Testing and Static Analysis

① Automatic Testing with ROS

gtest rostest

★ Why Automatic testing

⇒ Consider the following scenario:

⇒ You are fixing a bug

⇒ You have created a correction, and test the patch on your own computer.

⇒ If you automate the test, and submit it together with the patch, you will make it much easier for other developers to avoid reintroducing the bug in the first place.

⇒ And the continuous integration service will be able to detect such regressions automatically.

⇒ To get the benefits, some investment is necessary:

Development cost

⇒ You need to develop a test, which sometimes may be difficult or costly.

⇒ Sometimes it might also be nontrivial.

Maintenace cost

⇒ Regression tests and other automatic tests need to be maintained.

⇒ When the design of the component changes, a lot of tests become invalidated

★ Testing tools in Ros

⇒ For testing C++ code at the library level (at the C++ API level) Google test framework gtest should be used.

⇒ For testing at the ROS node level, involving ROS as a communication middleware, rostest is used together with gtest.

⇒ It is key that the tests are not only automatic, but are integrated in the project scripts, so that they are run by the build and test infrastructure, whenever the project is being tested.

⇒ To run the tests, you will need catkin/roslaunch integration

⇒ This may involve introducing a build dependency on rostest in package.xml and including a launcher for the test in the test file.

★ Levels of Testing

Level 1. Library unit test

Level 2. ROS node unit test

{ node unit tests start up your node and test its external API, i.e. published topics, subscribed topics, and services.

Level 3. ROS nodes integration / regression test

Integration tests start up multiple nodes and test that they all work together as expected.

Debugging a collection of ROS nodes is like debugging multi-threaded code: it can deadlock, there can be race conditions, etc...

② Gtest

⇒ Gtest has been converted to a rosdep and is available in ros_comm.

⇒ By convention, test programs for a package go in a test subdirectory.

```
1 // Bring in my package's API, which is what I'm testing
2 #include "foo/foo.h"
3 // Bring in gtest
4 #include <gtest/gtest.h>
5
6 // Declare a test
7 TEST(TestSuite, testCase1)
8 {
9 <test things here, calling EXPECT_* and/or ASSERT_* macros as needed>
10 }
11
12 // Declare another test
13 TEST(TestSuite, testCase2)
14 {
15 <test things here, calling EXPECT_* and/or ASSERT_* macros as needed>
16 }
17
18 // Run all the tests that were declared with TEST()
19 int main(int argc, char **argv){
20   testing::InitGoogleTest(&argc, argv);
21   ros::init(argc, argv, "tester");
22   ros::NodeHandle nh;
23   return RUN_ALL_TESTS();
24 }
```

Basic Structure of gtest in ros

Note: You need to initialize ROS if your tests are using ROS.

★ Test naming Conventions

⇒ Each test is a "test case," and test cases are grouped into "test suites."

- Test suites are **CamelCased**, like C++ types
- Test cases are **camelCased**, like C++ functions

★ Handling Exceptions

⇒ If you're testing code that can throw exceptions, then you need to try/catch them yourself, and then use the ADD_FAILURE and/or FAIL macros as appropriate.

⇒ ADD_FAILURE records a non-fatal failure, while FAIL records a fatal failure.

★ Package.xml

```
<package format="2">
  <test_depend>rosunit</test_depend>
</package>
```

★ CMakeLists.txt

```
## Add gtest based cpp test target and link libraries
catkin_add_gtest(${PROJECT_NAME}-test test/talker_tests.cpp)
if(TARGET ${PROJECT_NAME}-test)
  target_link_libraries(${PROJECT_NAME}-test ${PROJECT_NAME})
endif()
```

★ Run tests

⇒If your gtest initializes a rosnode via rosinit() you must initialize a roscore separately, then run all tests:

```
roscore
catkin_make run_tests
```

⇒OR to run a specific package simply tab complete to find the specific package tests you'd like to run:

```
roscore
catkin_make run_tests<TAB><TAB>
```

② rostest

⇒Integration test suite based on roslaunch that is compatible with xUnit frameworks.

⇒rostest is an extension to roslaunch that enables roslaunch files to be used as test fixtures.

★ Writing rostest files

⇒ In order to create a rostest, you first need to write some test nodes.

⇒Test nodes are nodes that also execute unit tests.

⇒ rostest files are 100% compatible with roslaunch.

⇒These files generally have a .test or .launch file extension.

⇒ It's highly recommended that you embed tests within your roslaunch files in order to verify that they are functioning properly.

⇒The difference between rostest and roslaunch is that rostest also processes <test> tags within your XML file. These <test> tags specify test nodes to run.

● rostests in CMakeLists.txt

Using the ROS-specific cmake build commands is highly recommended as it enables you to do useful test commands such as testing all packages that depend on your package.

```
if(CATKIN_ENABLE_TESTING)
  find_package(rostest REQUIRED)
  add_rostest_gtest(tests_mynode test/mynode.test src/test/test_mynode.cpp [more cpp files])
  target_link_libraries(tests_mynode ${catkin_LIBRARIES})
endif()
```

⇒ Here is a very simple `rostest` file, which you'll note is compatible with the roslaunch format:

```
<launch>
  <node pkg="mypkg" type="mynode" name="mynode" />
  <test test-name="test_mynode" pkg="mypkg" type="test_mynode" />
</launch>
```