Coverage Analysis

Verifying Test Quality

The purpose of coverage analysis is to verify the thoroughness of a test suite. For example, unit tests are used to validate the implementation of detailed design objects through comprehensive testing. Coverage analysis checks that the testing is, indeed, comprehensive by executing instrumented unit tests which records the complete execution path through the code and then calculating metrics indicative of the coverage achieved during execution.

Coverage analysis examines the output of a code instrumented to record every line executed, every conditional branch taken, and every block executed. It then generates metrics on:

- Percent of statements executed
- Percent of methods (and/or functions) executed
- Percent of conditional branches executed
- Percent of a method's (and/or function's) entry/exit branches taken.

The metrics give a general idea of the thoroughness of the unit tests. The most valuable aspect of most web-based coverage analysis tools is the color-coded report where the statements not exercised and the branches not taken are vividly evident. The color-coded coverage holes clearly show the developer where unit tests need improvement.

Using the coverage analysis reports, the LSST DM developer should determine code segments which have not been adequately tested and should then revise the unit test suite as appropriate. Coverage analysis reports should be generated in concert with the routine automated buildbot testing.

DM Coverage Analysis Metrics

Refer to Code Coverage Analysis, by Steve Cornett, for a discussion of coverage metrics and to Minimum Acceptable Code Coverage, also by Steve Cornett, for the companion discussion on determining 'good-enough' overall test coverage.

A specific metric for lines of code executed and/or metric for branch conditionals executed is expected to be defined for Construction.

Using Coverage Analysis Tools

C++

LSST **scons** builds will automatically instrument all object and link modules with coverage counters when invoked with:

```
scons profile=gcov
```

This passes `--coverage` to all compile and link builds; this is equivalent to `-fprofile-arcs -ftest-coverage` on compile and `-lgcov` on link.

Executing the instrumented program causes coverage output to be accumulated. For each instrumented object file, the associated files `.gcda` and `.gcno` are created in the object file's directory. Successive runs add to the `.gcda` files resulting in a cumulative picture of object coverage.

Use one of the following tools to create the coverage analysis reports to verify that your unit testing coverage is adequate. Editor's preference is for either **ggcov** or **tggcov** since only the local source files are processed; see below for details.

gcov

gcov is the original coverage analysis tool delivered with the GNU C/C++ compilers. The coverage analysis output is placed in the current directory. The analysis is done on all source and include files to which the tool is directed so be prepared for reports on all accessed system header files if you use **gcov**.

Use the following to generate coverage analysis on the LSST `<module>/src` directory:

```
cd <module>
scons profile=gcov
gcov -b -o src/ src/*.cc src.gcov >& src_gcov.log
```

ggcov

ggcov is an alternate coverage analysis tool to **gcov** which uses a GTK+ GUI. **ggcov** uses the same profiling data generated from a GCC instrumented code but uses its own analysis engine.

Use the following to bring up the **ggcov** GUI:

```
cd <module>
scons profile=gcov
ggcov -o src/
```

tggcov

tggcov is the non-graphical interface to **ggcov**.

tggcov creates its output files in the same directory as the source files are located. It creates analysis files for only the local source files (i.e. not the system files).

Use the following for a comprehensive coverage analysis. Output files will be in `src/*.cc.tggcov`:

```
cd <module>
scons profile=gcov
tggcov -a -B -H -L -N -o src/ src
```

gcov output files in git directories

gcov coverage output files should be identified as non-**git** files to avoid the **git** warning about untracked files. In order to permanently ignore all **gcov** output files, add the extensions `.gcno` and `.gcda`, to the `.gitignore` file.

Python

Note

No recommendations have been made for Python coverage analysis tools. The following are options to explore when time becomes available.

Coverage.py

Coverage.py, written by Ned Batchelder, is a Python module that measures code coverage during Python execution. It uses the code analysis tools and tracing hooks provided in the Python standard library to determine which lines are executable and which have been executed.

figleaf

figleaf, written by Titus Brown, is a Python code coverage analysis tool, built somewhat on the model of Ned Batchelder's Coverage.py module. The goals of figleaf are to be a minimal replacement of Coverage.py that supports more configurable coverage gathering and reporting.

Java

No options have been researched.

Python & C++ Test Setup

DM developers frequently use the Python unittest framework to exercise C++ methods and functions. This scenario still supports the use of the C++ coverage analysis tools.

As usual, the developer instruments the C++ routines for coverage analysis at compilation time by building with **scons profile=gcov**. The C++ routines generated from the SWIG `*.i` source are also instrumented. Later when a Python unittester invokes an instrumented C++ routine, the coverage is recorded into the well-known coverage data files `<src>.gcda` and `<src>.gcno`. Post-processing of the coverage data files is done by the developer's choice of C++ coverage analysis tool.