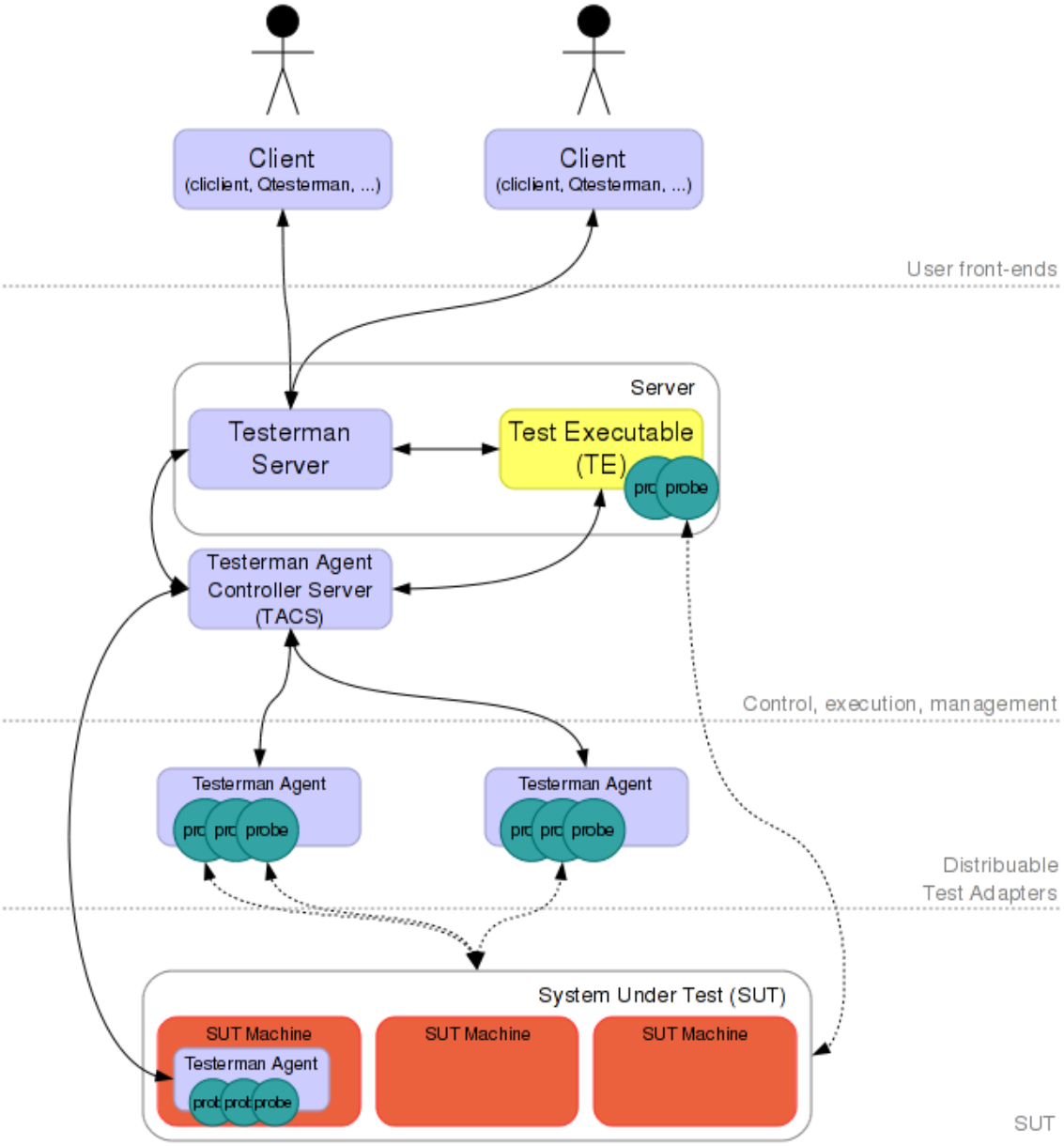
Testerman is not meant to replace or to compete with such a growing supported standard. Instead, it tries to fill the gap between the existing, closed-source, expensive TTCN-3 tools on the market and the other testing tools that are dedicated to one family of protocols, forcing the users to develop glues to synchronize them together, and create additional scripts to test non-protocol things such as log files, database updates, etc.

By using the same concepts as in TTCN-3, your time investment in Testerman can be fully reusable when migrating to the standard, if needed: the test logic implementation, objects and primitives remain the same, only the syntax changes.

Testerman Infrastructure

Testerman has been designed to be multi-user (client-server) with a centralized test cases repository, adaptable to most SUT topologies and network constraints (distributed test/system adapters), and extensible (test/system adapters can be developed easily, in any language).

[2] http://www.ttcn-3.org/StandardSuite.htm



Clients

Users interact with the system through a Testerman client. Any number of clients can connect to the same server, enabling to share the same ATS repository and the same technical resources (the agents, see below). Currently, the following clients are available:

- a command-line interface client, that can be used to call test suite executions from a shell script, a Makefile, or a continuous integration system
- a multi-platform, rich client named QTesterman providing an IDE to design, execute, control and analyze test suites
- a lightweigth web-based client, limited to running tests and get their results and logs. This client can be published as a front-end to your customers in a DMZ.

CLI and rich client should be installed on the user's workstation. However, developing a full fledged web-based client (with test designs and writing capabilities) is quite possible.

Server Components

The server is actually split into two components:

- The Testerman Server, which is a front-end to the clients, responsible for creating, running and controlling Test Executables (TE) from a source ATS written by the user. It is also able to push updates to the clients which implemented auto-updates, such as QTesterman.
- The Testerman Agent Controller Server (TACS), which is a backend process responsible for managing remote Agents and Probes so that TEs can use them when needed at runtime (see below). The TACS may be installed on another machine than the Testerman Server (and shared between multiple servers), but is typically colocated with it.

Test Executables

They are the result of a "compilation" of the ATS into something runnable by the operating system. In a real TTCN-3 execution system, the ATS, written in TTCN-3, would probably be transformed into Java or C++ code before being compiled with a standard compiler, and linked to specific libraries according to the tool vendor. These libraries then enable the TE to access a Platform Adapter (PA) and System Adapters (SA) modules that implement actual, technical services such as timers, message encoders/decoders, and connections to the SUT (actual socket management, writing and reading on the wire, etc).

In Testerman, the ATS, writing by the user in Python and calling Testerman-provided functions, is also transformed into a special Python program that embeds additional Testerman modules (the Python word for "libraries") to provide an access to the Testerman environment functions. In particular, these libraries enable the TE to access the probes either hosted on a distant agent ("remote probes") or run from the same location as the TE ("local probes").

Agents & Probes

A probe is the Testerman word for the implementation of what TTCN-3 would call a Test Adapter (aka a System Adapter). This is the piece of software that can actually connects to a SUT (System Under Test) using different protocols (such as tcp, udp, sctp, or higher level ones: http, rtsp, Oracle, SOAP, XMLRPC, ...).

Testerman provides a collection of probes to be able to interact with the SUT using most usual protocols but also to interact with it simulating user actions or anything needed to automate a test you can do manually. For instance, the probe whose type is `watcher.file` is able to monitor any (text) file on a system, and sends notifications when a new line matching some patterns was written to it. Some others enable you to simulate command line commands, as if you were opening a local shell on the machine and running some commands locally.

Probes can be run by the TE itself, running on the Testerman Server and thus initiating or expecting network connections on the Testerman Server's IP interfaces (which may not be possible in all SUT topologies), or can be run on any distant system. In this case, they need to be hosted on a Testerman Agent, responsible for managing the low-level communications with the Testerman Agent Controller Server (TACS).

Agents must be deployed (i.e. installed and started) manually by the tester. However, once deployed, an agent may be used by any TE to deploy (dynamically) any number of probes on it, according to its defined Test Adapter Configuration.

This Agent-oriented architecture enables to distribute probes anywhere in the network, either around the SUT or inside the SUT itself. This provides several benefits:

- You can stimulate the SUT from any IP address, not limited to the server's ones. Useful to simulate SIP end-points between multiple NAT gateways, test an Intrusion Detection System or any security rules, stimulate some interfaces that are only reachable from selected SUT subnets, etc.
- You may install an agent on the SUT machines themselves, even if they are behind a router disabling direct access to them (the agents tcp-connect to the TACS, so that you can NAT them), and then be able to execute commands on the SUT itself or observe events occurring inside the SUT only, such as log updates, file creations, child process creations or kills, etc. You are no longer limited to what the SUT exposes outside of it to interact with it.

- As a side effect, you may implement probes in any language, providing an agent exists for it - you are not stuck to Python and free to choose the best implementation language for your probe according to your likes, skills, or available libraries in the language (for instance, a SOAP probe using the .NET stack, a C++ based ISUP probe linked with HP OpenCall TDM modules, ...). If not using Python, however, your probes won't be able to run locally with the TE and will require an agent to host them.

For now, the main (and single) Testerman agent implementation available is written in Python (that's why it is dubbed PyAgent) and can host any probes that can also be run locally by the TE, sharing the same plugin interface. This PyAgent can also be updated remotely from the TACS+Testerman Server, making Testerman administrator's and user's lives easier when new updates are available.

Main Concepts

...

Quick Start

Before installing Testerman, make sure you had a look to TestermanOverview, in particular to understand its client/server architecture.

The following procedure will help you install and start the server components, set up a repository with some sample scripts, and deploy the QTesterman client.

Server Installation

Requirements

The Testerman server components currently run on Linux (may also run on Solaris 10) with Python 2.6 or better (however, Python 3 is not yet supported).

Installation (current development snapshot)

This procedure assumes that you have installed the subversion client on your machine, named in this sample `testermanserver`.

0. You may create a dedicated user to run the Testerman server components, or run it as a normal user. It is usually a bad idea to run it as root.

1. Get the current development snapshot using subversion (here "installed" to `testerman-svn`):

```
git clone http://testerman.fr/git/testerman.git testerman-git
cd testerman-git
```

2. Set up a document root (in this example, we are using the default, as configured in `conf/testerman.conf`: `~/testerman`) and a repository that will contain your shared test cases; we also fill it with some samples (WARNING: existing samples will be replaced with the one from the SVN tree):

```
bin/testerman-admin setup document-root
```

3. Publish the current (SVN version) QTesterman client and PyAgent on this server, advertising them as stable versions:

```
bin/testerman-admin -c testerman/components source-publish component qtesterman branch stable
bin/testerman-admin -c testerman/components source-publish component pyagent branch stable
```

4. You may check the published components with:

```
user@testermanserver$ bin/testerman-admin -c testerman/components show
Component   | Version | Branch | Archive File                    | Status
-------------------------------------------------------------------------
pyagent     | 1.1.3   | stable | /updates/pyagent-1.1.3.tgz      | OK
qtesterman | 1.2.0   | stable | /updates/qtesterman-1.2.0.tgz | OK
```

Starting the Server Components

Assuming you are using `~/testerman` as the document root, go to the directory where you installed or checked out Testerman (in the procedure above, into `~/testerman-svn`), and start the Testerman components, i.e. the Testerman Agent Controller Server (TACS) and the Testerman Server, with:

```
user@testermanserver$ bin/testerman-admin start all
```

You may check that both processes are running with:

```
user@testermanserver$ bin/testerman-admin status
 Component | Management Address     | Status  | PID
-----------------------------------------------------
 server    | http://localhost:8080 | running | 6625
 tacs      | 127.0.0.1:8087        | running | 6670
```

By default, the Testerman Server listens on `tcp/8080` (Ws interface, XML-RPC based Web Services), `tcp/8082` (Xc interface, a text-based event subscription service), `tcp/127.0.0.1:8081` (Ei interface). You may set Ws and Xc IP addresses explicitly in the `conf/testerman.conf` file if needed.

The TACS listens on `tcp/40000` (Xa interface, used by the agents to connect to their controller) and `tcp/127.0.0.1:8087` (Ia interface). Those are configurable in the same configuration file as well.

Installing the QTesterman Client

You can now proceed with the QTesterman client installation, pointing to `http://testermanserver:8080` were `testermanserver` is the name or the IP of the machine where you just installed and started the Testerman Server component.

Deploying and Starting a PyAgent

You may need to deploy some Testerman agents to host probes remotely.

You may directly start a Python-based agent, dubbed PyAgent, from the installation root:

```
bin/testerman-agent --name localagent --log-filename myagent.log -d
```

where `localagent` is a friendly identifier that will be used as the domain part in probe URIs when defining test adapter configurations. The `-d` flag indicates that the agent should deamonize. By default, it will try to connect to a controller on `localhost:40000`.

But starting an agent locally is not very useful, as it main interest is its capability to be distributed over the network. Several options exit to install such an agent on another machine, on which it has the following requirements:

- Any system with Python 2.4 or better (won't work/untested with python 3.x)
- according to the probes you plan to use on this agent, additional requirements may appear. Please see CodecsAndProbes.

The Easiest Way

Assuming `wget` is installed on your target machine, you can retrieve a pre-configured pyagent installer from the Testerman server with:

```
wget http://<server>:8080/pyagentinstaller
```

where <server> is your Testerman server hostname or IP address.

This fetches a Python script that is pre-configured to download and install the latest stable pyagent package from the server you provided in this url. Just execute it with:

```
python ./pyagentinstaller
```

(for other options, in particular to install a specific pyagent version or a testing version, see the inline help with `python ./pyagentinstaller --help`)

Once done, your pyagent is ready to be executed from your current directory with:

```
./testerman-agent.py -c testermanserver --name remoteagent [-d] [--log-filename remoteagent.log]
```

where `testermanserver` is the Testerman server hostname or IP address, and `remoteagent` the name that will identify this agent on the Testerman system. Use the `-d` flag to daemonize the agent, if needed, and `--log-filename` to add an optional log file if you want one.

Manual PyAgent Installation

If you don't want to use the `pyagentinstaller` script, you may copy the pyagent component package that was deployed into ~/testerman/updates/pyagent-X.X.X.tgz to a target machine that will run the agent (you can also fetch this file from `http://<testermanserver>:8080/components.vm`).

Once `pyagent-X.X.X.tgz` has been copied, just untar it (it will create a directory named `pyagent`) and execute the agent:

```
cd pyagent
python ./testerman-agent.py -c testermanserver --name remoteagent --log-filename remoteagent.log
```

where `testermanserver` is the hostname or IP address of the server the TACS is running on and `remoteagent` the name you want to assign to this agent instance (using the machine hostname can be a good start). The agent will connect to the TACS on startup (and keeps reconnecting in case of a connection failure) and will show up in QTesterman's probe manager when available.

Alternatively, you may check the correct agent deployment from the server's installation root with:

```
user@testermanserver$ bin/testerman-admin -c testerman/probes show all
URI                    | Type    | Location      | Version
-----------------------------------------------------------------------
agent:remoteagent      | pyagent | 192.168.13.17 | PyTestermanAgent/1.1.3
agent:localagent       | pyagent | 127.0.0.1     | PyTestermanAgent/1.1.3
```

And voila ! You are now ready to play with some samples from the QTesterman interface.

External Resources

TTCN-3

If you are new to TTCN-3, you may find the following links useful:

- The official TTCN-3 site[3], in particular the core language reference to download (and use it in conjunction with TestermanTTCN3 for feature comparisons) and the tutorials page[4] (don't miss C Willcock's introduction to TTCN-3[5])
- TTCN-3 Basics[6], a very clear introduction (courses ?) to TTCN-3 by Vesa-Matti Puro, from OpenTTCN

[3] http://www.ttcn-3.org
[4] http://www.ttcn-3.org/CoursesAndTutorials.htm
[5] http://www.ttcn-3.org/TTCN3UC2005/program/TTCN-3%20Introduction%20version%20T3UC05.pdf
[6] http://www.ttcn3basics.com/Day1/siframes.html

- Research in TTCN-3[7] is a collection of very useful links, including multiple tutorials and real world use cases

Python

New to Python ? try these:

- The official Python site[8] - at least one thing to read: the Python tutorial[9]. However, since it is based on the most recent Python version at date, some features may be not available on most Testerman deployments, running Python 2.4 or 2.5 as provided with your distribution
- Dive into Python[10] is a book by Mark Pilgrim that is freely available online. A good reading, too.

QTesterman

QTesterman is the preferred client to execute Testerman scripts, monitor and analyze their executions.

It is a cross-platform application (Windows, Unixes, Mac OS X) developed in Python with PyQt4. This is the main interface to a Testerman system most end-users will use.

Installation

QTesterman basically requires Python 2.x >= 2.4.4, and a PyQt4 4.x >= 4.4.2. You may install them the way you are used to, from sources, or following the instructions below. Please notice that actual package versions may evolve and the ones given here may be deprecated or no longer available as a direct link - in this case, just use the next or recommended version on the associated sites.

Installing Prerequisites

This section describes how you may install QTesterman dependencies on several operating systems.

If you already installed them, you may jump to the [#RunningtheQTestermanInstaller QTesterman installer execution].

Windows

To run QTesterman under Windows, you'll first need to install Python 2.x and the PyQt4 library. QTesterman is not compatible with Python 3 for now.

If you don't have any Python installation yet, you are advised to install Python 2.6.1(or latest 2.6.x version) that you can grab from the official Python site[11] (direct Windows Installer link for Python 2.6.5[12]). Once downloaded, simply run it and install the usual way.

Then, get PyQt4 from Riverbank Computing site[13]. You should get the Windows installer package that matches your Python version, for instance PyQt-Py2.6-gpl-4.7.7-1.exe[14] if you installed a Python 2.6.x. The binary PyQt4 package contains everything needed for QTesterman, in particular the QScintilla2 module. When installing the package, you can limit your installation options to the single "Qt libraries" option, discarding all development tools, documentation, and examples.

[7] http://www.site.uottawa.ca/~bernard/ttcn.html
[8] http://www.python.org
[9] http://docs.python.org/tutorial/
[10] http://www.diveintopython.net
[11] http://www.python.org/download/
[12] http://www.python.org/ftp/python/2.6.5/python-2.6.5.msi
[13] http://www.riverbankcomputing.co.uk/software/pyqt/download
[14] http://www.riverbankcomputing.co.uk/static/Downloads/PyQt4/PyQt-Py2.6-gpl-4.7.7-1.exe

Once these dependencies have been installed, you can proceed with the [#RunningtheQTestermanInstaller QTesterman installer execution].

QTesterman has been tested on the following combinations:

- Windows XP SP2 with Python 2.5.4 + PyQt 4.4.3
- Windows Vista SP1 with Python 2.5.4 + PyQt 4.4.3
- Windows 7 with Python 2.6.1 + PyQt 4.4.4

Debian-based Distributions

Under Debian and derivatives, you should apt-install the following packages (and their dependencies):

- `python-qt4`
- `python-qscintilla2`

QTesterman has been tested on the following distributions, with their standard up-to-date packages:

- Ubuntu 8.10..12.10
- Debian lenny, wheezy

Red-Hat-based Distributions

Under Red-Hat and derivatives (CentOS, Fedora, RHES, Mandriva, ...), you should install the following RPM (and their dependencies):

- `PyQt4`
- ... ?

RHES4/CentOS4 only provides Python 2.3, so it won't run on them.

Mac OS X

If you successfully installed QTesterman under an OS X version, please send me some feedback to complete this section !

Running the QTesterman Installer

Once the prerequisites are installed, you can download and execute the QTesterman Installer.

- Get it here, or get the [source:/trunk/qtesterman/Installer.py latest version from SVN]
- Run it:
  - Under Windows, double-clicking on it should be OK, unless you changed the standard Python associations. In this case, you know how to execute it with your favorite interpreter
  - Under Linux, run `python Installer.py` (`./Installer.py` may also work)
- Alternatively you may have run it directly from your browser, since you can delete it once executed

If your dependencies are correctly installed, you should get a minimalistic installer window enabling you to select your installation directory (in which a folder `qtesterman` will be created) and provide a Testerman server URL (`http://your-server:8080`). This server URL should point to a running Testerman server configured to deploy a QTesterman component. Your Testerman administrator is supposed to configure it for you correctly, or you may have a look to TestermanAdministrationGuide to do it yourself.

When ready, click install; the installer should detect a new QTesterman version available on the server, and prompt you for an upgrade. Accept, wait for the installation to complete.

Once over, QTesterman is installed in your `installation directory/qtesterman`. You may create a shortcut to the `qtesterman.py` file contained in it - this is the Python file to run to start the QTesterman client.

You may delete the downloaded `Installer.py` file.

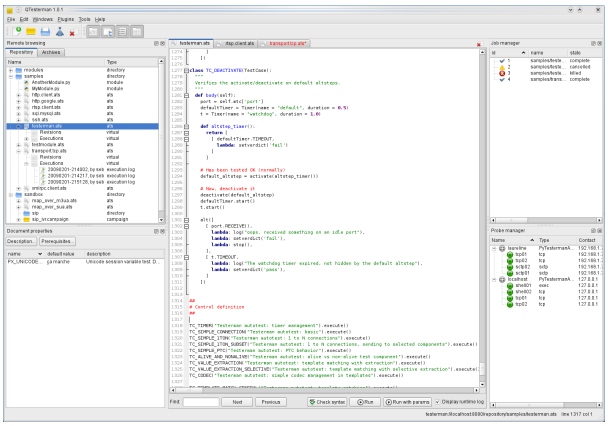
First Run

The first time you execute the `qtesterman.py` script, you'll need to provide the URL of your Testerman Server and a username that will identify you on this system.

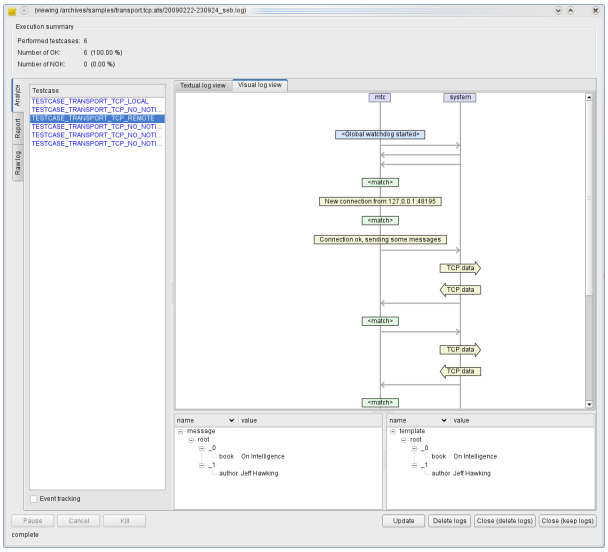
For now, this username is not associated to any password since no rights management is implemented yet. It just enables to have an idea of who is currently running jobs, who should be contacted by the administrator in case of a problem, etc.

Main Interface Quick Tour

Main interface:



Log viewer, here in visual view mode:



...

Your First ATS Execution

...

Building Reports

QTesterman comes with two default plugins enabling to create template-based reports at two levels:

- ATS documentation: the Test Specification Extractor plugin allows to create reports based on the ATS code and embedded docstrings. This could be used to create an external test case documentation, such as a HTML page, but also any kind of text-based documents.
- Execution results reports: from the Log Analyzer, you can access the Simple Reporter plugin that use templates to produce any kind of text-based reports, including HTML.

Both plugins can be used as a basis to built any kind of text-based reports, ranging from plain text to HTML pages, from CSV files to XML-based document (think ODF). They are based on a template engine that implements a subset of the Velocity engine[15] features.

From such templates, you can access to different variables and functions depending on the calling context (ATS documentation report or test execution report); in particular, you can easily include [TestermanReferenceGuide#DocumentationSystem documentation string tags].

Template Engine Syntax

A template is a text file that may contain particular tokens interpreted by the engine. Actually, all of the following is Velocity-compliant, and you can also refer to the Velocity engine documentation directly. Yet, some Velocity features are not implemented, in particular:

- Arithmetics/maths
- Range operator
- Velocity variables (velocityHasNext, velocityCount, ...)
- `#evaluate` directive

Additionally, there are no built-in/default velocimacros, though you can define your owns.

Comments

The following template

```
You won't see the following lines.
## This is a single-line comment

#*
While this
is a multi-line
comment
#*

Use \# to escape the pound sign.
```

will result in:

```
You won't see the following lines.


Use # to escape the pound sign.
```

Variable Assignments

you can create variables in a template dynamically:

[15] http://velocity.apache.org/

```
#set($foo = "QTesterman")
Hello $foo World!
```

Leads to:

```
Hello QTesterman World!
```

Variable References

You may reference a context variable at any time with a `$myvar` or `${myvar}` syntax.

The available context variables for each plugin are defined below.

```
My variable: $my_variable
Another variable available in my context: ${another_variable}
Say_${within_a_word}_here
```

If a reference is not found when applying the template, the template code is unchanged. For instance, with the template above, if `my_variable` evaluates to `2`, `within_a_word` to `"hello"`, but `another_variable` is not found, this will produce:

```
My variable: 2
Another variable available in my context: ${another_variable}
Say_hello_here
```

To substitude the placeholder with a blank instead of leaving the original template code, use a `$my_var` syntax:

```
Missing variable: $!{missing}
```

Would generate:

```
Missing variable:
```

If a variable cannot be evaluated to a string representation, an exception string is injected instead.

Some object referenced by such variables may expose additional properties or methods. In this case, you can access them with a dot-based notation:

```
This is an object attribute: ${testcase.id}
## Also works without {}:
$testcase.title
## And also with methods
## Assuming $a is a string:
$a.replace('\\n', '<br />')
```

Actually, you can also reference a call to an exposed function directly. Let's assume that a function `toHtml(s)` that escapes the usual HTML characters is provided in the current context:

```
<p>Description: $toHtml($description)
</p>
```

Applied with `$description` that evaluates to `"check that 100 > 10,\nthen make sure that the function returns within 10s"`:

```
<p>Description: check that 100 &gt; 10,<br />then make sure that the function returns within 10s
</p>
```

Iteration within a Sequence

Some variables are evaluated to a list of items. In this case, you may use the `#foreach` directive to iterate through them:

```
#foreach ($testcase in $testcases)
Testcase Identifier: $testcase.id
#end
```

Conditions

A if/elseif/else mechanism is available:

```
#if ($name)
Test case name: ${name}
#else
Undefined test case name
#end
```

The usual operators are supported: <, >, <=, >=, ==, =, && (and), || (or), ! (not).

```
this is#if ($a > 10) large#elseif ($a > 5) medium#else small#end, don't you think ?
```

Notice how you can mix directives and the normal text to procude the desired output.

Test Specification Extractor Plugin

This plugins exposes the following variables and functions:

Name	Type
`testcases`	list of `testcase` objects

...

Simple Reporter Plugin

This plugins exposes the following variables and functions:

Name	Type
`testcases`	list of `testcase` objects

...

Random Tips

- You may use `Ctrl+Mouse Wheel` to zoom in/zoom out in the Visual Log Viewer
- This zoom is also available in the main editor, using the same shortcuts
- A simple key recording feature is available in the editor: use `Alt+K` to start recording keys, `Alt+K` again to stop recording, `Ctrl+K` to replay (shamelessly inspired by nedit[16])
- `Ctrl+Shift+A` on a plugin in the Settings window will display some additional information about it

Testerman Reference Guide

This chapter describes features that are not related to the Testerman core librairies, implementing the TTCN-3 concepts, but instead the Testerman infrastructure and environment features.

[16] http://nedit.org

Main Concepts

Note: all concepts that from TTCN-3 are indicated with a * in this chapter.

- ATS (Abstract Test Suite)*: The script containing the automated testcases. It is normally written in TTCN-3. In Testerman, it is written in Python, using the Testerman-provided modules (libs) to provide TTCN-3 features and an access to a complete Test System* implementation.
- Test System*. An implementation framework to compile TTCN-3 scripts to TEs*, execute them, control them, analyse their logs, etc. A typical TSE system is described below. Testerman is one example of a Test system, though not TTCN-3 compliant.
- TE (Test Executable)*: the executable generated from an ATS that can be actually executed on a real machine to perform the tests. In Testerman, the TE is built as a python script from the user-written ATS*, slightly modified, and linked to Testerman-provided modules to connect to the execution environment, providing system, platform, and test adapters implementations.
- Campaign: a hierarchical collection of ATSes (or other campaigns) to execute conditionally in a row. Gathers your ATSes from multiple sources in a single campaign, and run all your tests in one click.
- ...
- SUT (System Under Test): the system to test. Identifying the SUT is essential to correct automated tests implementation in Testerman (and TTCN-3), especially because you'll have to identify the different stimulation and observation interfaces between the Test System and the SUT. These interfaces are then viewed in TTCN-3 as Test System Interface Ports*, or TSI Ports for short.
- userland: the "TTCN-3" or "Testerman" world, where the testcase is executed and designed by a writer, i.e. the abstract part where testcases, test components, messages, templates live without being bothered by low-level consideration such as encoding/decoding, physical transports, physical connections, etc. The userland is test logic-oriented, not test implementation-oriented.

Logical Architectures

TTCN-3 Test System

...

- *TCI* (Test Control Interface) and *TRI* (Test Runtime Interface) are two interfaces standardized in TTCN-3, with specific operations and call flows.

Testerman does not implement them directly nor completely, but tries to follow them as often as possible to keep the clean, flexible model of a TTCN-3 implementation.

Testerman System

...

Physical Architecture

Testerman implements these logical view as a distributed system, at different levels:

- the SA is made of distributable entities: agents. Agents are containers that can be deployed on any physical machine providing it has an access to an agent controller (and, not to be useless, to the SUT). This controller, dubbed the TACS (Testerman Agent Controller Server), is typically co-hosted with the Testerman Server (the main test system executor front-end), and exposes the connected agents to TACS clients:

[little diagram]

All TEs are clients to the TACS, enabling them an access to all the probes hosted on the agents. The Testerman Server itself is also a TACS client, for administration and control purposes.

On the front-end side, Testerman uses a client/server model, enabling multiple users to control and executes ATSes at the same time, sharing the same Testerman installation and repository.

[another little diagram]

Interfaces

Testerman names the following interfaces:

- **Ws** (Web Service): client/server control interface. Job control, repository access (read/write), agent/probe management, and some server administration. Implemented in XML-RPC, and documented here - useful if you plan to integrate Testerman in a SOA.
- **Xc** (eXternal interface, Client): subscription-based notification interface. Clients can subscribe to get job, probe, agent control-related events or real-time execution logs. Implemented using TestermanProtocol.
- **Xa** (eXternal interface, Agent): the interface between the TACS and the agents (TACS southbound interface). Implemented using TestermanProtocol.
- **Ia** (Internal interface, Agent): the protocol used internally by the TEs to access the remote probes through the TACS (TACS northbound interface). Also used for basic TACS management. Implemented using TestermanProtocol.
- **Il** (Internal interface, Logging): the interface between the TEs and the Test Logging (TL) module embedded within the Testerman Server. Whenever the TE needs to log something, it sends it to the TL through this interface. The TL is responsible for writing it to the proper log file. Will enable distributed PTCs over multiple TEs on different machine (but not yet). You can identify this interface as being a part of the TTCN-3 TCI. Implemented using TestermanProtocol.

Core Concepts

Most concepts at use for an ATS writers are described in TTCN-3: Core Language (ES 201 873-1, version 3.4.1)[17]. However, Testerman adds some new ones to fill the gap between the standard, dealing with abstractions only, and a real implementation.

Test Adapter Configuration

TTCN-3 does not provide a standard way to assign a test system interface port with an actual test adapter implementation, implementing on-the-wire sending and receiving operations.

Testerman introduces a way to bind these ports to test adapters, i.e. to probes, and calls this relationship a "binding". When defining a (named) Test Adapter Configuration (for now, programmatically in your ATS only), you defines the different bindings (i.e. what probe instance will be used to implement a test system interface port), including the binding configuration, i.e. the probe parameters, if any.

You may define multiple test adapter configurations and switch from one to another one depending on the test environment you're about to run your tests against.

The test adapter configuration is supposed to be the only thing that change from one test environment to another one. It usually defines:

- real SUT IP addresses, ports
- some low level variables such as ssh login/password (likely to change from one system to another one)
- etc

Application-oriented variables may still vary from one test environment to another one, but if your testcase is carefully designed, including with correct [#PreambleandPostamble Preambles], you should be able to minimize the amount of efforts needed to run your test on another instance of your SUT.

[17] http://www.ttcn-3.org/StandardSuite.htm

Codec Aliasing

The TTCN-3 code language does not completely ignore this, though.

Basic, General Purpose SUT Adapters

Once introduced to TTCN-3, you may wonder "OK, that's nice and powerful, but now, how do I connect to a tcp server, send and receive some data ?". Protocols in use are not a (direct) concern for the standard, you are right. We're dealing with application- and test-oriented message structures, but physical transport (such as tcp/udp/sctp or even ip) is not adressed. TTCN-3 has the concept of SUT addresses, but how do we control (perform and verify) actual connections, disconnections, stream-oriented data reception, and so on ?

Antti Hyrkkanen, from the Tampere University of Technology, defended his master thesis[18] about a general purpose SUT Adapter for TTCN-3, bringing socket-like structures and associated functions to TTCN-3. While this approach cannot be more flexible and complete, it renders ATSes harder to write for non-programmers, forced to take into account low level details in most cases (of course, if your tests are about testing the SUT's ability to handle tcp connections, disconnect them when expected, etc, this is fully adapted and even required).

[WARNING: feel free to correct me if I misunderstood Antti's work]

Testerman tries to find an intermediate solution to this problem by providing a collection of transport-related probes, interfaced in userland using the same kind of templates - quite similar to Antti's solution, but just less generic as the very low level (socket parameters) are embedded within the probe, and partially controllable through test adapter configurations.

(TODO: transport interface: to document)

Preamble and Postamble

Testcases may require some SUT preparation in order to be executed, typically data provisioning, configuration files settings, maybe some processes or applications restarts.

Once the test is over (independently from its verdict), the SUT needs to be restored in an "original" state so that, in particular, we can replay the testcase without any additional manipulations.

These SUT preparation and clean up phases are called "Preamble" and "Postamble" (P&P), respectively, in Testerman terminology.

Testerman provides a way to use its core features to implement an automated preamble (you may call it "automated test bed setup", "automated prerequisites set up", ...) at least for what Testerman can automate using its available probes and the available SUT provisioning interfaces - manual prerequisites may still be needed.

You may design campaign-level P&P, suitable for multiple ATSes (i.e. starting the Preamble at the beginning of a campaign, starting the Postamble when finished), or ATS-level P&P, where a Preamble/postable may be used for multiple testcases in a row, or testcase-level P&P, i.e. only valid for a particular testcase (in this case, they are typically embedded within the testcase definition itself).

Testerman Applications

The Testerman application framework is currently not available.

The idea is to provide a way to run "in the background" applications built using Testerman features to act as simulators either to help manual testing or to simulate/prototype new applications.

Basically, you can already develop such simulators in a testcase, but a testcase is not designed to run forever and not to return a verdict. A Testerman application will.

[18] http://www.ttcn-3.org/doc/GeneralPurposeTTCN3SA.pdf

Campaigns

A campaign is a structured collection of ATSes that can be executed conditionally.

It is basically a black and white tree (each node has two branches: one to follow if the current node is successful, the other one in case of an error) enabling to chain ATSes (or other campaigns), executing specific ATSes or campaign depending on the execution status of the current job.

Campaign Definition

A campaign is defined in clear text, declaring a job tree based on indentation:

```
job
 job
 job
  job
job
```

The indentation is defined by the number of indent characters. Valid indent characters are `\t` and `' '`.

A job line is formatted as:

```
[<branch> ]<type> <path> [groups <groups>] [with <mapping>]
```

where:

- `<branch>` indicates the execution branch the job belongs too. Must be a keyword in 'on_success', 'on_error', '*', or left empty. If not provided or set to 'on_success', it indicates that the job is in the *success* branch, and that it should be executed only if its parent job returns a 0-result. If set to 'on_error' or '*', this is the 'error' branch, and the job is executed only if its parent job returns a non-0 result.
- `<type>` is a keyword in 'ats', 'campaign', indicating the type of the job
- `<path>` is a relative (not starting with a /) or and absolute path (/-starting) within the repository refering to the ATS or the campaign to execute.
- `<groups>` is an optional string formatted as `GX_GROUP_NAME[,GX_ANOTHER_GROUP]*` enabling to select the groups to run in the ATS. This option is only valid for an ATS job. By default, all groups are selected.
- `<mapping>` is an optional string formatted as `key=value[,key=value]*` enabling to map or set job's parameters from the current context's parameters. See *below* for more details.

Comments are indicated with a #.

Example:

```
# Sample campaign
ats class5/call_forward_unconditional.ats # job1
 ats class5/call_forward_busy.ats         # job2
 ats class5/call_forward_no_answer.ats    # job3
ats clip/clip_base.ats                    # job4
ats clip/clir_base.ats                    # job5
```

reads:

- first start executing `call_forward_unconditional.ats`. If it's OK (retcode = 0), then execute `call_forward_busy.ats`, and (regardless of its retcode) `call_forward_no_answer.ats`
- always execute `clip_base.ats` then `clir_base.ats`

Session Parameters Flows in Campaigns

Additionally, session parameters are transmitted to the executed children. In the example above:

- job1 will be started with the campaign's initial session parameters (a merge from the user provided values, if any, and the default values)
- job2, if executed, will be started with the session output from job1
- job3, if executed (same condition as for job2), will be executed with the session output from job1, too (its parent)
- job4 will be executed with the campaign's initial session parameters
- job5 will be executed with the campaign's initial session parameters

You can also define some local mappings to adjust the parameters to pass to a child job.

Let's assume the script `class5/call_forward_unconditional.ats` takes two parameters: `PX_SUT_IP`, defaulted to 127.0.0.1, `PX_SUT_PORT`, defaulted to 5060, and `PX_SOURCE_URI`, defaulted to `'sip:john@testerman.fr'`. In a campaign defined as:

```
ats class5/call_forward_unconditional.ats with PX_SUT_IP=192.168.1.1,PX_SOURCE_URI=sip:campaign@s
```

the ATS will be executed with explicitly provided `PX_SUT_IP` and `PX_SOURCE_URI` values, but keeping the default ATS value for `PX_SUT_PORT` (5060).

However, hardcoding SUT-dependent values is probably not a good idea. Instead, we'd probably define the `PX_SUT_IP` parameter at campaign level, and set it on run or via its default value.

```
ats class5/call_forward_unconditional.ats with PX_SOURCE_URI=sip:campaign@somewhere.com
```

with a `PX_SUT_IP` defined as a parameter for the campaign.

Note: this is equivalent to:

```
ats class5/call_forward_unconditional.ats with PX_SUT_IP=${PX_SUT_IP},PX_SOURCE_URI=sip:campaign@s
```

Now, if you have several ATSes using the same parameter names for different purposes, for instance PX_SUT_IP, used to defined a SIP server in one ATS, and used to defined a LDAP interface in another ATS, you can design different parameters at campaign levels and map them to their local names when needed:

```
ats class5/sip_test.ats with PX_SUT_IP=${PX_SIP_SUT_IP}
ats class5/ldap_provisioning_test.ats with PX_SUT_IP=${PX_LDAP_SUT_IP}
```

and defining `PX_SIP_SUT_IP` and `PX_LDAP_SUT_IP` as parameters for the campaign.

You got it, `'${NAME}'` is the way to reference a session parameter named `NAME`. If such a parameter is not defined when requested, no substitution occurs (`'${UNKNOWN_PARAM}'` will be expanded to `'${UNKNOW_PARAM}'`).

As it is a mere string substitution, you may design campaigns whose parameterization is more user-friendly than the ATSes (or campaigns) they embed:

```
ats another_test.ats with PX_PROBE_URI=probe:_@${PX_AGENT}
```

The ATS `another_test.ats` was designed to make the whole probe URI configurable. In the campaign, only the agent is, indirectly reducing the amount of information to set.

Jobs

ATSes and campaigns are executed as "jobs" created internally by the server.

Job Control

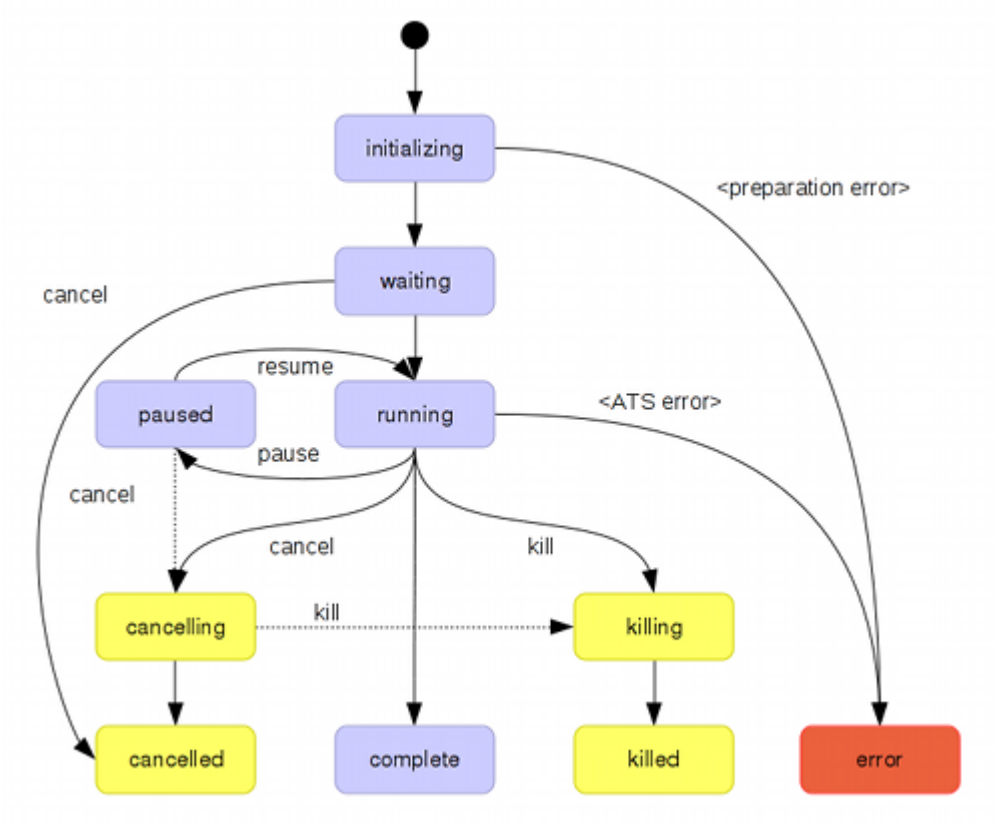
Some clients, for instance QTesterman, provides a user interface to control the scheduled or running testerman jobs.

Jobs are controlled sending *signals* to them, through the Ws interface using the `sendSignal(jobId, signal)` API. The job reacts differently according to its state when receiving the signal.

Signal	Description	**Acceptable states**	**Final state**
pause	pause the job	running	paused
resume	resume a paused job	paused	running
cancel	cancel a waiting job (preventing it from being executed), or stop a running job after its current ATS is over (for a campaign), or gracefully stop the current testcase (for an ATS) then stop the ATS. This automatically resumes the job if it was paused before cancelling it.	waiting, running, paused	cancelled
kill	kill a running job, not waiting for any pending testcase completion. This should only be used if the cancel operation does not work, as it may leave remote probe resources unfreed.	running, cancelling	killed

Job Lifecycle

The following diagram exposes the basic job state machine, from its birth to its multiple death possibilities:



Some state explanations:

- **initializing**: the job is being prepared and dependencies scanned: campaigns are parsed and missing ATSes or children are reported, leading to an error; imported modules in ATSes are checked.
- **waiting**: the job is now in the server's queue, waiting for its start according to its scheduled start time.
- **running**: the job is now running, either executing testcases for ATSes, or ATSes for campaigns
- **paused**: the job has been paused. Only meaningful for an ATS job. Running timers, if any, are not frozen during the pause. As a consequence, when resuming the job, several timers may expire immediately.

- **complete**: the job completes its execution "successfully", i.e. no technical errors (TTCN3-, Testerman- or Python-related errors/exceptions) occurred, and the job return code is 0 (you may alter it with the stop(retcode) statement in the control part). However, it does not mean that all testcases were OK.
- **cancelling**: the job is being cancelled, i.e. it waits for the pending testcase to finish, then stops.
- **cancelled**: the job has been cancelled, i.e. probably did not complete all its testcases (unless the cancel signal arrived during the last testcase execution). The associated log file is still valid and consistent to analyze testcases till the cancellation.
- **killing**: the job is being killed, i.e. stopped without waiting for a possible pending testcase to finish. This state typically lasts less than one second. Anyway, you can't do anything more to kill the job now.
- **killed**: the job has been killed. The associated log file may be inconsistent, especially regarding the running testcase when killed.
- **error**: a problem occurred either while preparing the job (campaign parsing error, temporary files creation problems, TE syntax error, ...) or a technical error occurred preventing the ATS continuation, typically a Python exception in the control part (incorrect testcase identifier, ...). In the first case, you should have a look to the server's logs to know what was wrong; in the second case, take a look at the log file in raw mode: the exception is probably logged. Additionally, the job's return code could help you diagnose the problem:

Return code	Description	**Associated state**	**Comments**
0	No error	complete	
1	Cancelled (by the user)	cancelled	
2	Killed (by the user)	killed	
3	Killed by the OS	error	Could be a segfault, out of memory, ... check the server's logs for the exact signal.
4	Complete, but some testcases were not executed successfully	complete	This status enables to quickly identify that at least one testcase was not passed, and the ATS may require your attention.
10	TE/Runtime: Unable to initialize the logger	error	Check the server's log for a possible additional trace. Check II interface settings and local firewall settings.
11	TE/Runtime: Unable to initialize core libraries	error	Check the ATS log file for more details, in raw mode.
12	TE/Runtime: TTCN-3 related error	error	You did something not compliant with the TTCN-3 logic in the control part. Check the ATS log file for more details, in raw mode.
13	TE/Runtime: Generic TE error	error	An exception occurred in the control part, probably a missing or invalid identifier. Check the ATS log file for more details, in raw mode.
20	Preparation: Unable to write the TE	error	Check disk space and rights to create a file in the document_root/archives folder.
21	Preparation: Python Syntax error	error	Look at the server's logs for the error line in the TE.
22	Preparation: Unable to check the TE	error	Look at the server's logs for more details.
23	Preparation: Unable to extract ATS parameters from its metadata	error	Look at the server's logs for more details.
24	Preparation: Unable to create input session file or TE dependencies	error	Check disk space and rights to create a file in the tmp_root.
25	Preparation: Unable to locate all module dependencies	error	Check `import` statements in ATS and imported modules (you can only import modules that are in the repository or Python system modules). The missing dependencies are reported to some client such as QTesterman. Alternatively, you may look at the server's logs.
26	Preparation: Unable to create the TE	error	The exact error is provided to the caller when submitting the job for an standalone run (i.e. outside a campaign). Typical errors include unsupported language APIs, or internal (but specified) errors.
>= 100	user defined, via stop(retcode) in control part	error	

Documentation System

Testerman provides a highly flexible framework to create documentation from ATS scripts, based on Python's docstring capabilities.

A docstring is a character string (usually multi-line) that is inserted just after defining a Python object, in particular a class or a function. Testerman re-uses this model to offer the ability to document user-created functions as well as test cases through the use of documentation plugins that extract these information to create a test specification document, or to export to a test management system, etc.

Raw Documentation

Basically, if you need to document a test case named `TC_MY_TESTCASE`:

```
class TC_MY_TESTCASE(TestCase):
  """
  This is the test case documentation.

  This test verifies that an SNMP trap
  is actually sent when rebooting the server.
  We first listen for a trap,
  then we reboot a machine using ssh reboot.
  We then should get our trap within 30 seconds.

  Created by John Smith on 2009-05-05.
  """
  def body(self, ...):
    """
    A documentation can also apply here
    """
    ...
```

Documentation Tags

It is usually convenient to structure your test case documentation into something more formal. A test case specification, for instance, may contain a purpose, the description of the steps to perform, some prerequisites, an author, a creation date, and so on.

The documentation may be left as is, using a plain English text, or turned into something more formal and still human-readable using tags. Tags are special markers in the plain text that indicates the different parts of the documentation. If you are familiar with code documentation system such as Epydoc, Javadoc, Doxygen, there is nothing new here – the syntax is (almost) the same:

```
class TC_MY_TESTCASE(TestCase):
  """
  This is the test case documentation.

  @purpose: This test verifies that an SNMP trap
  is actually sent when rebooting the server.
  @steps:
  We first listen for a trap,
  then we reboot a machine using ssh reboot.

  We then should get our trap within 30 seconds.

  @author: John Smith
  @date: 2009-05-05
  """
  def body(self, ...):
    """
    A documentation can also apply here
    """
    ...
```

`@purposes`, `@steps`, `@author`, `@date` are used to tag different parts of the raw documentation. In this sample, however, we left what could be a test case overview untagged, as the "natural", basic test case documentation.

Documentation plugins, for instance, can then access tag values directly to create some more formal test specification (or any other kind of documentation).

Any tags can be created at any time, anywhere in docstrings. However, they are mainly interpreted by plugins and you should match their expectations, according to the documentation strategy defined by your Testerman administrator and test managers. Testerman only provides a framework, regardless of the way you plan to use it.

Tag Format

A tag is defined as an identifier (`[a-zA-Z0-9_-]+`) following a `@` character, starting a new line. Its value starts just after the following `:` character.

Tags are case-insensitive: for instance, `@purpose` and `@Purpose` define the same tag `purpose`.

Tag Value Format

A tag value can be written over multiple lines. Actually, the current tags value only stops when another tag is started or when the docstring ends:

```
@purpose: This test verifies that an SNMP trap
is actually sent when rebooting the server.
```

Resulting in a tag value:

```
This test verifies that an SNMP trap is actually sent when rebooting the server.
```

Notice that leading and trailing spaces are stripped, so that this is equivalent to:

```
@purpose:
This test verifies that an SNMP trap
is actually sent when rebooting the server.
```

If you need to create a value that contains carriage returns, you must either leave an empty line or starts a new line with at least one blank character (space(s), tab(s)):

```
@steps:
We first listen for a trap,
then we reboot a machine using ssh reboot.
We then should get our trap within 30 seconds.
```

Resulting in the following tag value:

```
We first listen for a trap, then we reboot a machine using ssh reboot. We then should get our trap
```

Which is not really readable. So let's use:

```
@steps:
We first listen for a trap,
then we reboot a machine using ssh reboot.

We then should get our trap within 30 seconds.
```

Or

```
@steps:
We first listen for a trap, then we reboot a machine using ssh reboot.
We then should get our trap within 30 seconds.
```

Leading to:

```
We first listen for a trap, then we reboot a machine using ssh reboot.
We then should get our trap within 30 seconds.
```

General Concepts

Test Cases

...

Control Part

You may think of control part of an ATS as the main function in a program: this is where test cases are instantiated and executed in the order that fits your needs, with additional conditions or loops or the usual programming control primitives.

The control code is usually located at the end of the ATS, and it is a good practice to stick to this structure, defining all test cases classes before it, leading to the following advised ATS layout:

1. ATS header, comments
2. `import` directives
3. ATS-wide function definitions, including templates (non-shared functions or templates are better kept within their test cases)
4. Test Case and Behaviour definitions as Python classes
5. Control part

For instance:

```
##
# Some comments here
##


# Let's import some stuff shared with our colleages
from myteam.SharedFunctions_2_3 as CommonTeam
import mysite.utils as Utils


# We define some templates and functions
def mw_myTemplate(text, minsize = 10, author = any()):
  return { 'text': text, 'size': greater_than(minsize), 'author': author }

def f_sleep(duration):
  t_sleep = Timer(duration)
  t_sleep.start()
  t_sleep.timeout()


# Now, let's define some test cases
class TC_MY_TESTCASE(TestCase):
  """
  A first test case.
  """
  def body(self):
    ...

# Another one
class TC_MY_SECOND_TESTCASE(TestCase):
  """
  Don't forget to document me.
  """
  def body(self, loopCount = 10):
    ...


# Finally, the control part
TC_MY_TESTCASE().execute()
TC_MY_SECOND_TESTCASE().execute(loopCount = 20)
```

In the ATS control section, TTCN-3 allows several basic operations, and so does Testerman:

- Logging, using the [#ThelogStatement `log()` statement]

- Stopping the ATS explicitly, using the [#ThestopStatement `stop()` statement]
- [#ExecutingaTestCase Executing a test case], optionally modifying some test case information on the fly (such as its ID)

In addition, TTCN-3 provides a way to create and start timers from the control part. Testerman does not, however it offers several facilities to:

- [#ControllingLogLevel Control log level]
- [#TestAdapterConfiguration Define and use test adapter configurations]
- [#ControllingTestCasesExecution Control ATS autostop after a testcase failure]

Executing a Test Case

To execute a test case, you first have to instantiate its class, then call its `execute()` method. In practice, this translates to:

```
TC_MY_TESTCASE().execute()
```

The `execute()` method may take any optional or mandatory arguments, depending on how you defined the test case body. Naming the argument before assigning it a value is mandatory. Let's imagine you defined a test case this way:

```
class TC_MY_TESTCASE(TestCase):
  def body(self, clip, clir, cnip = False, cnir = False):
    ...
```

This defines a test case with 4 parameters: two of them are mandatory (`clip` and `clir`), while the two last (`cnip` and `cnir`) are optional, as they have a default value. You may then execute such a test case with:

```
# Minimal case: we only provide the mandatory parameters
TC_MY_TESTCASE().execute(clir = False, clip = True)
# We provide an additional, optional parameter value
TC_MY_TESTCASE().execute(clir = False, clip = True, cnir = True)
# Or all of them (notice that the argument order does not matter):
TC_MY_TESTCASE().execute(cnip = True, clir = False, clip = True, cnir = True)
```

The following cases would lead to a runtime error, as the mandatory paraemeters won't be set:

```
# Missing mandatory parameters
TC_MY_TESTCASE().execute()
# Missing argument name - not a syntax error, but won't fill clip/clir
TC_MY_TESTCASE().execute(False, True)
# Missing mandatory parameter
TC_MY_TESTCASE().execute(clir = True, cnir = False)
```

A call to `execute()` returns the test case verdict, as a character string in `"pass"`, `"inconc"`, `"fail"`, `"none"`, `"error"`. You may use this result to execute additional tests or stop the ATS conditionally (see [#ThestopStatement below] for some examples).

Test Case Instantiation

Upon test case instantiation, you also have the possibility to alter some of its static attributes to adapt them to the execution context:

```
TC_MY_TESTCASE(title = "Sample test case", id_suffix = "001").execute()
```

Both `title` and `id_suffix` are optional.

- `title` is a way to label your test case with a friendly short description. Such a title can then be used in log analyzers and reporters to create more clear and readable test execution reports.

- `id_suffix` enables to alter the test case ID, which is its class name (in this sample, `TC_MY_TESTCASE`) with an additional suffix. When such a suffix is present, an underscore character is automatically added before adding it to the native ID. In this case, for instance, the executed test case would have a final ID equals to `TC_MY_TESTCASE_001`.

These parameters may prove useful when looping over the same test case with different parameters:

```
i = 0
for clir in [ True, False ]:
  for clip in [ True, False ]:
    i += 1
    TC_MY_TESTCASE(id_prefix = "%3.3d" % i, title = "Call with clip=%s, clir=%s" % (clip, clir)).
```

will result in the four following test cases:

ID	Title
TC_MY_TESTCASE_001	Call with clip=True, clir=True
TC_MY_TESTCASE_002	Call with clip=False, clir=True
TC_MY_TESTCASE_003	Call with clip=True, clir=False
TC_MY_TESTCASE_004	Call with clip=True, clir=False

In particular, their IDs are now unique.

Controlling Log Level

Several functions are provided to control the logs that the execution may generate.

Generally, you don't need to use these functions, as the default log levels provide all that is necessary to construct acceptable log files for functional testing analysis. However, in several conditions, you may need to add additional internal traces, using:

```
enable_debug_logs()
```

or disable them later in your ATS, using:

```
disable_debug_logs()
```

Finally, if you don't care analysing your test cases (known to work already) and you need to boost test execution performances (for minimal load testing, for instance), you may use:

```
disable_logs()
```

to disable all traces except the core ones that will be used to identify ATSes and test cases executions.

A fine-grain log control is also available, using `enable_log_levels(*levels)` and `disable_log_levels(*levels)`.

Controlling Test Cases Execution

The `execute()` method of a Test Case returns its verdict. It enables to stop your ATS conditionally:

```
if TC_MY_TESTCASE().execute() != PASS:
  stop() # explicit ATS stop, this is a critical testcase
TC_MY_SECOND_TESTCASE().execute()  # we don't care about its verdict
if TC_MY_THIRD_TESTCASE().execute() != PASS:
  stop() # another important testcase
```

While this method offers a precise control over which testcase may be considered critical enough to justify the ATS abortion, it is not convenient when you want any testcase failure to stop the ATS.

In this case, you may use the `stop_ats_on_testcase_failure()` directive:

```
# auto stop on failure
stop_ats_on_testcase_failure()

TC_MY_TESTCASE().execute()
TC_MY_THIRD_TESTCASE().execute()
```

Test Adapter Configuration

A test adapter configuration is a Testerman context that enables to define a set of bindings, i.e. the mapping between test system interface ports and probes, with their location on the network and particular properties, to use when executing your test cases.

In the mid term, such configurations will be taken outside the ATS, as they can be different from one site (or user) to another one. This deals with ATS portability, and running an ATS to another site should not make the operator modify the source at all.

However, for now, the test adapter configuration can be done only programmatically, using the `TestAdapterConfiguration` Testerman class:

```
myconfig = TestAdapterConfiguration('myconfig')
myconfig.bind('ldapServer', 'probe:ldap', 'ldap')
myconfig.bind('hlr', 'probe:sctp@remoteagent', 'sctp', listening_port = 14001)
```

This will create a test adapter configuration labelled/named `default`, binding a ldap probe to the test system interface port name `ldapServer` and a remote SCTP probe bound to `hlr`. This probe is also pre-configured to listen on port 14001.

Then, to activate a test adapter configuration, use:

```
use_test_adapter_configuration('myconfig')
```

where `name` is the name of your defined configuration, for instance `'myconfig'` in the example above.

Activating a configuration automatically deactivates the previous one, if any.

However, it is not mandatory to create a test adapter configuration explicitely. You may also use the `bind` instruction directly, and it will use a built-in, default test adapter configuration:

```
bind('ldapServer', 'probe:ldap', 'ldap')
bind('hlr', 'probe:sctp@remoteagent', 'sctp', listening_port = 14001)
```

is actually enough to be able to use the tsiPort `'ldapServer'` and `'hlr'` in your testcases.

Basic Statements

The `stop()` Statement

This statement can be used in multiple locations, leading to different effects according to the calling context.

- When called from a (running) PTC (either from its Behaviour body or any of the functions called from it, including in alternative actions), it stops the PTC whose final local verdict will then be its last known local verdict.
- When called from the MTC, i.e. from the test case body or any of the functions called from it, it stops the test case itself, requesting all running PTC to stop, merging all current local verdicts at the time of the stop action as the final test case verdict.

A typical usage pattern can be:

```
setverdict('fail')
stop()
```

used when something went wrong and you don't need to continue your test case anymore. Notice that you don't need to stop whenever you set the verdict to `fail` to fail your test case since this verdict value can't be overriden.

In case of explicit test case stop, beware that you may skip the postamble part of your test case that is supposed to restore the SUT state to what it was before the test case started. You should use this statement carefully.

- In addition, you may call `stop()` from anywhere in the ATS control part to stop the ATS explicitly. In this case, Testerman provides an additional, optional integer argument to this function so that you can control the ATS return code - useful to control a campaign continuation, or simply state that the ATS was "failed" or "passed". This code is also returned by the Testerman CLIclient in synchronous execution mode, enabling to check an ATS status from shell scripts, Makefile, continuous integration engines or the likes.
- An ATS is considered complete if its return code is 0 - this is the default behaviour. This status is independent from the executed test cases results: it just indicates that the ATS was run up to its end.
- An ATS is considered not complete if its return code is greather or equals to 1. Return codes from 1 to 99 (included) are reserved for runtime errors and ATS control (abnormal terminations, either system or user-triggered). Return codes >= 100 (and <= 255) are for your own use, and you should only pass return codes in this range to `stop()`.

You may control this return code with:

```
# Don't continue if this test case is not passed,
# and consider the ATS should report an error to the ATS executor
# (a campaign, or a Makefile, ...)
if TC_VERY_IMPORTANT_TESTCASE().execute() != PASS:
  stop(100)

# In this case, we stop the ATS for optimization reasons:
# if the first TC fails, we assume that all others will fail too.
# Since they lasts a very long time, we don't want to waste our time waiting
# for their completion.
if TC_FIRST_OF_A_SERIES_OF_SAME_KIND_OF_TC().execute(param = value1) != PASS:
  stop()
TC_FIRST_OF_A_SERIES_OF_SAME_KIND_OF_TC().execute(param = value2)
TC_FIRST_OF_A_SERIES_OF_SAME_KIND_OF_TC().execute(param = value3)
TC_FIRST_OF_A_SERIES_OF_SAME_KIND_OF_TC().execute(param = value4)
```

Setting a return code in a `stop()` called from a test case (i.e. not from the control part) has no effect.

The `log()` Statement

You may log a user message at any time, anywhere in your ATS, using the following function:

```
log("This is a user message")
```

This produces a user log element in the logger system that can be analyzed and displayed in the various log viewers available for Testerman. Such a logged message is automatically attached to a logging entity depending on the calling context:

- when called from the control part, the logged message is not attached to any test case.
- when called from the test case body or any function called from here, the logged message is attached to the main test component (MTC). The QTesterman visual log viewer, for instance, is able to make this association visible to the end-user.
- when called from a behaviour body or any function called from here, the logged message is attached to the parallel test component (PTC) running the behaviour. The QTesterman visual log viewer is able to make this association visible to the end-user, too.

`log()` only takes a single string or unicode string argument. The usual Python formatting methods apply, such as:

```
log("Current verdict: %s ; based on parameter p1=%s" % (getverdict(), p1))
```

Test Components

...

Timers

Timers can be defined at any time as soon as a test case is running (TTCN-3 allows the use of timers in the control part, this is not the case for Testerman). They can be declared and manipulated in the MTC, PTC, or any functions called from them.

Once started, a timer emits a timeout event on expiry. It can be restarted at any time (while running, stopped or expired), stopped before its timeout, and provides a way to measure the elapsed time since its start.

To declare a timer, use:

```
t_myTimer = Timer()
```

Alternatively, you may assign a default duration to it, expressed in seconds (as a non-negative float), and a name to track it in execution logs:

```
t_myTimer = Timer(2.0, "my timer") # defines a 2-second timer labeled "my timer"
```

Both arguments are optional, and you may use named arguments as well:

```
t_myTimer = Timer(duration = 2.0, name = "my timer") # defines a 2-second timer labeled "my timer
```

If no name is provided, an identifier is automatically generated. It is unique for each timer. If no duration is provided, you will have to provide one when starting it. Consider this is a default duration.

To start a timer, use:

```
t_myTimer.start() # raise a runtime error (exception) if no duration was provided on declaration
```

or

```
t_myTimer.start(5.0) # start the timer with a 5 second duration, overriding the default duration,
```

If you start an already running timer, it is restarted with the new duration, or, if not provided, the default duration provided during the timer declaration.

At any time you may stop it:

```
t_myTimer.stop()
```

This has no effect if the timer was not running (already stopped, expired, or never started).

While the timer is running, you may use:

```
elapsed = t_myTimer.read()
```

which returns the number of seconds, as a float, since the timer start. Returns `0.0` if the timer is not running; and

```
if t_myTimer.running():
  ...
```

which returns a boolean value indicating if the timer is running. Returns False only if the timer is stopped, expired, or never started.

To catch the timeout event, you may use, in a `alt` statement:

```
alt([
  [ t_myTimer.TIMEOUT, # timer timeout
    lambda: log("Do something on timer expiry"),
    lambda: ...
  ],
  ...
])
```

or the shortcut:

```
t_myTimer.timeout()
```

if you don't have any other events or messages to match in parallel. This is equivalent to `alt([[t_myTimer.TIMEOUT]])`.

In particular, this is used to implement a sleep/wait function:

```
# Sleep of 1.5s, the TTCN-3/Testerman way
t_myTimer = Timer(1.5)
t_myTimer.start()
t_myTimer.timeout()
```

It is important to use this pattern for a sleep/wait implementation instead of the standard Python library `time.sleep()` to ensure that your code is interruptible by a `ptc.stop()` or equivalent.

Notes:

- `t_myTimer.timeout()` blocks until the timeout event for this timer is detected. However, it will returns immediately if the timer is not running.
- `t_myTimer.TIMEOUT` event can be matched only once. It will me matchable again once the timer has been restarted.

Template Matching

Matching Mechanisms

Template matching is one of the most interesting feature of TTCN-3. It enables to detect if we receive a quite precise message without any manual checks or conditional value traversal. In addition to constant matching, several matching mechanisms are available to act as wildcards or conditions. These mechanisms are used in place of a constant in a template.

Testerman implements the following template matching mechanisms:

mechanism	arguments	applies to*	description
`any()`		any type of values	matches any but non-empty value. May be used as a single element wildcard in a list, too.
`any_or_none()`		any type of values	matches any values, including empty values (lists, dicts, strings) or absent values (in a dict). May be used as "any number of elements" wildcard in a list, too
`greater_than(value)`	integer, float	integer or float	matches values v as `value` <= `v`
`lower_than(value)`	integer, float	integer or float	matches values v as `v` <= `value`
`between(a, b)`	integer, float	integer or float	matches values v as `a` <= `v` <= `b`
`empty()`		list, dict, string	matches empty lists, dicts, or strings
`pattern(pattern)`	Python regular expression	string	matches strings that matches the regular expression `pattern`
`omit()`		any value in a dict	enables to match a dict only if the associated field is not present in the dict (i.e. the entry has been omitted)
`ifpresent(template)`	a Testerman template	any value in a dict	enables to apply the `template` to the value if the field present in a received dict, or still accept to match the dict if the field is not present
`superset(*template)`	any number of elements of any type	lists	matches any list that contains at least one time each of the given elements, in any order
`subset(*template)`	any number of elements of any type	lists	matches any list that contains only elements in the given elements, 0 or more times, in any order
`complement(*template)`	any number of elements of any type	lists	matches any list that does not contain any of the given elements
`length(template)`	a Testerman (scalar) template	list, string, dict	extracts the length of the received message, and matches it against the provided template

* "applies to" means "can be used to match"

Mechanisms can be combined together. See the examples below.

Examples

message	template	matched ?	comments
`1.0`	`any()`	yes	
`0`	`any()`	yes	
`[]`	`any()`	no	
`[]`	`any_or_none()`	yes	
`hello`	`pattern(r'^hell.*')`	yes	
`{ 'key': 123, 'password': 'secret' }`	`{ 'key': between(100, 200) }`	yes	
`{ 'key': 123, 'password': 'secret' }`	`{ 'key': lower_than(200), 'password': omit() }`	no	the field `password` should not be present
`{ 'key': 123 }`	`{ 'key': any(), 'password': any_or_none() }`	yes	
`{ 'password': 'secret' }`	`{ 'key': any(), 'password': any_or_none() }`	no	the `key` field must be present (but may have any value)
`{ 'key': 123 }`	`{ 'key': any(), 'password': ifpresent('secret') }`	yes	`password` was made optional
`{ 'key': 123, 'password': 'hello' }`	`{ 'key': any(), 'password': ifpresent(pattern(r'secret.*')) }`	no	`password` is now present, but does not match the sub-template
`'verylongpassword'`	`length(greater_than(16))`	yes	
`[1, 2, 3]`	`subset(3, 2, 4, 1, 5, 6)`	yes	
`[1, 2, 3, 1]`	`superset(2, 1)`	yes	
`[1, 2, 3, 2]`	`superset(1, 2, 3, 4)`	no	
`[1, 1, 2, 2]`	`complement(3, 4, 5, 6, 7)`	yes	

List matching

List matching may be tricky because ordered. Several mechanisms, however, can help you matching exactly what you need, even if you don't know the complete list you may receive (optional elements, etc). In particular, you can use `any()` and `any_or_none()` as `?` and `*` wildcards, respectively:

message	template	matched ?	comments
`[ 1, 2, 3, 4, 5, 6 ]`	`[]`	no	
`[ 1, 2, 3, 4, 5, 6 ]`	`[ 1, 2, 3, 4, 5, 6, 7 ]`	no	
`[ 1, 2, 3, 4, 5, 6 ]`	`[ 6, 5, 4, 3, 2, 1 ]`	no	not in the correct order
`[ 1, 2, 3, 4, 5, 6 ]`	`[ 1, 2, 3, 4, 5, 6 ]`	yes	
`[ 1, 2, 3, 4, 5, 6 ]`	`[ 1, 2, any(), 4, 5, 6 ]`	yes	`any()` can replace any single element...
`[ 1, 2, 3, 4, 5, 6 ]`	`[ 1, 2, 3, any(), 4, 5, 6 ]`	no	...but this element must be present
`[ 1, 2, 3, 4, 5, 6 ]`	`[ 1, 2, any_or_none(), 5, 6 ]`	yes	`any_or_none()` can replace any number of elements...
`[ 1, 2, 3, 4, 5, 6 ]`	`[ 1, 2, 3, any_or_none(), 4, 5, 6 ]`	yes	...even zero
`[ 1, 2, 3, 4, 5, 6 ]`	`[ any_or_none(), 3, any(), 5, 6 ]`	yes	you may combine `any()` and `any_or_none()`
`[ 1, 2, 3, 4, 5, 6 ]`	`[ any_or_none(), 3, any_or_none() ]`	yes	equivalent to `superset(3)`, which may be more readable

And you may combine any other matching mechanism as well:

message	template	matched ?	comments
`[ 1, 2, 3, 4, 5, 6 ]`	`[ lower_than(10), 2, 3, any_or_none() ]`	yes	
`[ 1, 2, 3, 4, 5, 6 ]`	`[ any_or_none(), lower_than(2), any_or_none(), 3, 4, 5, any() ]`	yes	
`[ 1, 2, 3, 4, 5, 6 ]`	`superset(greater_than(5))`	yes	

Matching Mechanisms Valuation

To avoid writing different templates for both sending and receiving purposes, Testerman proposes an extension to TTCN-3 to valuate some matching mechanisms.

For instance, if you defined the following template:

```
mw_received_message = { 'location': 'Grenoble, France', 'weatherForecast': { 'temperature': betwee
```

You may also send it though it does not contain only fully qualified values due to the `between` matching condition. In this example, the sent value will use a `temperature` field valuated to a random integer between 0 and 30 (inclusive).

The following table provides the possible valuations for the matching mechanisms that implement one:

template matching mechanism	valuation type	valuation value	comments
`between(a, b)`	integer	random integer >= a, <= b	
`lower_than(a)`	a's type	value of a	
`greater_than(a)`	a's type	value of a	
`any()`	`None`	`None`	

All valuations are implemented so that they lead to a value that matches the corresponding matching mechanisms. When trying to send a message that contains non-value-able matching mechanisms, a Testerman exception occurs.

Alternatives

Alternatives are a way to express the branching of a test behaviour upon the reception of messages on selected ports, timer events, or termination of (parallel) test components. Basically, you can see them as a kind of `switch..case` operating asynchronously and self-reevaluating in an infinite loop, until one branch is selected. This is a pooling loop where we wait for SUT (or Timer or PTC) events.

The event is called a "branch condition", and is associated to some code to execute if the condition is met, i.e. when the branch is selected. Branches can be of five different types:

- *receiving-branch*: such a branch is selected when a message matching a template has been receiving on a port. The associated condition is expressed, in Testerman, using `port.RECEIVE(template)`, as detailed below.
- *timeout-branch*: this kind of branch is selected when a timer expires. The associated condition is expressed using `timer.TIMEOUT`.
- *done-branch*: the associated branch is selected when a PTC is complete; it is denoted `ptc.DONE` in Testerman.
- *killed-branch*: the associated branch is selected when a PTC is killed; denoted `ptc.KILLED` in testerman.
- TTCN-3 also specifies an *altstep-branch*, which is currently not supported in Testerman.

Notice that all branch conditions syntaxes use methods or members in uppercase. It helps differentiate them from the operations `port.receive(template)`, `timer.timeout()`, `ptc.done()`, `ptc.killed()` respectively, which are basically shortcuts to a `alt()` statement containing only a single branch condition.

Additionally, a branch condition can be optionally guarded, i.e. only considered if an additional condition evaluates to true. If not provided, the guard is assumed to be always fulfilled, and the branch condition is always taken into account.

Finally, the branch contains some code to execute if its branch condition is matched. In Testerman, this code is written as a list of `lambda` functions (aka anonymous functions).

Note for the curious readers: lambda functions are used to prevent evaluating their contained statements when calling the `alt()` function - since we basically write Python code in the Testerman ATS, Python naturally evaluates all arguments to a function before being able to call it. In our case, this is not what we want, since we want to execute code conditionally. As an alternative (no pun intended), you may call a single function that contains all your code for the branch. This is, by the way, your only choice if you need multiple-line lambda functions (containing control structures such as `if/else`, `while`, ...).

These complete branches (optional guard, branch condition, code) are technically written as a list, and all these branches are gathered into another, ordered list which is passed as the single argument to the `alt()` function, leading to the following kind of construct:

```
alt([
  [ port01.RECEIVE(mw_myTemplate),
    lambda: log("This is a receiving-branch"),
  ],
  [ port02.RECEIVE(mw_mySecondTemplate),
    lambda: log("This is another receiving-branch, on an other port"),
  ],
  [ port01.RECEIVE(),
    lambda: log("Still a receiving-branch, matching all messages on port01"),
    lambda: log("You can use several lambda in the 'code block'"),
    lambda: setverdict('fail'),
  ],
  [ lambda: a >= 1, port02.RECEIVE(mw_myThirdTemplate),
    lambda: log("This is another receiving-branch, guarded only considered if a >= 1"),
  ],
  [ t_timer.TIMEOUT,
    lambda: log("This is a timeout-branch"),
  ],
  [ ptc.DONE,
```

```
    lambda: log("This is a done-branch"),
  ],
  [ ptc.KILLED,
    lambda: log("This is a killed-branch"),
  ],
])
```

The order branches are written does matter, due to the following rules when entering an `alt()`:

- whenever we enter a alt or loop over it (once all conditions have been checked and mismatched, or due to an explicit repeat using `REPEAT`), a "snapshot" of the current system is taken, memorizing all message queues states on all ports that are involved in the alt (actually, this snapshot is not implemented for now - ticket:20 - just consider it should be the expected behaviour, however) - in our example, `port01` and `port02` - as well as the current known PTC and timer event,
- we try to match the snapshot messages/events against the different branch conditions `in their order of appearance` - providing their guards are fulfilled (they are re-evaluated at each loop)
- if the branch condition is matched, then the associated code is executed. If the last executed statement evaluates to `REPEAT`, we restart the loop from scratch, with the matched message consumed, re-snapshooting the current system state. If the last executed statement evaluates to anything else, we exit the `alt()` call, with the matched message consumed, but all other messages on other ports unchanged. Matching a condition is the only way to exit a `alt()` call.
- if the branch condition is mismatched, we continue with the next branch condition
- once we mismatched all conditions in the alt, we discard the mismatched message, and restart our pass with the next message.

Since matching is "first-match" and not "best-match" based, the order does matter. In particular, in something like:

```
alt([
  [ port01.RECEIVE(),
    lambda: log("This condition hides the next one"),
  ],
  [ port01.RECEIVE(mw_myTemplate),
    lambda: log("This log will never be displayed"),
  ],
])
```

both conditions are expecting a message on the same port, but even if we receive a message that matches `mw_myTemplate`, it will first match the default template implicitly provided in `port01.RECEIVE()`, keeping the second branch from being selected.

Branch Conditions

...

Code Block

A code "block" is typically a list of lambda functions to call in this order. While this is quite convenient for small actions, such as sending a message back, setting a verdict, logging something, or event stop the current test component or test case, this is sufficient.

However, if you need to execute more complex statements, in particular control statements such as `if/elif/else`, `for..in`, or even variable assignment, you won't be able to do it from a lambda. In this case, you'll need to implement a function external to the `alt` call, or, if you just want to assign a variable, use something like a `StateManager` instance, which has been designed for this kind of case.

Examples: If you need an additional condition or loop in a branch:

```
def f_messageHandler(message):
  if message['method'] == 'POST':
    action1()
  elif message['method'] == 'PUT':
    action2()

def f_doSomeLoop():
  for i in range(10):
    port02.send(m_response(count = i))

alt([
  [ port01.RECEIVE(mw_request(), value = 'msg'),
    lambda: f_messageHandler(value('msg')),
  ],
  [ port02.RECEIVE(),
    lambda: f_doSomeLoop(),
  ],
  ...
])
```

Notice that most conditions could be embedded into the matching template too. In the example above, we may have just use 2 branch conditions, one on `port01.RECEIVE(mw_postRequest())`, another one on `port01.RECEIVE(mw_putRequest())`.

Loops, to a certain extent, can also be collapsed to a single line instruction using Python list comprehensions: `lambda: [ port02.send(m_response(count = i)) for i in range(10) ]` would have been equivalent to the `f_doSomeLoop()` code above, but may be less readable.

In case of variable assignation, something like `lambda: a = 1`, you'll simply get a syntax error. You can't assign a variable directly from a lambda, but you can assign a member variable, or use a `StateManager`:

```
# This sample counts the number of requests received on a port
count = StateManager(0)

alt([
  [ port01.RECEIVE(mw_request()),
    lambda: count.set(count.get() + 1), # increment a
    lambda: REPEAT, # repeat the alt
  ],
  [ port02.RECEIVE(),
    # no action - just exit the alt when something has been received on port02
  ]
])

log("OK, we received %d requests on port01" % count.get())
```

`StateManager` objects can be quite convenient to implement state machines (hence their names), as we will see below.

Repeating a Alt

By default, when a branch is selected, its code block is executed and the alt returns. If you need to re-enter the loop, waiting for another event, you may use the special "Testerman keyword" `REPEAT` as the (usually last) action in your code block:

```
alt([
  [ port01.RECEIVE(mw_interestingMessage()),
    lambda: log("Got it !") # then exit the alt
  ],
  [ port01.RECEIVE(mw_keepAlive()),
    lambda: log("KA message received, sending response")
    lambda: port01.send(m_keepAliveResponse()),
```

```
    lambda: REPEAT, # repeat the alt
  ],
])
```

Repeating the alt is the default behaviour when receiving a message that does not match your templates. So it's not use adding an alternative branch if you don't need to perform any particular action on this message.

`REPEAT` can also be returned by the function called in your code block. Combined with the fact that if `REPEAT` is not the last action of your code block, it breaks your action sequence to restart the alt, discarding the remaining actions, this can lead to interesting things:

```
count = 0

def f_conditionalLoop():
  count += 1
  if count < 10:
    return REPEAT
  return False # or anything != REPEAT

alt([
  [ port.RECEIVE(),
    lambda: action1(),
    lambda: f_conditionalLoop(), # if it returns REPEAT, action2 won't be executed this time.
    lambda: action2()
  ]
])

# action2 will be executed only in the last loop, before leaving the alt
```

However, this example would rather be implemented with two alternative guarded branches (more readable).

Returning from a Alt

By default, when a branch is selected, its code block is executed and the alt returns, so you don't need to return explicitly. However, you may implement conditional return in your list of actions via external functions:

```
def f_shouldWeContinue():
  if count > 10 and timer.read() < 10.0:
    return RETURN # don't continue
  return True # or anything != RETURN

alt([
  ...
  [ port.RECEIVE(),
    lambda: action1(),
    lambda: f_shouldWeContinue(), # if it returns RETURN, action2 won't be executed
    lambda: action2()
  ]
])
```

Returning the "Testerman keyword" `RETURN` immediately returns from the alt, discarding the subsequent actions. Writing it statically is also possible, but would have exactly the same effect as commenting out the subsequent actions:

```
alt([
  [ port.RECEIVE(),
    lambda: action1(),
    lambda: RETURN,
    lambda: action2(),
    lambda: action3()
  ]
])
```

```
# equivalent to:
alt([
  [ port.RECEIVE(),
    lambda: action1(),
  ]
])
```

Guards

Guards are defined as callable/0 Python objects, that is functions that do not take any argument. If the first element of a branch declaration list is such a callable object, Testerman assumes this is a guard. If not, the first element of the list is assumed to be the branch condition - this is the way the guard can be optional.

The most usual way to implement such guards is, once again, `lambda` functions:

```
class TC_GUARD(TestCase):
  def body(self):
    a = StateManager(0)

    alt([
      [ port01.RECEIVE(m_something()),
        lambda: log("let's increment a"),
        lambda: a.set(a.get() + 1),
        lambda: REPEAT,
      ],
      [ lambda: a.get() >= 1,
        port01.RECEIVE(),
        lambda: log("ok, now we are sure that we received at least once m_something() on port01"),
      ],
    ])
```

...

Alt Execution

You can execute one `alt()` per test component "thread". An alt is interruptible via `ptc.kill()` or `ptc.stop()` from any other test component, or only by a matching event. So be careful when entering the function, be sure to have a watchdog timer or a way to stop the polling loop gracefully.

Default Behaviours may help you with setting such watchdog timers for all `alt()` at once.

Default Behaviours

It is not unusual to have one or several events to catch systematically in a `alt` to execute some default actions, such as stopping the test case on error due to receiving a non-explicitly handled message, a global watchdog timeout keeping your test case from running indefinitely, or dealing with "background", uninteresting messages such as keep-alive probes.

In these case, you may appreciate to implement one or several default behaviours.

Basically, a default behaviour is a set of alternative branches that are automatically appended to any `alt()` defined branches. This set can be activated and deactivated at any time. It is also possible to activate multiple default behaviours - however they will be handled in the order or their activation.

Activating a Default Behaviour

To activate, i.e. register, a default behaviour, use:

```
myDefaultBehaviour = activate([
  [ t_watchdog.TIMEOUT,
    lambda: log("Global watchdog expiry - stopping testcase"),
    lambda: setverdict("fail"),
    lambda: stop()
  ],
  [ port.RECEIVE(),
    lambda: log("Unknown message received. Strict mode: stopping testcase"),
    lambda: setverdict("inconc"),
    lambda: stop()
  ],
])
```

You noticed that `activate` takes only one argument that is exactly constructed the same as for a `alt()`. It may contain as many branches as needed. `activate` returns an identifier that is suitable for a call to `deactivate` (see below), in case of you need to deactivate thisdeault behaviour.

Since you can activate multiple default behaviours in a row, it may be convenient to separate the branches sets according to their functions. For instance, one set for a global watchdog, one set for background error management:

```
defaultWatchdog = activate([[t_watchog.TIMEOUT, lambda: setverdict('fail'), lambda: stop()]])
defaultError = activate([[port.RECEIVE(mw_errorMessage()), lambda: setverdict('fail'), lambda: sto
```

An activation is only valid within a single test component, depending on where you activated it from. As a consequence, if you want to use a default behaviour in the MTC and in each PTC you create, you have to activate it from each PTC in addition to the MTC.

Notes:

- a default behaviour activated from within an alternative branch will only be taken into account in the next `alt` call and not in the current one, if it is repeated.

Deactivating a Default Behaviour

Once a default behaviour has been activated using `activate()`, it is taken into account in all subsequent calls to `alt()` (or functions that embeds an alt, such as `timer.timeout()`, `port.receive()`, etc) for the current test component. At any time, however, you may deactivate it using the identifier returned during its activation:

```
defaultWatchog = activate([ ... ])

...

deactivate(defaultWachdog)
# From now on, no more default watchdog handling in alt()
```

You can only deactivate a default behaviour that was activated in the same test component. Deactivating an already-deactivated behaviour has no effect.

Notes:

- a default behaviour deactivated from within an alternative branch will only be discarded in the next `alt` call and not in the current one, if it is repeated.

Template Value Extraction

Full Extraction

TTCN-3 defines a single way to extract a value from a message that matched a template, using the `->` syntax and `value` (and `sender`) keywords. For instance:

```
port.receive(my_template) -> value m, sender s;
```

will store the received message that matched the template `my_template` to the local variable `m`, and the address of the sender (either a test component reference or a SUT address) to the local variable `s`. Once stored, you may traverse the received message as any other structured value to find the field of interest. For instance, if we assumed we received a SIP request to which we should reply with a response using the same call-id, we may use:

```
type record SipRequestType {
  charstring method,
  ...
  charstring callId,
  ...
}

template SipRequestType mw_sipRequest()
{
  method := ?,
  ...
  callId := ?,
  ...
}

// ...

charstring callId;

port.receive(nw_sipRequest) -> value request;
callId := request.callId

// Now reinject the callId into a response

resp = m_sipResponse(callId)
// ...
```

Testerman offers a similar mechanism to match the complete received message (and the associated sender, if needed). The syntax, however, is different:

```
port.receive(m_myTemplate, value = 'm', sender = 's')
```

This will store the received message matching the template `my_template` to an internal structure whose value can be retrieve later within the same test component "thread" (i.e. within the behaviour/PTC "thread" or the main/MTC "thread") using:

```
matchedMessage = value('m')
```

The sender (either a reference to a test component (`TestComponent` instance) or the SUT address (Python built-in `string`) can be retrieved a similar way within the current test component "thread" with:

```
messageSender = value('s')
```

Of course, it is not mandatory to store the matched value to a variable; however you are advised to do so, as the matched value may be overriden on the next template match if you use the same value/sender name.

The above SIP example then translates to:

```
def mw_sipRequest():
  return { 'method': any(), 'callId': any() }

port.receive(sip_request(), value = 'request')
callId = value('request')['callId']

resp = m_sipResponse(callId)
...
```

Selective Extraction

The mechanism above is quite convenient to get a whole message. Sometimes, however, you may prefer get only a part of the matched message to avoid a structure traversal, especially when this structure is not as trivial as in the example above or when wildcards and lists are involved.

For example, if your template is something like:

```
# In SUA protocol, we may get an undefined list of parameters that contain a tag and a value.
# We are only interested in one of these parameters, but we cannot control the order
# we received them.
mw_myTemplate = [ any_or_none(), { 'tag': 0x06, 'value': any() }, any_or_none() ]
# This is equivalent to my_template = superset({ 'tag': 0x06, 'value': any() })
```

Once we matched it, we have to find the interesting parameter manually, checking the tag value in each entry of the matched list (provided all these entries contains a`tag` field, which may be not mandatory depending on the message structure / involved codec). Instead of traversing the list manually, Testerman proposes a selective extraction mechanism that is tightly bound to the template:

```
mw_myTemplate = [ any_or_none(), { 'tag': 0x06, 'value': extract(any(), 'my_val') }, any_or_none(
```

Using the `extract(<matching mechanism>, <name>)` feature, you can directly get the value you want to extract without requiring a full message traversal. The matched value, if the template has matched, is then available through the `value(<name>)` syntax as for full message extraction.

Full example:

```
mw_myTemplate = [ any_or_none(), { 'tag': 0x06, 'value': extract(any(), 'my_val') }, any_or_none(

alt([
  [ port.RECEIVE(mw_myTemplate),
    lambda: log("parameter 0x06 value: %s" % value('my_val'),
  ]
])
```

Notice that you can freely use `extract` in sending templates providing the wrapped template matching condition has a tangible valuation (typically `between`, `greater_than`, `lower_than`, ... - selective extraction is meaningless, but fully usable, to extract constants).

Limitations:

- This selective extraction mechanism does not work with `any_or_none()` (TTCN-3 `*`) wildcard.
- Calls to `value(name)` where `name` is a string referring to a selected extraction in a template that did not match is undefined (may or may not return a value, depending on when the mismatch was detected).

Verdict Management

Each test component has a local verdict that can be set and retrieved at any moment, from anywhere during the test component execution.

This verdict is said to be local as it is only valid for the running test component: this is the PTC verdict in a behaviour body or any functions called from it, or the MTC verdict in the test case body or any functions called from it.

You can only manipulate (get or set) the local verdict of your current context. The test case verdict is automatically computed from merging the different local verdicts.

In Testerman, verdict values are string literals instead (while they are keywords in TTCN-3). However, the full verdict values are available, and some Testerman constants are provided for convenience to avoid using string values:

TTCN-3 verdict	Testerman verdict value	description
`none`	`NONE` (or `'none'`)	default verdict, unset
`pass`	`PASS` (or `'pass'`)	the test component logic considers the SUT reactions it observed were what it expected. Test case passed successfully.
`fail`	`FAIL` (or `'fail'`)	the test component logic considers the SUT reactions it observed were not the expected ones. Due to the verdict values precedence rules, if at least one local verdict is set to `fail`, it implies that the Test case verdict will be `fail`, too.
`inconc`	`INCONC` (or `'inconc'`)	inconclusive: not enough elements have been observed to tell if the SUT reactions were correct or not. We can't tell that the test case failed because what we planned to test was not tested actually. Useful when some prerequisites cannot be set up or verified.
`error`	`ERROR` (or `'error'`)	an execution error occurred. Automatically set by the test execution system on runtime exception; cannot be set by the user.

Local verdicts are automatically merged to create a "test case" verdict using the following rules:

- the test case verdict is initialized to `none`
- whenever a test component is over (either done or killed), the test case verdict is updated with its local verdict,
- when updating a verdict (including the test case verdict), the following precedence rules apply: `none` < `pass` < `inconc` < `fail` < `error`, which can also be visualized as indicated in the table below, indicating the resulting verdict after an update:

current verdict	new verdict			
	pass	inconc	fail	none
none	pass	inconc	fail	none
pass	pass	inconc	fail	pass
inconc	inconc	inconc	fail	inconc
fail	fail	fail	fail	fail

The `error` value overwrites all others.

Notice that the order of verdict merges does not affect the final test case verdict (when we wait for N PTCs to complete, for instance).

Setting a Verdict

You can only set a local verdict; the test case verdict is automatically computed by the system according to the different local ones.

To update the current local verdict, use the Testerman function `setverdict` anywhere in your test case or behaviour body or in a called function, for instance:

```
setverdict(PASS) # you may use setverdict('pass') if not using the pre-defined constant
```

Setting a verdict that is not in `[ 'error', 'none', 'pass', 'fail', 'inconc' ]` leads to a run-time exception.

However, as a best practice to make your code more readable and more reusable, you should only set verdicts from behaviour an test case `body` methods, while functions you call should never decide for such a verdict: only the caller should know how to interpret a function result.

Getting a Verdict

Local Verdict (Test Component)

You can get a verdict at any time during a test component execution using the Testerman function `getverdict`:

```
v = getverdict()
```

The returned value is the current local verdict, as a string in `[ 'error', 'none', 'pass', 'fail', 'inconc' ]` (i.e. in '' [ERROR, NONE, PASS, FAIL, INCONC]'').

Additionally, when a test component is over (either done or killed), you may retrieve their local verdict with the `getverdict()` method:

```
ptc = create()
ptc.start(MyBehaviour())
ptc.done()
v_ptcVerdict = ptc.getverdict()
```

Test Case Verdict

The test case verdict is returned as a result to its execution; if you don't store it in a variable, it is lost. For instance:

```
TC_SAMPLE_01().execute()
v = TC_SAMPLE_02().execute()
if v != 'pass':
  stop()
```

could be a way to stop an ATS when a particular test case fails (or more precisely - does not succeed), avoiding executing additional test cases whose outcomes would already be known since this basic test case failed.

Reference: Testerman API

This API may evolve, but its backward compatibility is guaranteed, so that your ATSes can still work in next Testerman versions.

The whole API is made accessible directly in the ATS namespace. You should NOT import any Testerman modules in your ATS, as their names and contents may evolve without notice.

Timer Objects

Constructor:

```
Timer(duration = None, name = None)
```

Methods:

```
start(duration = None)
stop()
running()
timeout()
read()
```

TestComponent Objects

Constructor: N/A (constructed from a testcase only)

Methods (meaningless on MTC):

```
alive()
running()
start(behaviour, **kwargs)
stop()
kill()
done()
```

Members (meaningless on MTC):

```
DONE
KILLED
```

Specials:

```
get_item[name] # provides a reference to a TC port - creates it dynamically if needed
```

Port Objects

Constructor: N/A (constructed dynamically when calling `tc['portname']`)

Methods:

```
send(message, to = None)
receive(template = None, value = None, sender = None, from_ = None)
start()
stop()
clear()
RECEIVE(template = None, value = None, sender = None, from_ = None)
```

Behaviour Objects

Constructor:

```
Behaviour()
```

TestCase Objects

Constructor:

```
TestCase(title = None, id_suffix = None)
```

Methods:

```
set_description(description)
create(name = None, alive = False)
execute(**kwargs)
stop_testcase_on_failure(stop = True)
```

Functions Callable while a Testcase is Running

Can be used anywhere (functions, altsteps, behaviour, testcases):

```
stop()
log(msg)
getverdict(verdict)
alt(alternatives)
get_variable(name, default_value = None)
set_variable(name, value)
value(name)
```

```
sender(name)
match(message, template)
action(message, timeout = 5.0)
```

Should be used in behaviours and testcases only (avoid setting the local verdict from anywhere):

```
setverdict(verdict)
```

Should be used in testcases only (avoid dynamic test reconfiguration):

```
connect(portA, portB)
disconnect(portA, portB)
port_map(port, tsiPort)
port_unmap(port, tsiPort)
```

Default alternatives management:

```
activate(altstep)
deactivate(id_)
```

Templates matching mechanisms:

```
greater_than(value)
lower_than(value)
between(a, b)
any()
any_or_none()
empty()
pattern(pattern)
omit()
ifpresent(template)
length(template)
superset(*templates)
subset(*templates)
complement(*templates)
```

Selective message value extraction:

```
extract(template, value)
```

Codec :

```
with_(codec, template)
```

Control Part

Functions:

```
enable_debug_logs()
disable_debug_logs()
disable_logs()
enable_logs()
get_variable(name, default_value = None)
set_variable(name)
with_test_adapter_configuration(name)
bind(tsiPort, uri, type_, **kwargs)
stop_ats_on_testcase_failure(stop = True)
```

TestAdapterConfiguration Objects

Constructor:

```
TestAdapterConfiguration(name)
```

Methods:

```
bind(tsiPort, uri, type_, **kwargs)
```

Test-unrelated

Functions:

```
octetstring(s)
```

StateManager Objects

Constructor:

```
StateManager(self, state = None)
```

Methods:

```
get()
set(state)
```

Best Practices

Naming Conventions

ATS Naming Conventions

Testerman adapts the TTCN-3 naming conventions[19] to its context:

[19] http://www.ttcn-3.org/NamingConventions.htm

Language element	Naming convention	Prefix	Example	Comments
message template	mixedCase	m_	m_sipInvite	These templates are both usable as sending and receiving templates. This naming convention applies to both templates functions and templates constants.
message template with wildcard or matching expression	mixedCase	mw_	mw_sipInviteResponse	These templates are (typically) only valid as receiving templates, as wilcards and maching mechanisms may prevent from valuating them when sending them. This naming convention applies to both templates functions and templates constants.
port	mixedCase	(none)	ssh, rtp01, sipSignalling	
port name	mixedCase	(none)	system['iscInterface'], mtc['sipSignalling']	
test component	mixedCase	(none)	endpoint01, hlrSimulator	
function	mixedCase	f_	f_executeCommand	
timer	mixedCase	t_	t_watchogTimer	
ATS parameter	UPPER_CASE	PX_	PX_HOST_IP_PORT	The QTesterman client enforces this name convention when defining new ATS parameters.
test case (class name)	UPPER_CASE	TC_	TC_ORACLE_CONNECTION_001	The class name is also used as a test case identifier in log reporters
behaviour (class name)	CamelCase	(none)	SipEndpoint, HlrSimulator	
constant	mixedCase	c_	c_suaAspActive = 0x04	
parameters (test case, behaviour, functions, ...)	mixedCase	(none)	controlPort, autoConnectSctp	
preamble (class name)	UPPER_CASE	PRE_	PRE_VOICEMAIL_INITIALIZATION	
postamble (class name)	UPPER_CASE	POST_	POST_VOICEMAIL_CLEANUP	

This naming convention is suggested to make a difference between what belongs to the userland (user-written) and what is part of the Testerman API availablt to the user, which uses a `lower_case` convention.

Paths and Filenames

Some rules on filenames:

- Use ascii characters and lower case only for ATS and Campaign filenames, and _ as a word separator: `my_sample.ats`, `unconditional_call_forward.ats`, `class5_services.campaign`, ...
- Use ascii characters and CamelCase (with initial and acronyms considered as a word) for module filenames: `SipMessages.py`, `DiameterServerBehaviours.py`, ...

- Use lower case only for directory names, and _ as a word separator. In particular, do NOT use a dot (`.`) in a directory name. This would prevent modules from being imported correctly.

Additionally, the following repository structure is advised:

- `/` : no files in the repository root
- `/samples/`: contains samples (either Testerman's provided, or provided by your teams - in this case creating subdirectories in it could be a good idea)
- `/sandbox/`: a place where users can play and save their test documents, without working in a test project context yet (feasibility tests, demo, troubleshooting, ...)
- `/modules/`: user created or site-related shared modules, available to all (SIP endpoint behaviours, convenience functions according to your organization's use of Testerman, ...)
- `/...`: any other directory, according to your organization, projects, teams, etc. Mapping this subtree to a subtree in an external test system could be a good idea, too.

Reusability

This section suggests several best practices to maximize the reusability of the objects you design.

Testcases

...

Behaviours

...

Templates

...

Functions

...

Simple Functions

...

Functions Involving Ports

...

ATS

Control Part

...

ATS Parameters (Variables)

...

Module

...

CHAPTER

THREE

CODECS AND PROBES

Codecs and Probes Overview

Testerman comes with a collection of codecs and probes to interact with your SUTs.

A *probe* (a.k.a. a Test Adapter or SUT Adapter in TTCN-3 terminology) is the entity that is responsible for actually implementing the interaction with the SUT.

It could be reading and writing packets on the wire, reading/writing a file somewhere, etc, and is usually configurable to provision actual SUT IP addresses or port or other SUT-dependent and test logic-independent parameters.

The probe is what is behind a test system interface port, and both are associated through what Testerman calls a *binding*. The binding is also the moment when you can set those SUT-dependent parameters for the probe: they are called probe *properties*.

A *codec* is something different.

It does not perform any SUT adaptation, but only provides a way to encode/decode high level, structured userland messages to/from a lower level payload. Codecs can be stacked so that it is trivial to build things such as transporting a MAP message encoded in base64 in a XML element sent over UDP to a remote target; in this dummy example, we would stack MAP, base64, and XML codecs to build a full payload that would be sent through a UDP probe, actually injecting raw data over the network to a SUT address that could be defined as a probe property (when not test-dependent).

Codecs

IT Oriented Codecs

Codec ID	Description
`xml`	a simple XML coder/decoder - you may also have a look to CodecXerLite
`xer.lite`	CodecXerLite, a XER (XML Encoding Rules)-like codec that only uses a subset of XML capabilities (no attributes)
`http`	CodecHttp, HTTP request/response codecs
`soap11.ds`	CodecSoapDs, a codec that enables to sign (and verify signatures of) SOAP 1.1 message, using Ws-Security standards.

Multimedia and VoIP Oriented Codecs

Codec ID	Description
`rtsp.*`	RTSP Codecs
`sdp`	CodecSdp, a RFC4566 SDP (Session Description Protocol) codec

Telecom Oriented Codecs

Codec ID	Description
`sua`	CodecSua, a RFC3868 SUA (SCCP User Adaptation) codec
`tcap`	CodecTcap, a TCAP (ITU) codec
`map.*`	CodecMap, a collection of codecs for MAP Phase 2+ PDUs
`sms.tpdu.*`	CodecSmsTpdu, a collection of codecs for SMS TPDU SM-TL layer (GSM 030.40)
`tbcd`	A telephony BCD codec
`packed.7bit`	A codec for 7-bit packed strings (useful for SMS encoding)

Test Adapters (Probes)

Protocol Oriented Probes

These probes are useful to test a SUT at protocol level.

- General purpose probes: transport over IP probes: TCP Probe, UDP Probe, SCTP Probe (could be used as a base for implementing SIP testing, SUA, HTTP, ... most protocols)
- HTTP Probe, a simple, dummy HTTP client probe without any automated behaviours (no redirect following, etc)
- RTSP Probe, a simple RTSP client probe
- RTP Probe, a RTP stream sender/listener

Tools Probes

These probes are mainly useful to interfact with a SUT at high-level. They are mainly convenience tools to develop the glue between several domain testing, and are meant to integrate usual actions associated with testing: connecting to remote machines, checking files, running commands, changing configuration files, checking an SQL database, etc.

- SSH Probe, a probe to execute non-interactive commands remotely through SSH, ProbeExec to execute them locally, ProbeExecInteractive to execute them locally, but interactively
- SQL connector probes: MySQL Probe, Oracle Probe
- Directory Watcher Probe, a probe that monitors a directory and notifies you when an entry has been added/removed (useful to check if a lock file was created/removed, ...)
- File Watcher Probe, a probe that monitors an ascii file and notifies you of new lines (think of it as a combination of `tail -f` and `grep`, useful to check log files for instance)
- Configuration File Probe, a probe that can access in read-write to configuration files, supporting most used configuration formats and extensible to match your needs (useful to prepare a SUT for testing in particular configuration conditions)
- `autogen/ProbeXmlRpc`, a probe to invoke remote operations via XML-RPC
- LDAP Probe, a probe to access a LDAP directory
- Selenium Probe, a probe to execute web-oriented tests through Selenium RC
- File Manager Probe, a probe to create, move, delete files dynamically (convenient to create temporary files, to inject resources into the SUT that are kept embedded in the ATS that you don't have to manage additional dependencies at runtime)

Codecs and Probes References

This section provides the reference documentation for each codec and probe.

XML Codec

Encode/decode XML payloads.

Identification and Properties

Codec ID: `xml`

Properties:

Name	Type	Default value	Description
`encoding`	string	`utf-8`	encoding: use this to encode the payload. Decoding: assumes the payload follows this encoding if the xml prolog encoding attribute is missing.
`write_prolog`	boolean	`True`	encoding: if True, write the `<?xml version="1.0" encoding="..."?>` prolog.
`prettyprint`	boolean	`False`	encoding: if True, pretty print the XML (carriage returns, hard tabs)

Overview

This codec enables to encode/decode XML strings from/to Testerman message structures.

Limitations:

- Comments, entity nodes are not decoded (nor can be encoded)
- Nested elements are only reported if not preceded by a text. In particular, this makes this codec unsuitable for full XHTML parsing, but should make it more convenient in most testing cases

Something like

```
<p>In this text, <i>some elements</i> are nested</p>
```

will be decoded to:

```
('p', { 'value': u"In this text, <i>some elements</i> are nested" })
```

I.e. the nested element won't be constructed; whereas:

```
<p><i>nested element</i></p>
```

will be decoded to

```
('p', { 'children': [ ('i', 'value': u"nested element") ] })
```

If you don't need to manage attributes and CDATA sections, you may consider CodecXerLite.

Decoding

Input:

```
<element attr="attrval">
  <subelement>subvalue</subelement>
  <subelement2><![CDATA[cdata value]]></subelement2>
</element>
```

Decoded message:

```
('element', { 'attributes': { 'attr': 'attrval' }, 'children': [
  ('subelement', { 'value': 'subvalue', 'cdata': False }),
    ('subelement2', { 'value': 'cdata value', 'cdata': True }),
  ]}
)
```

`cdata` is always provided during decoding.

Encoding

Input:

```
('element', { 'attributes': { 'attr': 'attrval' }, 'children': [
  ('subelement', { 'value': 'subvalue' }),
  ('subelement2', { 'value': 'cdata value', 'cdata': True }),
]})
```

Encoded message:

```
<element attr="attrval">
  <subelement>subvalue</subelement>
  <subelement2><![CDATA[cdata value]]></subelement2>
</element>
```

If `cdata` is not present during a value element encoding, it is assumed to be `False`.

If both `children` and `value` are provided, `children` has a greater precedence and `value` will be ignored.

TTCN-3 Types Equivalence

```
      type record Element
      {
      Attributes attributes,
      // if children is present, cdata and value are not present.
      // if cdata and value are present, children it not present.
      boolean cdata optional,
      universal charstring value optional,
      record of Element children optional
      }

type record Attributes
{
  // field name depends on the available attributes. All types are universal charstring
  universal charstring name
}
```

RTSP Codecs

Encode/decode RTSP requests and responses.

Identification and Properties

Codec ID: `rtsp.request` and `rtsp.response`

Properties:

name	type	default value	description
`lower_case`	boolean	`False`	decoding: transforms header names to lower case, making header name matching easier (when performing non-protocol oriented testing)
`version`	string	`RTSP/1.0`	encoding: the version to set in the request or response line, if not provided by the user

Overview

These two codecs encode/decode RTSP requests and responses respectively from/to the following TTCN-3 equivalence types (optional fields are always provided when decoding), as provided below.

During encoding, they automatically compute the content-length if not provided as a header, and add it only if a body is present.

Content-Length verification is also performed on decoding automatically. If the length is not valid, the decoding attempt fails.

These codecs are usually used with the UDP probe to implement solutions that can test an RTSP implementation.

Availability

All platforms.

Dependencies

None.

TTCN-3 Types Equivalence

```
type record RtspRequest
{
  charstring method,
  charstring uri,
  charstring version optional, // default: RTSP/1.0
  record { charstring <header name>* } headers optional, // default: {}
  charstring body optional, // default: ''
}

type record RtspResponse
{
  charstring version optional, // default: RSTP/1.0
  integer status,
  charstring reason,
  record { charstring <header name>* } headers optional, // default: {}
  charstring body optional, // default: ''
}
```

TCP Probe

Send/receive TCP payloads.

Identification and Properties

Probe Type ID: `tcp`

Properties:

Name	Type	Default value	Description
`local_ip`	string	(empty - system assigned)	Local IP address to use when sending packets
`local_port`	integer	`0` (system assigned)	Local port to use when sending packets
`listening_ip`	string	`0.0.0.0`	Listening IP address, if listening mode is activated (see below)
`listening_port`	integer	`0`	Set it to a non-zero port to start listening on mapping
`size`	integer	`0`	Fixed-size packet strategy: if set to non-zero, only raises messages when `size` bytes have been received. All raised messages will hage this constant size.
`separator`	string	`None`	Separator-based packet strategy: if set to a character or a string, only raises messages when `separator` has been encountered; this separator is assumed to be a packet separator, and is not included in the raised message. May be useful for, for instance, x00-based packet protocols.
`enable_notifications`	boolean	`False`	If set, you may get connection/disconnection notification and connectionConfirm/Error notification messages
`default_sut_address`	string (ip:port)	`None`	If set, used as a default SUT address if none provided by the user
`default_decoding`	string	`None`	If set, must be a valid codec name (aliases are currently not supported). This codec is then used to decode all incoming packets, and only the probe only raises an incoming message when the codec successfully decoded something. This is particular convenient when used with an incremental codec (such as `'http.request'`) that will then be responsible for identifying the actual application PDU in the TCP stream.
`default_encoding`	string	`None`	If set, must be a valid codec name (aliases are currently not supported). This codec is then used to encode all outgoing packets, without a need to use it when sending the message through the port mapped to this probe.
`use_ssl`	boolean	`False`	If set, all outgoing and incoming traffic through is probe is transported over SSLv3. All TLS negotiations are performed by the probe. However, ...
`ssl_key`	string	`None`	The SSL key to use if `use_ssl` is set to `True`. Contains a private key associated to `ssl_certificate`, in base64 format. If not provided, a default sample private key is used.
`ssl_certificate`	string	`None`	The SSL certificate to use if `use_ssl` is set to `True`. Contains a certificate in PEM format that will be used when a certificate is needed by the probe connection(s). If not provided, a default one that matches the default private key, is used.
`connection_timeout`	float	`5.0`	The connection timeout, in s, when trying to connect to a remote party.
`auto_connect`	boolean	`True`	When sending a message, autoconnect to the provided address if there is no existing connections with this peer yet.
`ssl_require_client_cert`	string	`None`	When `use_ssl` is set to `True` and the probe is used on server side (`listening_port` > 0), request the SSL client to provide a client-side certificate issued by one of the CA whose certificate is provided as base64 in this property. This enables to test mutual SSL authentication.
`auto_connect`	boolean	`True`	When sending a message, autoconnect to the provided address if there is no existing connections with this peer yet.

Overview

This is a general purpose probe to transport anything over TCP, with basic control on connections/disconnections (you can get optional incoming connection notifications or outgoing connection confirmations, or simply focus on payload exchanges), and a basic support for SSL (v3).

Such a probe may be used as a base to test any protocol transported over TCP.

Combined with the `http.request` and `http.response` codecs, this is enough to test anything based on HTTP/HTTPS. You may also use the `diameter` or `sua` codec, actually any codec that comes with an incremental decoding implementation. You just have to define such a codec as the `default_decoder` property (used to decode incoming stream) or the `default_encoder` property (used to encode outgoing messages).

ADPU Identification

The probe first waits for `size` bytes (if the `size` property is set) or (exclusively) for the `separator` character(s) (is the `separator` property is set). If none of those properties are set, the probe only considers what it read in the stream (which is system-dependent).

Then, the default decoder, if set, tries to decode this first raw segment. If it needs more input, it waits for the next raw segment. If multiple APDUs are detected, multiple incoming messages are raised. If undecodable data is detected, the raw segment is ignored.

If no decoder is set, the raw segment is raised as raw data.

Basic SSL Support

When the property `use_ssl` is set to True, the probe automatically performs SSL negotiations after a TCP connection (probe as a client) or when accepting a new incoming connection (as a server).

If `enable_notifications` is True, the `connectionConfirm` message will contain the server's certificate in DER format. The `connectionNotification` message is planned to contain the client's certificate as well, but it is currently not possible to force the (server side) probe to request it.

In addition, received certificates are not validated, and hostnames are not verified.

This probe offers very little control on these negotiations and is not meant to test SSL-level stuff (for instance, how a SUT implemented SSL itself). This support is provided as a convenience to interact with a SUT through higher-level protocols that have been ported over SSL (HTTP, SIP, ...).

When using this probe as a server in SSL mode, if you don't provide the `ssl_key` and `ssl_certificate` parameters, a default pair is used. The default certificate is:

```
-----BEGIN CERTIFICATE-----
MIICPTCCAaYCCQDP2Zlsj6TqTDANBgkqhkiG9w0BAQUFADBjMQswCQYDVQQGEwJG
UjERMA8GA1UECBMIR3Jlbm9ibGUxDzANBgNVBAcTBk1leWxhbjESMBAGA1UEChMJ
VGVzdGVybWFuMRwwGgYDVQQDExNzYW1wbGUudGVzdGVybWFuLmZyMB4XDTEwMDcx
MzE1NTMwNloXDTIwMDcxMDE1NTMwNlowYzELMAkGA1UEBhMCRlIxETAPBgNVBAgT
CEdyZW5vYmxlMQ8wDQYDVQQHEwZNZXlsYW4xEjAQBgNVBAoTCVRlc3Rlcm1hbjEc
MBoGA1UEAxMTc2FtcGxlLnRlc3Rlcm1hbi5mcjCBnzANBgkqhkiG9w0BAQEFAAOB
jQAwgYkCgYEA4scEJ3BC7WcWHYHM0Z7O5rRz5W4231OfKoOCFROoV9QWzqGhvkpJ
IOLUVkV9SEc9tZICUdHRnmTMacwRG75YyUV8wg4w1EqVyIysRVyQySXOr8YzjZ5N
BJ924nw3lwk7pms2EGIBAiTmrUdEXYD97qrzaOSER3GMUmZWaSwFGf0CAwEAATAN
BgkqhkiG9w0BAQUFAAOBgQDET13MT8ctRiuQkfCLO8D9iyKjT94FRe+ocPbUjOts
gjkrh5miH91MhabQrqEcwOArF+2yuakvfgdPP3KbHgBXDwtYg/+wqAj4e/74D6Ai
Ud0h6vHFVZreLZm7F1QNLpgUSrPxra2xkTiH7NxEMYlJheGCnJ4F6YP3IdKMUicZ
Og==
-----END CERTIFICATE-----
```

Availability

All platforms. However, SSL support depends on the Python SSL module, provided by default with Python 2.6 and later on Unix platforms.

Dependencies

None.

See Also

Other transport-oriented probes:

- SCTP Probe
- UDP Probe

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `TransportProbePortType` port type as specified below:

```
type union NotificationType
{
  record { octetstring certificate optional } connectionNotification, // new incoming connection e
  charstring disconnectionNotification, // contains a human readable reason to the disconnection
  record { octetstring certificate optional } connectionConfirm, // connection request OK
  charstring connectionError, // contains a human readable error after a connection request
}

type union RequestType
{
  any connectionRequest, // request a new tcp-connection
  any disconnectionRequest, // request a disconnection. Except a disconnectionNotification later
}

type TransportProbePortType
{
  in RequestType;
  out NotificationType;
  in, out octetstring;
  out any; // if the default_decoder is used, the raised structure is the decoder's output
  in any; // if the default_encoder is used
}
```

UDP Probe

Send/receive UDP packets.

Identification and Properties

Probe Type ID: `udp`

Properties:

Name	Type	Default value	Description
`local_ip`	string	(empty - system assigned)	Local IP address to use when sending packets
`local_port`	integer	0 (system assigned)	Local port to use when sending packets
`listen_on_send`	boolean	True	Once something has been sent (from `local_ip:local_port`), keep listening for a possible response on this address. Only stops listening on unmapping. When set to False, immediately closes the sending socket once the packet has been sent.
`listening_ip`	string	0.0.0.0	Listening IP address, if listening mode is activated (see below)
`listening_port`	integer	0	Set it to a non-zero port to start listening on mapping. May be the same ip/port as `local_ip:local_port`. In this case, `listen_on_send` is meaningless.
`default_sut_address`	string (ip:port)	None	If set, used as a default SUT address if none provided by the user

Overview

This probe is a very simple adapter that enables to send and receive UDP packets on the network.

It can work as a listening probe, raising an event/message each time a new UDP packet is received on its `listening_ip:listening_port` address, or send an UDP packet to either `default_sut_address` or to an explicit sutAddress if one was provided in the `send()` port command.

When sending a packet, the UDP source address is set to `local_ip:local_port`.

Listening and sending modes can work in parallel on the same probe.

Availability

All platforms.

Dependencies

None.

See Also

Other transport-oriented probes:

- SCTP Probe
- TCP Probe

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `TransportProbePortType` port type as specified below:

```
type TransportProbePortType
{
  in, out octetstring;
}
```

SCTP Probe

Send/receive SCTP payloads.

Identification and Properties

Probe Type ID: `sctp`

Properties:

Name	Type	Default value	Description
`local_ip`	string	(empty - system assigned)	Local IP address to use when sending packets
`local_port`	integer	`0` (system assigned)	Local port to use when sending packets
`listening_ip`	string	`0.0.0.0`	Listening IP address, if listening mode is activated (see below)
`listening_port`	integer	`0`	Set it to a non-zero port to start listening on mapping
`style`	string in `'tcp'`, `'udp'`	`'tcp'`	SCTP style: UDP or TCP (stream)
`enable_notifications`	boolean	`False`	If set, you may get connection/disconnection notification and connectionConfirm/Error notification messages
`default_sut_address`	string (ip:port)	`None`	If set, used as a default SUT address if none provided by the user

Overview

This probe was used to implement SUA-based testing with a TCPA/MAP stack to simulate HLRs.

Availability

All platforms.

Dependencies

None.

See Also

Other transport-oriented probes:

- TCP Probe
- UDP Probe

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `TransportProbePortType` port type as specified below:

```
type union NotificationType
{
  record {} connectionNotification, // new incoming connection established
  charstring disconnectionNotification, // contains a human readable reason to the disconnection
  record {} connectionConfirm, // connection request OK
  charstring connectionError, // contains a human readable error after a connection request
}

type union RequestType
{
  any connectionRequest, // request a new tcp-connection
  any disconnectionRequest, // request a disconnection. Except a disconnectionNotification later
}

type TransportProbePortType
{
  in RequestType;
  out NotificationType;
  in, out octetstring;
}
```

Directory Watcher Probe

Watch for new/deleted files in a directory.

Identification and Properties

Probe Type ID: `watcher.dir`

Properties:

None.

Overview

This probe watches one or more directories locally and sends notifications whenever a new entry whose name matches a pattern is created or removed in one of them.

You should first send a startWatchingDirs command, specifying the directories to monitor (absolute paths, wildcards accepted) and an optional list of regular expression patterns the interesting entry names should match. These patterns may contain named group, such as in `r'errorlog_(?P<number>[0-9+])\.log'`. In this example, if a file (or directory or link) matching this pattern is created or removed in one of the watched dirs, you will receive a notification containing the watched dir, the complete entry name that matched the pattern, and an additional `matched_number` string entry containing the matched group.

For instance, if you're watching `['/tmp', '/var/lock']` with the patterns `[r'(?P<application>[a-z]+)_(?P<number>[0-9+])\.log', r'(?P<application>[a-z]+).lock']`, as soon as the file `'/tmp/testerman_1.log'` is created, you should expect a Notification message such as:

```
('added', {'dir': '/tmp', 'name': 'testerman_1.log', 'mached_application': 'testerman', 'matched_
```

When the file `'/var/lock/testerman.lock'` is removed, expect:

```
('removed', {'dir': '/var/lock', 'name': 'testerman.lock', 'mached_application': 'testerman'})
```

On start watching, the probe checks for changes in the monitored dirss each second (by default). The interval between two checks can be configured via the `interval` startWatchingFiles field. The probe is aware of reset/recreated or new born dirs (when monitoring a dir that has not been created yet). Be aware that you may miss notifications if some files are created/deleted faster than the interval allows to detect.

When you do not need to watch these dirs any more, send a stopWatchingDirs command.

The probe automatically stops watching dirs on unmap and when the current test case is over.

If you send a startWatchingDirs command while the probe is already in watching mode, the monitoring is restarted with the new watching parameters.

The `patterns` startWatchingDirs field may contain several regular expression. Only the first one that matches new lines in watched files is used to generate `matched_*` notification fields.

Possible use cases for this probe:

- Checking file rotation (the oldest should be removed, a new one should be added)
- Checking lock files (created on application start, deleted when stopped, etc)

Availability

All platforms.

Dependencies

None.

See Also

- File Watcher Probe, a probe that watch a directory for new/removed files

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the *DirWatcherPortType* port type as specified below:

```
type union WatchingCommand
{
  StartWatchingDir startWatchingDirs,
  anytype           stopWatchingDirs
}

type record StartWatchingDirs
{
  record of charstring dirs,
  record of charstring patterns optional, // defaulted to [ '.*' ]
  float interval optional, // defaulted to 1.0, in second
}

type union Notification
{
  DirNotification added, // entry added
  DirNotification removed, // entry removed
}

type record DirNotification
{
  charstring dir,
  charstring name, // the matched added or removed entry in dir
  charstring matched_* optional, // matched groups, if defined in patterns
}

type port DirWatcherPortType message
```

```
{
  in  WatchingCommand;
  out Notification;
}
```

File Watcher Probe

Watch for changes in files.

Identification and Properties

Probe Type ID: `watcher.file`

Properties:

None.

Overview

This probe watches one or more text files locally and sends notifications whenever a line matching a pattern appeared in one of them.

You should first send a startWatchingFiles command, specifying the files to monitor (absolute paths, wildcards accepted) and an optional list of regular expression patterns to match. These patterns may contain named group, such as in `r'resource (?P<name>.*) played'`. In this example, if a line matching this pattern is detected in one of the watched files, you will receive a notification containing the source file filename, the complete line that matched the pattern, and an additional `matched_name` string entry containing the matched group.

On start watching, the probe checks for new lines in the monitored files each second (by default). The interval between two checks can be configured via the `interval` startWatchingFiles field. The probe is aware of reset/recreated or new born files (when monitoring a file that has not been created yet). In case of a file reset, you may miss some matching lines if new lines are created and the file is reset before the next file check, but this should not be a show-stopper considering the typical use cases for this probe.

When you do not need to watch these files anymore, send a stopWatchingFiles command.

The probe automatically stops watching files on unmap and when the current test case is over.

If you send a startWatchingFiles command while the probe is already in watching mode, the monitoring is restarted with the new watching parameters.

The `patterns` startWatchingFiles field may contain several regular expression. Only the first one that matches new lines in watched files is used to generate `matched_*` notification fields.

WARNING: in most cases, you should set a pattern list that avoids raising a notification for each new line in watched files. This is particularly true when watching application log files that may generate hundred lines per second, raising as many notifications to the Testerman Test Executable, saturating the whole system.

Possible use cases for this probe:

- Checking a log file, verifying that the application logs what we expect according to external stimuli
- Using a log file to check the application behaviour; for instance, when testing an IVR (Interactive Voice Response) server, it could be convenient to check in log files the file that is assumed to be played instead of using voice recognition or RTP pattern analysis.
- Applied to telecom systems, could be used to check CDR (Call Detail Reports) files, possibly in conjunction with a custom codec to apply to the notified lines
- Checking that no error is dumped in a log file during a test. In this case, the probe is probably controlled by a background, dedicated behaviour, setting its verdict to fail as soon as it received an error line notification.

Known Bugs

This probe may not detect a file recreation on some file systems such as ext3, in the following case:

- the file is recreated/replaced with a larger file than the initial one,
- AND the file's inode is not changed (which is the case with ext3, apparently).

In this case, only the delta lines between the old file and the new ones are reported, instead of reporting all the lines of the file.

Availability

All platforms.

Dependencies

None.

See Also

- Directory Watcher Probe, a probe that watches a directory for added/removed entries.

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the *FileWatcherPortType* port type as specified below:

```
type union WatchingCommand
{
  StartWatchingFile startWatchingFiles,
  anytype           stopWatchingFiles
}

type record StartWatchingFiles
{
  record of charstring files,
  record of charstring patterns optional, // defaulted to [ '.*' ]
  float interval optional, // defaulted to 1.0, in second
}

type record Notification
{
  charstring filename,
  charstring line, // the matched line
  charstring matched_* optional, // matched groups, if defined in patterns
}

type port FileWatcherPortType message
{
  in  WatchingCommand;
  out Notification;
}
```

Local Execution Probe

Execute a command in a local shell.

Identification and Properties

Probe Type ID: `exec`

Properties:

Name	Type	Default value	Description
`shell`	string	`None`	The shell to use when executing the command line. On Unixes, this is defaulted to `/bin/sh`, on Windows, this is the shell as specified via the COMSPEC environment variable.

Overview

This probe implements a single shot command execution interface (the same as ProbeSsh) for locally executed commands.

Basically you just specify a command to execute that will be executed within a shell, and you get a response once its execution is over. The response contains both an integer return code and the whole command output.

If you consider the command execution is too long (no response received), you can cancel it at any time from the userland. Such a cancellation terminates all the subprocess tree with a SIGKILL signal on POSIX platforms, and a SIGTERM to the started process (not all its subtree) on Windows. Once cancelled, you should not expect a command response anymore.

No interaction is possible during the command execution.

Notes:

- when starting daemons from this probe, make sure that your daemon correctly closes standard output, otherwise the probe never detects the command as being complete.

Availability

All platforms. Tested on Linux (kernel 2.6), Solaris 8, Solaris 10.

Dependencies

None.

See Also

- SSH Probe, implementing the same port type for execution through SSH (avoiding the installation of an agent on the target machine)
- Local Interactive Execution Probe, to run a command line program and interact with it (CLI testing, etc)

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `ExecPortType` port type as specified below:

```
type union ExecCommand
{
  charstring execute,
  anytype    cancel
}
```

```
type record ExecResponse
{
  integer status,
  charstring output
}

type charstring ErrorResponse;

type port ExecPortType message
{
  in  ExecCommand;
  out ExecResponse, ErrorResponse;
}
```

Local Interactive Execution Probe

Execute a command in a local shell and interact with it.

Identification and Properties

Probe Type ID: `exec.interactive`

Properties:

Name	Type	Default value	Description
`separator`	string	`None`	Should we notify only complete lines (or whatever) based on this separator ?
`timeout`	float	`0.5`	Maximum amount of time to wait for new data before notifying of it, when no separator is used
`encoding`	string	`'utf-8'`	The encoding to use to turn the output into unicode (also used to encode data sent to the started process)

Overview

This probe enables to execute a shell-based program interactively, i.e. start a program, be notified of what it outputs to standard output (stdout, stderr), send it some input or signals.

Output can be notified on a per-line basis (or any separator - the separator is not included in the event sent back to the userland) or a timeout basis (useful if the program does not outputs lines only, for instance prompts).

This probe does not require the started program to use unbuffered stdout.

Typical use cases include:

- CLI testing
- Program output watching, for instance something that can dump real-time traces to stdout, avoiding the use of combined `exec` (that executes a blocking program whose output is redirected to a temp file) and `file.watcher` (that watches the temp file created by the program we should watch the output from) probes.

Basically you just specify a command to start that will be executed within a shell, providing an optional list of regular expressions the output must match before being sent to userland (a `egrep` equivalent).[[BR]] Whenever you output to stdout/stderr is detected, based on `separator` and `timeout` properties, it checks that it matches at least one of the regular expressions provided above before enqueing a `OutputNotification` message to userland. If the matched regular expression contains named groups, corresponding fields are automatically added in this message (as for the ProbeFileWatcher).

At any time, you may send some input to the started process using the `input` choice of the `InteractiveExecCommand` message (don't forget possible trailing carriage returns, as they are not sent automatically) or send a signal using its POSIX integer value and the `signal` choice of the `InteractiveExecCommand` message.

When the process is over (for whatever reason), a `ExecStatus` message is enqueued, containing the execution status as a shell executing the command would have provided.

Limitations:

- stderr output is currently reported within the stdout stream
- interleaved stdout/stderr output are not garanteed to be delivered in the correct order

Availability

All POSIX platforms. Windows platforms are not supported.

Dependencies

None.

See Also

- Local Execution Probe, a single shoot command, non-interactive execution.

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `InteractiveExecPortType` port type as specified below:

```
type union InteractiveExecCommand
{
  StartCommand start,
  universal charstring input,
  integer signal,
}

type record StartCommand
{
  charstring command,
  record of charstring patterns optional, // defaulted to [ r'.*' ]
}

type record OutputNotification
{
  universal charstring output,
  universal charstring matched_*,
  charstring stream, // 'stderr' or 'stdout'
}

type record ExecStatus
{
  integer status,
}

type port InteractiveExecPortType message
{
  in  InteractiveExecCommand;
```

```
  out ExecStatus, OutputNotification;
}
```

File Manager Probe

Create, move/delete/rename files and links.

Identification and Properties

Probe Type ID: `file.manager`

Properties:

None.

Overview

This probe performs basic local file management operations:

- file creation (with content injection)
- file move, deletion, renaming
- link creation, deletion
- file content read (the content is returned as a result choice, set to None if nothing can be read)

One of its primary purposes is providing the ability to create files on the fly from resources that were embedded in your ATS or its dependencies. This way, you may have no additional requirements on the SUT: you can embed some reference files directly into the Testerman userland instead of instructing the user to get and copy files from another source.

Availability

All platforms.

Dependencies

None.

See Also

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `FileManagerPortType` port type as specified below:

```
type union FileManagementCommand
{
  CreateFileCommand createFile,
  CreateLinkCommand createLink,
  RemoveCommand remove,
  MoveCommand move,
  CopyCommand copy,
  ReadCommand read,
}
```

```
type record CreateFileCommand
{
  universal charstring name,
  octetstring content optional, // defaulted to an empty content
  boolean autorevert optional,  // backup existing file, restore it on unmap, defaulted to False
}

type record RemoveCommand
{
  universal charstring path,
}

type record MoveCommand
{
  universal charstring source,
  universal charstring destination,
}

type record CopyCommand
{
  universal charstring source,
  universal charstring destination,
}

type record CreateLinkCommand
{
  universal charstring name,
  universal charstring target,
  boolean autorevert optional, // backup existing file, restore it on unmap, defaulted to False
}

type record ReadCommand
{
  universal charstring path,
}

type record CommandStatus
{
  integer status,
  universal charstring errorMessage optional, // only if status > 0
}

type union FileManagementResponse
{
  CommandStatus status,
  universal charstring result, // for read command, null if the file could not be read
  // more to come: fileExists response, fileType response, ...
}

type port FileManagerPortType message
{
  in  FileManagementCommand;
  out FileManagementResponse;
}
```

SSH Probe

Execute a command via a remote SSH shell.

Identification and Properties

Probe Type ID: `ssh`

Properties:

Name	Type	Default value	Description.
`host`	string	`'localhost'`	the host to connect onto to execute the commands.
`username`	string	(none)	the username to use to log onto `host` to execute the commands.
`password`	string	(none)	the `username`'s password on `host`.
`timeout`	float	`5.0`	the maximum amount of time, in s, allowed to __start__ executing the command on `host`. Includes the SSH login sequence.
`convert_eol`	boolean	`True`	if set to True, convert `\r\n` in output to `\n`. This way, the templates are compatible with ProbeExec.
`working_dir`	string	(none)	the diretory to go to before executing the command line. By default, the working dir is the login directory (usually the home dir).
`strict_hostkey`	boolean	`True`	if set to False, the probe removes the target host from $HOME/.ssh/known_hosts to avoid failing when connecting to an updated host. Otherwise, the connection fails if the SSH key changed.
`max_line_length`	integer	`150`	the max number of characters before splitting a line over multiple lines with a -based continuation. Currently the splitting algorithm is pretty dumb and may split your command line in the middle of a quoted argument, possibly changing its actual value. Increasing this size may be a workaround in such cases.

Overview

This probe implements a single shot command execution interface (the same as ProbeExec) through SSH.

Basically you just specify a command to execute that will be executed within a shell, and you get a response once its execution is over. The response contains both an integer return code and the whole command output.

If you consider the command execution is too long (no response received), you can cancel it at any time from the userland. Such a cancellation terminates all the subprocess tree with a SIGKILL signal. Once cancelled, you should not expect a command response anymore.

No interaction is possible during the command execution.

Like ProbeExec, this probe is not re-entrant and is only able to execute one command at a time. You can execute another one (using the `ExecCommand execute` message template) only when the previous command execution was either complete (i.e. you received a `ExecResponse` message) or cancelled via a `ExecCommand cancel` command (no response in this case).

If you need to execute multiple commands in parallel, you should use multiple probes - consider that each one is as if you had one open ssh terminal connection to your SUT.

Notes:

- When starting daemons from this probe, make sure that your daemon correctly closes standard output, otherwise the probe never detects the command as being complete. For poorly written daemonized programs, adding a `>/dev/null 2>&1` in the `execute` command line is usually enough to make the probe return in such cases.

Availability

All POSIX platforms. Windows platforms are not supported.

Dependencies

Requires a ssh client installed on the machine running the probe, as it is only a wrapper over it.

See Also

- Local Execution Probe, implementing the same port type for local execution (convenient when you have no SSH access to your machine, but you must deploy an agent on it)
- Local Interactive Execution Probe, to run a command line program and interact with it (CLI testing, etc)

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `ExecPortType` port type as specified below:

```
type union ExecCommand
{
  charstring execute,
  anytype    cancel
}

type record ExecResponse
{
  integer status,
  charstring output
}

type charstring ErrorResponse;

type port ExecPortType message
{
  in  ExecCommand;
  out ExecResponse, ErrorResponse;
}
```

Configuration File Probe

Get or set a key in a configuration file.

Identification and Properties

Probe Type ID: `configurationfile`

Properties:

Name	Type	Default value	Description
`plugin_properties`	dict[string] of dict	`{}`	properties to pass to sub-plugins, indexed by the format they support

Properties for the `ini` plugin:

Name	Type	Default value	Description
`comments`	list of strings	`['#', ';']`	List of characters or strings that identify a comment in a .ini file

There are no properties for the other `xml` and `conf` plugins.

Overview

This probe is specialized in updating configuration files, so that you can easily integrate small configuration changes prior to running a particular test (and rollback this change after).

In can only work with files located on the system the probe is running on (i.e. you cannot update a file remotely via this probe, but you can still deploy your probe on a remote agent, so technically you can still update a configuration file anywhere on your distributed SUT).

This implementation relies on probe-specific plugins (yes, nested plugins ! where this over-engineering spree will end ?) to update the files. These plugins simply provide `get` and `set` operations for a specific key, and the way the key is identified is plugin-specific.

Currently the following configuration file formats are supported:

- XML files (extension: `.xml`): the key format is a XPath request, so you can change an element value or attribute.
- Shell sourceable configuration files (extension: `.conf`): the typical key=value configuration file that can be read by `sh`, using `#` for comments, and a flat structure (no sections).
- INI files (extension: `.ini`): Windows-like configuration files, with sections. The key format is "section/key". The characters used for comments in such a file are configurable (see the `comments` property) so that it could adapt to some custom INI file formats.

The plugin to use to handle a configuration file is automatically detected based on the file extension, unless you force one (which is convenient to support INI files named `something.conf`).

To get the value of a key in a file (here, the value of the key name `key` in section `[section]` in the INI configuration file `/pat/to/my/config.ini`), simply send the following message:

Then, expect a result message:

If the key or the configuration file was not found or not readable, the returned value is an empty string.

To set the previous key to a given value:

Then, optionally expect a result message (if you don't mind if the operation succeeded, don't expect anything):

Other received messages indicate a set error.

As usual, it may be convenient to embed these message-based calls into a convenience function to use in your ATSes.

Availability

All platforms.

Dependencies

None.

See Also

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `ConfigFileProbeType` port type as specified below:

```
type union Command
{
  SetKeyCommand set,
  GetKeyCommand get,
```

```
}

type record SetKeyCommand
{
  charstring filename,
  charstring keypath,
  charstring value,
  charstring format optional, // default: autodetect based on extension
}

type record GetKeyCommand
{
  charstring filename,
  charstring keypath,
  charstring format optional, // default: autodetect based on extension
}

type union Result
{
  charstring errorResult, // contains a human readable error string
  charstring getResult, // empty string if not found
  bool setResult, // True if OK, False otherwise
}

type port ConfigFileProbeType
{
  in Command;
  out Result
}
```

Properties: || *plugin_properties* || *dict[string] of dict* || *'{}* || properties to pass to plugins, indexed by the format they support ||

Properties for the *ini* plugin: || *comments* || list of strings || *['#', ';']* || List of characters or strings that identify a comment in a .ini file ||

RTP Probe

Send/receive RTP streams.

Identification and Properties

Probe Type ID: `rtp`

Properties:

Name	Type	Default value	Description
`listening_ip`	string	`'0.0.0.0'`	Default listening IP address, when starting listening without providing one explicitly
`listening_port`	integer	`0`	Default listening UDP port, when starting listening without providing one expliclity. `0` means a system-assigned port.
`local_ip`	string	(empty)	Default source IP address when sending packets, if not provided explicitly when starting sending. An empty value means a system-assigned IP address.
`local_port`	integer	`0`	Default UDP source port when sending packets, if not provided explicitly when starting sending. An empty value means a system-assigned port.
`stream_timeout`	float	`0.5`	The maximum interruption interval allowed, in second, before considering an incoming stream has been stopped
`ssrc`	integer	`1000`	The default SSRC to use in sent RTP packets, if not provided explicitly when starting sending a stream
`payload_type`	integer	`8`	The default payload type to use in sent RTP packets, if not provided explicitly when starting sending a stream
`frame_size`	integer	`20`	The default frame size to use when sending RTP packets, if not provided explicitly when starting sending a stream
`packet_size`	integer	`160`	The default packet size to use when sending RTP packets, if not provided explicitly when starting sending a stream
`sample_rate`	integer	`8000`	The default sample rate to use in sent RTP packets, if not provided explicitly when starting sending a stream

The default payload type/frame size/packet size/sample rate values are chosen to enable a default stream in G.711 a-law (PCMA) / 20.

Overview

Through this probe, you can control sending and receiving RTP (Real Time Protocol) streams.

It can be used as a probe companion of a UDP transport probe to simulate a SIP endpoint, or with a RTSP probe to act as a , and of course any other cases where controlling sending and receiving of RTP streams is required, without requiring to inspect each RTP packets individually.

This probe is based on "signals" that change its state. You can:

- start sending a new RTP stream to a remote destination, setting the payload characteristics that will be used until the stream is stopped,
- stop sending a started stream
- start listening for, i.e. waiting for a possible incoming stream
- stop listening

And you may receive the following "notifications" if the probe was in listening state:

- `StartedReceivingEvent`, indicating the payload properties (source ip/port, ssrc, payload type), when a new stream is detected or has been updated,
- `StoppedReceivingEvent`, when a stream being received was interrupted (or updated its properties

Additionally, you may inject at any time while sending RTP a recorded payload using the `PlayCommand`. Currently the probe only supports:

- WAV file format: the contained payload must be encoded with the correct codec (payload type). The probe won't check it.
- raw format: the payload is encoded using the correct codec, no header, no container.

The probe sticks to packetize the payload according to the current frame size parameter. No transcoding mechanism is provided.

By default, when starting sending a RTP stream, a default sound is played according to the payload type:

Payload type	Default sound
0 (PCMU/G711 u-law)	a tone
8 (PCMA/G711 a-law)	another tone
Other codecs	Some very scratchy sound

Anyway, the idea is that, by default, the probe makes some sound or noise.

If you prefer start sending a silent sound, you should set the argument `defaultPayload` of `startSendingRtp` to what fits your need according to the codec. For reference:

Payload type	Payload to use for a silence
0 (PCMU/G711 u-law)	`'\xff'`
8 (PCMA/G711 a-law)	`'\x00'`
Other codecs	...

You may, of course, inject another payload as a default sound.

Notes:

- A probe can send/receive at most one stream in a way (i.e. can send, receive, or send+receive).
- No RTCP support for now
- No RFC2833 support for now (excepted by playing a wav file with RFC2833 payload, but this is not integrated enough to be usable)

Availability

All platforms. Tested on Linux (kernel 2.6), Solaris 8, Solaris 10.

Dependencies

None.

See Also

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `RtpPortType` port type as specified below:

```
type record StartSendingCommand
{
  integer payloadType optional, // default: payload_type (8)
  integer frameSize optional, // default: frame_size (20 (ms))
  integer ssrc optional, // default: ssrc (1000)
  integer packetSize optional, // default: packet_size (160 bytes, corresponding to 20ms in G711a
  integer sampleRate optional, // default: sample_rate (8000 (Hz))

  // Local port may be controlled dynamically - useful when negotiating ports via SDP, etc
  integer fromPort optional, // default: local_port
  charstring fromIp optional, // default: local_ip
```

```
  // Default payload to play (infinite loop)
  octetstring defaultPayload optional, // default: see above, according to the codec
  charstring defaultPayloadFormat optional, // choice in wav, raw ; default: wav
}

type record StopSendingCommand
{
}

type record StartListeningCommand
{
  // Should we control the listening ip/port dynamically ? ...
  integer onPort optional, // default: listening_port
  charstring onIp optional, // default: listening_ip
  integer timeout optional, // timeout to detect interrupted incoming stream, in s
}

type record StopListeningCommand
{
}

type record StartedReceivingEvent
{
  integer fromPort,
  charstring fromIp,
  integer payloadType,
  integer ssrc,
}

type record StoppedReceivingEvent
{
  integer fromPort,
  charstring fromIp,
  enum { interrupted, updated } reason,
}

type record PlayCommand
{
  octetstring payload,
  integer loopCount optional, // default: 1
  charstring format optional, // choice in wav, raw ; default: wav
}

type union Command
{
  StartSendingCommand startSendingRtp,
  StopSendingCommand stopSendingRtp,
  StartListeningCommand startListeningRtp,
  StopListeningCommand stopListeningRtp,
  PlayCommand play
}

type union Notification
{
  StartedReceivingNotification startedReceivingRtp,
  StoppedReceivingNotification stoppedReceivingRtp
}

type port message RtpPortType
{
  in Command,
  out Notification
```

```
}
```

RTSP Probe

Act as a RTSP client over TCP.

Identification and Properties

Probe Type ID: `rtsp.client`

Properties:

Name	Type	Default value	Description
`strict_mode`	boolean	`False`	When `True`, disables automatic CSeq management (generation and check on response), and transfer response headers as is, without modifying the case (use this mode for protocol-oriented testing). When `False`, CSeq is automatically generated if not provided, checked when receiving a response, and reponse header names are converted to lower case to make matching easier.
`version`	string	`RTSP/1.0`	The RSTP version string to use in the request line
`auto_connect`	boolean	`False`	If set to `True`, tcp-connect on mapping, otherwise only connect hen sending a message. Has no effect for udp transport.
`maintain_connection`	boolean	`False`	If set to `True`, does not tcp-disconnect once a response has been received. Has no effect for udp transport.
`host`	string	`localhost`	The IP address or hostname of the RTSP server
`port`	integer	`554`	The transport port of the RTSP server
`transport`	string in `'tcp'`, `'udp'`	`tcp`	The transport to use to reach the RTSP server. Only `tcp` is implemented for now
`local_ip`	string	`''` (empty)	The local IP address (or hostname) to use for outgoing packets. Leave it empty for automatic selection by the system. For udp transport, this is also the address the probe listens for a response on
`local_port`	integer	`0`	The local port for the outgoing packets. Leave it to 0 for automatic port selection by the system. For udp transport, this is also the port the probe listens for a response on

Overview

This probe implements a very simple RTSP encoder/decoder over tcp. It encapsulates the RTSP codecs, feeding it only when the whole payload has been received (that's why a probe is required), and is based on RFC 2326[20].

Optionally, the probe can manage CSeq generation and correlation on response (see the `strict_mode` property), enabling both high-level and more procotol-oriented testing.

Availability

All platforms.

Dependencies

None.

[20] http://www.ietf.org/rfc/rfc2326.txt

See Also

For pure protocol testing, you may also consider using RTSP codecs with an UDP probe (stateful decoding not required), or with a TCP probe.

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `RtspClientPortType` port type as specified below:

```
type record RtspRequest
{
  charstring method,
  charstring uri,
  charstring version optional, // default: 'RTSP/1.0', or as configured
  record { charstring <header name>* } headers,
  charstring body optional, // default: ''
}

type record RtspResponse
{
  integer status,
  charstring reason,
  charstring protocol,
  record { charstring <header name>* } headers,
  charstring body,
}

type port RtspClientPortType message
{
  in RtspRequest;
  out RtspResponse;
}
```

LDAP Probe

Act as a LDAP client.

Identification and Properties

Probe Type ID: `ldap.client`

Properties:

Name	Type	Default value	Description
`server_url`	string	`'ldap://127.0.0.1:389'`	LDAP server url, including protocol to use (ldap or ldaps)
`ldap_version`	integer	`2`	LDAP server version
`bind_dn`	string	`None` (undefined)	The DN entry to bind, if not provided through a request
`password`	string	(empty)	The password to use by default for binding
`timeout`	float	`60.0`	The maximum amount of time allowed to perform a search/write/delete operation before raising an error response
`base_dn`	string	(empty)	A base DN to suffix all DNs used in search/write/delete operations. It is not used for the
`username`	string	`None` (undefined)	Deprecated. Use `bind_dn` instead.

Overview

This probe simulates a LDAP client that can connect to a LDAP server to perform the following LDAP operations:

- bind
- unbind
- search
- add and update
- delete

It can connect to LDAP v2 and v3 servers, using a standard or a secure (ldaps, SSL/TLS) connection.

SASL connections are currently not interfaced.

By default, the probe automatically binds using the properties `bind_dn` and `password` prior to executing a search/write/delete command, unless an explicit bind command was performed by the user before.

The probe automatically unbinds on unmap.

To add or update an entry, use a `WriteCommand` message. The probe automatically detects if it should be a new entry (don't forget mandatory attributes according to the entry's schema) or an update. In case of an update:

- only provided attributes are modified. The other ones are left unchanged
- all values of the existing attributes are replaced with the new ones (no values merge)
- you can delete an attribute by specifying an empty value list for it

To make ATSes more portable and more simple to manage, you may use the `base_dn` property to set a DN that will be appended to all DNs in use in:

- delete operation (`dn` parameter)
- write operation (`dn` parameter)
- search operation (`baseDn` parameter)

and automatically removed from the dn values as returned in `SearchResult.SearchEntry.dn` structures.[[BR]] In other words, all the dn values, in the userland, will be relative to that `base_dn`.

This `base_dn`, however, won't be suffixed to the `bind_dn`.

Notes:

- Bind, write and unbind operations are synchronous, i.e. it's not use arming a timer to cancel them from the userland: they are not cancellable, and only return when they are complete. However, you still must wait for a `BindResult` or `WriteResult` before assuming the operation is complete.
- The synchronous bind and write implementations may be replaced with asynchronous equivalent ones one day. This won't have any impact on your testcases if you wait for the `BindResult` and `WriteResult` as explained above.

Availability

All platforms.

Dependencies

This probe depends on the openldap library and its associated Python wrapper.

On Debian-based system, this is `python-ldap` and its dependencies.

See Also

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `LdapPortType` port type as specified below:

```
type union LdapCommand
{
  BindCommand   bind,     // binds to the server set in properties, with the defaut or given crede
  UnbindCommand unbind,   // unbinds from the server
  SearchCommand search,   // searches entries according to a ldap search filter
  WriteCommand  write,    // updates or adds a new entry or attribute
  DeleteCommand delete,   // deletes an entry
  AbandonCommand abandon, // abandon the current command, if any
  CancelCommand abandon,  // cancel the current command, if any (only supported on RFC3909 compli
}

type record BindCommand {
  charstring bindDn optional, // the distinguished name of the entry to bind
  charstring password optional,
}

type any UnbindCommand;

type record SearchCommand {
  charstring baseDn, // suffixed by the probe's base_dn property, if any
  charstring filter,
  charstring scope optional, // enum in 'base', 'subtree', 'onelevel', defaulted to 'base'
  record of charstring attributes optional, // defaulted to an empty list, i.e. all attributes are
}

type record WriteCommand {
  charstring dn, // suffixed by the probe's base_dn property, if any
  record of Attribute attributes optional, // defaulted to an empty list
}

type record Attribute {
  record of <natural type> <name> // dynamic names, natural types (charstring/universal charstrin
}

type record DeleteCommand {
  charstring dn, // suffixed by the probe's base_dn property, if any
}

type any AbandonCommand;

type any CancelCommand;

type boolean DeleteResult;

type boolean BindResult;

type boolean UnbindResult;

type record of SearchEntry SearchResult;

type record SearchEntry {
  charstring dn, // does not contain the base_dn property part, if any
  record of Attribute attributes,
}

type boolean WriteResult;
```

```
type charstring ErrorResponse;

type union LdapResponse
{
  BindResult bindResult,
  UnbindResult unbindResult,
  DeleteResulet deleteResult,
  SearchResult searchResult,
  WriteResult writeResult,
  ErrorResponse error,
}

type port LdapPortType message
{
  in  LdapCommand;
  out LdapResponse;
}
```

Oracle Probe

Oracle Database client.

= Identification and Properties =

Probe Type ID: `sql.oracle`

Properties:

Name	Type	Default value	Description
`host`	string	`'localhost'`	The Oracle server IP address or hostname. Must be the same as defined in the TSN on the server's connector.
`port`	integer	`1521`	The Oracle TCP listening port on `host`
`sid`	string	(empty)	The Oracle System ID corresponding to your instance
`user`	string	(empty)	The user to use to connect to the database `db` above
`password`	string	(empty)	The password to use, if required to connect to the database `db` above for user `user`

Overview

This probes allows to connect to an Oracle database and perform any valid SQL command that are valid for this DBS. Just send the SQL request as a string through a port bound to the probe.

As a result, you should expect a `SqlResult` choice structure, whose `'error'` arm is a simple charstring indicating an error (connection- or SQL- related), and `result` indicates a successful query.

In case of a non-SELECT query, `result` is an empty list. In case of a SELECT query, it contains a list of dictionaries corresponding to the selected entries, whose keys are the names of the returned columns. The associated values are natural Python equivalent to SQL types. However, the following types have not been tested yet:

- `NULL`
- blobs
- date/time

(If you encounter any problem with these SQL types or other ones, please contact us or create a ticket).

These structures and mechanisms are common to all `sql.*` probe types.

The supported Oracle versions depend on the underlying Oracle libs you are using, and are independent from the probe.

It has been tested successfully against a Oracle 10g R2 and 11g R2 servers.

Availability

All platforms.

Dependencies

This probe requires the cx_Oracle Python module, than can be found at http://cx-oracle.sourceforge.net.

The cx_Oracle module is a wrapper that requires Oracle librairies.

Here is a possible path to a working probe:

- Get the Oracle Instant Client from the Oracle site[21] (requires free registration) for the platform that will run the probe (the Basic Lite version is sufficient)
- Decompress it wherever you want
- Under Windows, copy `oraociicus11.dll` and `oci.dll` to a directory included in the `PYTHONPATH` of the agent (in particular, you may put them into its `plugins/probes/` directory)
- Under Linux, copy `?.so` and `oci.so` to `plugins/probes/` under your agent tree, but be sure to add a LD_LIBRARY_PATH to this path before restarting your agent. To make the probe work from the TE (i.e. not hosted on a agent), you should copy them to the `core/` Testerman directory, since it is automatically included in all TE's `LD_LIBRARY_PATH`.

Make sure to download Oracle Instant Client and a cx_Oracle module that are compatible with each other and with your Oracle database version.

As far as it has been tested, using libs for Oracle 11g correctly works with a Oracle 10g server.

See Also

- MySQL Probe, a probe to access a MySQL database.

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `SqlPortType` port type as specified below:

```
type charstring SqlRequest;

type union Result
{
  charstring error,
  record of SqlResult result
}

type record SqlResult
{
  any <field name>* // according to your request
}

type port SqlPortType message
{
  in SqlRequest,
  out SqlResult
}
```

[21] http://www.oracle.com/technology/tech/oci/instantclient/index.html

MySQL Probe

MySQL Database client.

Identification and Properties

Probe Type ID: `sql.mysql`

Properties:

Name	Type	Default value	Description
`host`	string	`'localhost'`	The MySQL server IP address or hostname
`port`	integer	`3306`	The MySQL service listening port on `host`
`db`	string	(empty)	The database to use
`user`	string	(empty)	The user to use to connect to the database `db` above
`password`	string	(empty)	The password to use, if required to connect to the database `db` above for user `user`

Overview

This probes allows to connect to a MySQL database and perform any valid SQL command that are valid for this DBS. Just send the SQL request as a string through a port bound to the probe.

As a result, you should expect a `SqlResult` choice structure, whose `'error'` arm is a simple charstring indicating an error (connection- or SQL- related), and `result` indicates a successful query.

In case of a non-SELECT query, `result` is an empty list. In case of a SELECT query, it contains a list of dictionaries corresponding to the selected entries, whose keys are the names of the returned columns. The associated values are natural Python equivalent to SQL types. However, the following types have not been tested yet:

- `NULL`
- blobs
- date/time

(If you encounter any problem with these SQL types or other ones, please contact us or create a ticket).

These structures and mechanisms are common to all `sql.*` probe types.

The supported MySQL versions depend on the underlying MySQL DB libs you are using, and are independent from the probe.

Availability

All platforms.

Dependencies

This probe requires the MySQLdb Python module.

- Debian package: `python-mysql` (+ dependencies)
- Windows package, egg package, sources available at http://sourceforge.net/projects/mysql-python.

The Windows package also requires a MySQL ODBC connector, as available here[22].

[22] http://www.mysql.com/products/connector/odbc/

See Also

- Oracle Probe, a probe to access Oracle databases.

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `SqlPortType` port type as specified below:

```
type charstring SqlRequest;

type union Result
{
  charstring error,
  record of SqlResult result
}

type record SqlResult
{
  any <field name>* // according to your request
}

type port SqlPortType message
{
  in SqlRequest,
  out SqlResult
}
```

HTTP Probe

A simple HTTP client.

Identification and Properties

Probe Type ID: `tcp`

Properties:

Name	Type	Default value	Description
`local_ip`	string	(empty - system assigned)	Local IP address to use when sending HTTP requests.
`local_port`	integer	`0` (system assigned)	Local port to use when sending HTTP requests.
`host`	string	`localhost`	The HTTP server's hostname or IP address.
`port`	integer	`80`	The HTTP server's port.
`version`	string	`HTTP/1.0`	The HTTP version to use in requests.
`protocol`	string	`http`	The HTTP variant:`http` or `https`. For now, only `http` is supported.
`maintain_connection`	boolean	`False`	If set to True and HTTP version is 1.1, the probe keeps the tcp connection opened once a response has been received, until the server closes it.
`connection_timeout`	float	`5.0`	The connection timeout, in s, when trying to connect to a remote party.

Overview

This probe acts as a simple HTTP client and is fully based on TCP Probe with `HTTP codecs`.

It can be used to connect to an HTTP server, send and receive requests at low-level. This client won't follow HTTP redirections, enable authentication, or anything fancy. However, it is able to maintain the TCP connection between requests (set `maintain_connection` to `True`) and manageq a connection timeout to automatically fails the probe after a given delay (see the `connection_timeout` property).

For HTTPS connectivity, you should consider using TCP Probe with `HTTP codecs` as default codecs instead.

Availability

All platforms.

Dependencies

None.

See Also

Instead of using the probe, you may consider a TCP Probe with some `HTTP codecs` set as default encoder/decoders, which offer a greater control and support HTTPS as well (with `use_ssl` set to `True`).

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `HttpClientPortType` port type as specified below:

```
type record HttpRequest
{
  charstring method optional, // default: 'GET'
  charstring url,
  record { charstring <header name>* } headers,
  charstring body optional, // default: ''
}

type record HttpResponse
{
  integer status,
  charstring reason,
  charstring protocol,
  record { charstring <header name>* } headers,
  charstring body,
}

type port HttpClientPortType
{
  in HttpRequest;
  out HttpResponse;
}
```

Selenium Probe

A Web driver using Selenium.

Identification and Properties

Probe Type ID: `selenium`

Properties:

Name	Type	Default value	Description
`rc_host`	string	`localhost`	Selenium-RC hostname or IP address
`rc_port`	integer	`4444`	Selenium-RC port
`browser`	string	`firefox`	The browser Selenium should use on the Selenium host
`server_url`	string	`None`	The server URL to browse when opening the browser. Providing it is mandatory.
`auto_shutdown`	boolean	`True`	When set, Selenium closes the browser window when the test case is over. It may be convenient to set it to False to leave this window open to debug a test case on error.

Overview

This probe enables to perform Selenium-based test through the Selenium Remote Control (RC) / Server.

It uses two kinds of command interfaces:

- a "loose command" approach that enables to trigger any actions you may record via the Selenium IDE - in this case all commands have at most two optional parameters: target, and value. Adapting a Selenium IDE script to Testerman is in this case trivial and a QTesterman plugin will be made available soon to do it for you.
- a "strict command" approach that enables to name the different parameters a Selenium command supports. Only commands specified in the TTCN-3 equivalence below are supported in this mode.

In both cases, some commands may return a result (for instance, isTextPresent returns a boolean result). You may expect such a result in a `alt()` loop as usual.

The probe automatically starts the browser on the RC host as soon as it is mapped, and everything is cleaned up once the test case is over.

A basic example is available in `samples/selenium.ats`.

Limitations: for now, only the "loose command" mode is implemented.

Availability

All platforms.

Dependencies

The probe needs a configured (and running) Selenium Server (provided by Selenium RC) that it can reach.[[BR]] You may install it on the host you want to run the browsers from.

See Also

Generate Testerman test cases with Selenium IDE

Summary Selenium IDE[23] is a Firefox plugin which records user interaction with a web site. The plugin can then export the list (of user actions) to different formats. The plugin enabling Selenium IDE to export is called a formatter.

The Selenium team provides bindings for several programming languages[24]. You can write a test in one of these languages and use the provided libraries to remote control a browser. Selenium IDE formatters do exactly that: convert the recorded list into source code for a specific language.

The Testerman ATS Formatter let's you export your recorded interactions directly to a Testerman test script (*.ats). The generated code will use the Selenium Probe or the Selenium Probe (WebDriver) to control the browser. This document covers only Selenium RC but the usage of the Selenium Webdriver probe is largely equivalent.

Install The formatter is a Firefox plugin, you have to install it on top of the Selenium IDE plugin. You may find a build script in plugin/probes/seleniumrc/selenium-testerman-formatter. Run the script and install the install via Firefox: Select File -> Open File ... and select the *.xpi file.

Installing a newer versions of the formatter Before installing a newer version, make sure to reset the options (Selenium IDE -> Options -> Options -> Formats -> Testerman ATS Formatter -> Reset Options) in order to follow the latest changes. Please be carefully though, as this will erase all your personal configurations.

How To, Best Practises and Information Please read the following instructions carefully (yes, it is long *sigh*). There are a lot of hidden tricks in SeleniumIDE/Testerman one has to know to efficiently generate ats files.

You should be familiar with writing Testerman ats files manually (and thus you are familiar with the send/receive concept, alt statements, verdicts, ...) and know Selenium IDE a little bit (record test cases, exporting), too. You will find a lot of documentation for Selenium IDE here[25].

Understanding Selenium Selenium is a suite of different tools. We will use Selenium IDE (recording user interactions with a web interface) and Selenium RC (remote executing of "simulated" user actions on a web interface, acutally, this is done by the Selenium Probe/Selenium Probe (WebDriver)).

Selenium RC has a set of different type of commands. There are commands like click(myButton), type(myInput, blablub), getText(myElement), isEditable(myInput).

Selenium IDE uses these commands but in a slightly changed way. Some commands are the same (click, open, type), but some others are not available in Selenium IDE, at least not in their orinigal form. Selenium IDE uses generated commands from the Selenium RC accessors (getText, isEditable, ... every command where you would expect a result). Instead of getXXX, one can use assertXXX, verifyXXX, storeXXX, and/or waitForXXX. These generated commandes are ultimately translated into their source command (= the command Selenium RC expects). It will note change anything for Selenium RC whether you use assertText(id=myElement, 42) or verifyText(id=myElement, 42); the actual command in both cases will be getText(id=myElement). To see which Selenium RC command will be produced, have a look at the reference tab in the lower bar of the Selenium IDE gui. When adding a command, you will see it's argument and it's base command appear. It is up to the formatter (e.g. the source code generator) to handle the different command types while still sending the base command to Selenium RC. Selenium IDE is of course not able to "record" these generated commands. It knows where you clicked, but it doesn't know your expectations concering returned results.

Selenium commands are often refered to as "selenese".

Using Selenium IDE generated commands As an example consider you want to fill in a form and expect a certain text after clicking on the submit button. During recording, you would type your text into the form and click the button. Selenium will list something like this:

[23] http://seleniumhq.org/projects/ide/
[24] http://seleniumhq.org/about/platforms.html#programming-languages
[25] http://seleniumhq.org/docs/02_selenium_ide.html

```
commands        target          value
type            id=input_pw     1234
click           id=login
```

Now, to check whether your expected text appears on the following page, right-click on the element where the text should appear (or select the text, ...). The context menu will propose you a range of commands to check the elements content. After selecting one command, Selenium IDE will list something like this:

```
verifyText      id=header   Login successful
```

The command will become "getText(id=header)" for Selenium RC. If you had used assertText() instead, the outcoming Selenium RC commands would have been the same. The difference lies in the handling of the returned result (see below for a differentiation of assert and verify). The target argument is often a "locator", i.e. the element on the web interface you address. To get an element's locator, you could check the HTML source (Firebug!) for id or name attributes. However, it is a probably a good idea to let Selenium IDE choose the locator: "Record" a dummy click (or something) action on that element to get it's locator listed and delete the dummy action then.

See http://release.seleniumhq.org/selenium-core/1.0.1/reference.html for more details

Using Testerman's parameter system The Testerman ats formatter supports Testerman's parameter system (global variables, often called PX_PASSWORD or PX_HOST_IP). There is a support for "static" and "dynamic" parameters. We call theme "static", when they are defined in the meta data block at the beginning of the generated code regardless of the content. You can setup these static parameters in the formatter's option (Selenium IDE -> Options -> Options -> Formats). Define here the parameter name, default value and type ("string" or "integer"). Every parameter listed here will always be dumped. Remember that the parameter's value can be overwritten by the campaign configuration (when launching a campaing in Testerman).

The Format is "name:value:type\n". The name has to match **/!^PX_[_A-Z0-9]+$/**. Examples:

```
PX_DB_SERVER:string:127.0.0.1
PX_DB_PASSWORD:string:1234
PX_MAX_CONNECT:integer:10
```

On the other hand, "dynamic" parameters are set up within the actual selenese. After recording your actions with Selenium IDE and before formatting (exporting), you could change some command arguments in order to make the test more flexible. By replacing a "real" value with it's parameter name (and it's default value), the test can be adapted to another test environment. To define a dynamic parameter, replace the original value by ${PX_PARA_NAME:orginalValue}.

Example:

```
commands        target          value
type            id=input_pw     1234
```

The above command generates:

```
sel.send(["type", "id=input_pw", "1234"])
```

Now consider the following

```
commands        target          value
type            id=input_pw     ${PX_PASSWORD:1234} ** define parameter name and default value be
```

generates:

```
# <parameter name="PX_PASSWORD" default="1234" type="integer"><![CDATA[]]></parameter>
...
sel.send(["type", "id=input_pw", str(PX_PASSWORD)])
```

As you can see, a new parameter is defined and during the selenium command, this parameter is used. The parameter definition is to be found in the meta data block, the value can be overwritten by external campaing config as mentioned before. Of course, you do/can not redefine the same parameter several times. Once, your PX_XXX is defined you can use in later commands:

old:

```
commands        target      value
type            id=input_pw     1234
assertValue     id=blub     1234
```

generates:

```
sel.send(["type", "id=input_pw", "1234"])
sel.send(["getValue", "id=blub"])
alt([
    [ sel.RECEIVE(template = "1234"),
...
```

new:

```
commands        target          value
type            id=input_pw     ${PX_PASSWORD:1234}
assertValue     id=blub         ${PX_PASSWORD}
```

generates:

```
# <parameter name="PX_PASSWORD" default="1234" type="integer"><![CDATA[]]></parameter>
...
sel.send(["type", "id=input_pw", str(PX_PASSWORD)])
sel.send(["getValue", "id=blub"])
alt([
    [ sel.RECEIVE(template = str(PX_PASSWORD)),
...
```

The parameter can easily be referred to. This enables you to use static parameters as well. You do not have to define a default value because they are defined anyway. Remember that you have to adapt the selenese *before* exporting. Their is one disadvantage: Once you changed your values to 'PX_SOMETHING' you won't be able to replay your test case with Selenium IDE any more.

The rule of thumb is:

`${PX_XXX}` –> replace this by a variable called `PX_XXX`

`${PX_XXX:1234}` –> replace this by a variable called `PX_XXX`, which has been set to 1234 in the meta data block

`PX_XXX` –> take this literally (do not replace anything, used for `storeXXX()`, see later)

The replace mechanism generally applies to the last argument of a command (often called "pattern" in Selenium IDE).

The px parameters can be used in conjunction with regular expressions. The value field is `regexp:${PX_VAR_NAME:the_actual_expression}`. Fur further details see the section for regular expressions.

```
commands        target      value
assertValue     id=blub     regexp:${PX_PASSWORD:[a-z]?}
```

Store Commands Seleniums provides the possiblity to store values in variables. You will find a lot of storeXXX() fonctions (storeText(), storeAlert(), storeChecked(), ...). Whenever you use a store command, the Testerman Formatter will print a send and a receive command (to the port). When using storeXXX(), the second argument of the command is the name of the variable to which the value will be assigned. There are two possibilities to store values. Either you use a "normal" variable name or you use a name like PX_XXX (matching the abouve regexp).

```
commands        target      value
storeText       link=home   myvari
storeText       link=home   PX_MYVAR
```

The generated code will be as follows:

```
# store (selinium): myvari = getText(link=home)
sel.send(["getText", "link=home"])
sel.receive(value = 'myvari')
myvari = value('myvari')
log('myvari = %s' % myvari)
# store (selinium): PX_MYVAR = getText(link=home)
sel.send(["getText", "link=home"])
sel.receive(value = 'PX_MYVAR')
PX_MYVAR = value('PX_MYVAR'))
log('PX_MYVAR = %s' % str(PX_MYVAR))
```

To refer to the stored variables, use the **${nameOfTheVariable}** syntax as mentioned above. If you used a PX_XXX variable you could even store a value and define a default value in the metadata block:

```
commands        target      value
storeText       link=home   ${PX_MYVAR:42}
```

The generated code will be as follows:

```
# <parameter name="PX_MY_VAR" default="42" type="integer"><![CDATA[]]></parameter>
...
# store (selinium): PX_MYVAR = getText(link=home)
sel.send(["getText", "link=home"])
sel.receive(value = 'PX_MYVAR')
PX_MYVAR = value('PX_MYVAR'))
log('PX_MYVAR = %s' % str(PX_MYVAR))
```

This would enable you to use PX_MYVAR (with it's default value) even before storing it. Of course, it's pretty hard to find a use case ;)

Let's take another example. Imagine you want to change a value on the web interface, check that the settings are saved correctly and then reset the original value. The naive command list would be:

```
commands        target            value
type            name=port         12345
clickAndWait    id=submit_button
assertValue     name=port         12345
type            name=port         1234 (supposing the original value was 1234)
clickAndWait    id=submit_button
```

While this is pretty forward (and easily recordable), it is not very flexible. When you are not sure of the system's state before test execution, it will be hard to reset the "right" default value (in this case, the original value was 9999 or so). Use storeXXX() to copy and paste. After recording your test with Selenium IDE and before exporting to an ats file, change the following:

```
commands        target            value
storeValue      name=port         PX_DEFAULT_PORT
type            name=port         ${PX_NEW_PORT:12345}
clickAndWait    id=submit_button
assertValue     name=port         ${PX_NEW_PORT}
type            name=port         ${PX_DEFAULT_PORT}
clickAndWait    id=submit_button
```

Just save the value before changing it, then, during "cleanup" reuse the given name. The above example will produce the following code:

```
# <parameter name="PX_NEW_PORT" default="12345" type="integer"><![CDATA[]]></parameter>
...
# store (selinium): PX_DEFAULT_PORT = getValue(name=port)
sel.send(["getValue", "name=port"])
sel.receive(value = 'PX_DEFAULT_PORT')
PX_DEFAULT_PORT = value('PX_DEFAULT_PORT'))
log('PX_DEFAULT_PORT = %s' % str(PX_DEFAULT_PORT))
```

```
#change to new port
sel.send(["type", "name=port", str(PX_NEW_PORT)])
sel.send(["click", "id=submit_button"])
sel.send(["waitForPageToLoad", "30000"])

#verify changings
sel.send(["getValue", "name=port"])
alt([
    [ sel.RECEIVE(template = str(PX_NEW_PORT)),
        lambda: self.setverdict(PASS),
        lambda: log('getValue(name=port) == ' + str(PX_NEW_PORT) + ' [using PX_NEW_PORT] -> Good!
    ],
    [ sel.RECEIVE(template = any_or_none()),
        lambda: self.setverdict(FAIL),
        lambda: log('getValue(name=port) != ' + str(PX_NEW_PORT) + ' [using PX_NEW_PORT] -> Bad!'
        lambda: stop(),
    ],
])

#clean up
sel.send(["type", "name=port", str(PX_DEFAULT_PORT)])
sel.send(["click", "id=submit_button"])
sel.send(["waitForPageToLoad", "30000"])
```

Regular Expression Sometime you need to check a certain value, but you know only the expected pattern. Selenium IDE has build-in support for regular expression (in fact, it says [in our case] to the formatter, that the user wants to use a regexp). The following example should answer all questions:

```
commands            target          value
assertValue         name=search     blub
assertValue         name=search     regexp:^[A-Z]?$
assertValue         name=search     regexp:${PX_MYPATTERN}
assertTextPresent   regexp:[a-z].?
```

generates:

```
sel.send(["getValue", "name=search"])
alt([
    [ sel.RECEIVE(template = "blub"),
...
sel.send(["getValue", "name=search"])
alt([
    [ sel.RECEIVE(template = pattern(r"^[A-Z]?$")),
...
sel.send(["getValue", "name=search"])
alt([
    [ sel.RECEIVE(template = pattern(str(PX_MYPATTERN))),
...
sel.send(["isTextPresent", "regexp:[a-z].?"])
alt([
    [ sel.RECEIVE(template = True),
...
```

You can combine px parameter definitions with regexp:

```
commands        target      value
assertValue     name=search regexp:${PX_REGEXP:[a-e]?}  ** define new px parameter (with default
```

Regular expression are a bit tricky though. In fact, Selenium RC supports regexp by itself and Testerman does so (vie Python's re.search()), too. The "problem" is that sometimes Selenium RC has to use it to determine the result of a command and sometimes Testerman can just wait for the result and then check it against the regular expression. Whenever you use a selenese generated from getXXX() (like assertText(), verifyConfirmation(), ...)

the Testerman's (=Python's) regular expressions will be used. You can see them in the output (Testerman's function pattern()). On the other hand, commands like assertTextPresent() (e.g. commands where you expect a boolean response) will use Selenium RC's regexp. When dumping these commands, you will the that the literal "regexp:" is still present because it will be send to Selenium RC. There doesn't seem to be a way to unify the two regular expression systems, although it probably won't be a problem.

Note: You can not use a variable stored previously via storeXXX() in an regexp if the variable is **not** a PX_XXX parameter. By the way, Selenium IDE doesn't implement variable translation (with ${variableName}) in regular expressions at all. It's the Testerman Formatter which adds this feature only for PX_XXX parameters.

See also: http://release.seleniumhq.org/selenium-core/1.0.1/reference.html#patterns

Assert vs. Verify assertXXX() and verifyXXX() produce nearly the same code. In both cases, an alt statement checks whether the returned response matches the expected result. The difference is that assert() will stop the test case if the check failed, verify will not (-> assert() generates an extra "lambda: stop()" for the failing template). Most of the time, you would probably want to use assert(), but verify() can be handy for debugging purpose or small and unimportant checks.

Adding comments There are to types of comments possible. The first one is inserted via Selenium IDE -> Edit -> Insert new comment. This will generate a simply python comment. Or, use the command echo(), which will use Testerman's log() method.

setverdict() The formatter will dump a setverdict(FAIL) during every alt step for the branch with the not-matching template (no setverdict() in storXXX(), though). A setverdict(PASS) is only printed in the very last alt branch. In fact, before dumping the code, the formatter counts all accessors (= all commands generated from isXXX() or getXXX() = every command where a result is returned). This enables him to print the setverdict(PASS) only at the very end.

Best Practices If Selenium IDE is not recording your actions (although the red button is pressed), the base url is most likely wrong. Selenium IDE will only record user interaction with the site given in the base url. You can change it in the upper "address bar" in Selenium IDE gui.

When using stuff like waitForCondition (e.g. functions with a timeout), you could consider to set up a high timeout and use a Testerman timer as watchdog with a lower timeout instead. If the selenium call fails (=condition didn't become true until the given timeout), the Selenium Probe will throw an exception; whereas if the Testerman timer times out, the test can "fail correctly"

Make sure you use [assert|verify|storte]Value() when checking the text of an input and not [assert|verify|store]Text(). Inputs do not have text, they have values. Applying getText() on an input will return an empty string.

Once you did your assertions etc (e.g. your test verdict is set), reset the system to the state before the test.

Wherever you want in the command list in Selenium IDE, you can use "Edit -> Insert new comment" to add a comment in the generated source coude. For example, before resetting the system to the former state you could add a comment saying "clean up"

Import ats files to Selenium IDE It is possible to revert the process, e.g. import a generated ats file to Selenium IDE. It does not work with every ats file, though. In fact, the generated ats files contain the list of selenese and the import mechanism exploits this list. Thus, ats without this list cannot be imported. To open an ats file with Selenium IDE, you have to change the expected format: Selenium IDE -> Options -> Format -> Testerman Anevia ATS Formatter. If this option is not available, enable it via Options -> Options -> Enable experimental features.

Note that the formatter is NOT able to import ats scripts containing several test cases.

ATTENTION: Make backups of your original ats file. The formatter will try to warn you if the thinks the file has been touched by hand

Known Issues

Selenese returning an array Some commands return an array (typically commands with [assert|verify|waitFor]All[Buttons|Links|...](), like assertAllLinks()). They are handled by the formatter now in a experimental version. The generated code may not be what you expected. The formatter will warn you.

Extensions

If you are using Selenium IDE to record events from your browser, you can user the following formatter to generate Testerman code directly: Generate Testerman test cases with Selenium IDE.

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `SeleniumProbePortType` port type as specified below:

```
type union SeleniumStrictCommand
{
  record { charstring key, charstring value } addCustomRequestHeader,
  record { charstring strategyName, charstring functionDefinition } addLocationStrategy,
  record { charstring scriptContent, charstring scriptTagId } addScript,
  record { charstring locator, charstring optionLocator } addSelection,
  record { charstring allow } allowNativeXpath,
  record {} altKeyDown,
  record {} altKeyUp,
  record { charstring answer } answerOnNextPrompt,
  record { charstring locator, charstring identifier } assignId,
  record { charstring fieldLocator, charstring fileLocator } attachFile,
  record { charstring filename, charstring kwargs } captureEntirePageScreenshot,
  record { charstring kwargs } captureEntirePageScreenshotToString, // then expect a response as c
  record { charstring type } captureNetworkTraffic, // then expect a response as charstring
  record { charstring filename } captureScreenshot,
  record {} captureScreenshotToString, // then expect a response as charstring
  record { charstring locator } check,
  record {} chooseCancelOnNextConfirmation,
  record {} chooseOkOnNextConfirmation,
  record { charstring locator } click,
  record { charstring locator, charstring coordString } clickAt,
  record {} close,
  record { charstring locator } contextMenu,
  record { charstring locator, charstring coordString } contextMenuAt,
  record {} controlKeyDown,
  record {} controlKeyUp,
  record { charstring nameValuePair, charstring optionsString } createCookie,
  record {} deleteAllVisibleCookies,
  record { charstring name, charstring optionsString } deleteCookie,
  record {} deselectPopUp,
  record { charstring locator } doubleClick,
  record { charstring locator, charstring coordString } doubleClickAt,
  record { charstring locator, charstring movementsString } dragAndDrop,
  record { charstring locatorOfObjectToBeDragged, charstring locatorOfDragDestinationObject } drag
  record { charstring locator, charstring movementsString } dragdrop,
  record { charstring locator, charstring eventName } fireEvent,
  record { charstring locator } focus,
  record {} getAlert, // then expect a response as charstring
  record { charstring attributeLocator } getAttribute, // then expect a response as charstring
  record {} getBodyText, // then expect a response as charstring
  record {} getConfirmation, // then expect a response as charstring
  record {} getCookie, // then expect a response as charstring
  record { charstring name } getCookieByName, // then expect a response as charstring
```

```
  record { charstring locator } getCursorPosition, // then expect a response as integer
  record { charstring locator } getElementHeight, // then expect a response as integer
  record { charstring locator } getElementIndex, // then expect a response as integer
  record { charstring locator } getElementPositionLeft, // then expect a response as integer
  record { charstring locator } getElementPositionTop, // then expect a response as integer
  record { charstring locator } getElementWidth, // then expect a response as integer
  record { charstring script } getEval, // then expect a response as charstring
  record { charstring expression } getExpression, // then expect a response as charstring
  record {} getHtmlSource, // then expect a response as charstring
  record {} getLocation, // then expect a response as charstring
  record {} getMouseSpeed, // then expect a response as integer
  record {} getPrompt, // then expect a response as charstring
  record { charstring selectLocator } getSelectedId, // then expect a response as charstring
  record { charstring selectLocator } getSelectedIndex, // then expect a response as charstring
  record { charstring selectLocator } getSelectedLabel, // then expect a response as charstring
  record { charstring selectLocator } getSelectedValue, // then expect a response as charstring
  record {} getSpeed, // then expect a response as charstring
  record { charstring tableCellAddress } getTable, // then expect a response as charstring
  record { charstring locator } getText, // then expect a response as charstring
  record {} getTitle, // then expect a response as charstring
  record { charstring locator } getValue, // then expect a response as charstring
  record { charstring currentFrameString, charstring target } getWhetherThisFrameMatchFrameExpress
  record { charstring currentWindowString, charstring target } getWhetherThisWindowMatchWindowExpr
  record { charstring xpath } getXpathCount, // then expect a response as integer
  record {} goBack,
  record { charstring locator } highlight,
  record { charstring ignore } ignoreAttributesWithoutValue,
  record {} isAlertPresent, // then expect a response as boolean
  record { charstring locator } isChecked, // then expect a response as boolean
  record {} isConfirmationPresent, // then expect a response as boolean
  record { charstring name } isCookiePresent, // then expect a response as boolean
  record { charstring locator } isEditable, // then expect a response as boolean
  record { charstring locator } isElementPresent, // then expect a response as boolean
  record { charstring locator1, charstring locator2 } isOrdered, // then expect a response as bool
  record {} isPromptPresent, // then expect a response as boolean
  record { charstring selectLocator } isSomethingSelected, // then expect a response as boolean
  record { charstring pattern } isTextPresent, // then expect a response as boolean
  record { charstring locator } isVisible, // then expect a response as boolean
  record { charstring locator, charstring keySequence } keyDown,
  record { charstring keycode } keyDownNative,
  record { charstring locator, charstring keySequence } keyPress,
  record { charstring keycode } keyPressNative,
  record { charstring locator, charstring keySequence } keyUp,
  record { charstring keycode } keyUpNative,
  record {} metaKeyDown,
  record {} metaKeyUp,
  record { charstring locator } mouseDown,
  record { charstring locator, charstring coordString } mouseDownAt,
  record { charstring locator } mouseDownRight,
  record { charstring locator, charstring coordString } mouseDownRightAt,
  record { charstring locator } mouseMove,
  record { charstring locator, charstring coordString } mouseMoveAt,
  record { charstring locator } mouseOut,
  record { charstring locator } mouseOver,
  record { charstring locator } mouseUp,
  record { charstring locator, charstring coordString } mouseUpAt,
  record { charstring locator } mouseUpRight,
  record { charstring locator, charstring coordString } mouseUpRightAt,
  record { charstring url } open,
  record { charstring url, charstring windowID } openWindow,
  record {} refresh,
  record { charstring locator } removeAllSelections,
  record { charstring scriptTagId } removeScript,
```

```
  record { charstring locator, charstring optionLocator } removeSelection,
  record {} retrieveLastRemoteControlLogs, // then expect a response as charstring
  record { charstring rollupName, charstring kwargs } rollup,
  record { charstring script } runScript,
  record { charstring selectLocator, charstring optionLocator } select,
  record { charstring locator } selectFrame,
  record { charstring windowID } selectPopUp,
  record { charstring windowID } selectWindow,
  record { charstring logLevel } setBrowserLogLevel,
  record { charstring context } setContext,
  record { charstring locator, charstring position } setCursorPosition,
  record { charstring pixels } setMouseSpeed,
  record { charstring value } setSpeed,
  record { charstring timeout } setTimeout,
  record {} shiftKeyDown,
  record {} shiftKeyUp,
  record {} shutDownSeleniumServer,
  record { charstring formLocator } submit,
  record {} testComplete,
  record { charstring locator, charstring value } type,
  record { charstring locator, charstring value } typeKeys,
  record { charstring locator } uncheck,
  record { charstring libraryName } useXpathLibrary,
  record { charstring script, charstring timeout } waitForCondition,
  record { charstring frameAddress, charstring timeout } waitForFrameToLoad,
  record { charstring timeout } waitForPageToLoad,
  record { charstring windowID, charstring timeout } waitForPopUp,
  record {} windowFocus,
  record {} windowMaximize,
}


// With this "simplified" model, the command can be anything
// (assumed to be valid for Selenium), and its arguments
// are made of 0, 1 or 2 arguments (none, target, or target + value)
type union SeleniumLooseCommand
{
  record { charstring target optional, charstring value optional } *,
}

type SeleniumProbePortType
{
  in SeleniumStrictCommand;
  in SeleniumLooseCommand;
  out any; // depends on the invoked command
}
```

Selenium Probe (WebDriver)

A Web driver using Selenium WebDriver.

Identification and Properties

Probe Type ID: `selenium.webdriver`

Properties:

Name	Type	Default value	Description
rc_host	string	localhost	Selenium server hostname or IP address
rc_port	integer	4444	Selenium server port
browser	string	firefox	The browser Selenium should use on the Selenium host
server_url	string	None	The server URL to browse when opening the browser. Providing it is mandatory.
auto_shutdown	boolean	True	When set, Selenium closes the browser window when the test case is over. It may be convenient to set it to False to leave this window open to debug a test case on error.

Overview

This probe enables Testerman to perform Selenium-based tests through the Selenium Webdriver using a standalone Selenium Server.

The probe automatically starts the browser on the host as soon as it is mapped, and everything is cleaned up once the test case is over.

A basic example is available in `samples/selenium_webdriver.ats`.

Availability

All platforms.

Dependencies

The probe needs a configured (and running) Selenium Server that it can reach (depending on the browser, you may need different servers).[[BR]] You may install it on the host you want to run the browsers from.

TTCN-3 Types Equivalence

The test system interface port bound to such a probe complies with the `SeleniumWebdriverProbePortType` port type as specified below:

```
type record SeleniumSelenese
{
 charstring command,
 charstring target optional,
 charstring value optinal
}

type SeleniumWebdriverProbePortType
{
  in SeleniumSelenes
  out any; // depends on the invoked command
}
```

Developing New Codecs and Probes

TODO.

Notice that the internal APIs are stable and can be safely used to develop new codecs/probes plugins.

CHAPTER

FOUR

TESTERMAN ADMINISTRATION

Testerman Administration Guide

The Document Root

The document root (or docroot for short) in the directory used by the server processes to store local repository files, component updates, built test executables and execution logs.

The document hierarchy is:

- `repository/`: this directory is the root directory exposed to the user through the Ws interface. This is the local directory that is mounted as / in the exposed virtual file system. It typically contains all user ATS and module files, unless you configured additional file system backends to point to other locations or backend types. Since it contains user data, this is a directory to backup.
- `MOTD`: the Message Of The Day optional file. If present, it is retrieved by QTesterman clients and displayed on startup. You may use it to communicate about the server administrations, updates or maintenance announcements, etc.
- `archives/`: this directory contains executed ATSes, as Test Executables and associated log execution files. If you need to troubleshoot a TE, you may need to access this folder. It can also be browsed through the Ws interface, and is actually interfaces by QTesterman, so that the user can retrieve any archived execution log.
- `updates.xml`: this file is used to describe available component updates for component that supports autoupdates (for instance, QTesterman (via Ws) and the PyAgent (via Xa)). This file is mandatory to enable updates distribution.

Additionally, when distribution updates from the servers, we suggest you use the following convention:

- `updates/`: store component update packages (tar or tar.gz files). The complete path within the document root to the update is provided within the `updates.xml`, so it is not mandatory to use this folder name. However, distributable packages must be located below the document root.

This document root is typically used by both the Testerman Server and the Testerman Agent Controller Server (TACS). However, since the TACS only uses it for distributing updates to agents, you may create a document root dedicated to it, containing at least a `updates.xml` file, and typically a `updates/` folder with the agent packages referred by the xml file.

Update Management

Updates Metadata Format

The file `docroot/updates.xml` provides all the needed information to update an agent or a client. It is a xml file whose structure is something like:

```
<?xml version="1.0" encoding="utf-8" ?>
<updates>
  <update component="qtesterman" branch="stable" version="1.0.0" url="/updates/qtesterman-1.0.0.ta
    <!-- optional properties -->
    <property name="release_notes_url" value="/updates/rn-1.0.0.txt />
    <!-- ... -->
  </update>
  <update component="qtesterman" branch="experimental" version="1.0.1" url="/updates/qtesterman-1
  <update component="pyagent" branch="stable" version="1.0.1" url="/updates/pyagent-1.0.1.tar" />
  <update component="pyagent" branch="stable" version="1.0.0" url="/updates/pyagent-1.0.0.tar" />
</updates>
```

Whenever you want to make a new update available to agents/clients, you should update this file to declare the new update, providing the followinf information as attributes to the `update` element:

- `component`: a name identifying the component the update refers to. This name depends on the client/agent. See below for details.
- `branch`: a branch classifies an update. By convention, it is either `stable` (well tested update), `testing` (should be OK in most cases, but the user should be aware of some remaining potential problems with it - early deployment), or `experimental` (beta or alpha testing, specific purposes). These branches enables you to deploy updates without impacting all production users (using only stable updates), while leaving the opportunity to some users to test new probes or new QTesterman features (testing or experimental, when testing very specific updates with one or two users). Once a version has been correctly tested, you may switch its branch from `testing` to `stable` (or anything else).
- `version`: the version of a component. Must be formatted as `A.B.C` (for instance `1.0.1`, `2.10.13`). The meaning of each digit is your own choice for the component you developed. For components distributed with the project, C is a incremented on bugfix or small enhancement (A.B fixed), B on normal enhancement (A fixed, C reset to 0), A on major changes (B, C reset to 0).
- `url`: the location of the update archive within the document root. Must be a file format supported by the component.

Additional, optional properties may be defined. In this case, they are component-dependent.

Standard Updatable Components

Testerman provides the following standard component that can be updated through the system above:

Component name	Description	Supported archive formats	Supported properties	Comments
`qtesterman`	The QTesterman rich client	`tar`, `tar.gz`	(none)	The archive file must contain files in a `qtesterman/` base folder. Updated through Ws.
`pyagent`	The Python Testerman Agent (with probes)	`tar`, `tar.gz`	(none)	The archive must contain a (single) subdir containing the file to update. Updated through Xa.

You may develop your own components that could be distributed through this infrastructure as well.

CHAPTER

FIVE

CONTRIBUTING

Testerman Internals

This document provide some entry points if you want to contribute to the Testerman code.

Source Tree Organization

The source tree contains the following folders:

- `cliclient/`: the testerman command line client, useful to integrate tests execution from a Makefile or a continuous integration system
- `core/`: the core modules and binaries: the Testerman Server, the TACS, and all TTCN-3 related librairies used to create the TE
- `common/`: common modules used to create a Testerman Node (messages format, testerman protocol handling, etc), including a Testerman client
- `docs/`: sphinx-based documentation. This contains this document as well as most of the online documentation, managed along the source code to maintain consistency. A part of this documentation is autogenerated from the plugins (codecs and probes) sources.
- `plugins/probes/`: shared probes implementations, as plugins, both usable by a TE (as a local probe) or by a pyagent (as a remote probe)
- `plugins/codecs/`: shared codecs implementations, as plugins, usable by any shared probes or any ATS
- `pyagent/`: a python-based agent implementation, able to use the shared probes above
- `qtesterman/`: the QTesterman rich client, in python + PyQt4
- `samples/`: default samples ATSes and campaigns, to copy to the `$docroot/repository/samples/` during a nominal server installation. They should be maintained as carefully as the other files.
- `docs/`: the whole project documentation, based on Sphinx, in reStructuredText (rst) format. Also includes some tools to extract probe and plugins documentation from their code.

Notice the source tree is not exactly the same as the server's runtime tree (cf TestermanAdministrationGuide), even if you should be able to run the different components directly from it during their development.

Core Files

The `core/` folder contains:

- `backends/`: FileSystemBackend implementations, as plugins.
- `CodecManager`: main file for codec plugins: contains codec plugins interfaces, base classes, factories, and registration facilities. This module is adapted by `TestermanCD` when used by an ATS/TE, or by `PyTestermanAgent` when used from the pyagent.

- `ConfigManager`: server config, as singleton. One instance used by the Testerman Server process, another one for the TACS.
- `CounterManager`: for future usage. Will maintain statistics counters to expose through Ws, and maybe through SNMP one day.
- `EventManager`: server-side Xc stub, server-side Il stub.
- `FileSystemBackendManager`: main backend dispatcher, according to registered backend plugins.
- `FileSystemBackend`: base class for backend plugins implementation.
- `FileSystemManager`: the main entry point to any read/write access to the repository. Transforms Ws-like file path to something lower level, and calls the correct backend to manage these files.
- `JobManager`: the main job queue and job state manager.
- `ProbeImplementationManager`: main file for probe implementation plugins: contains probe implementation plugins interfaces, base classes, factories, and registration facilities. This module is adapted by `TestermanSA` when used by an ATS/TE (to create local probes), or by `PyTestermanAgent` when used from the pyagent (to create remote probes).
- `ProbeManager`: the server-side module responsible for probe management functions, using the TACC (server-side Ia client).
- `SimpleXMLRPCServer`: fixed XML-RPC server, for Python < 2.4.
- `TEFactory`: the TE creator, called from the job manager. One day, maybe, we'll implement a TE factory taking a real TTCN-3 script as input.
- `TestermanAgentControllerClient` (or TACC for short): the client stub to access the TACS. Used by the TEs, but also by the server itself to get an access to the TACS for agent/probe management purposes. Client-side Ia client stub.
- `TestermanAgentControllerServer`: the TACS process. Server-side Ia server, server-side Xa server, and TACS business logic.
- `TestermanCD`: Codec management part. Manages codecs as plugins, and interfaces them when the TTCN3 adaptation module needs to call them.
- `TestermanPA`: Platform adapter part: through a TRI-like interface, implements a timer library.
- `TestermanSA`: SUT Adapter part: through a TRI-like interface, interfaces the probes (via TACC) for the mapped test system interface ports. Also provides super classes to create local probes and remote probe stubs as plugins.
- `TestermanServer`: the server's main module. Initializes everything, starts the different components of the server.
- `TestermanTCI`: logging facilities for the TE. Partially implements the TCI TTCN-3 interface, in particular the TLI subset (Test Logging Interface). Generates log events sent to the TL module implemented by EventManager. Client-side Il stub.
- `TestermanTTCN3`: the main TTCN-3 adaptation module, providing TTCN-3 features to ATSes. It relies on Testerman{TCI,CD,SA,PA} to implement them, though TLI- and TRI-like interfaces.
- `Tools`: some low level tools.
- `Versions`: interfaces and implementations versions.
- `WebServices`: the functions of this module are exposed as being part of the WsInterface API.

Sequence Diagrams

These sequence diagrams are limited to remote interactions; they do not detail every internal calls within a module.

Starting a Job with Event Subscriptions

(including job states till its completion)

Testerman Internal Protocol

Most internal interfaces in Testerman, including Xa, uses the same kind of protocol and in particular encodes messages the same way.

Message Format

They are text-based, human-readable messages heavily inspired by HTTP-like protocols:

```
METHOD testerman:probe@agent XA/1.0
Type: request
Seq: 1
Content-Type: application/json

...body...
```

Messages are \x00 separated, thus not read based on a content-length. As a consequence, you should make sure that \x00 is never transmitted as a part of your message. Some classes, such the one as provided by source:trunk/common/TestermanNodes.py and source:trunk/common/TestermanMessages.py make you sure to use preferred methods to encode payloads (using JSON or python pickle) so that the problem does not happen.

The following headers are mandatory:

- Type: “request” or “notify”
- Seq: an integer, should be unique (in the system ?)
- Content-Type: indicates what the body deals with.

Response format:

```
XA/1.0 200 OK
Seq: 1
Content-Type: application/json

...body...
```

Just like HTTP (for instance). Notice the Seq header that must match the request’s one.

The message line separator is a single \n (and not a \r\n).

Testerman Log Format

Testerman logs, as generated by the server, are formatted as XML file.

However, they do not use the full feature of the description language, in particular they are not structured, and only consist in a series of basic “events”, where each event is mapped to a particular XML element (this is mostly due to the fact that real-time logs send these elements, so that the analyzer code for real-time or offline log parsing can be the same).

The following events are managed:

event	element	attributes	sub-elements	class
User log	user	tc (optional)	CDATA	user
Internal log (for debug)	internal		CDATA	internal
Message sent between two test component ports	message-sent	from-tc, from-port, to-tc, to-port	message	event
Test Case created	testcase-started	id		event
Test Case started	testcase-started	id	CDATA (title)	event
Test Case stopped	testcase-stopped	id, verdict	CDATA (description, as it can be modified on runtime)	event
Timer started	timer-created	id, tc, duration		event
Timer stopped	timer-stopped	id, tc, running-time (float, in s)		event
Timer expiry	timer-expiry	id, tc		event
Test Component created	tc-created	id		event
Test Component started	tc-started	id, behavior (id as string)		event
Test Component stopped	tc-stopped	id, verdict		event
Test Component killed	tc-killed	id		event
Verdict updated	verdict-updated	tc, verdict		event
Template match	template-match	tc, port	message, template	event
Template mismatch	template-mismatch	tc, port	message, template	event
Timeout branch selected	timeout-branch	id (the timer id)		event
Done branch selected	done-branch	id (the tc id)		event
Killed branch selected	killed-branch	id (the tc id)		event
Message sent to the SUT (i.e. completely encoded payload)	system-sent	tsi-port	label (string), payload (string)	system
Message received from the SUT (i.e. completely encoded payload)	system-received	tsi-port	label (string), payload (string)	system

Additionally, all elements have a timestamp element, indicating the time of the event as a float unix timestamp, in s.

`<message>` and `<template>` sub-elements are encoded as the XML representation of any valid Testerman message structure:

- tuples `(a, b)`, corresponding to a TTCN-3 `union`, are encoded as an element:

```
<c name="a">b'</c>
```

where `b'` is the XML serialization of `b` (the tag `c` is a shortcut for `choice`).

- dict `{a: b, c: d}`, corresponding to a TTCN-3 `record`, are encoded as an element:

```
<r>
  <f name="a">b'</f>
```

```
  <f name="c">d'</f>
</r>
```

where `b'`, `c'` are the XML serializations of `b`, `c` (the tag `f` is a shortcut for `field`, `r` is a shortcut for `record`).

- lists `[ a, b, ...]`, corresponding to a TTCN-3 `record of`, are encoded as an element:

```
<l>
  <i>a'</i>
  <i>b'</i>
</l>
```

where `a'`, `b'` are the XML serializations of `a`, `b` (the tag `l` is a shortcut for `list`, `i` is a shortcut for `item`).

Ws Interface

For now, you can directly browse the [source:/trunk/core/WebServices.py WebServices] file that contains, normally, all the required documentation for the Ws interface.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- search