Robot Unit Testing*

Andreas Bihlmaier and Heinz Wörn

Institute for Anthropomatics and Robotics (IAR),
Intelligent Process Control and Robotics Lab. (IPR),
Karlsruhe Institute of Technology (KIT),
76131 Karlsruhe, Germany
{andreas.bihlmaier,woern}@kit.edu

Abstract. We introduce Robot Unit Testing (RUT) as a methodology to bring modern testing methods into robotics. Through RUT the range of robotics software that can be automatically tested is extended beyond current practice. A robotics simulator is used to bridge the gap between well automated tests that only check a robot's software and time consuming, inherently manual tests on robots of alloy and circuits. An in-depth realization of RUT is shown, which is based on the Robot Operating System (ROS) framework and the Gazebo simulator due to their prominence in robotics research and inherent suitability for the RUT methodology.

1 Introduction

Software testing, not as merely a good practice, but rather as a software development paradigm, has (re)gained widespread popularity. In particular test-driven development (TDD) is as a core concept of agile software engineering methods. Regardless of whether TDD provides a benefit compared to other types of software development processes (cf. [10]), the importance of test coverage for complex software systems is not called into question. Unit testing together with integration and regression testing is an enabler for high quality and long-term supportability of large scale software systems. Therefore, it is hardly surprising that the Robot Operating System (ROS) relies on and encourages unit, integration and regression testing. To state a few benefits of testing as regarded by the ROS community[1]: faster incremental code updates, more reliable refactoring, better code design and prevention of recurring bugs.

However, something crucial is missing from these tests – in ROS and other robotic frameworks: robots. As we try to show in the following, this does not have to be the case. Of course, in contrast to the universal Turing machine, there is (so far) no universal physical machine. Additionally, the physical machine must be unbreakable or to be exact infinitely many times breakable. In Robot Unit Testing there is an additional gap between all tests passing and the system working correctly. If the test coverage of a software is high and all tests pass, one

* This research was carried out with the support of the German Research Foundation (DFG) within project I05, SFB/TRR 125 "Cognition-Guided Surgery".

[1] http://wiki.ros.org/UnitTesting

© Springer International Publishing Switzerland 2014

is justified to believe the software will work correctly, but it is not guaranteed. The additional gap in RUT stems from an additional artificiality, test data is not real data and simulated robots are not real robots. Yet, if our proposed RUT methodology is implemented there will remain gaps, but the current gaping abyss will narrow.

The following section (Sec. 2) concerns the current best practices in automated software testing. The further text is divided into two major parts, the first one (Sec. 3) details what we mean by Robot Unit Testing, its methodology. The second part (Sec. 4) applies RUT to ROS enabled robots, translates the abstract terminology of the first part into ROS terms, fills in details and provides results. A short summary (Sec. 5) concludes our presentation of RUT.

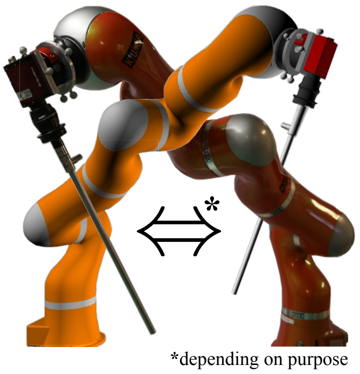

Fig. 1. The real KUKA LWR4+ and its simulated counterpart are clearly distinguishable in this image. They can not be said to be equivalent. However, from the viewpoint of our proposed Robot Unit Testing methodology, they are sufficiently similar in order to avoid breaking one of them by automated breaking of the other one.

2 Automated Software Testing

This section will not mention any manual testing methods, such as program inspection, walkthroughs or reviews. Although we find these to be valuable tools, manual methods are not our concern here. We bring up Robot Unit Testing as a methodology that enables *automated* testing of robots.

In the following short overview of automated testing, we follow the recently updated classic presentation by Myers et al. [8]. Because we do not introduce a new software-only testing methodology, such as the material discussed in this section and due to the limit space, no extensive state of the art for software testing is provided here. The purpose is to shortly introduce the most important aspects and vocabulary of testing (cf. [11]) necessary for the following sections.

First, tests can be distinguished by whether the internal workings of software components are known and exploited for testing. In case they are, one speaks of white box testing. Otherwise, it is called black box testing. Different software metrics apply for the two cases. The most important difference is that white box testing allows to precisely state the statement, condition and other types of coverage about the tested code. The next categorization separates functional from non-functional (e.g. stress) testing. Furthermore, one can distinguish between the scope included in a single test. While unit testing examines small fragments of the system, integration testing pertains to their interaction. The last terminological distinction that has to be introduced concerns the testing purpose. Common purposes include the early detecting of problems, the verification of the API contract and the avoidance of regressions through changes. All these are within reach of robotics research once the RUT methodology is utilized.

Complementary to testing as a technique there are testing methodologies that systematise testing into engineering. The most important methodologies for the paper at hand are test-driven development (TDD) and continuous integration (CI). TDD is about writing tests even before the code that is required to make them succeed [1]. CI is not as clearly defined, but all variants have in common that there must be a high degree of automated software build processes together with extensive test coverage [4].

We conclude this condensed survey of automated software testing methodology with two remarks: First, the testing mindset reckons software development as an iterative process and the test coverage provides the much needed non-formal loop invariant for the iteration. Second, testing is all about greater – justified – confidence in software quality. The following section explains why we deem RUT to be a crucial missing link in current robot software engineering.

3 The RUT Methodology

Sir the Simple Robot

To make the description more vivid, we'll assume a SImple Robot *Sir* made up of a manipulator, its arm, and a camera sensor, its eye. The scenario for the robot is to recognize a certain landmark, the red button, approach the landmark from its surface normal and continue a few millimeters, i.e. press it. Sir's arm can be controlled by specifying a Cartesian pose, which is converted to joint positions. The joint positions are interpolated and finally sent to the black box motor control unit. Sir's eye observes the world as a video stream of two megapixel color images at 30 frames per second seen as originating from a black box. Each image is preprocessed and compared against a database of known objects, if a match is found the object's position is known by Sir.

Requirements I: Robotics Simulator

Robot Unit Testing rests on the assumption that modern real-time robotics simulations are able to substitute real sensors and actuators *sufficiently* well. The plausibility of this assumption now rests on the notion of "sufficient" that we adopt. Sufficient here means that average robots in average environments doing averagely difficult tasks are simulated with an accuracy that makes them succeed or fail with the same average probability as in the real scenario. Here probability does not necessarily have to be interpreted in a frequentist manner, rather it can be seen as a degree of belief (cf. [5]). The point being, if the simulated robot can perform a kind of task reliably, so should the real robot and vice versa. It does not matter whether the simulation is accurate enough to also pass this test for other kinds of tasks. Therefore, no quantitative criterion for geometric, kinematic or dynamic realism has to be met in order to utilize and profit from RUT. We will come back to this point later in the section.

Requirements II: Robot Software

As will be shown, what is important for RUT are well designed robot software architectures. The robot software must permit to reroute the data flow to actuators and from sensors, ideally at several layers of abstraction. It is not strictly necessary, yet clearly preferable, if the rerouting function is available without directly changing the software, i.e. being a runtime instead of compile time option. In our example, the commands to Sir's arm position could be accessed as target pose, target joint position and interpolated joint position. Sir's accessible sensor data would consist of the video stream and the stream of detected object positions. Fortunately, this kind of flexible robot software is not a desirable exception anymore, but rather the state of the art in academia and becoming a standard in industry.

Bringing Things Together: The Big Picture

There are two ideas at the core of the Robot Unit Testing methodology: First, to redirect the actuator and sensor data at the *lowest* feasible level of abstraction to a robotic simulator. Second, to write unit tests that interact with the robot at the *highest* feasible level of abstraction.[2] Fig. 2 shows the general structure of RUT.

Again in the example, Sir is simulated at the level of interpolated joint positions and rendered camera images. In other words the simulator interface receives interpolated joint positions and sends rendered camera images. The unit tests control Sir by Cartesian poses, whose calculation is based on detected object positions.

Get Out of the Rut: Benefits of RUT

Obviously, neither using a robotic simulator nor writing tests are recent inventions in robotics research or software engineering, still thus far their full potential

[2] We do not presuppose a layered architecture or any specific architecture for that matter. We use the terms high and low level to refer to the semantic level of data, i.e. in our usage of these terms a Cartesian pose is on a higher level than a vector of joint positions of some kinematic.

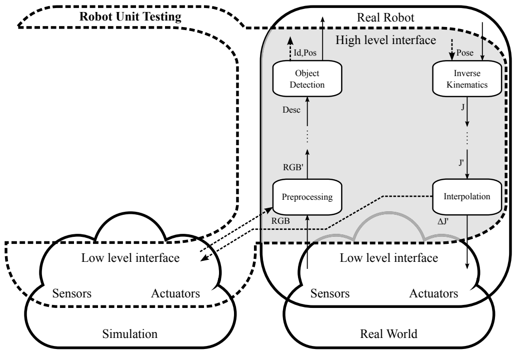

Fig. 2. The figure illustrates how Robot Unit Testing is related to the software and hardware components of the real robot. The shaded area highlights the software components utilized by both the real robot and its unit tests.

lies idle. What we want to contribute to with the notion of Robot Unit Testing is a systematic and sound presentation of a new methodology in robotics research and engineering. Its benefits are a novel level of automated test coverage for robots – as entities of software embedded in sensors and actuators. We propose to *test robots*, not just their isolated (software) components. Once a RUT framework is in place, we expect the same level of quality increase that tightly integrated testing brings to complex software systems (cf. [7]).

To elaborate on the new level of test coverage, we want to point out that *all software components of the real robot* under test besides very low level hardware components take part in RUTs. In case of Sir the tested components include the high level interface code, inverse kinematics, interpolator, image preprocessing and object detection. The only components left untested are the motor controller and camera firmware, which are usually black boxes in the real world, too. This large test coverage does not have to go along with large or complicated RUT code, as will be discussed later on.

Bringing Things Together: The Details

To recapitulate, the interface we write RUTs against is the highest level of abstraction offered by the robot software. The interface to the robotics simulator is the lowest available level of abstraction. RUTs should obey general rules established in the software testing community and conversely avoid its anti-patterns. One anti-pattern is particularly noteworthy: Avoid to build up system state. Each RUT should be executable on its own, not dependent on others and should also not interfere with them. This will be achievable with any robot framework defined as suitable for RUT above. Ideally the tests can be run in parallel by rerouting each test instance to a different robotics simulator instance. If this is not supported, it must only be ensured that the simulators state is completely reset between the sequentially running tests.

RUTs can be written to test a single robot function – our notion of "unit" in robotics – or a complete robot skill or task capability. What the test checks in order to determine success or failure is completely up to the purpose of the particular test. In testing terminology the success or failure condition is often referred to as the test oracle. A RUT can monitor actuator and sensor data in a very close or a rather coarse manner. In case of Sir, we can initialize its arm in a known pose and send a single motion command. Depending on what invariant the test should guarantee about Sir, we can compare each position of each joint in each simulation period to our ground truth. Alternatively, we may only care about whether Sir's arm reached its goal position after a specified amount of time. If the invariant is about Sir's kinematics we would most likely select the former, if we are currently modifying this robot subsystem the latter one seems more useful. To test Sir's button pressing skill, the RUT would initialize an environment with a red button at a known location and execute the skill. Again, depending on the test's purpose, it could suffice to test whether Sir's arm stops in a button press position. The complete RUT would consist of placing Sir and the red button in a known position, starting the skill and monitoring Sir's simulated arm position to check it against the correct final pose. This is everything required to ascertain that no later modification – anywhere in Sir's system – introduces a regression with respect to all involved actuator and sensor systems.

Here the point has come to say more about required simulation realism and accuracy. We only required the simulation to be sufficiently similar for a certain kind of task, so that this task will work similarly well on the real and simulated robot. For the RUT of Sir's button pressing skill two accuracy requirements must be met. Geometrical accuracy on the one hand and visual accuracy on the other hand. Sir's simulated kinematic must be accurate enough for the following to hold: For all relevant button positions the delta between simulated and real final pose is small enough to count as a button press in both cases. An analogous proposition must hold for Sir's visual perception. That is, the simulated scene must – for the relevant button positions – be visually close enough to the real one in order for the button to (wrongly) count as detected in reality and simulation. This vague notion ("relevant positions", "count as") is not a shortcoming of our approach, but part of the RUT methodology. Ground truth is always relative to what we judge to be sufficiently close to our use cases. As robotics simulators become more accurate, the RUT coverage can increase and include cases previously inaccessible to RUT. However, our point is that current simulators are *already* perfectly suited to facilitate RUT now. As long as the RUT methodology is minded, adopting Robot Unit Testing can provide a justified higher level of confidence in the quality of robot software development.

Cost of RUT in Terms of Additional Components and Effort

Talking about benefits of a methodology must be accompanied by elaborating on the costs of adopting it. In the following section, we'll detail element by element what it takes to implement RUT on ROS in terms of components, models and effort. Beforehand, we give a more abstract presentation of the requirements together with a glance to what state of the art robotics frameworks and simulators already provide. The difference between the two is the cost to implement RUT.

Considering Fig. 2, we can discern four necessary elements: Code to implement each test case, simulation models, simulated actuators with an appropriate interface and simulated sensors with an appropriate interface. In the previous subsection, we have already elaborated on the test code. As should be the case in non-robot testing, simple tests only require a few lines of code: Create input data, call the interface with the input data and compare the result with a reference. Since we use the robot's high level interface, the only open question relates to getting the test results. In some cases the result is directly provided by the robot software, e.g. whether Sir sees a certain object. In other cases the result is a state of the simulated world. Fortunately, the simulated world is fully observable from the RUTs point of view – of course not from the simulated robot. Thus, no substantial effort is required to query this world state. Simulation models represent the next element. It can require a substantial amount of work to produce these. Yet, in our experience most of the work is typically already done. Either because the robots are already simulated or at least CAD models are available. Often the CAD models were already created in order to built the robot in the first place. The last two elements, namely simulated actuators and sensors with the appropriate interface, can be dealt with together. All the major robotics simulators such as Gazebo[3], MORSE[4] and V-Rep[5] provide a variety of simulated actuators and sensors. Most actuators and sensors can already be accessed through the standard interface of common robot software frameworks. On the other hand it can take substantial work to remedy a situation, where the simulated item is not available or does not provide the required interface. To summarize, the effort required to adopt RUT is very low if the robot under test is already simulated for other purposes or at least a visualization exists. Otherwise, the effort increases along the dimensions of uncommon actuators and sensors, uncommon robotics simulators and uncommon robot frameworks.

4 RUT for ROS

The Robot Operating System (ROS) [9] is a well known and prevalent framework for modular and distributed robot software. This section describes our implementation of Robot Unit Tests for ROS. There are several open-source robotics simulators with ROS support available. We selected the Gazebo simulator because it seems to be the one currently favored by the ROS community. Furthermore, the simulator used in the DARPA virtual robotics challenge (VRC) is closely based on Gazebo [6]. In the remainder of the section, our goal is to relate everything generally said about methodology to the tangible implementation.

[3] `http://gazebosim.org/`
[4] `http://www.openrobots.org/wiki/morse/`
[5] `http://www.coppeliarobotics.com/`

Robots under Test

Our example implementation was done to test the software stack for two lightweight robot arms, the KUKA LWR4+ and the Universal Robots UR5, and their attached sensors. An industrial camera and an endoscope camera together with optics used for minimally invasive surgery (MIS) are mounted to the arms, shown in Fig. 1. To keep the presentation short and concise, the component graph of Sir the SImple Robot used as example in the previous section, has a lot in common with the actual ROS software. Fig. 3 shows the annotated graph of the actual ROS nodes.[6]

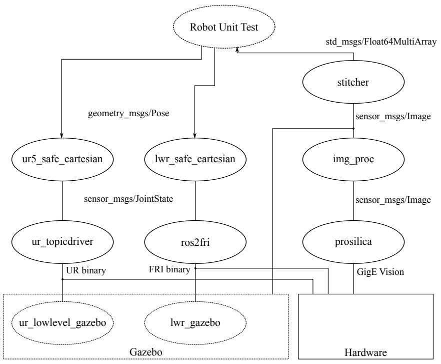

Fig. 3. The extended ROS graph used as example for simple and complex Robot Unit Tests is shown. Edges are annotated with the ROS message types or the protocol passed between the nodes. All solid framed nodes are part of the system independent of whether RUT is integrated.

Initial Effort Required for RUT

All solid framed nodes in Fig. 3 depict software that is required independently of simulation and RUT. Only the Gazebo model plugins for the two robot arms are supplements. In our case, they expose the respective vendor specific binary protocol interface and convert received commands to movements of the Gazebo joints and vice versa.

It is important to understand why we required, according to our RUT methodology, that one should go down to the level of vendor specific binary protocols and when one can and should stay at a higher interface level: Due to our specific demands in surgical robotics, the ros2fri and ur_topicdriver node are our own implementation to ROS-ify the robot arms. Therefore, we wanted the RUTs to include this part of the robot as well – following the RUT rule to test robots, not components. Had our test *purpose* been different, we could have used a generic joint model plugin already provided by Gazebo. In this case there would have been no additional code required to start with RUTs.

[6] We assume the reader to be familiar with ROS terminology, otherwise please refer to the above cited reference or `http://wiki.ros.org/ROS/Concepts`

All nodes required for the industrial camera are provided by the ROS community. We regarded them together with the camera firmware as black boxes outside our test domain. Therefore, the RUT methodology suggests sensor_msgs/Camera as the appropriate interface. This is fortunate because the ROS sensors plugins for Gazebo provide exactly this interface to the simulated camera.

Thus, the only elements missing for RUT are the simulation models for Gazebo. To create these in the form of meshes together with a URDF[7] or SDF[8] description, can take substantial effort. However, since the advent of ROS Industrial[9] the models are already available for a large number of (industrial) robots. This is the case for the UR5. Even if this doesn't apply to a certain robot, such as the LWR4+, there is a good chance that the model must be created anyway. The same model formats are required for visualization with rviz[10]. Making this another case where RUT does not involve additional effort. To sum up, most of the required RUT elements are already available for a robot that is well integrated with ROS. Let's put them to a novel use in RUTs.

A Basic RUT

A basic, yet essential, Robot Unit Test for the robot arms is to test self-collision free reachability of poses. The simulated environment is initialized with a certain robot arm configuration, i.e. world to base transformation and robot joint positions. Referring to the node names in Fig. 3, the test input is a Cartesian pose to the safe_cartesian nodes. The test result consists of two outputs: Detected collisions and the final pose of the simulated robot arm. For poses reachable by the arm, a test counts as successful if and only if no collision occurs and the arm reaches the target pose after a specified amount of time and stops there. For an unreachable pose, we define success as the arm not moving at all. Depending on the context, other definitions are conceivable, such as stopping in the point closest to the target pose that is reachable. Whether a pose should be reachable, i.e. the ground truth, is either tested with the real robot or validated through a human observer. We only test poses – together with environments, whose ground truth has been once verified, either one way or the other, to be correct. All these poses together make up the test data for this particular RUT. As usual in testing, effort and coverage increases with a larger set of test data.

ROS Tools

Before describing a more complex RUT, a note on tool support provided by ROS. Rostest[11] is an extension to ROS launch files, that takes care of starting up all the required nodes together with the simulator. Furthermore, a combination

[7] The Unified Robot Description Format, cf. `http://wiki.ros.org/urdf`
[8] The Simulation Description Format, cf. `http://gazebosim.org/sdf.html`
[9] `http://rosindustrial.org/`
[10] `http://wiki.ros.org/rviz`
[11] `http://wiki.ros.org/rostest`

of ROS command-line tools started from the launch file, especially rostopic, already provides most of the necessary test infrastructure. We will soon release code to the community that provides a similarly generic interface to check for certain events in the Gazebo simulation, e.g. something reaches a certain pose. Combined, with the include capability of launch files, each RUT only consists of four lines: Include the other launch file to start everything up, parametrize the simulated world, send a target pose, begin checking for success or failure.

An Extended RUT

To show the capabilities of RUT regarding complex tests, a short description about testing a continuous real-time image stitching node [2][3] on the same robots follows. The stitching node only uses the camera video as input because the algorithm trades image processing requirements against movement restrictions. Traditional software-only unit tests and ROS node test are used to ensure some of the image (pre)processing and performance invariants. Nevertheless, ultimately the camera video – the camera being attached to the arm's flange – depends on the arm trajectory. At this point we leave behind the state of the art in testing and enter the new realm of RUT. In the RUT methodology, we now have to consider what deviation from the current result should be brought to our attention through a failed test. The stitching result is a panorama image. Should we compare it to one of a correct run? No, or at least not if we do not want to define a new ground truth for every change in the simulators rendering pipeline.

The question to be answered according to our methodology is: What exactly is the test's *purpose*? It is not to check rendering details of the simulated camera image. Rather, we want to assert the robot's skill (cf. what we said about Sir in the previous section) of continuous stitching. We made sure the task is of a kind, where the simulation is accurate enough. Which means that the rendered camera images provide a sufficiently similar level of detail, e.g. edge and ORB features, to suffice for the robot's skill under test. Having gone through these methodological considerations, a test emerges that is easy to implement and robust to changes we do not care about: An integral part of stitching the images together, is calculating their relative offsets based on extracted features. If the assumptions to do RUT were satisfied in the first place, the offsets will be sufficiently stable and can be used as test oracle. After following through the RUT methodology, we can test all components involved into this non-trivial robot skill by a few lines of code that compare image offsets. The recent effort to create tools that automatically and continously execute ROS tests [12], based on the Jenkins continous intergration server[12], further facilitate adoption by the robotics community.

Results

By now the methodology of Robot Unit Testing as theory and practice should have become clear. Therefore, it is time to evaluate the claimed benefits for

[12] `http://jenkins-ci.org`

robotics research and robot software development. As it is often the case in software engineering, we provide qualitative results because their quantification would not bring further insight at this point in time. Once we arrived at the mindset behind what is written down in this paper as RUT methodology, we can clearly see the following improvements: Finding more bugs and finding them earlier. Introducing less regressions through code changes, or to be more precise finding regressions before deploying the new version to the real robot and having to hit the emergency stop button. Hence, less testing is required on the real robot, which is inherently manual if one doesn't want to risk breaking it. We can allow us to have greater confidence during development cycles. As long as all RUTs give us green light, we can postpone the manual testing of the new version. On that account we gain development speed – without having to fear that everything fails once we conduct the manual tests.

The proponents of test-driven development make a point that writing software in a way that facilitates testing is a best practice in itself. We affirm their position from our own experience. Previous versions of our ROS interface, consisted of less independent nodes. For example the intermediate nodes in Fig. 3 were integrated with their parent nodes. In terms of functionally the inverse kinematics did not have a public interface to the joint space interpolator. We refactored the nodes because this eased writing RUTs. This only superficially sounds like inverted reasoning. The objective is high quality robot software in order to obtain reliable robots. But in the process of ensuring long-term quality through writing RUTs, the very means tend to further their objective.

5 Conclusions

We presented Robot Unit Testing as a novel methodology based on existing technologies that advances the state of software development for robots. The goal and ultimate purpose is to extend the reach of automated tests beyond software components to whole robots, seen as entities of software embedded in sensors and actuators. Robot unit tests comprise *all* software components of the real robot down to the level selected as simulator interface. As explicitly shown in our RUT implementation for ROS, this level can be as low as vendor specific interfaces. In this case everything above what is a black box in hardware is included in the tests. Implementing the methodology for ROS is not only valuable in itself and to give a more detailed account, but our results also warrant to claim the benefits of RUT. We conclude our presentation of RUT by stating once more that it will make a difference to start doing *continuous, automated testing of robots* instead of robot components.

References

1. Beck, K.: Test-Driven Development By Example. Addison-Wesley, Amsterdam (2002)
2. Bihlmaier, A., Wörn, H.: Automated endoscopic camera guidance: A knowledge-based system towards robot assisted surgery. In: Proceedings for the Joint Conference of ISR 2014 (45th International Symposium on Robotics) Und ROBOTIK 2014 (8th German Conference on Robotics), pp. 617–622 (2014)
3. Bihlmaier, A., Wörn, H.: Ros-based cognitive surgical robotics. In: Workshop Proceedings of 13th Intl. Conf. on Intelligent Autonomous Systems (IAS 2013), pp. 253–255 (2014)
4. Duvall, P.M.: Continuous Integration. Addison-Wesley (2007)
5. Hjek, A.: Interpretations of probability. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Winter 2012 edition (2012)
6. Levi, N., Kovelman, G., Geynis, A., Sintov, A., Shapiro, A.: The DARPA virtual robotics challenge experience. In: 2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR), pp. 1–6. IEEE (2013)
7. Lewis, W.E., Dobbs, D., Veerapillai, G.: Software Testing and Continuous Quality Improvement, 3rd edn. CRC Press, Boca Raton (2008)
8. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing, 3rd edn. Wiley and Sons, New Jersey (2011)
9. Quigley, M., Conley, K., Gerkey, B., 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, vol. 3 (2009)
10. Rafique, Y., Misic, V.B.: The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis. IEEE Transactions on Software Engineering 39(6), 835–856 (2013)
11. Saha, G.K.: Understanding Software Testing Concepts. Ubiquity, 2:1 (February 2008)
12. Weisshardt, F., Kett, J., de Freitas Oliveira Araujo, T., Bubeck, A., Verl, A.: Enhancing software portability with a testing and evaluation platform. In: Proceedings for the Joint Conference of ISR 2014 (45th International Symposium on Robotics) Und ROBOTIK 2014 (8th German Conference on Robotics), pp. 219–224 (2014)