Adding Unit Tests to an Analysis Release

Introduction

The analysis release build system, SConsTools, provides a mechanism for integrating unit tests. Each package in the release, or a package that a user is developing, can have its own tests. All tests can be run by a simple command. Users may find this useful for testing their packages. For packages that psana developers add to the analysis release, these tests are automatically run during the nightly build. This page is primarily for psana developers, to go over how to add unit tests to test packages that are a part of the release. There are special considerations for tests that are run during the nightly build.

Creating a Package test directory

As an example, lets make a package with both a Python and a C++ unit test. For the example below, I am starting from a directory where I want to create a psana release directory. You can cut and past these lines into your terminal (assuming you are on a psana interactive node, and set up your environment for analysis):

```
newrel ana-current unitTestTutorial
cd unitTestTutorial
sit_setup
newpkg MyPkg
mkdir MyPkg/test
```

Now when one does

```
scons test
```

You are building and running the test target. SConsTools will look in the test subdirectory for all packages. It looks for unit tests in these test subdirectories. It looks for:

- Any file without an extension is treated as a test script. This script will be installed and run.
    - WARNING: beware of editors such as emacs that leave backup files such as myscript~, they will be treated as a separate test
- Any file that looks like C or C++ code (.c, .cpp extension, etc) is treated as a test program. It will be compiled, installed, and run.
- If a test script or program returns non-zero, it failed and scons will report this.

Files with extensions that do not look like C/C++ code are ignored.

We will start with some simple examples to demonstrate the testing mechanism. Then we will look at examples using testing frameworks. In practice, developers will most likely want to use testing frameworks.

Python Unit Test

Add the file MyPkg/test/myfirst

```
#!@PYTHON@
import sys
if __name__ == '__main__':
 print "Running myfirst test - it should fail:"
 sys.exit(1)
```

When you do scons test, you will get the output

```
Running UnitTest: "build/x86_64-rhel5-gcc41-opt/MyPkg/myfirst"
********************************************************************************
*** Unit test failed, check log file build/x86_64-rhel5-gcc41-opt/MyPkg/myfirst.utest ***
********************************************************************************
```

This is because the script returned something non-zero. If you look at the file build/x86_64-rhel5-gcc41-opt/MyPkg/myfirst.utest you see the output of the script, i.e:

```
$ cat build/x86_64-rhel5-gcc41-opt/MyPkg/myfirst.utest
Running myfirst test - it should fail:
```

This is not very informative, but demonstrates the underlying mechanism by which the system is told if a test passed or failed. Below we'll see how the testing frameworks provide more information in the output.

The syntax @PYTHON@ is explained in the SConsTools page.

Change the sys.exit(1) to sys.exit(0) and the test will succeed. You can also take out the sys.exit line, be default Python will return 0 after the script runs.

C++ Unit Test

A simple C++ test would like this, create the file MyPkg/test/mysecond.cpp

```
#include <iostream>

int main() {
 std::cout << "Cpp test" << std::endl;
 return -1;
}
```

This test will also fail. Note, scons test stops after the first test fails. If you have not changed myfirst to return 0, only one of myfirst and mysecond will be run before failure is reported.

Using Frameworks for Unit Tests

It is worthwhile to learn how to use a testing framework. Psana developers are encouraged to use unittest for Python, and boost::unit_test for C++ in order to be consistent with existing tests in the release. However this is not necessary. You can use whatever framework you like.

Python unittest Framework

Below is an example of using unittest with Python. Add the file MyPkg/test/using_python_framework with the following content:

```
#!@PYTHON@

import sys
import unittest

class MyTest( unittest.TestCase ) :

    def setUp(self) :
        """
        Method called to prepare the test fixture. This is called immediately
        before calling the test method; any exception raised by this method
        will be considered an error rather than a test failure.
        """
        pass

    def tearDown(self) :
        """
        Method called immediately after the test method has been called and
        the result recorded. This is called even if the test method raised
        an exception, so the implementation in subclasses may need to be
        particularly careful about checking internal state. Any exception raised
        by this method will be considered an error rather than a test failure.
        This method will only be called if the setUp() succeeds, regardless
        of the outcome of the test method.
        """
        pass

    def testMyTestOne(self):
        a=3
        b=4
        self.assertEqual(a,b)

if __name__ == '__main__':
  unittest.main(argv=[sys.argv[0], '-v'])
```

after doing scons test, you will get failure. After looking at the utest output file, one will find

```
**********************************************************************************************************
*** Unit test failed, check log file build/x86_64-rhel5-gcc41-opt/MyPkg/using_python_framework.utest ***
**********************************************************************************************************
scons: *** [build/x86_64-rhel5-gcc41-opt/MyPkg/using_python_framework.utest] Error 256
scons: building terminated because of errors.
psana1302:~/rel/unitTestTutorial $ cat build/x86_64-rhel5-gcc41-opt/MyPkg/using_python_framework.utest
testMyTest (__main__.MyTest) ... FAIL
======================================================================
FAIL: testMyTest (__main__.MyTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "build/x86_64-rhel5-gcc41-opt/MyPkg/next_test", line 31, in testMyTest
    self.assertEqual(a,b)
AssertionError: 3 != 4
----------------------------------------------------------------------
Ran 1 test in 0.000s
FAILED (failures=1)
```

I believe the mechanism by which Python unittest works is

- The script we wrote runs unittest.main and passes it the script name. We also pass the verbose argument so there will be more output in the logfile.
- The unittest module then inspects the contents of the script, finds a class that is derived from unittest.TestCase.
- It then creates an instance of this class, calling the setUp() function
- Next it looks for any method of the class that starts with "test" these are the "test methods" referred to in the comments for setUp and tearDown.
    - Other methods are ignored
- All of the test methods are called, finally tearDown is called (assuming all tests passed, but I'm not sure about that)

Refer to the documentation https://docs.python.org/2/library/unittest.html for more information on unittest and some good tutorials.

Boost unit_test Framework

For an example of using the boost C++ unit test framework, create the file MyPkg/test/using_boost_framework.cpp with the contents (this is mostly copied from the boost website:

```
/**
 * Simple test suite for module psevt-unit-test.
 * See http://www.boost.org/doc/libs/1_36_0/libs/test/doc/html/index.html
 */


#define BOOST_TEST_MODULE MyTest
#include <boost/test/unit_test.hpp>

int add( int i, int j ) { return i+j; }

BOOST_AUTO_TEST_CASE( my_test )
{
    // seven ways to detect and report the same error:
    BOOST_CHECK( add( 2,2 ) == 4 );        // #1 continues on error

    BOOST_REQUIRE( add( 2,2 ) == 4 );      // #2 throws on error

    if( add( 2,2 ) != 4 )
      BOOST_ERROR( "Ouch..." );            // #3 continues on error

    if( add( 2,2 ) != 4 )
      BOOST_FAIL( "Ouch..." );             // #4 throws on error

    if( add( 2,2 ) != 4 ) throw "Ouch..."; // #5 throws on error

    BOOST_CHECK_MESSAGE( add( 2,2 ) == 4,  // #6 continues on error
                         "add(..) result: " << add( 2,2 ) );

    BOOST_CHECK_EQUAL( add( 2,2 ), 4 );      // #7 continues on error
}

BOOST_AUTO_TEST_CASE( my_test_fail )
{
    BOOST_CHECK_EQUAL( add( 2,2 ), 5 );
}
```

The second test is designed to fail, and the output in the log file is

```
$ cat build/x86_64-rhel5-gcc41-opt/MyPkg/using_boost_framework.utest
Running 2 test cases...
MyPkg/test/using_boost_framework.cpp(38): error in "my_test_fail": check add( 2,2 ) == 5 failed [4 != 5]
```

For more examples, one can look in the test subdirectories of packages like AppUtils, ConfigSvc, XtcInput, psana, psana_test, and Translator.

Nightly Build Considerations - External Test Data

There are several things psana developers need to consider when writing tests for packages that are part of the analysis release that will be run as part of the nightly build. This mostly involves how to work with external test data files.

- The nightly build is (presently) run on psdev, both rhat5 and rhat6 machines.
  - psdev has no access to the experiment data
  - The same unit test may be running under both rhat5 and rhat6 at the same time, from the same release directory, but on different host machines.
- The nightly build runs under the user account psrel
  - psrel cannot read files private to your directories. It may not have the same group permissions that you do.

Unit tests should not reference experimental test data. In addition to the above, experiment data may be removed as per the data retention policy. In light of this, there are several choices for incorporating test data

- Check it in as part of your package
- Make a copy of it in a place accessible to psrel running on the psdev machines

Test Data Checked into the Package

Ideally we do not want to keep large amounts of data under version control. For test data, I think 10 kilobytes or so is Ok, but when it gets larger one should use the external location discussed below.

However for data files checked in with the package, the general problem is where to put them, and how to find them at run time. Probably the best practice is to use the data subdirectory. This is intended for application data with the package, so you may want to make a subdirectory under it for testing files. The advantage of the data directory is that it is wired into the release. So if we add the directories

- MyPkg/data/testdata

and then create the file

- MyPkg/data/testdata/mytestfile.txt

there that starts with the string "my", then we can write a unit test that uses a psana utility to find the file and test that it starts with "my". The psana utility is the Python class AppDataPath in the AppUtils package (there is a C++ version there as well). The unit test would look like

```
    def testMyFile(self):
        import os
        from AppUtils.AppDataPath import AppDataPath
        testFileDataRelPath = os.path.join('MyPkg', 'testdata', 'mytestfile.txt')
        testFilePath = AppDataPath(testFileDataRelPath).path()
        assert len(testFilePath)>0 , "test file (relative to release data dir): %s not found." %
testFileDataRelPath
        fileText = file(testFilePath, 'r').read()
        self.assertTrue(fileText.startswith("my"),
                        msg="Test file=%s doesn't start with my" % testFilePath)
```

It may be worth understanding the mechanism by which AppDataPath works. At run time, SconsTools will create two directories:

- unitTestTutorial/data # release data dir
- unitTestTutorial/data/MyPkg # soft link to unitTestTutorial/MyPkg/data

Moreover, when sit_setup was run, it will set the environment variable SIT_DATA. SIT_DATA is a : separated list of paths, the first being the absolute path to unitTestTutorial/data, the second being the absolute path to the data directory of the base release. AppDataPath goes through these paths in order, returning the first match. One thing to note, scons test only runs tests for packages that are part of the working release. It does not run tests for all the packages that are part of the base release. Given this, there is no reason to search the base release, but there should be no harm as well. Harm could conceivably befall a developer who was modifying a test that is checked into an existing package in the base release. Were the developer to change the name of the test file in the working/test release, but not modify the unit test code to use the new name, then AppDataPath would find the old test file in the base release data directory.

External Test Data Location

A directory for test data has been set up here:

`/reg/g/psdm/data_test`

that was created expressly for the purpose of storing test data for the analysis releases. Presently it holds xtc files, and some calibration constant files. We do not want to copy entire xtc files from the experiments into this location as they are to big. We need to select the parts of the xtc file necessary for testing. The current organization of the data_test directory is

data_test /Translator	samples from approximately 80 different xtc files that cover a broad range of psana types and Translator issues. A unit tests will typically work with one of these xtc files at a time.
data_test /multifile	samples from 8 different experiments, suitable for unit tests that work with the psana datasource string specification to work with a set of xtc files from an experiment
data_test /types	soft links to files in data_test/Translator to easily identify a file with a given type
data_test/calib	calibration test data. Same structure as calib directory to an experiment

Keeping the test data files small makes the preparation of xtc test data tedious. One must identifying the parts of the xtc file that you need for your test. An xtc file is comprised of datagrams. Event data are in L1Accept datagrams, but a properly formatted xtc file includes transition datagrams that precede and follow the L1Accept datagrams. At the low most tedious level, making a small xtc file for testing involves identifying the beginning and ending offsets of the datagrams you need to make your file. I'll give an example below of how I do this with tools I've written. Other people are welcome to add examples of using other tools that they find easier to work with.

Once you have prepared some test data, you can either add it to the Translator subdirectory, or the multifile subdirectory, or create a new sub directory, maybe with your package name (like I did when I made the Translator subdirectory). If you want to add it to Translator or multifile, please contact me (davidsch) as these files have specific naming conventions and there are unit tests in the psana_test package that access them. Creating a new subdirectory requires less coordination, however if you think the test data is going to be useful to others, we should work together on it. One of the benefits of using the psana_test package, is I have a mechanism for checking in the md5 checksums of the test data into svn. This allows the unit tests to verify that the test data has not changed.

Making a small xtc test file

This section covers different methods to make small xtc test files. Presently the largest xtc test file in data_test is about 1GB, which is bigger than it needs to be. I think we should be able to keep test files down to 20-100 MB, smaller files mean faster unit tests as well.

Using psana_test

Psana_test includes a library of Python code with a function to copy out a few datagrams from each xtc file for a run. An example of use is

```
import psana_test.psanaTestLib as ptl
ptl.copyToMultiTestDir('cxie9214',63,1,2,'/reg/g/psdm/data_test/multifile/test_012_cxie9214')
```

For experiment cxie9214, run 64, 1 calib cycle, the first 2 events from this calib cycle (for each stream) are copied into xtc files with the same name in the given directory. Moreover a 'index' subdirectory will be made and index files will be written there.

Using xtclinedump

Before writing that function, I would do things by hand. Suppose we want some test data for Epix100aConfig. We know it is somewhere in this file:

`/reg/d/psdm/xcs/xcsi0314/xtc/e524-r0213-s03-c00.xtc`

One of the tools in psana_test is xtclinedump, some documentation is in the psana - Module Catalog. it is a line oriented header dump of xtc files. One can use grep to filter the output so one only sees the datagram headers (which include file offsets in the file) and any xtc headers for a type that has epix as a part of it. Here is a command line that lets me see that there is epix in datagram 5, and to see the offset of where datagram 6 begins. This will be the first part of the xtc I want to save in my small test file. I am also going to want to get some transitions at the end of the file to form a correct xtc file - however this is not necessary, psana can handle xtc files that end abruptly.

```
~/rel2/unitTestTutorial $ xtclinedump xtc /reg/d/psdm/xcs/xcsi0314/xtc/e524-r0213-s03-c00.xtc | grep -i
"dg=\|epix" | head -10
dg=    1 offset=0x00000000 tp=Event sv=      Configure ex=1 ev=0 sec=54754FDF nano=1E31D0E8 tcks=0000000
fid=1FFFF ctrl=84 vec=0000 env=0000161C
xtc d=2  offset=0x00022F4C extent=00108834 dmg=00000 src=01003069,19002300 level=1 srcnm=XcsEndstation.0:
Epix100a.0 typeid=84 ver=1 value=10054 compr=0 compr_ver=1 type_name=Epix100aConfig plen=1083424 payload=0x0B...
dg=    2 offset=0x0012B780 tp=Event sv=       BeginRun ex=0 ev=0 sec=54755EBD nano=35A7112E tcks=0000000
fid=1FFFF ctrl=06 vec=0000 env=000000D5
dg=    3 offset=0x0012B820 tp=Event sv=BeginCalibCycle ex=0 ev=0 sec=54755EBE nano=00D5EE07 tcks=0000000
fid=1FFFF ctrl=08 vec=0000 env=00000000
dg=    4 offset=0x0012C188 tp=Event sv=         Enable ex=0 ev=0 sec=54755EBE nano=0124366D tcks=0000000
fid=1FFFF ctrl=0A vec=0000 env=80000000
dg=    5 offset=0x0012C228 tp=Event sv=       L1Accept ex=1 ev=1 sec=54755EBE nano=05EE73D6 tcks=005094A
fid=144F9 ctrl=8C vec=146F env=00000003
xtc d=2  offset=0x0012C264 extent=0010A454 dmg=00000 src=01003069,19002300 level=1 srcnm=XcsEndstation.0:
Epix100a.0 typeid= 1 ver=1 value=10001 compr=0 compr_ver=1 type_name=Xtc
xtc d=3  offset=0x0012C278 extent=0010A440 dmg=00000 src=01003069,19002300 level=1 srcnm=XcsEndstation.0:
Epix100a.0 typeid=75 ver=2 value=2004B compr=0 compr_ver=2 type_name=EpixElement plen=1090604 payload=0x00...
dg=    6 offset=0x00266284 tp=Event sv=       L1Accept ex=1 ev=1 sec=54755EBE nano=06EE644D tcks=0050974
fid=144FF ctrl=8C vec=1475 env=00000003
xtc d=2  offset=0x002662C0 extent=0010A454 dmg=00000 src=01003069,19002300 level=1 srcnm=XcsEndstation.0:
Epix100a.0 typeid= 1 ver=1 value=10001 compr=0 compr_ver=1 type_name=Xtc
```

so if I copy bytes [0,0x00266284) I will get up to the first datagram with an EpixElement in it. Next I take a look at just the datagram headers at the end of the file:

```
$ xtclinedump dg /reg/d/psdm/xcs/xcsi0314/xtc/e524-r0213-s03-c00.xtc | tail -5
dg= 1204 offset=0x5C225D2C tp=Event sv=       L1Accept ex=1 ev=1 sec=54755EFA nano=01E1D524 tcks=00506D4
fid=1993E ctrl=8C vec=3089 env=00000003
dg= 1205 offset=0x5C35FDE8 tp=Event sv=       L1Accept ex=1 ev=1 sec=54755EFA nano=04DB98AE tcks=0050974
fid=19950 ctrl=8C vec=308F env=00000003
dg= 1206 offset=0x5C499EA0 tp=Event sv=        Disable ex=0 ev=0 sec=54755EFA nano=172A7720 tcks=0000000
fid=1FFFF ctrl=0B vec=0000 env=00000000
dg= 1207 offset=0x5C499F40 tp=Event sv=  EndCalibCycle ex=0 ev=0 sec=54755EFA nano=17A3F59A tcks=0000000
fid=1FFFF ctrl=09 vec=0000 env=00000000
dg= 1208 offset=0x5C499FE0 tp=Event sv=         EndRun ex=0 ev=0 sec=54755EFA nano=19054BF0 tcks=0000000
fid=1FFFF ctrl=07 vec=0000 env=00000000
```

So I also want bytes [0x5C499EA0, end of file). By using the unix command ls -l I can see the file length is 1548329088.

Part of the psana_test package is a library function to copy out portions from a source file to a destination file. From a Python shell, I do

```
In [9]: import psana_test.psanaTestLib as ptl
In [10]: ptl.copyBytes('/reg/d/psdm/xcs/xcsi0314/xtc/e524-r0213-s03-c00.xtc', [[0,0x00266284],[0x5C499EA0,
1548329088]], '/reg/g/psdm/data_test/Translator/test_089_xcs_xcsi0314_e524-r0213-s03-c00.xtc')
copying 2 sets of bytes:  [0,2515588)  [1548328608,1548329088)  from src=/reg/d/psdm/xcs/xcsi0314/xtc/e524-
r0213-s03-c00.xtc to dest=/reg/g/psdm/data_test/Translator/test_089_xcs_xcsi0314_e524-r0213-s03-c00.xtc
```

To create the test file.

After creating it, I take a look at it in the data_test directory and make sure the permissions look good for psrel, basically that it is world readable. I also remove my write permission to make it read only.

Using psana

Short xtc file can also be created using psana module `PSXtcOutput.XtcOutputModule` as shown in the configuration file `psana-reduce-xtc.cfg`

```
[psana]
files = exp=xcsi0314:run=213
skip-events = 5
events = 10

modules = PSXtcOutput.XtcOutputModule

[PSXtcOutput.XtcOutputModule]
dirName  = ./
```

This procedure can be launched by the command

```
psana -c psana-reduce-xtc.cfg
```

or with additional parameters

```
psana -c psana-reduce-xtc.cfg -s 5 -n 10 exp=xcsi0314:run=213
```

which creates reduced xtc file `e524-r0213-s00-c00.xtcf` in local directory. In this example the number of skipped and selected events is controlled by the parameters `skip-events` and `events`, respectively. More sophisticated event selection can be done with psana filtering modules included in the list of modules before `PSXtcOutput.XtcOutputModule`, for example

```
[ImgAlgos.TimeStampFilter]
filterIsOn  = yes
print_bits  = 31
tsinterval  =  2012-02-06 153629 / 2012-02-06 153630

[ImgAlgos.EventNumberFilter]
filterIsOn    = yes
evtstring     = 10,20,30,90
print_bits    = 6
```

See psana - Module Catalog for more detail on the filtering modules. Information on PSXtcOutput can be found at psana - Reference Manual. Note, xtc files created in this fashion are xtc files that "end abruptly", that is the last datagram will be a L1Accept as opposed to the transition sequence EncCalibCycle, EndRun.

Make a Unit Test to Process Output

Suppose you have prepared a test file. Sometimes we are testing the output of Psana modules. It can be awkward to get that output into a testable form. One thing I do is run psana on test data using the Python subprocess package. I gather the stdout and stderr, looking for errors from psana as well as checking the module output. Sometimes I parse the module output looking for expected strings, sometimes I save the md5 of the entire output and test against that. Another thing you might want to do before running the test is check if the xtc file has changed.

A useful psana module in the psana_test package is psana_test.dump (see psana - Module Catalog). This dumps information on all the types and data psana sees. We first run it on the above file and inspect the output. After satisfying ourselves that it is correct, we make a unit test as follows. We run the original xtc and the dump output through md5sum. We record these, and then write a unit test that does both md5 checks and compares them.

The commands we do to prepare our test are

```
psana1302:~/rel/unitTestTutorial $ md5sum  /reg/g/psdm/data_test/Translator/test_089_xcs_xcsi0314_e524-r0213-
s03-c00.xtc
58357b0bf7b6d630c4c2e9633d5b4ca2  /reg/g/psdm/data_test/Translator/test_089_xcs_xcsi0314_e524-r0213-s03-c00.xtc

psana1302:~/rel/unitTestTutorial $ psana -m psana_test.dump /reg/g/psdm/data_test/Translator
/test_089_xcs_xcsi0314_e524-r0213-s03-c00.xtc
# this has a lot of output that we inspect

psana1302:~/rel/unitTestTutorial $ psana -m psana_test.dump /reg/g/psdm/data_test/Translator
/test_089_xcs_xcsi0314_e524-r0213-s03-c00.xtc | md5sum
cc830e794d292eb6cd1414fdad1e2b10  -
```

Our unit test looks like:

```
    def test_dump_on_test89(self):
        import os
        import psana_test.psanaTestLib as ptl

        test_data = '/reg/g/psdm/data_test/Translator/test_089_xcs_xcsi0314_e524-r0213-s03-c00.xtc'
        assert os.path.exists(test_data), "Test data: %s not found" % test_data
        md5sumXtc = '58357b0bf7b6d630c4c2e9633d5b4ca2' # from running md5sum on file
        md5sum = ptl.get_md5sum(test_data)    # demonstrate ptl function to run md5sum on file
        self.assertEqual(md5sum, md5sumXtc, msg="md5sum of xtc file has changed.\nold=%s\nnew=%s" % (md5sumXtc,
md5sum))
        cmd = 'psana -m psana_test.dump %s | md5sum' % test_data  # pipe dump through md5sum, compare to
previous
        md5sumOnDump = 'cc830e794d292eb6cd1414fdad1e2b10'  # from previous testing
        md5out,stderr = ptl.cmdTimeOut(cmd)   # demonstrate ptl function to capture input/output
        md5out = md5out.split(' -')[0].strip()  # the output of md5sum ends with -, discard it
        self.assertEqual(stderr,'',msg="There were errors in cmd=%s\nstderr: %s" % (cmd,stderr))
        self.assertEqual(md5out.strip(),
                         md5sumOnDump,
                         msg="md5sum for psana_dump is wrong.\nold='%s'\nnew='%s'\nRun cmd=%s and investigate"
% (md5sumOnDump, md5out, cmd))
```