A Framework for Improved Regression Testing Based Upon CTest and CDash

Release 0.00

Bradley C. Lowekamp[1] and David T. Chen[1]

July 31, 2009

[1]National Library Of Medicine

Abstract

The existing CMake and CTest environment provides an excellent platform for execution and submission of software quality tests. However, there is little support or aid for regression type tests provided to lessen the burden of writing tests and verifying the results. We propose an additional facility to compare measurements generated by a software quality test to a based-line result. Our approach enables the ability to use multiple images as measurements, along with integer, floating point and string values for regression tests. We describe a new testing class framework which provides the functionality of reading, writing, comparing and differentiating named measurements.

Latest version available at the Insight Journal [http://hdl.handle.net/1926/???]

Contents

We propose an improved process for running regression tests which fully integrates with CTest and CDash and leverages the existing XML based submittion process. Adoption of this framework will enable the improvement of regresstion tests by providing the structure for comparing numerous measurement values in the CTest process. The provided base classes to the XML measument system are generic enough to be used with any project, while the implementation of a derived class extends the measuments to specific Insight

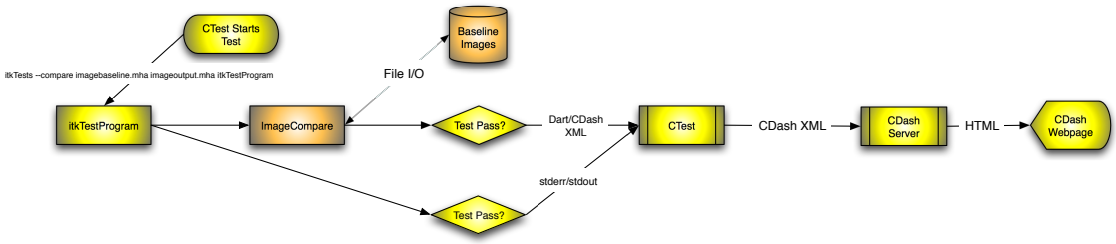


Figure 1: Existing Insight Testing data flow, the orange is used to hightlight the difference.

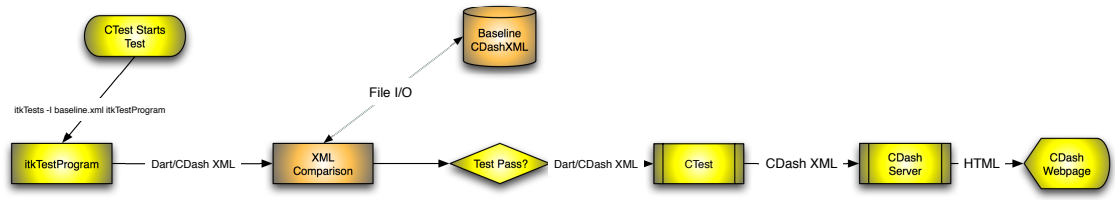


Figure 2: The proposed data flow where XML is used for named measurements and a base-line.

Toolkit data objects. In this paper we will provide some background, describe the implementation of our testing framework, illustrate the framework with some examples and then discuss the positives and negatives of our approach.

1 Background

Often there are a variety of types of tests that need to be performed to evaluate software quality in the extreme programming environment. Among these tests are smoke, black box, white box, and regression tests [2]. Empowered by the integration of CTest with CMake, the smoke tests of configuration, compilation, and program execution for a project are easily accomplished with CMake scripts and submitting the results to a CDash server for viewing, aggregation and archival. For compilation or build tests the output is carefully parsed and converted into CDash's XML schemata (`http://public.kitware.com/Wiki/CDash:XML`) to be submitted the server. The process by which CTest executes program tests is similar to build tests.

Each test defined in a CMake script is run and the standard output, exit value, and completion status along with the executed command line and execution time are submitted to CDash the server as XML. The CTest environment enables a basic smoke program test to verify that a program runs, with a successful completion status and return value. If a regression-type test, which would verify its output and results, were needed, then the test program must internally provide support for evaluating the results. This restriction may place an excessive burden on the developers of the regression tests by requiring excessive amounts of time and effort to internally verify the correctness of the execution.

Independently of CMake or CTest, the Insight Toolkit provides a common testing main function (see "itkTestMain.h"), which provides support for validating a single image per test and setting global toolkit parameters such as the number of threads. There is still the burden of writing code to compare commonly used types, such as double, `itk::Vector` or `itk::ImageRegion`, in both tolerant and exact methods. Frequently the expected values will be hard-coded into the test program, making it valid for only a single input. An interesting feature of the Insight Toolkit's testing main function is when the test image and the base-line image do not match. In this situation XML will be printed to the standard output, and then CTest

will transmit it to the CDash server (See figure 1). The generated XML will display the base-line image, the test image, and then the difference between the two. This feature was the inspiration for our framework.

2 Implementation of Testing Framework Extension

We propose that program tests generate named XML measurement, which are then compared to base-line XML for verification for regression tests (See figure 2).

If we examine the XML for the CDash (http://public.kitware.com/Wiki/CDash:XML), specifically the file which describes the valid XML for tests "ValidationSchemata/Text.xsd", we see the `NamedMeasurement` tag. This tag applies to the fields that are displayed on the CDash's dashboard for each test. NamedMeasurement can show a variety of types with the `type` attribute. Previously Dart, the predecessor of CDash, explicitly defined the following types: `numeric/integer`, `numeric/float`, `numeric/double`, `numeric/boolean`, `text/string`, `text/plain`, `image/png`, `image/jpeg`.

Our testing framework produces and analyzes the XML tags of `DartMeasurements` or `NamedMeasurements`. The generated tags of a test are compared against a separate XML base-line file containing verified results. Any significant differences detected are printed to the standard output so that CTest can report it to a CDash server. To produce a base-line file, the output of the test should be carefully verified manually for correctness and accuracy. What follows is the file "tfRegressionTestTest2.xml" which is a sample base-line used in the testing of the framework:

```
<?xml version="1.0" encoding="US-ASCII"?>
<!-- created on victoria at Thu Apr 30 14:35:53 2009 -->
<output>
<DartMeasurement name="TextString" type="text/string">string text value</DartMeasurement>
<DartMeasurement name="TextPlain" type="text/plain">this is
 some plain text
</DartMeasurement>
<DartMeasurement name="NumericInteger" type="numeric/integer">0</DartMeasurement>
<DartMeasurement name="NumericFloat" type="numeric/float">1.1</DartMeasurement>
<DartMeasurement name="NumericDouble" type="numeric/double">2.222222222</DartMeasurement>
<DartMeasurement name="NumericBoolean" type="numeric/boolean">1</DartMeasurement>
</output>
```

An object oriented programing approach was taken to implement the interface for program tests with a class called `testutil::RegressionTest`. This class provides the functionality to help generate and compare measurements. It does this by operating in two modes: compare mode and non-compare mode. In non-compare mode the measurements are simply outputted to a file. In compare mode a file of the base-line measurements is loaded, and then the test measurements generated are compared and output by an extendible visitor design pattern [1]. Some highlights of the `RegressionTest` interface follow:

```
void SetRelativeTolerance(double tol);
double GetRelativeTolerance(void) const;

int MeasurementTextString(const std::string &str, const std::string &name);
int MeasurementTextPlain(const std::string &str, const std::string &name);

int MeasurementNumericInteger(long i, const std::string &name, bool tolerant = false);
int MeasurementNumericFloat(float f, const std::string &name, bool tolerant = true);
```

tfRegressionTestTest3 Passed

Execution Time (s)	0.00 (mean:0.00 std:0.00)
Command Line	/playpen/blowek1/IJTestingFramework/utilities/TestingFramework/tests/tfTests2 tfRegressionTestTest3 -I /nfs/mead/Users/blowek1/src/papers/IJTestingFramework/Source/utilities/TestingFramework/tests/xml/tfRegressionTestTest3.xml
Completion Status	Completed
Test TextString	difference string text value
Baseline TextString	string text value
Difference TextPlain	1c1 < this is --- > this is different
Test NumericInteger	1
Baseline NumericInteger	0
Test NumericFloat	2.2
Baseline NumericFloat	1.1
Test NumericDouble	4.44444
Baseline NumericDouble	3.333333333333333
Test NumericBoolean	0
Baseline NumericBoolean	1

Figure 3: This is the output of a test displayed on a CDash server designed to show all the differences for the basic measuement types.

```
int MeasurementNumericDouble(double d, const std::string &name, bool tolerant = true);
int MeasurementNumericBoolean(bool b, const std::string &name, bool tolerant = false);

/// the function to perform the test and create measurements
int Test(int argc, char *argv[]) = 0;

/// This should be passed the command line arguments. It will the
/// call ParseAndRemoveArguments to extract the input and output
/// files. Next it will try to open them, read the input file, and then invoke Test
/// with the remaining arguments. Finally the files will be
/// closed.
///
/// If in compare mode, the the return value is the number of
/// measurements that did not match. Otherwise it is the return
/// value of test.
int Main(int argc, char *argv[]);
```

The command line arguments that the `RegressionTest::Main` method knows are "-I infilename.xml" and "-O outfilename.xml". If "-I" is not specified then the regression test will run in non-compare mode. When the "-O" argument is specified the test will direct the output to an XML file. Otherwise the test will direct the output of the difference visitor to the standard output so that CTest will parse and submit the XML to a CDash server.

$$\mathbf{u} \approx \mathbf{v} \iff (|\mathbf{u}-\mathbf{v}| \leq \varepsilon \cdot \mathbf{u}) \vee (|\mathbf{u}-\mathbf{v}| \leq \varepsilon \cdot \mathbf{v}) \tag{1}$$

The visitor design pattern was used for the purposes of comparing two measurements and outputting the difference between measurements. The standard compare visitor performs an exact match for text based measurements, but it can electively perform an approximate comparison with a symmetric relative tolerance (see equation 1) for numeric types. For numeric and string text types, the standard difference visitor outputs both the base-line and then test measurements. However, for plain text a "diff" algorithm is used based on the longest common subsequences of lines. Using this pattern enables custom comparisons of existing types or of custom `type` attributes. Figure 3 show a CDash page where the difference output of many different types was performed.

itktfMeasurementFileImageInsightTest2 Passed

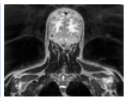
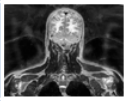
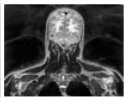
Difference Different Images PNG	
Baseline Different Images PNG	
Test Different Images PNG	
Difference Different Images PNG to JPEG	
Baseline Different Images PNG to JPEG	
Test Different Images PNG to JPEG	
Execution Time (s)	0.00 (mean:0.00 std:0.00)
Command Line	/playpen/blowek1/IJTestingFramework/itktfMeasurementFileImageInsightTest2 -I /nfs/mead/Users/blowek1/src/papers/IJTestingFramework/Source/xml/itktfMeasurementFileImageInsightTest2.xml
Completion Status	Completed
ImageError Different Images PNG	2407
ImageError Different Images PNG to JPEG	7694

Figure 4: The CDash display of single test which has two test images not matching base-lines.

3 Example Tests

To customize the testing framework, classes were derived from `RegressionTest` and the two visitor classes. To these classes we added functionality that enables the uses of custom measurements based on the Insight Toolkit's commonly used types and object, including `itk::Size`, `itk::Index`, `itk::Region`, `itk::Point`, `itk::Vector`, `itk::Matrix` and image files. Most of these measurements are composite types, meaning they will output multiple XML tags. The ability to use images as measurements is perhaps the most important feature of our framework. The functionality mimics"TestMain", which is based on `itk::DifferenceImageFilter`. Figure 4 illustrates an example of utilizing multiple image file base-lines.

3.1 ShiftScaleInPlaceImageFilterTest

The ShiftScaleInPlaceImageFilter test was ported to this framework from the existing Insight test locating in "Testing/Code/BasicFilters". We chose this test because the test had been passing even though the execution of the filters had changed. To verify the pipeline execution a string measurement was created which contains what was previously output to `std::cout`. By capturing the output and using it as a string measurement, we are able to compare the output each time the test is run. The `FilterWatcher` class was modified to direct the output to the same `stringstream` as the measurement. Doing so enables us to verify which filters are run on each update of the pipeline.

Previously some get member functions were invoked with the output going to `std::cout`. These invocations were changed to individual numeric integer measurements. Also previously verification of the pixel's values for the filter was performed via hard coded checks with each check producing a custom error message if it failed. Since the pixel types for the filters are `char`, each one of these checks was also converted into numeric integer measurements. This change resulted in over 35 lines of unnecessary code being removed from the test.

3.2 PipeToXML

"PipeToXML" is a simple program for converting existing tests to the new XML wrapped framework. It executes another program and converts the standard output into a string that can be used as a single plain text argument. By wrapping the program's output as a named measurement, existing tests can also have their outputs compared and any differences detected and displayed.

4 Discussion

There are strengths and weaknesses to our approach. Unfortunately, we have added another interface and more code to an already complex system, which further steepens the learning curve. The separation of the base-line XML, may need more data stored to verify the tests. With the ease of producing more test values, additional care must be taken to verify that the measurements in the XML are correct and portable.

On the positive side our approach separates the base-line from the test data. This separation enables a single test to be run with multiple inputs, and so there can be multiple correct base-lines to match those inputs. Because the XML output can specify multiple images, a single test can have multiple images verified for correctness. As shown in the first example, this framework can significantly reduce the amount of code required to perform common testing tasks. More named measurement types that can be displayed on CDash can make understanding and fixing failed tests easier. Also by redirecting some output to a separate string measurement, detecting when the execution of a test changes can be more apparent.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995. 2

[2] K. Martin and B. Hoffman. *Mastering CMake: Updated for CMake 1.8*. Kitware, Inc. ISBN 1-930934-09-2, http://www.kitware.com, 2003. 1