A Generic Testing Framework for Test Driven Development of Robotic Systems

Ali Paikan, Silvio Traversaro, Francesco Nori, and Lorenzo Natale

Istituto Italiano di Tecnologia (IIT), Genova, Italy
{ali.paikan,silvio.traversaro,francesco.nori,lorenzo.natale}@iit.it

Abstract. This paper proposes a generic framework for test driven development of robotic systems. The framework provides functionalities for developing and running unit tests in a language and middleware independent manner. Tests are developed as independent plug-ins to be loaded and executed by an automated tool. Moreover, a fixture manager prepares the setup (e.g., running robot drivers or simulator) and actively monitors that all the required resources are available before and during the execution of the tests. These functionalities effectively accelerate the development process and cover different levels of robotic system testing. The paper describes the framework and provides realistic examples to show how it has been used to support software development on our robotic platform.

Keywords: Robot testing framework, Unit testing, Test-driven development, Software engineering, Robotics

1 Introduction

Autonomous robots have evolved in complex systems that are increasingly difficult to engineer and develop. A possible approach to tame such complexity is to divide the system into simpler units that are independently developed, tested and integrated at a later stage. Further testing is consequently performed on the whole system; this may trigger re-development or debugging of the individual components in an iterative process. This strategy is known as the test-driven development [3] and it has gained increasing attention as one of the core extreme programming practices. Proper application of this technique requires i) alternating writing tests and developing functional code in small and rapid iterations and ii) executing tests automatically to ensure that modifications to existing code (new components, bug fixes or new features) do not disrupt existing functionalities.

Developers have created varieties of tools for supporting test-driven development. These frameworks usually focus on providing support for a specific programming language or for automating unit test execution. For example JUnit [10] is a unit testing framework for Java based on the xUnit [11] test patterns. AUnit [2] is a set of Ada packages based on the xUnit family of unit testing framework which is intended to make it easy to develop and run unit tests. The framework supports easy composition of sets of unit tests to provide flexibility in determining what tests to run for a given purpose. CppUnit [5] is another implementation of the xUnit pattern which supports writing unit

tests for C as well as C++ with minimal source modification. It also provides a graphical user interface for monitoring test execution and facilities for generating test results in XML format. Google Test [7] is a multi-platform unit testing library for the C++ programming language, based on the xUnit architecture. It allows for developing different types of tests such as unit tests, integration tests and acceptance tests. The framework also includes a graphical test runner that executes the test binaries.

A unit is the smallest possible testable software component. There is some debate about what exactly constitutes a unit test [9]. For robotic systems a testable unit can be a piece of code, a software component, a driver or even a hardware component such as a sensor or an actuator. A robot testing framework should support different levels of tests, i.e.: component interface tests, integration tests, stress tests and ideally system (application) tests [1]. Tests should be performed on the real robots and on simulation (the latter is a fundamental requisite, it gives more control on the environment and allows to perform fully automated testing). This implies that the framework should be able to setup the required resources (i.e., fixture), like, for example, running drivers to control the motors, executing the simulator and/or possible preparatory routines (e.g. for calibration). Equally importantly, the framework should monitor that these resources remain functional during the execution of the tests, offering hooks to handle failure appropriately (i.e. restarting the robot, performing parking routines, etc.). This is a fundamental requirement for robotics which is not supported by the available testing frameworks. Finally, robotic applications rely on various languages, software libraries and middlewares such as YARP [12], ROS [15] and OROCOS [4]. The testing framework should therefore be designed to offer maximum flexibility to the user in terms of language, dependencies and middleware of choice.

This article proposes the Robot Testing Framework (RTF)[1], a generic testing framework for test driven development of robotic systems. The framework provides functionalities for developing and running unit tests in a language and middleware independent way. Individual tests are developed as independent plug-ins (i.e., using scripting languages or built as dynamically loadable libraries) that are loaded and executed by an automated test runner. A fixture manager prepares the setup (e.g., running the robot drivers or the simulator and its interfaces) and actively monitors that all the dependencies for running the tests remain functional during their execution. These functionalities along with other facilities such as the test result collector, result formatter and remote interface allow for rapid development of tests that cover the various testing levels of robotic systems.

2 Robot Testing Framework

The RTF architecture is based on the well–known xUnit test patterns, which includes a test runner, test result formatters and a test fixtures manager. In addition it provides functionalities for defining test cases (i.e. unit tests), suits and assertions. However, to fulfill the requirements of robotic systems, the RTF design provides abstraction levels for the platform (i.e. operating system), the middleware and the programing language.

[1] The source code and documentation of the RTF can be accessed on-line at `http://robotology.github.io/robot-testing/index.html`.

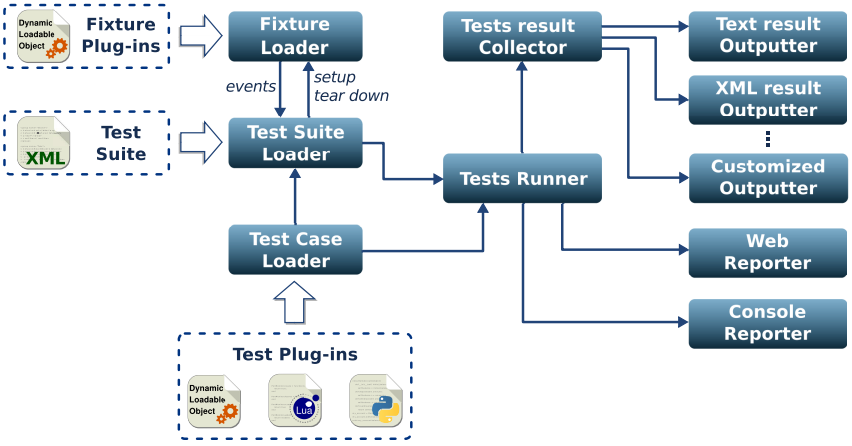

Fig. 1. The architecture of the Robot Testing Framework. Test cases can be developed as independent plug-ins using different programming languages and grouped in test suites which are represented using XML files. A fixture manager can be assigned to every test suite which setups and monitors the fixture during the execution of the tests. The results of the tests can be stored in different formats and monitored from the consoler or a Web browser. See text for details.

Moreover, RTF provides functionalities for managing complex fixtures which support stress testing at the level of individual components (robot hardware like sensors or actuators) as well as integrated (sub) systems.

Fig. 1 demonstrates the main components of the RTF architecture. Test cases can be developed as independent plug-ins using scripting languages or can be built as dynamically loadable libraries. The plug-ins are loaded by the Test Case Loader and are executed by the Test Runner. Test cases can also be grouped in different test suites which are represented using XML. In the latter case, the Test Suite Loader parses the XML file and, using the Test Case Loaders, it loads the corresponding test plug-ins. Each test suit can optionally have a fixture manager which is implemented as a separate plug-in (which is loaded by the Fixture Loader). This fixture plug-in is responsible for setting up the fixture and informing the Test Suite Loader when the fixture fails (e.g. crashes). In this case, the Test Suite Loader restarts the fixture and resumes execution of the remaining test cases. In Section 2.3 we describe an example of a fixture plug-in which has been implemented for the YARP middleware. The result of the tests can be monitored from the console (through the Console Reporter) or remotely from a Web browser (through the Web Reporter). The Test Result Collector allows storing data in different formats (for example Fig. 1 shows two components for storing output in text format or XML, Text result Outputter and XML Result Outputter respectively).

2.1 Middleware and language abstraction

Within the robotic community, researchers have been developing a large number of software components using different robotic middlewares and varieties of software li-

braries. Test cases for those components have the same dependencies. To be as generic as possible, an ideal testing framework should allow unit tests to be developed independently and without posing conflicting constraints due to their dependencies.

Middleware independency in RTF is achieved by allowing unit tests to be implemented as independent plug-ins. Each plug-in can be separately compiled and built with the required libraries. This approach separates dependencies between different tests. The test driven development paradigm requires writing automated tests in small and rapid iterations. Therefore, it is important that test development is done easily and with minimal amount of programming. The RTF provides features that allow to easily develop and run the test case plug-ins for different platforms. Listing 1.1 demonstrates an example that shows how to implement a test plug-in that checks a generic sensor using the YARP middleware [2].

```
#include <TestCase.h>
// include test-dependent libraries ...
#include <yarp/os/all.h>

// prepare the plugin
PREPARE_PLUGIN(SensorTest)

class SensorTest : public RTF::TestCase {
    yarp::os::BufferedPort<yarp::sig::Vector> port;

public:
    // initialization goes here...
    virtual bool setup(int argc, char** argv) {
        RTF_ASSERT_ERROR_IF(port.open("/sensor"),
                            "Failed to open the port!");
        // intialize the rest ...
        return true;
    }

    // test implementation goes here
    virtual void run() {
        RTF_TEST_REPORT("Reading sensors...");
        yarp::sig::Vector *data = port.read();
        RTF_TEST_CHECK(data, "reading sensor error!");
        RTF_TEST_CHECK(data->size() == 6, "sensor size error!");
    }

    // finalization goes here...
    virtual void tearDown(){
        port.close();
    }
};
```

Listing 1.1. Implementing a test case plug-in using C++.

Listing 1.1 demonstrates that a test case in RTF requires only to write a few lines of code. This example is written using the C++ language. A test case is an instance of a class that derives from the abstract `RTF::TestCase` class; the developer is required only to fill the `run()` method with the test functional code. Optionally the test can specify its own context (i.e. a fixture) which can be prepared in the `setup()` method. This method is called to initialize the context before executing the test functional code

[2] In YARP data from generic sensors are published and read from Port objects. BufferedPort is a specialization of Port that support streaming operations.

(`run()`). Similarly the `tearDown()` method is called to terminate the context after running the test. The `PREPARE_PLUGIN` macro adds the required code for generating the plug-in. The RTF also provides macros for error assertion and condition checking. During test execution these macros generate detailed information which are uniformly formatted to be stored or reported to the user.

Unlike most of the available unit testing frameworks, RTF does not enforce adoption of a single programming language. Indeed the framework provides a clean and simple abstraction layer for developing test case plug-ins using different scripting languages. Listing 1.2 shows an example that demonstrates the code for testing an encoder using the Lua [8] programing language and the YARP middleware.

```
1  --import test-dependent libraries ...
2  require("yarp")
3
4  -- initialization goes here...
5  TestCase.setup = function(parameters)
6      TestCase.setName("EncoderTest")
7      port = yarp.BufferedPortBottle()
8      if port:open("/encoder") ~= true then
9          TestCase.assertError("Failed to open the port!")
10     end
11     -- intialize the rest ...
12     return true
13 end
14
15 -- test implementation goes here
16 TestCase.run = function()
17     TestCase.testReport("Reading encoder...")
18     local data = port:read()
19     TestCase.testCheck(data ~= nil, "reading encoder error.")
20 end
21
22 -- finalization goes here...
23 TestCase.tearDown = function()
24     port:close()
25 end
```

Listing 1.2. Implementing a test case plug-in using Lua.

Listing 1.2 is similar to the example of Listing 1.1 for C++. The above script, simply opens a YARP port and checks whether robot encoder data is available or not. The test functional code is implemented in the function `TestCase.run()` whereas initialization and termination of fixtures are done in the functions `TestCase.setup()` and `tearDown()` respectively. Similarly to the C++ example, error assertions and condition checking functions are also available for Lua.

2.2 Test Suites

A test suite is a set of test cases which share the same test fixture [11]. In RTF a set of test cases (plug-ins) can be grouped as a test suite using an XML file and executed using the test runner. This allows the unit tests to be easily organized in different test suites which are easy to maintain and extend. Listing 1.3 shows an example of a test suite (called BasicChecking) that checks a sensor and an encoder using the test cases

described in Section 2.1.

```
<?xml version="1.0" encoding="UTF-8"?>

<suit name="BasicChecking">
    <description> checking the robot </description>
    <environment> simulator </environment>
    <fixture param="--launch sim.xml"> MyFixtureManager </fixture>

    <!-- add the unit plug-ins -->
    <test type="dll" param=""> SensorTest </test>
    <test type="lua" param=""> EncoderTest </test>
    ...
</suit>
```

Listing 1.3. An example of a test suite. The XML file describes the test suite BasicChecking which groups the SensorTest and EncoderTest so that they can be executed sequentially using the Test Runner.

The above example demonstrates a test suite called `BasicChecking`. Two test cases (`SensorTest` and `EncoderTest`) are added to the test suite using the `<test>` tags; the property `type` specifies the type of the plug-in (e.g., dll or lua for C++ and Lua respectively). Each test case can optionally have parameters (i.e. `param="..."`) which can be accessed inside the test's `setup()` method for its initialization. The `<environment>` tag can specify a set of parameters which form a common *environment* for the test cases in the suite. Using these variables test cases can be quickly adapted and re-used for execution in different situations (i.e. real robot versus simulator). As previously discussed a test suite has a fixture manager (specified by the `<fixture>` tag) which prepares the test contexts for the unit tests. Fixture managers are also plug-ins so that different implementations can exist for different platforms (i.e. operating systems and/or middleware). The following section provides more details about the Fixture Manager.

2.3 Fixture Manager plug-in

System tests in robotics require execution of a complex fixture which provides the set of basic functionalities for running the tests (i.e. low-level software components for interfacing with the robot or the simulator, etc.). Importantly the fixture should be monitored during the execution of the tests to ensure robustness against failures. We decided to implement the fixture manager as a separate plug-in to avoid reimplementing or simply interfering with the deployment policy of the robotic framework of choice. Middlewares usually have their own policies for configuring and deploying components and executing applications. For example, Orocos components are compiled as dynamic loadable libraries, configured using XML files and launched using the Orocos component deployer (i.e., `deployer-corba`). ROS components (nodes) are configured using launch files and executed by the `roslaunch` toolset. In the YARP middleware the `yarpmanager` [14] that gives users the possibility to execute and monitor the lifecycle of applications from an XML description file [3]. In short an application XML file

[3] Please refer to `http://wiki.icub.org/yarpdoc/yarpmanager.html` for the structure and syntax of the YARP application description file.

contains the necessary information for configuring and launching the individual executables (called modules in YARP terminology) using the `yarpmanager`. The framework provides a rich set of the functionalities such as launching executables on a cluster of computers, monitoring their execution and recovering programs from failure. Listing 1.4 demonstrates the implementation (pseudo-code) of a fixture manager plug–in which uses the `yarpmanager` libraries to set up and monitor a test fixture.

```
#include <FixtureManager.h>
// include middleware-dependent libraries ...
#include <yarp/manager/manager.h>

// prepare the plug-in
PREPARE_FIXTURE_PLUGIN(MyFixtureManager)

class MyFixtureManager : public RTF::FixtureManager,
                                yarp::Manager {
public:
    virtual bool setup(int argc, char** argv) {
        // parse the arguments, load the application
        // xml file and run all the modules
        yarp::Manager::loadApplication(...);
        RTF_ASSERT_ERROR_IF(yarp::Manager::run(),
                            "failed to run!");
        // monitor the modules execution
        yarp::Manager::enableWatchDog();
        return true;
    }

    virtual void tearDown() {
        yarp::Manager::stop()  // stop the modules
    }

    // this is called from the yarp::Manager
    // if the execution of the modules fails
    virtual void onExecutableFailed(void* which) {
        // inform the test suite/runner that the fixture
        // has been collapsed.
        getDispatcher()->fixtureCollapsed("reason...");
    }
};
```

Listing 1.4. An example of implementing the fixture manager plug-in for the YARP middleware.

As it is shown in Listing 1.4, a fixture manager plug-in (i.e., `MyFixtureManager`) is an instance of a class that derives from `RTF::FixtureManager` and implements the `setup()` and `tearDown()` methods. The test runner calls the function `setup()` before running the test cases in a test suite. In the above example, we use the `yarpmanager` functionalities (implemented in the `yarp::Manager` class) to load an XML application file and run the modules that make the corresponding fixture. When terminating the execution of a test suite, the test runner calls `tearDown()` to tear down the fixture and stop all modules. In the `setup()` method we also enable the `yarpmanager` watchdog functionality; this feature monitors the proper execution of all the modules in the fixture. The `onExecutableFailed()` method is inherited from the `yarp::Manager` class. The failure of one of these modules (which are monitored in the `yarp::Manager`) triggers the `onExecutableFailed()` method; this function notifies the test runner by calling the `fixtureCollapsed()` function. This event-sender function interrupts the test runner which, in turn, tries to restart the fixture

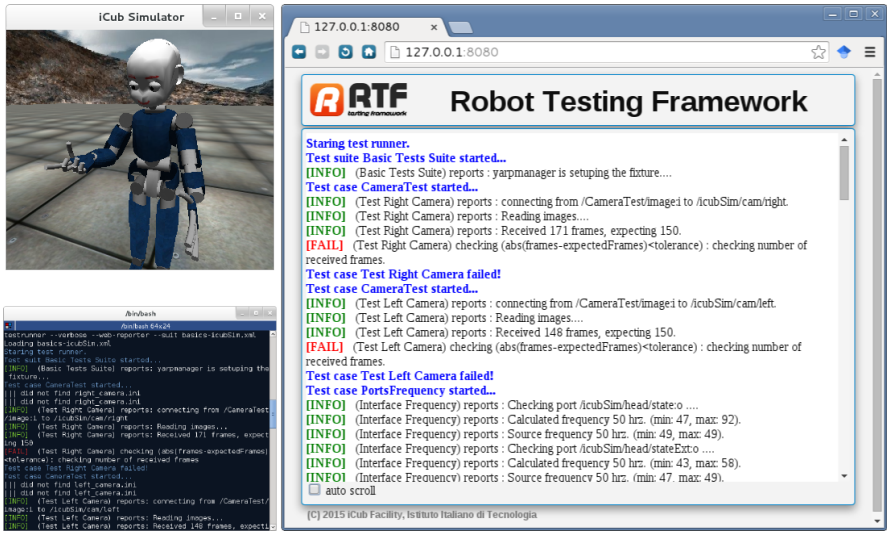

Fig. 2. Testing the iCub robot using its simulator. The RTF test runner was launched from the console (at the left side) and the iCub simulator was automatically launched using the YARP fixture manager. The right side shows how the test results can be monitored from a Web browser.

by calling the `setup()` method. If this operation succeeds the test runner continues execution of the next test case, otherwise, the proper error message is reported.

2.4 Running and monitoring unit tests

As described previously (See also Fig. 1), the test cases (i.e., as shared libraries or scripts) can be uniformly executed using the RTF test runner. The test runner [4] is a multi-platform tool that loads and runs the test cases in various ways. It can run a single test case, multiple test cases from a given path, a full test suite or multiple test suites from a given path. The test runner utility also provides functionalities to monitor the tests execution progress and the results remotely using a Web browser. Moreover, RTF provides web services (via a standard Ajax [6] framework) which can be used for developing any Web-based graphical user interfaces.

Fig. 2 shows an example of using the RTF to tests the iCub [13] robot using its simulator. A set of test case plug-ins were developed and grouped in a test suite (similar to Listing 1.3). The test suite was configured with the YARP fixture manager to run the iCub simulator. The figure shows the output of the test runner launched from a console (left side); the iCub simulator was automatically launched using the YARP fixture manager before executing individual tests. Finally the right side of Fig 2 shows the results of the test as they are collected and monitored from a Web browser. The

[4] Please refer to `http://robotology.github.io/robot-testing/documentation/testrunner.html` for the documentation of the test runner.

source code of the example and further test units for YARP middleware and iCub robots are available at `https://github.com/robotology/icub-tests`.

3 Conclusions

In this paper we have described RTF, a generic testing framework for the test driven development of robotic systems. We have discussed the requirements that we consider peculiar to the application and that motivated the development and our design choices, namely: i) the necessity to separate the dependencies of individual tests to avoid mutual constraints and ii) the need to deal with different level of testing from individual components to system tests. The latter is particular important because it required to introduce a sophisticated mechanism for dealing with complex test fixtures.

We have described the architecture of RTF by demonstrating different examples. RTF provides functionalities for developing unit tests in a platform, middleware and language independent manner. Middleware and language independency in RTF is achieved by allowing developers to implement and build the unit tests as independent plug-ins using different programming languages. A fixture manager prepares the setup (e.g., running robot interfaces, simulator) and actively monitors that all the requirements for running the tests are satisfied during the execution of the tests. At the moment we are actively developing tests for the iCub humanoid robot and the YARP middleware; using these features we plan to deploy fully automated tests not only for hardware components but also for robot controllers and complex behaviors that rely on the correct execution of different layers of software. These tests are available on the following repository: `https://github.com/robotology/icub-tests`. The framework has been currently tested with the YARP middleware but it can be used for any test-driven development and other middlewares and robots.

Acknowledgments. The research leading to these results has received funding from the European FP7 ICT project No. 611832 (WALK-MAN) and No. 600716 (CoDyCo).

References

1. Abran, A., Bourque, P., Dupuis, R., Moore, J.W.: Guide to the software engineering body of knowledge-SWEBOK. IEEE Press (2001)
2. AdaCore: Ada unit testing framework (2012), `http://libre.adacore.com/tools/aunit`.
3. Beck, K.: Test-driven development: by example. Addison-Wesley Professional (2003)
4. Bruyninckx, H.: Open robot control software: the OROCOS project. In: IEEE International Conference on Robotics and Automation. vol. 3, pp. 2523–2528. IEEE (2001)
5. freedesktop: cppUnit test framework (2013), `http://freedesktop.org/wiki/Software/cppunit`.
6. Garrett, J.J., et al.: Ajax: A new approach to web applications (2005)
7. Google: Google C++ Testing Framework (2013), `http://code.google.com/p/googletest`.
8. Ierusalimschy, R., De Figueiredo, L.H., Celes Filho, W.: Lua–an extensible extension language. Software: Practice & Experience 26(6), 635–652 (1996)

9. Janzen, D., Saiedian, H.: Test-driven development: Concepts, taxonomy, and future direction. Computer (9), 43–50 (2005)
10. Link, J.: Unit testing in Java: how tests drive the code. Morgan Kaufmann (2003)
11. Meszaros, G.: xUnit test patterns: Refactoring test code. Pearson Education (2007)
12. Metta, G., Fitzpatrick, P., Natale, L.: Towards Long-Lived Robot Genes. Elsevier (2007)
13. Metta, G., Sandini, G., Vernon, D.: The iCub humanoid robot: an open platform for research in embodied cognition. Proceedings of the 8th workshop on performance metrics for intelligent systems pp. 50–56 (2008)
14. Paikan, A.: yarpmanager: a way of running and managing multiple programs on a set of machines (2011), `http://wiki.icub.org/yarpdoc/yarpmanager.html`.
15. Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA Workshop on Open Source Software (2009)