Testing with rostest and GTest

Christoph Steup

October 20, 2015

Table of contents

Some Examples

- Gripper Integration of the Last Meeting
  - Interface changed
  - Hardware configuration changed
  - Developer was away
  - A lot of guessing

Some Examples

- Gripper Integration of the Last Meeting
  - Interface changed
  - Hardware configuration changed
  - Developer was away
  - A lot of guessing
- Ariane 5
  - 370 million US$ damage
  - Overflow of inertial system led to deactivation
  - Direct usage of Ariane 4 software
  - Lack of exception handling in Ariane 4 code
  - No test of Ariane 4 code within Ariane 5

Regressions

Definition: Regression

Violation of a specific part of the expected behaviour of a piece of Software after a code change.

Regressions

Definition: Regression

Violation of a specific part of the expected behaviour of a piece of Software after a code change.

- Happens a lot in software development
- Tightly coupled code is more prone for regression
- If detected timely can be fixed easily
- Hard to find after a lot of code changes

Regressions

Definition: Regression

Violation of a specific part of the expected behaviour of a piece of Software after a code change.

- Happens a lot in software development
- Tightly coupled code is more prone for regression
- If detected timely can be fixed easily
- Hard to find after a lot of code changes

Solution

Automatically test software on each code change and report results.

Unit Test: Definition

Definition: Unit Test

Test of a specific expected property of a single small piece of software, which can not be further decomposed.

Unit Test: Definition

Definition: Unit Test

Test of a specific expected property of a single small piece of software, which can not be further decomposed.

Benefits

- Eases integration
- Finds bugs during development
- Documents API
- Checks refactoring

Unit Test: Examples

Unit to Test

```
1  int Factorial(int n); // Returns the factorial of n
```

Unit Test: Examples

Unit to Test

```
1  int Factorial(int n); // Returns the factorial of n
```

GTest Primer Example

```
1  // Tests factorial of 0.
2  TEST(FactorialTest, HandlesZeroInput) {
3    EXPECT_EQ(1, Factorial(0));
4  }
5
6  // Tests factorial of positive numbers.
7  TEST(FactorialTest, HandlesPositiveInput) {
8    EXPECT_EQ(1, Factorial(1));
9    EXPECT_EQ(2, Factorial(2));
10   EXPECT_EQ(6, Factorial(3));
11   EXPECT_EQ(40320, Factorial(8));
12 }
```

Example Setup

GitHub Project

http://github.com/steup/Ros-Test-Example

Example Setup

GitHub Project

http://github.com/steup/Ros-Test-Example

Download Command

```
git clone https://github.com/steup/Ros-Test-Example.git
```

Example Setup

GitHub Project

http://github.com/steup/Ros-Test-Example

Download Command

```
git clone https://github.com/steup/Ros-Test-Example.git
```

Compile Command

```
cd Ros-Test-Example && catkin_make
```

Example Setup

GitHub Project

http://github.com/steup/Ros-Test-Example

Download Command

```
git clone https://github.com/steup/Ros-Test-Example.git
```

Compile Command

```
cd Ros-Test-Example && catkin_make
```

Create Documentation

```
cd src/cars && Doxygen common/Doxyfile
```

Accessing Documentation

Webbrowser on src/cars/doc/html/index.html

Run the Simulation

Setup Workspace

```
. devel/setup.bash
```

Run the Simulation

Setup Workspace

```
. devel/setup.bash
```

Basic Simulation

A simple car simulation using *R*VIZ to show the current state. The cars are remotely controlled using individual nodes. The simulation node handles new cars, state updates and collision checking. The simulation's speed can be set using the `frameRate` parameter.

Run Simulation

```
roslaunch cars run.launch
```

Adding Cars

Adding Cars

Each car has its own unique identifier. The first car is always `"BEG_IN_001"`. The car's id can be set using the `numberPlate` parameter.

Add Car

```
roslaunch cars car.launch numberPlate:=<something unique>
```

Unit Tests

Running Unit Tests

Unit test are integrated within catkin using the `CMakeLists.txt`.

Run Unit Tests

```
catkin_make run_tests
```

Result of Unit Tests

Unit tests report their result in the console after the tests are finished. Additionally, a XML-file containing the results can be found in the `build`-folder of the workspace. In this case: `build/test_results/cars/gtest_cars_test.xml`

rostests

Running rostest-Tests

rostest-Tests are integrated using *roslaunch*'s XML notion. They are executed using `rostest`. They spawn their own *roscore* on a different port to avoid interfering with an existing instance.

Run Simulation Test

```
rostest cars simHz.test
```

Run Car Test

```
rostest cars carHz.test
```

Result of rostest-Tests

*r*ostest-Tests report their result in the console after the tests are finished. However, this is only a brief overview. The full result is an XML-file in the user's `.ros`-folder, in this case: `.ros/test_results/cars/...`

Writing *G*Test Tests

Test Environments

The *G*Test Framework needs the declared tests to be be executed in a program using the following functions :

`InitGoogleTest(...)` : This function parses testing relevant command parameters and sets up the environment

`RUN_ALL_TESTS()` : This function runs all specified tests, outputs the results and returns 0 on success

Writing *G*Test Tests

Test Environments

The *G*Test Framework needs the declared tests to be be executed in a program using the following functions :

`InitGoogleTest(...)` : This function parses testing relevant command parameters and sets up the environment

`RUN_ALL_TESTS()` : This function runs all specified tests, outputs the results and returns 0 on success

Defining Tests

A test is always associated to a test suite and has a unique name. There exist two specifications:

`TEST(Suite, test)` : Declares an isolated test named `test` associated to the test suite `suite`.

`TEST_F(Suite, test)` : Declares a test `test` using a Fixture named after the test suite `suite`.

ASSERT vs. EXPECT

Basic Test Operations

The following test operations are automatically recognized and evaluated by *G*Test:

`ASSERT_TRUE()` : If passed parameter evaluates to `false` stops execution of the test and fails it

`EXPECT_TRUE()` : If passed parameter evaluates to `false` continues with the test, but test fails anyway

`..._EQ()` : Test for equality of items (typically using ==)

`..._NE()` : Test for inequality of items (typically using !=)

`..._LT()` : Test for if first item is less then second item (typically using <=)

`..._STREQ()` : Test for equality of two C string (`(const) char*`)

`..._STRCASEEQ()` : Test for equality of two C string ignoring case (`(const) char*`)

Fixtures

Definition

A Fixture is class used as an environment template for test execution. It provides a repeatedly created similar environment for each test, isolating the individual test.

Fixtures

Definition

A Fixture is class used as an environment template for test execution. It provides a repeatedly created similar environment for each test, isolating the individual test.

```
1  class CarTestSuite : public ::testing::Test{
2      ros::ServiceServer mServer;
3      bool callback( cars::NewCar::Request& req,
4                     cars::NewCar::Response& res){
5        res.success = true;
6        return true;
7      }
8    public:
9      CarTestSuite() {
10       ros::NodeHandle n;
11       mServer = n.advertiseService( "carRegistry",
12                                     &CarTestSuite::callback,
13                                     this);
14     }
15 };
```

Advanced Tests

Advanced Test Operations

The general test operation cannot grasp all language constructs that are test-worthy. One very important example is exception handling. Fortunately, the advanced test operations of *G*Test already contain appropriate operations:

`..._THROW(statement, exception_type)` : fails if the `statement` does not throw an exception of type `exception_type`

`..._NO_THROW(statement)` : fails if the `statement` throws exception

Advanced Tests

Advanced Test Operations

The general test operation cannot grasp all language constructs that are test-worthy. One very important example is exception handling. Fortunately, the advanced test operations of *G*Test already contain appropriate operations:

`..._THROW(statement, exception_type)` : fails if the `statement` does not throw an exception of type `exception_type`

`..._NO_THROW(statement)` : fails if the `statement` throws exception

Many More

There exist many more specialized test operations in *G*Test to be presented in this talk. Have a look yourself, when you feel the need to!

catkin Integration

CMakeList.txt

Typically, there will be one *G*Test program per packet. This program needs to be registered with Catkin through the `CMakeList.txt` using `catkin_add_gtest`. The signature of the statement is similar to `add_executable`. However, if libraries need to be linked to the test program additional work needs to be done. As a template the following CMake snippet may be used:

```
1 ## Add gtest based cpp test target and link libraries
2 catkin_add_gtest(${PROJECT_NAME}-test src/Test.cpp)
3 if(TARGET ${PROJECT_NAME}-test)
4   target_link_libraries(${PROJECT_NAME}-test
5                         ${catkin_LIBRARIES}
6                         ${PROJECT_NAME})
7 endif()
```

Writing a rostest-Test

*r*ostest *r*oslaunch XML-Files

*r*ostest uses the *r*oslaunch XML description. The tag `<test>` is especially reserved for *r*ostest and is ignored by *r*oslaunch. `<test>` is similar to a node accepting `<param>` and being grouped in a namespace. The code executed by such a node needs to be a set of test suites implemented using *G*Test. It may be integrated in your `CMakeLists.txt` using `add_rostest_gtest(name, testFile, GTestSource)`.

```
1  if(CATKIN_ENABLE_TESTING)
2    find_package( rostest REQUIRED )
3    add_rostest_gtest( ${PROJECT_NAME}_test_<name>
4                       launch/<name>.test src/Test_something.cpp )
5    target_link_libraries( ${PROJECT_NAME}_test_<name>
6                           ${catkin_LIBRARIES} ${PROJECT_NAME} )
7  endif()
```

```
1  <launch>
2    <test test-name="<project>_test_<name>"
3          pkg="<project>" type="<project>_test_<name>" />
4  </launch>
```

HzTest

Pre-defined Publish Rate Test

*r*ostest already provides a pre-build test. The so-called *hz*test tests publish frequency of nodes. It is very configurable using different parameters like:

topic : Topic to listen to (namespace aware)

hz : target frequency of the publisher

hzerror : allowed difference between measured frequence and hz value

test_duration : how long the test should run

HzTest Example

```
1  <launch>
2    <include file="$(find␣cars)/launch/sim.launch">
3      <arg name="frameRate" value="0.1"/>
4    </include>
5    <test test-name="hztest_test" pkg="rostest" type="hztest"
6          name="simHz" >
7      <param name="topic" value="roads" />
8      <param name="hz" value="10" />
9      <param name="hzerror" value="1" />
10     <param name="test_duration" value="5.0" />
11   </test>
12 </launch>
```

Next Time

- Testing interfaces
- Mocking interfaces using *GMock*
- Using *selftest* to test on run time
- Using *git* to find regressions