# Yuma Test Harness

## Contents

# 1 Introduction

This document presents an overview of the Yuma Testing Strategy. It includes the following:

- Pre-requisites for using the test harness
- Guide to the Test Harness Directory Structure
- A high level overview of the Yuma Test Harness

## 1.1 Glossary

| | |
|---|---|
| **NETCONF** | An IETF network management protocol that provides mechanisms for create, retrieve, modify and delete operations on configuration data |
| **Yuma** | An OpenSource NETCONF Implementation |

## 1.2 References

For more information on Yuma / Netconf operation see the following references:

| | |
|---|---|
| **NETCONF Wikipedia Definition** | http://en.wikipedia.org/wiki/Netconf |
| **Netconf Central** | http://www.netconfcentral.org/ |
| **BOOST** | http://www.boost.org |
| **C++ Coding Standards Sutter / Alexandrescu** | |

# 2 Pre-Requisites

## 2.1 System Configuration

The table below identifies the system configuration necessary for building and running he Yuma Test Harness:

| Component | Version | Description |
|---|---|---|
| **Operating System** | Ubuntu 10.04 | |
| **Compiler** | GCC 4.4 or later | |
| **Boost** | Boost 1.47 | The Boost C++ libraries, including Boost::Test |
| **Lib Xml** | libxml2 / libxml2-dev | Gnome XML library, used by Yuma |
| **Python** | 2.6 | Python interpreter, used for simple scrpits |
| **GDB** | 7.1-1ubuntu2 | GNU Debugger |
| **Libbz2-dev** | 1.05-4ubuntu0.1 | Bzlib2 is required to build Boost |
| **Doxygen** | | Source code documentation tool |
| **Graphviz** | | Graphical Visualisation Tool used by Doxygen |
| **Texlive-font-utils** | | Text formatting used by doxygen |
| **Libncursesw5** | 5.7 | Shared libraries for terminal handling |
| **libssh2-1-dev** | 1.2.2-1 | SSH2 ssh2 client-side libary |

## 2.2 Yuma Sources

The Yuma source code can be checked out from the following location:

https://yuma.svn.sourceforge.net/svnroot/yuma

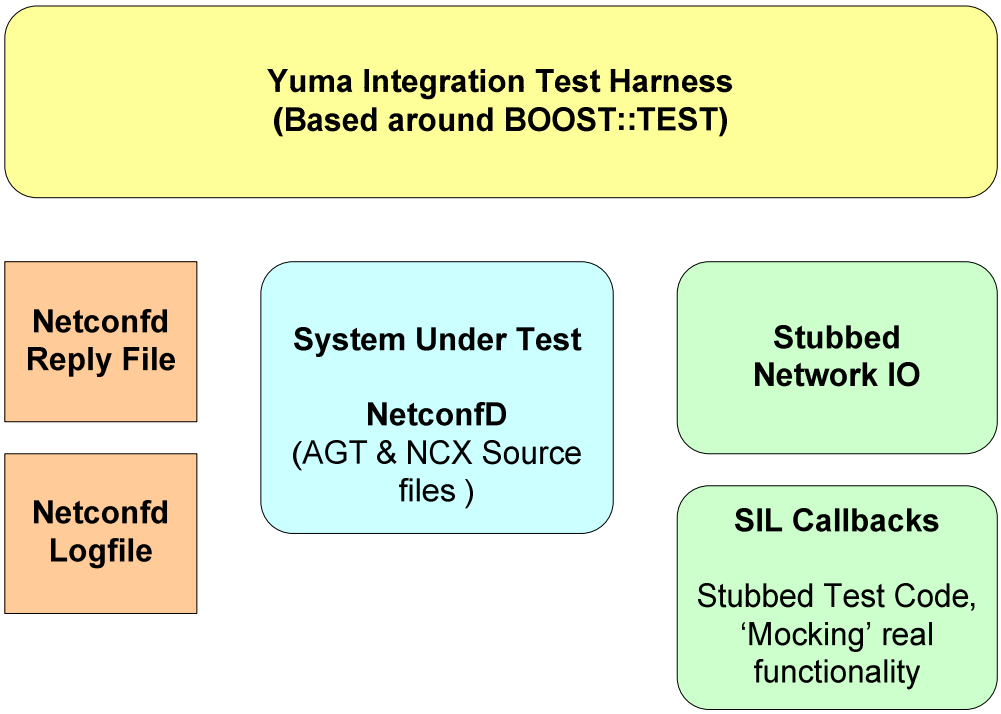
# 3 Overview

## 3.1 Yuma Test Strategy Overview

The Yuma test strategy consists can be split into two sections, Integration Testing and System Testing. The test harnesses are based around the BOOST::TEST libraries and utilise C++0x extensions available with the GCC 4.4 C++ Compiler.

### 3.1.1 The Integration Test Harness

The Integration test harness is built as a stand alone executable that includes most of the Yuma agt and ncx sources (which make up the system under test). The diagram below presents an overview of the Integration Test Harness:




### 3.1.2 The System Test Harness

The System test harness is a stand alone program that is capable of running full Netconf sessions against a full Yuma/Netconf Server (the system under test). The System test harness provides a fast way of verifying the behaviour of a full Yuma/Netconf system. It behaves in the same way as a real Netconf client. To use this test harness the Netconf Server must have the appropriate Yang and SIL modules installed.

### 3.1.3 Commonality between the System and Integration Test Harnesses

The majority of tests should be developed so they can be used by both the System and Integration test harnesses.

## 3.2 Test Harness Source Code Layout

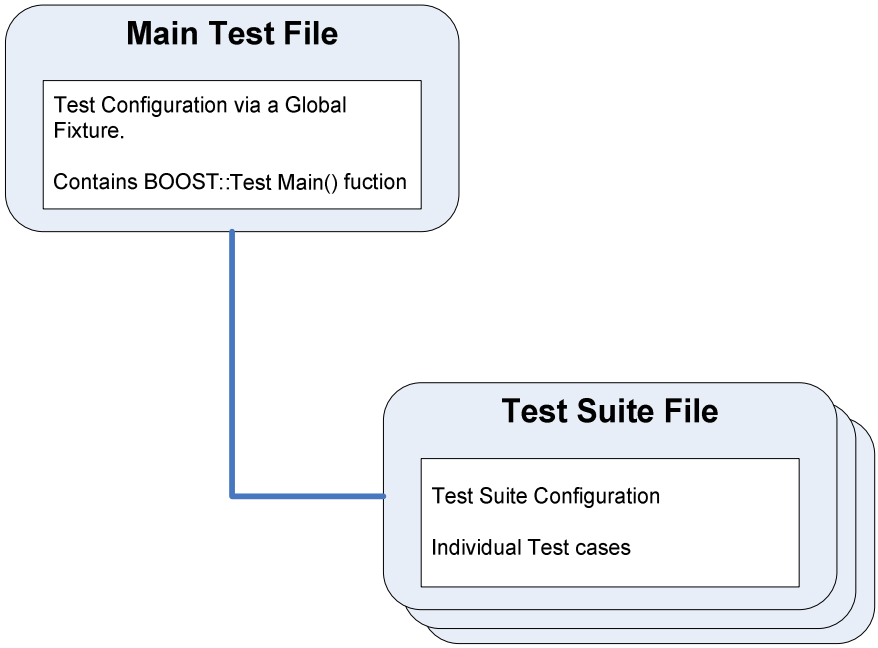
The Yuma test harness source code is located in the netconf/test directory. The table below presents a breakdown of the test harness directory contents:

| Directory | Description |
|---|---|
| netconf | The top level directory for Netconf source code. |
| netconf/agt | The Yuma Netconf Agent (agt) sources |
| netconf/ncx | The Yuma Core (ncx) sources |
| netcond/test | The root directory of the test harness |
| integ-tests | Home directory for all integration tests. |
| make-rules | Common make rules |
| modules | Root directory for all Netconf modules used by the test harness. |
| modules/yang | Test harness yang modules |
| modules/sil | Test harness sil modules |
| Modules/build-sil | Root directoru for building SIL modules |
| stubs | Root directory for all stubbed out functionality, as used by the integration test harness |
| stubs/agt | Stubs for agt components |
| stubs/ncx | Stubs for ncx components |
| support | Root directory for all test support sources |
| sys-test | Home directory for all system tests |
| test-suites | Root directory for all test suites |
| test-suites/common | Test suites that can be run by either the Integration Test Harness or the System Test Harness |
| test-suites/integ | Test suites that are specific to the Integration Test Harness |
| test-suites/system | Test suites that are specific to the System Test Harness |
| Utils | Miscellaneous utilities. |

# 4 Test Harness Overview

The Yuma test harness is driven by BOOST Test, with 'AUTOMATIC' registration of test cases. See [http://www.boost.org/doc/libs/1_47_0/libs/test/doc/html/utf.html]

Most tests consist of a Main Test Module and one or more Test Suites.




The Main Test Module which is responsible for all test wide initialization (initialization of BOOST::Test and Yuma).

The Test Suites contain one or more atomic test cases.

## 4.1 Atomic Testing

All tests must be 'atomic' and must not rely on the actions of previous tests or the order of testing. This means that all tests should revert the changes that were made before terminating.

This allows the execution of either all tests, selected test suites or individual tests.

## 4.2 Test Fixtures

A test fixture is essentially a combination of setup and teardown functions, associated with test case. Test fixtures are implemented as C++ classes whose constructor is run at the start of a test case and whose destructor is run at the end of a test case. Test fixtures can be attached as follows:

- **Globally** the test fixture is instantiated once, for the entire test run,.
- **Shared test suite level fixture** – the same test fixture is shared by all test cases, it will be instantiated at the start of every test.
- **Per test case fixture** – a specific test fixture is specified for the individual test case. It is instantiated only at the start of the specific test case.

### 4.2.1 Global Test Fixtures

The test harness makes use of global test fixtures to initialise and configure the test run and to cleanup after the test has completed.

The global test fixture for the Integration Test Harness performs the following operations:

- Configures the test harness context so that tests are run integration test mode, this involves instantiating a number  concrete classes that use direct function calls to interrogate Yuma.
- Initialisation of Yuma. This is a replacement for Yuma's *cmn_init()* function.
- The destructor simply shuts down the Yuma system, tearing down the ncx and agt modules.

The global test fixture for the System Test Harness performs the following operations:

- Configures the test harness context so that tests are run system test mode. This involves instantiating a number of concrete classes that use SSH connections to a remote Netconf Agent..

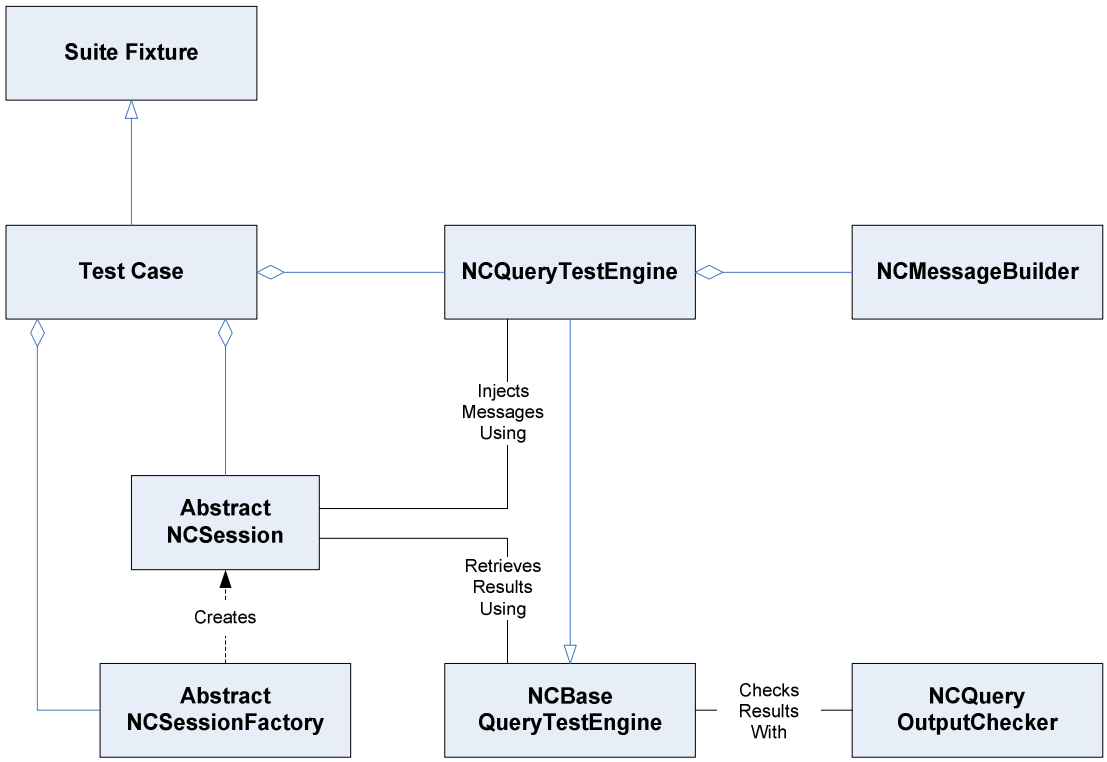
### 4.2.2 Suite Shared and Test Specific Fixtures

The suite shared / test specific fixtures perform initialisation and cleanup for each individual test case. BOOST::Test derives the test case from the specified fixture. This means that the test case can freely access any public or protected members of the fixture class.

The Yuma test harness uses these fixtures to implement functions that are common to a number of tests.

## 4.3 High Level Test Overview

The diagram below presents a high level overview of the test harness software.




Each test case uses it's NCQueryTestEngine to inject Netconf Messages into Yuma and check their result. The actual injection / retrieval of the results is performed by a concrete NCSession (e.g. SpoofNCSession). Detailed descriptions for these classes and their member functions can be found as Doxygen comments in the source code.

## 4.4 Generation of Doxygen Documentation for the Test Harness

Doxygen Documentation can be generated by:

**cd <YUMA_SRC>/netconf/test**
**./make-doxygen.sh**

## 4.5 Building the Test Harness

The test harness must be built using the following procedure:

1. Set the $YUMA_SRC environment variable to the top level of the Yuma checkout that is being tested, e.g.

   ***export YUMA_SRC = ~ /yuma/branches/v1-dev-001***

2. Build the test SIL libraries.

   a. **cd $YUMA_SRC/Netconf/test/modules/build-sil/XXXX**

   b. **make**

3. Build the test harnesses

   a. **cd $YUMA_SRC/Netconf/test/integ-tests**

**b. make**

The integration test harness can now be run.

### 4.5.1 Running All Tests

All test harnesses (and all tests) can be run using the Python utility script alltests.py, e.g:

**./alltests.py**

### 4.5.2 Running Individual Tests

To run a specific test:

*./test-name –run_test=<suite name>/<test case name>*

# 5 Guidelines

The test harness follows a 'loose coding standard' based around the guidelines in the set out by Herb Sutter and Andrei Alexandrescu in the book C++ Coding Standards. A few key rules to follow are identified below:

## 5.1 Doxygen

All functions should be documented with Doxygen comments in the header file. These comments make up the main documentation for the test harness.

## 5.2 Tests must be Atomic

As stated earlier, all tests **must** be atomic and **must not** rely on be run after any other test.

## 5.3 Give Each Entity One Cohesive Responsibility

Give each class function or variable a single well defined responsibility. This helps with maintenance and future understanding of the software. As a guideline keep functions small – preferably small enough to fit completely on one screen.

## 5.4 No Code Duplication

Code duplication must be avoided wherever possible. Any functionality that might be useful to future tests should be placed inside a separate class / function within one of the support directories.

## 5.5 Code for Scalability

- When writing test support functions use the most generic and abstract means to implement a piece of functionality
- New functionality must not be bolted on to existing functions – place it in a new function or class.

- Common functionality should be generalized. Use templates to minimize code duplication

# 6 Installing BOOST

To build, configure and install BOOST follow the instructions below:

1. Remove any previously installed versions of boost using Synaptic Package Manager

2. Download the BOOST libraries from www.boost.org

3. Create a local directory (e.g. ~/boost ) and copy in the downloaded boost archive.

4. Extract the archive

   **tar xvfj boost_1_47_0.tar.bz2**

5. Configure Boost build boostrap

   **cd boost_1_XX_XX**
   **./bootstrap.sh --prefix=/usr/local**

6. Build boost (this will take about 20 minutes)

   **./b2**

7. Install Boost – this should put the boost libraries in */usr/local/include* and */usr/local/lib*

   **sudo ./b2 install**

8. Add */usr/local/lib* to the runtime library search path

   **sudo cat '#boost libraries' >> /etc/ld.so.conf.d/libboost.conf**
   **sudo cat '/usr/local/lib' >> /etc/ld.so.conf.d/libboost.conf**

9. Refresh library search path

   **sudo ldconfig**