Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELOR THESIS

Integration of Software Development Testing Practices in Robotics

MIN Faculty
Department of Informatics
at Research Group Technical Aspects of Multimodal Systems, TAMS

Finn-Thorben Sell

finn-thorben.sell@studium.uni-hamburg.de
B.Sc. Informatik
Sutdent ID Number: 7047081

Supervisor: Marc Bestmann
Second Supervisor: Dr. Andreas Mäder
Date: July 6, 2021

Abstract

Verification of software correctness is an essential part of software development but also often difficult to do. This is especially the case in the development of complex autonomous systems like soccer-playing robots. For this bachelor thesis, a complete system has been developed to form a consistent and easy-to-use environment in which the RoboCup team Hamburg Bit-Bots can write its tests. A build automation platform has also been set up to automatically run these tests as checks whenever merge requests are created in the team's repositories. Afterwards, this system has been evaluated using a combination of qualitative feedback and quantitative data. The results of this evaluation found the developed solution to be generally valuable and performing as expected.

Zusammenfassung

Verifikation der Korrektheit von Software ist ein essentieller Teil beim Entwickeln der selbigen jedoch häufig schweirig umzusetzen. Dies ist besonders dann der Fall, wenn die betroffene Software ein komplexes autonomes System ist wie etwa fußballspielende Roboter. Im Rahmen dieser Bachelorarbeit wurde eine Komplettlösung entwickelt, um eine möglichst konsistente und einfach zu benutzende Umgebung bereitzustellen, in der die Mitglieder des RoboCup Team Hamburg Bit-Bots Tests erstellen und ausführen können. Es wurden außerdem Automatisierungsdienste aufgesetzt, welche ebendiese Tests automatisch ausführen, wenn Änderungsanfragen in den Repositories des Teams erstellt werden. Anschließend wurde das Komplettsystem anhand von gesammeltem Feedback und erhobenen qualitativen Daten evaluiert. Diese Evaluation ergab, dass die Entwickelte Lösung allgemein gut geeignet und wertvoll für das Team ist.

Contents

1 Introduction

With the increasing use of autonomous systems, their correctness becomes ever more important but also ever more challenging to verify. One example where this is the case is the software system of the RoboCup team *Hamburg Bit-Bots*. The *RoboCup Humanoid League* is an international competition in which teams participate to operate a completely autonomous humanoid robot that is supposed to play soccer on a level equal to humans by the year 2050. In this competition, the *Hamburg Bit-Bots* participate since 2012. Accordingly, its software has grown massively over the years, with many different people involved in different parts of it. This has resulted in an increasingly complex and interconnected software system that is very hard to test, especially when considering all possible edge cases of such a large system. As a consequence, the team currently does not use any form of automated or even standardized testing procedures but instead has to verify each component manually. Doing so has resulted in several problems for the team because not every member is an expert in every part of the system, and it is tough to verify that a slight change in one component does not break another. Because of that, the team voiced the need for a more standardized way to conduct testing, which led to this bachelor thesis. However, the methods in which testing can be done differ broadly and include but are not limited to regular automated tests or just guidelines for conducting manual testing. The goal of this thesis is thus to evaluate the different methods and establish a system under which the team can more efficiently test its software. Although the solution is developed with the teams' requirements in mind, it is intended to be generally reusable for any project using ROS ecosystem.

Part of the proposed solution is the combined usage of *Unit testing* as well as *System testing* with direct integration into the team's build tooling. These tests are, however, only authored using the two programming languages *Python* [39] and *C++* because those are the only two languages used by the team with *Python* being the primarily used language and *C++* only being used for specific components that have to be very performant. Additionally, build automation tools play an essential role in continuously executing the implemented tests and verifying that the team's software is always stable. But since many different solutions for build automation have been created, an analysis must be performed to choose one.

Due to the team's commitment to the open source philosophy, all the accompanying source code of the software created during this thesis is available under an MIT license in the GitHub repositories `bit-bots/bitbots_tools`[1], `bit-bots/bitbots_containers`[2] and `bit-bots/bitbots_jenkins_library`[3].

This thesis is structured as follows. At first, important terminology is introduced in chapter 2. Afterwards, scientific literature regarding testing methods as well as difficulties and solutions existing in the field of robotics are discussed in chapter 3. To establish a suitable testing methodology, the base toolset upon which to build is discussed and chosen in chapter 4. More details about the team's specific requirements on these tools are also discussed there. However, because the chosen tools do not fully provide a complete testing solution, they are further extended with the implementation of these extensions being described in chapter 5. Afterwards, the testing solution as a whole is evaluated qualitatively and quantitatively in chapter 6 with a final discussion following in chapter 7.

[1]https://github.com/bit-bots/bitbots_tools
[2]https://github.com/bit-bots/containers
[3]https://github.com/bit-bots/bitbots_jenkins_library

2 Fundamentals

This chapter defines the necessary terminology and concepts which are essential to this thesis. It first differentiates between the different kinds of tests based on their goal, scope, and underlying design knowledge in section 2.1. Afterwards essential software and frameworks are explained in section 2.2 and section 2.3. Finally, a differentiation is made between continuous integration and build automation in section 2.4 after which container technologies are described in section 2.5.

2.1 Different Kinds of Tests

The *IEEE Standard Glossary of Software Engineering Terminology* [90] defines different kinds of test practices.

Unit testing

One of these is *unit testing* which is defined as "Testing of individual [. . .] units or groups of related units" [90]. According to most software engineers, an additional property of unit tests is that they are "executed by the developer in a laboratory environment, that should demonstrate that the program meets the requirement" [Boy00]. Another important fact is that unit tests are purely technical tests designed and executed by developers without much influence from quality assurance or dedicated testing teams. Likewise, unit tests verify the correctness of a system in terms of what the developer expects, which is not necessarily what other stakeholders expect [Boy00].

For the context of this thesis, no differentiation between *Unit testing*, *Component Testing*, and *Interface Testing* is made because all of them are defined as testing a singular component or unit out of context of a larger system [90].

System testing

Another fundamentally different kind of test is the practice of *system testing* which can be defined as "Testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements" [90]. Its goal is to verify that all system components have been properly integrated by placing them in real-world usage scenarios [SBC12]. Similar to *system testing* is *integration testing* which is defined as "testing in which [. . .] components are combined and tested to evaluate the interaction between them" [90]. The critical difference between these two is that the former verifies that a system produces an intended outcome without any care as to how this was achieved. In contrast, the latter aims to verify the correct interaction between components and thus exactly the way in which the outcome was achieved.

White Box testing

Black Box testing

While *System testing* and *Unit testing* differ in their system-under-test, the knowledge with which a test is written can also be differentiated. If the test is based on an analysis of internal workings and the structure of a piece of software, it is called *White Box testing*. It could be used to, for example, verify that all branches of a control flow are correctly implemented by writing test cases specific to them. In opposition to this stands the practice of *Black Box testing* in which tests are written “based on the analysis of the specifications of a piece of software without reference to its internal working” [Kha10]. Black box tests are often more fundamental because they only verify that input is correctly accepted and output is correctly produced with no regard to the internal structure of the system under test. While this can be used in functional tests like system tests, it is employed in stress testing scenarios but also usability tests that require user interaction. In the real world, however, these two definitions are not always clearly applicable because the two methods are mixed, which is then called Grey Box testing [Kha10].

Property testing

A particular form of black box testing is *Property testing* which has the goal of determining whether a system under test has a predetermined property or is far from having it. This should be performed by inspecting the system’s behavior when given a small, possibly randomly selected subset of possible inputs. The testing algorithms are designed to compute an approximate and not exact decision on whether the system under test fulfills the defined property because it is simply unfeasible to prove correctness by testing all possible inputs [Ron01].

This testing method can be useful when verifying a system with large input space. For example, an encryption algorithm should fulfill the property of being reversible, thus reproducing the input again. Property testing has been popularized by the introduction of *quickcheck* [CH11] to Haskell, which has since been ported to other languages [30] and inspired similar projects [37].

2.2 Standard Solutions for Writing Tests

Most modern programming languages like *Python* [39], *Rust* [32], *Go* [34], *Java* [23] and others have a part of their standard library dedicated to writing Unit Tests. Figure 2.1 shows the structure of a simple unit test in the *unittest* [36] framework which is built into *Python*.

It is evident that language designers have recognized the need for writing tests and have thus created the possibility to do so simply and easily. However builtin frameworks often only provide rudimentary features which prompted the creation

```python
import unittest

class TestMath(unittest.TestCase):
    def test_multiply(self):
        self.assertEqual(6, 3*2)
```

Figure 2.1: A simple unit test in python consisting of a class inheriting from `TestCase` and defining one or more `test_*` methods.

of extensions like *nosetest* [26], *nose2* [38] or *pytest* [29] in python. These simplify the execution and definition of tests further while also providing customization options through plugins.

2.3 ROS – Robot Operating System

ROS [Qui+09], the *Robot Operating System*, is contrary to its name, not an operating system but a framework for building robot software[1]. It was originally created by *WillowGarage* but is now developed and maintained by the nonprofit *Open Source Robotics Foundation (OSRF)* [27]. It was designed to

- manage the communication between multiple independent processes on one or more computers,
- support various programming languages,
- provide many reusable tools,
- achieve modularity in large systems through standardized interfaces, and
- be free and open source. [Qui+09]

In practice, ROS has a large variety of existing packages, which abstract away many challenges when dealing with the control of robots. In 2017, the usage of ROS for information exchange between software components of a humanoid soccer robot was proposed. The proposal was designed to allow the reuse of software between different RoboCup teams in the humanoid league and shows how ROS can be used to achieve this [Bes17].

The ROS architecture is composed of the following relevant concepts. Of course, more exist because of the open nature of ROS, but only these are relevant to this thesis.

[1]This thesis focuses on ROS 1, not 2

Nodes are the processes running in a ROS system that perform arbitrary computation. Each process must always have a node associated with it in order to interact with the rest of the system.

Messages are strictly typed data structures which nodes pass between each other on named **topics**. Interface stability is achieved by standardizing and stabilizing these since that is the primary API a node exposes to the rest of the system. Each node can also freely chose when it subscribes to which topics, and any number of nodes may publish and subscribe on any given topic.

Packages are similar to packages in other contexts and bundle software. They also include metadata and build information.

Roscore is a central management server to which all processes connect. It manages the list of known nodes, topics, parameters, and other general management activities.

Launch Files are XML files that describe how nodes are started, which parameters to load onto the parameter server, and which dependent nodes are also started.

Catkin is the *CMake* [12] based build system used in ROS packages. It provides CMake functions that are required to interact with the ROS system correctly. There is also a python package named *catkin_tools* [9] which provides a `catkin` executable and is used to manage catkin based builds.

2.4 Build Automation

CI

Recently the practice of *Continuous Integration (CI)* has increased in popularity with ever more service offerings for it. The term has since become muddled and is interpreted by some people to mean automation of builds and tests whenever changes occur. While that is indeed an essential part of continuous integration, it is *only* a part, whereas continuously integrating software originally described the “practice where members of a team integrate their work frequently” [Fow06]. This thesis is partly concerned with introducing continuous integration practices into the Hamburg Bit-Bots team but, more importantly, describes how a build automation platform can be designed to support it.

While on this topic, it is helpful to disambiguate between Continuous Integration and *Continuous Delivery (CD)*. While CI aims to always have team members work as closely as possible on the primary copy, CD aims to deliver or deploy this main copy as soon and often as possible.

Job

Build automation platforms often introduce a few core concepts. One of these is the concept of a *job* which is an instantiation of a *build pipeline*. This means that a *build pipeline* is the configuration about exactly what a build automation platform should do, whereas a job is the execution of the pipeline. Additionally, jobs are often not directly running on the build automation platform's hardware but instead on a separate computer connected to it. These separate computers are called *workers*.

Worker

2.5 Containers

Along with build automation came the need to quickly and in scale create build environments as well as running software in a stable and reproducible environment. This need was met by containerization technologies like *Docker* [17]. Containers are standardized packages that include the software and everything needed to run it, like a runtime, tools, and libraries. To be precise, the standardized software packages are *container images* and consist of a complete filesystem and some metadata about the image. Only when an image is started, and a process runs inside this filesystem it becomes a *container*. Containers are often isolated from the host computer running it and only have access to what was explicitly granted.

Image

Container

3 Related Work

The difficulties of ensuring the correctness of autonomous systems have been discussed extensively. Afzal [Afz18; Afz+20a; Afz+20b] and Timperley et al. [Tim+18] have conducted extensive research into the general feasibility of testing robotic systems in simulated environments. They found that while there is a non-negligible gap between reality and what a simulator can simulate, most errors can nonetheless be discovered. According to their findings, most bugs surface under relatively simple conditions caused by configuration, setup, or simple to discover programming errors. As a result, Timperley et al. propose the use of black-box testing to probe a system's general correctness and discover these simple errors [Tim+18].

Outside of robotics in the general software development industry, automated software testing has increased in popularity and prompted several surveys into how companies conduct testing. Most important for this thesis are the works of Runeson as well as Daka and Fraser. Their surveys have found that testing is generally a process that tightly involves software developers and should not be done by a separate quality assurance department. This is due to the fact that developers are the ones actually writing software and reacting to test results. Because testing is such a developer-involved process, they found that a successful testing strategy must be as developer-friendly as possible. This entails the need for tests to produce quick feedback, which a developer can immediately integrate into their development, good integration into existing infrastructure so that results are always available when needed, and good APIs which a test author can use to author their tests [Run06; DF14].

Of course, there has also been some previous effort to develop testing frameworks specific to robotics. One such framework was developed by Paikan et al. [Pai+15]. They have introduced a generic framework in which test cases are written using *Python*, *Lua*, or *C++*, test fixtures are created using dynamically loaded libraries, and both are tied together using an XML based test suite definition. A test runner then executes the thereby defined test suite. This framework is very generic and not integrated into the team's development environment (notably ROS). While such an integration would theoretically be possible, ROS already supports some other existing tools as is described in chapter 4, which is why the framework introduced by Paikan et al. is not directly used in this thesis. However, the design considerations that went into it, as well as essential conclusions, were taken into account when choosing and designing the framework proposed in this thesis.

As previously mentioned, build automation and continuous integration play an important part in verifying software correctness and increasing productivity. Hilton et al. have analyzed the usage, costs and benefits of using continuous integration technologies in “Usage, costs, and benefits of continuous integration in open-source projects”. Their results indicate that most projects intend to or are already using CI. Additionally the primary reason for not using CI is not it’s high maintenance costs or ineffectiveness but rather that most projects are not familiar enough with CI systems or do not have any automated tests in the first place [Hil+16]. This research also indicates that using build automation is indeed an important when developing a test methodology. The discovered costs and best practices have been considered during this thesis.

4 Tool Analysis

This chapter outlines precisely what kind of solution the Hamburg Bit-Bots team needs by formulating its requirements. First, some general design guidelines and requirements are formulated here with more detailed ones following in section 4.1 and section 4.2.

General requirements on the complete solution as formulated by the team are:

- Run tests automatically when new code is requested to be merged into our repositories and manually whenever necessary.
- Enable simulator-based integration testing of multiple components in different scenarios as well as unit testing for single components.
- Control physical tests (like hardware control) over the same interface as software tests.
- Support at least unit tests written in all of the team's programming languages (Python, C++).
- Be as simple as possible to use and configure.

These form a base set of requirements that are taken into account when choosing any of the following solutions and additionally serve as guidelines for the latter extension done in chapter 5.

4.1 Build Automation platform

Since one of the major requirements is for tests to run automatically on merge requests, some service needs to exist to do so. Build automation platforms have been created specifically to build *and test* code changes whenever they occur, which makes them an obvious choice for this problem.

However, because there are many different platforms available to choose from, a list of requirements and other additional factors has been created in section 4.1.1 and section 4.1.2 with 18 platforms being evaluated according to these requirements in section 4.1.3.

4.1.1 Requirement definition

This section describes hard requirements which a build automation platform must fulfill in order for it to be chosen.

Job Configuration-As-Code

This requirement dictates that the build jobs that will run on the chosen platform should be configurable via a configuration file or defined through code that the platform executes on each job. While it might seem obvious, it contrasts with configuration done solely through a graphical user interface.

This has the benefit that a job configuration can more easily be reused by other internal and also foreign projects requiring a similar, if not the same, configuration. It also has the added benefit of making a platform's configuration reproducible should there ever be a need to set up the system from scratch again.

On Premise Workers

One of the use cases intended for the test system by the Hamburg Bit-Bots team is to run expensive tests of the whole system in a simulated environment. This is computationally difficult, and all hosted build automation platforms are restrictive about how much CPU, Memory, compute-time, or drive capacity a job is allowed to allocate, so we need to be able to supply our own workers, which do not have these kinds of restrictions. Additionally, the team uses custom hardware, e.g., an *Intel Neural Compute Stick* [22] which are also intended to be subjected to automated testing. This can realistically only be done if the platform supports supplying your own hardware. The job management, user interface, and other non-worker components may be offered as a service by a provider.

Low Cost

The Hamburg Bit-Bots team is a student-organized team that receives all of its funds either through sponsoring or as a budget by the University of Hamburg. Most of these funds are spent on hardware for robots or computers in the workspace. These preconditions require the team to be very careful about additional spending, especially recurring subscriptions, and we avoid it if we can. Because of that, the chosen build automation platform should either have a free-tier that fulfills the remaining requirements or a special policy for open source projects under which the team falls. Furthermore, this requirement ensures that the developed testing solution has a low entry barrier and can easily be reused by others.

4.1.2 Beneficial Aspects

This section describes aspects that are not hard requirements of the chosen platform but instead serve as an additional set of criteria that are beneficial for a platform to fulfill in order for it to be chosen.

Job abstraction

The Hamburg Bit-Bots team has its code organized over several git repositories, all with similar structures. Because of this, job configurations will largely be repetitive throughout these repositories. It is, therefore, desirable for a build automation platform to support a way to abstract job configuration so that each repository only configures a job in the aspects that differ between them and leaves common configuration to the abstraction.

Platform Configuration-As-Code

Suppose the chosen build automation platform is entirely hosted on-premise. In that case, it should itself be completely configurable through configuration files for the same reasons that job configurations are required to be configurable as code.

Explicit GitHub Integration

The git repositories used by the Hamburg Bit-Bots team are hosted on GitHub, which offers additional features to being a hosted git repository such as *pull request status checks* [2]. These checks offer an easier evaluation of merge requests, and it is desirable for a build automation platform to explicitly integrate with GitHub to annotate merge requests with these checks.

Other integration features such as the automatic discovery and registration of an organization's repositories and immediate creation of new jobs are also nice to have.

Existing Community

Because of the earlier *low cost* requirement, the availability of premium-level support is not guaranteed if problems arise on the chosen platform. Instead, a fallback on community resources such as documentation, blogs, or forums should be possible. These, however, only exist if the platform is popular enough for a community to have formed around it that created said resources.

This aspect is subjective, and a rating is given based on my impressions while evaluating the different build automation platforms.

Open Source

The Hamburg Bit-Bots team believes strongly in the open source vision and performs all of its development publicly and under an MIT license. It would be in line with this belief for a build automation platform to also be developed and maintained as an Open-Source project under one of the licenses approved by the *Open Source Initiative* [25].

4.1.3 Fulfillment Analysis

Table 4.1 shows a list of build automation platforms and which of the previously defined requirements they fulfill. Platforms that do not support the usage of custom toolchains or do not have explicit support for *ROS* are not included in this list at all.

The platforms which passed all requirements have been reviewed further by analyzing any additional aspects. Table 4.2 shows the result of this analysis.

Jenkins is the clear winner of this analysis since it is the only platform fulfilling all requirements and offering all additional beneficial aspects. Nevertheless, and although the Open Source aspect is important to the team, it could be argued that GitHub Actions, while not being Open Source, offers the same amount of features with the addition of taking care of a potentially sizeable administrative burden.

The final decision nonetheless fell onto Jenkins because of the following reasons. First, the team already had a configured Jenkins instance from previous automation experiments and has already gained some experience with it. Additionally, GitHub Actions is an offering by the for-profit company GitHub and thus inherently intends to generate profits. Moreover, although GitHub offers free tiers for open source projects that would apply to the Hamburg Bit-Bots, the team would still be reliant on GitHub not changing its policy or risk not being able to run its workload. Finally, GitHub Actions only supports GitHub itself as a version control system, whereas Jenkins can run pipelines from anywhere. Additionally Jenkins offers a lower entry barrier because it is usable by everyone even if they do not qualify for GitHub's free tier.

[1]Is on-premise but pricing depends on the number of agents.

[2]Atlassian provides free cloud licenses for Open-Source projects, but Bamboo is not a cloud offering.

[3]Workers are only supported with a separate subscription.

[4]Has a free tier that is restricted to less than the team's expected workload.

[5]Relatively new but very popular and growing fast

[6]Open Source Community edition and proprietary Enterprise Edition

[7]Supports GitHub but only for free during the year of 2020

Platform	Job-CaC	On-Premise-Workers	Low cost
AWS CodeBuild [4]	✓	✗	✗
Azure Pipelines [5]	✓	✓	❋[4]
agola [3]	✓	✓	✓
Bamboo [6]	✓	❋[1]	✗[2]
Buddy [7]	✓	❋[3]	✓
Buildbot [8]	✓	✓	✓
Circle CI [15]	✓	❋[3]	✓
Concourse [13]	✓	✓	✓
Codeship [11]	✗	✗	✗
Drone [16]	✓	✓	✓
GitHub Actions [19]	✓	✓	✓
GitLab CI [20]	✓	✓	✓
GoCD [28]	✓	✓	✓
Google Cloud Build [10]	✓	✗	❋[4]
Jenkins [24]	✓	✓	✓
Semaphore [14]	✓	✗	✓
TeamCity [33]	✓	✓	✓
TravisCI [35]	✓	✗	✓

Table 4.1: The list of evaluated build automation platforms and whether they fulfill the given requirements. Platforms that fulfill all requirements are marked green.

Platform	Job abstraction	Platform-CaC	GitHub	Community	Open Source
agola	✗	✓	✗	✗	✓
Buildbot	✓	✓	✓	✗	✓
Concourse	✓	✓	✓	✗	✓
Drone	✓	✓	✓	❋[5]	❋[6]
GitHub Actions	✓	n/a	✓	✓	✗
GitLab CI	✗	✓	❋[7]	✓	❋[6]
GoCD	✗	✓	✓	✓	✓
Jenkins	✓	✓	✓	✓	✓
TeamCity	✓	✗	✓	✓	✗

Table 4.2: Platforms that passed all requirements and whether they have any additional beneficial aspects. GitHub Actions is externally hosted, which makes the *Platform-CaC* aspect not applicable. It is marked with *n/a*. Platforms that offer all aspects are marked green.

4.2 Testing frameworks

As explained in section 2.2, different test frameworks exist for different programming languages and environments. Because this thesis aims to introduce a testing methodology to the team Hamburg Bit-Bots, appropriate frameworks need to be evaluated and chosen for all use cases. Generally, the kinds of tests that the team wishes to perform can be sorted into the three categories *Unit testing Python code*, *Unit testing C++ code* and *System testing*.
One of the major requirements laid out at the beginning of this chapter was that all of these tests need to be runnable through the same interface. This interface was chosen to be *catkin_tools* because it supports running different kinds of tests, is a commonly used tool in the ROS ecosystem, and is thus well supported and already known well by the team. In general *catkin_tools* supports any test framework which can output its result in the *Xunit* format [40]. However, there are some which are more directly supported. This means that *catkin* natively defines commands for registering tests authored in these frameworks and knows how to run them. Which test frameworks these are and how they can be used for the team's use cases is described in the paragraphs below.

4.2.1 Python Unit Tests

For writing unit tests in *Python*, *catkin* supports tests written using the *nosetest* [26] package. This package is an extension of the standard libraries *unittest* [36] that automatically discovers tests and renders results in a format chosen by the user. *Catkin* automatically calls *nosetest* with the correct arguments for its registered tests and for writing results in an *Xunit* formatted file.

Because it is a small package that does not interfere with the inner workings of *unittest*, many plugins such as *Hypothesis* [37] are still usable to support specific scenarios like property testing. Additionally, *nosetest* is a well-established and known framework in the *Python* ecosystem. Unfortunately though, it is discontinued, so no new features and development are expected to happen. Nonetheless, *nosetest* is a solid foundation upon which to build.

4.2.2 C++ Unit Tests

For writing unit tests in C++ *catkin*, supports the *GoogleTest* [21] framework. It is relatively new in the C++ ecosystem but very popular and supports many different use cases like mocking, test fixtures, parameterized tests. For these reasons, it was chosen as the framework in which the team will write its C++ unit tests.

4.2.3 System Tests

Unit tests are one thing, but arguably the more significant tests are those that verify that the whole system behaves as intended. Because of that, the majority of tests are expected to be system tests, and of course, a framework for those needs to be chosen as well. For the *ROS* ecosystem, there already exists a package called *rostest* [31] which allows tests to run not just on their own but instead in the context of a fully-fledged ROS system. This is implemented by *rostest* executing *ROS launch files* on an isolated *roscore*. On this *roscore* it then starts all *nodes* that are defined in the *launch file* normally. Additionally, *nodes* that are defined in the *launch file* using a `<test/>` tag instead of a normal `<node/>` tag, are recognized to be test nodes, started and monitored accordingly, and their results are collected after they have finished executing. Once this is the case, all remaining *nodes* are also stopped.

Similar to *nosetest*, *rostest* implements functionality for registration and management of test environments but does not offer any tools or utilities when writing actual test code. As a result, many common operations necessary for asserting the system to be in a specific state must be re-implemented for every test. An example of such a common operation would be the need to assert that a particular node in the system is still running and responding to pings or that it has not logged anything on the *error* or *critical* level.

5 Implementation

Chapter 4 discussed the tools upon which this bachelor thesis builds. It has highlighted the need for extensions of the chosen tools by showing their limitations. This chapter first describes in section 5.1 how the chosen build automation platform, *Jenkins*, is configured and how a library has been created to ease pipeline configuration. Afterwards, the needed framework extensions are described in section 5.2.

5.1 Build Automation Platform

For deploying Jenkins on the team's server as well as running the automation jobs, a container-based workflow has been chosen. It has the benefit of being universally usable in multiple different environments without having to install system dependencies beyond a container runtime. It also eases changing the job environment because only a container image needs to be updated instead of all worker machines. For this, two container configurations have been created which can be viewed in the GitHub repository `bit-bots/containers`[1]:

1. A *bitbots_builder* container which is dedicated to being able to compile, document and test our codebase. It exists so that not all dependencies need to be reinstalled during every job but are instead included in this base image.
2. A *jenkins* container, which includes the Jenkins software and all necessary plugins.

5.1.1 Jenkins Configuration

Jenkins itself consists of only a small core application with a sophisticated plugin architecture to extend nearly every part. Following is a list of the most important plugins and the reason why they are used:

- **Blue Ocean** is a modern graphical user interface that makes interacting with *Jenkins* easier for team members who do not have much experience working with it.

[1] `https://github.com/bit-bots/containers`

- **Configuration as Code** is a plugin that persists Jenkins platform configuration into a file so that setting up the Jenkins server is reproducible.
- **Git** is a plugin for retrieving the team's source code.
- **GitHub** is an integration into GitHub, which allows setting pull request checks and automatic discovery of repositories. During the repository discovery process, webhooks are automatically set up so that jobs are triggered immediately when a new commit is pushed, or a merge request is created.
- **Kubernetes** is a build cloud implementation that simplifies scheduling of builds onto workers supports the desired container-based workflow.
- **Pipeline** is the plugin that implements pipeline configuration as code.

As mentioned above, Jenkins itself is running in a container, and jobs are too. Consequently, one of the additional jobs which Jenkins performs besides automation of our software tests is to rebuild its own and the *bitbots_builder* container images periodically and when their specifications are changed. This process enables automatic update installation transparently and efficiently. The configuration for this can be viewed in the GitHub repository *bit-bots/containers*[1] in the file *Jenkinsfile*[2].

For jobs, Jenkins is configured to periodically scan the team's GitHub organization for all repositories containing a file called *Jenkinsfile*. This file acts as the pipeline configuration file for the repository it is located in.

5.1.2 Pipeline Abstraction

As mentioned before, Jenkins offers the ability to abstract pipeline configuration with so-called shared pipeline libraries [18]. As part of this thesis, one such library has been developed, which can be viewed in the GitHub repository *bit-bots/bitbots_jenkins_library*[3].

Its use can be seen in fig. 5.1 which is now used to explain the main concepts of the library:

Line 1 Import *bitbots_jenkins_library* into this pipeline so that its commands can be used.

Line 3 Call `defineProperties()` to set properties of this pipeline, e.g, whether multiple pipelines for the current repository may run concurrently.

Line 5 Create a new `pipeline` variable which is an instance of `BitbotsPipeline`. `BitbotsPipeline` is the primary content of the library and contains most of the logic. The given constructor arguments are necessary for it to call *Jenkins*

[2]https://github.com/bit-bots/containers/blob/main/Jenkinsfile
[3]https://github.com/bit-bots/bitbots_jenkins_library

functions. The constructor also accepts further configuration that affects all packages configured later in the pipeline. For example, documentation publishing can be generally enabled or disabled for change request builds here.

Line 6 Call `configurePipelineForPackage()` on the `pipeline` variable which instructs the pipeline to run for a ROS package that is located in the current repository and further defined as the functions arguments over the next lines.

Line 7 Create a new `PackagePipelineSettings` instance, which is a data structure that holds information about what the pipeline should do for a specific package. For example, building the package's documentation can be enabled or disabled via this data structure. By default, all pipeline actions are enabled if not explicitly disabled.

Line 8 Create a new `PackageDefinition` instance, which is part of `PackagePipelineSettings` and defines where the package is located in the repository and what its name is. The invocation shown here defines a package named *example_package* that is located in a folder with the same name.

Line 11 Execute the pipeline. At this point, worker resources will get allocated, containers will be set up, and all the configured actions will be run.

For comparison, an equivalent non-abstracted pipeline, which includes all the necessary logic that is currently defined in *bitbots_jenkins_library* would take up nearly 180 lines of code, with 50 of them repeating for each package in the repository. In contrast, the abstracted pipeline takes ten lines of code with five of them (six to ten in fig. 5.1) repeating for each package in the repository.

```
@Library("bitbots_jenkins_library") import de.bitbots.jenkins.*;

defineProperties()

def pipeline = new BitbotsPipeline(this, env, currentBuild, scm)
pipeline.configurePipelineForPackage(
        new PackagePipelineSettings(
                new PackageDefinition("example_package")
        )
)
pipeline.execute()
```

Figure 5.1: An example *Jenkins* job configuration that uses *bitbots_jenkins_library* to configure that a package named *example_package* is located in the current repository and all standard actions should be done for it.

5.2 Testing Framework bitbots_test

Section 4.2 described which frameworks were chosen for the different kinds of expected tests. It also highlights how some of the chosen frameworks are a good starting point and solve the problem of interacting with ROS but lack utilities and features when actually writing tests. Because of that, a new package called *bitbots_test* has been created during this thesis which addresses the shortcomings mentioned above and offers a unified and pleasant developer experience. Its source code is available in the GitHub repository *bit-bots/bitbots_tools*[4] in the directory *bitbots_test*[5]. It is important to note that *bitbots_test* does not aim to replace any of the chosen tools but instead extends them so that tests can be written in a more reusable and developer-friendly manner.

The features of *bitbots_test* are described in the following sections.

5.2.1 Test Auto Discovery

When directly using *rostest* and *GoogleTest*, a test author needs to separately register all the tests they have written in a package's build script, whereas *nosetest* already supports test discovery when given a directory in which multiple tests can be defined. The behavior of *nosetest* has been taken as an inspiration to implement

[4]https://github.com/bit-bots/bitbots_tools

[5]https://github.com/bit-bots/bitbots_tools/tree/master/bitbots_test

a similar behavior for the other test types in *bitbots_test*. The result is that a test author only needs to enable testing of their package in a general manner using the cmake function `enable_bitbots_tests()`[6] provided by *bitbots_test*.

This cmake function accepts the directories in which tests are searched as arguments but also defines sensible defaults for them. It then automatically registers python unit tests using *nosetests* builtin discovery mechanism. Additionally, it searches the given directories for C++ unit tests and system tests and registers them with the corresponding test framework. For C++ unit tests, it is often necessary to compile the tests together with the package's C++ source code. This is supported by `enable_bitbots_tests()` by using a cmake function argument.

Enabling tests for a package is then as simple as calling `enable_bitbots_tests()` in the package's cmake build script which is shown in fig. 5.2.

```
if (CATKIN_ENABLE_TESTING)
    find_package(catkin REQUIRED COMPONENTS bitbots_test)
    enable_bitbots_tests()
endif()
```

Figure 5.2: An example cmake build script that uses `enable_bitbots_test()` to set up testing for the current package.

5.2.2 General Test Authoring Utilities

While test auto-discovery is a nice feature, it is still not a useful addition for authoring tests. For that problem *bitbots_test* also offers some utilities and functions which were found to be commonly required but missing in the existing frameworks.

Notably, the most important addition is the ability to restrict test execution via dynamically evaluated conditions. This is a necessary feature for the team because it allows different kinds of tests to be run in different scenarios. For example, a test that verifies that a hardware interface is working correctly should only be executed when the hardware is connected.

Another important feature is the introduction of tags for test cases. This enables test cases to have one more tags associated with them. A user can then specify which subset of tests they want to execute by specifying a list of requested test tags or a list of forbidden test tags. The former restricts test execution to tests

[6] `https://github.com/bit-bots/bitbots_tools/blob/master/bitbots_test/cmake/enable_bitbots_tests.cmake.in`

which have one of the tags associated with them whereas the latter forbids their execution.

Another important addition is support for writing tests that do not immediately have a result available that can be validated. This is important when computation happens in a multi-threaded environment but also when writing system tests.

5.2.3 Rostest Utilities

Arguably the most essential parts of *bitbots_test* are its additional assertions and utilities for writing system tests using rostest. These include the following:

- Assertions for verifying the log. These can be used in different contexts for example that a certain node does not log anything on the log levels *Warning*, *Error* or *Fatal* or that a message matching a *RegEx* pattern is logged.
- Assertions for verifying another node's state. For example, it is possible to assert that it is running and responding to pings.
- Assertions that verify published messages on specific topics. An example usage would be asserting that a specific message is published in response to one sent by the test.

Some of these assertions need to be set up and torn down during the test life-cycle. To make the management of this more manageable, an additional `TestCase` subclass, `RosNodeTestCase`, has been created, which manages the setup and teardown of these assertions as well as the test's ROS node automatically.

5.2.4 Team-Specific Utilities

In addition to general rostest utilities, some team-specific ones were also implemented. They were added to ease interacting with the team's simulator setup and include assertions and utilities to set up the robot in the simulator and verify the state of the simulation. For example, using the implemented utilities, it is possible to spawn a robot in a specific position in the simulator and then later verify that it has moved a significant distance without falling down after sending a command to move forward.

6 Evaluation

In chapter 5, a build automation system was set up to automatically build and test merge requests created on the team's GitHub repositories. Additionally, a testing framework was developed to ease the creation of new tests. Of course, these solutions need to be evaluated in order to determine their usefulness to the team. This evaluation is done in two steps. First, a qualitative evaluation was performed, which is described in section 6.1, and afterwards, a quantitative evaluation is described in section 6.2.

In addition to these two evaluations, *bitbots_test* has been used during the development of *bitbots_test* to author tests asserting its own correct behavior.

6.1 Qualitative Evaluation

The qualitative evaluation aimed at collecting as much feedback from the team about the developed solution. First, an explanation about the method with which this feedback was collected is given, with the results described afterwards.

Method

For gathering as much substantial feedback as possible, a testing event was organized, that the team was asked to attend. At this event, a detailed introduction to writing tests was given by me, and extensive documentation was also provided. The team members were then asked to implement tests of their own choice and on their own accord. The event was also accompanied by a survey which the team members were asked to fill out individually. The survey consisted of three parts, of which the first two were filled out ahead of implementing tests and the third one afterwards.

The first section of the survey was designed to gauge a team member's responsibility and amount of experience with the team's software as a whole. Immediately following was the second section, which aimed to discover how the team members currently perform different testing-related tasks, how difficult they found these tasks, and how much confidence they have in the correctness of the team's software. The third section was only filled out after a team member had taken part in the event and implemented their own tests. It asked each participant how they plan to perform the same workflows as asked in section two in the future. It also asked the participants how difficult and effortful they found working with the proposed system. Additionally, it asked each participant to make a prediction about the team's future software quality.

Results

Unfortunately, only a few team members attended the event and filled out the survey completely. Because of this, no statistical analysis has been performed to discover statistically significant results. However, some feedback could still be extracted from the survey.

On the positive side, all participants said they think that, in general, testing will be easier in the future and that software quality will improve. When asked how they would perform testing in the future, participants freely chose to use *bitbots_test* as a framework in which to author their tests. On the downside, however, two shortcomings were still uncovered. The first one is that components exist in the team's software that are not completely deterministic and are thus difficult to test in a reproducible way. Another important issue is that due to many components (e.g. *catkin*, *rostest* and *bitbots_test*) interacting with each other it is often difficult to debug not the system under tests but the tests themselves.

These two downsides are especially detrimental when considering that a good developer experience is important when authoring tests.

6.2 Quantitative Evaluation

For the quantitative evaluation, the tests created during the testing events, as well as additional tests authored later, were used to analyze past commits of the team's software. The goal is to illustrate how many errors found by a human could have also been found using automated testing.

Method

This evaluation was done by performing the following steps:

1. For each package that has at least one automated test written using *bitbots_test*:

a) For each of the commits in the packages git history that include the word “fix”:
   i. Checkout the found commits parent commit so that the fix is not yet applied.
   ii. Run all tests but in their newest version.
   iii. Checkout the found commit so that the fix is applied.
   iv. Run all tests again in their newest version.
   v. Categorize the fixing commit using the following rules:
      - If a test fails without the fix applied and passes with the fix applied mark as `fail2pass`.
      - If a test passes without the fix applied and fails with the fix applied mark as `pass2fail`.
      - If all tests pass regardless of whether the fix is applied mark as `not_covered`.
      - If a test fails without the fix applied but still fails with the fix applied investigate further to differentiate between tests that are not applicable to the checked out software state (mark as `unrelated_error`) and the fix not correctly fixing the problem (mark as `broken_fix`).

Applying these rules results in a data set in which each package has commits associated with one or more marks. Using the described strategy also includes merge commits which resulted from bug fixing merge requests because the team has an internal policy to prefix all bugfix branches with “`fix/`” which is then included in the merge commit message.

Results

In this evaluation, 384 commits from seven different packages were found to contain the word “fix” and classified according to the defined criteria. In comparison the team actively maintains 55 packages with commits dating back to 2018. This was done using a combined number of 55 tests throughout the seven packages; however, 40 of these are tests for the new *bitbots_test* package. Of these 55 tests only *bitbots_test* contained unit tests whereas all other packages contained just system tests. This results in the unit tests additions and utilities for *C++* and *Python* not being evaluated well outside of the tests contained in *bitbots_test*. The combined number of commits for each classification mark can be seen in fig. 6.1. Notably the *bitbots_test* package had the highest number of tests and no broken fixes or `pass2fail` marks.

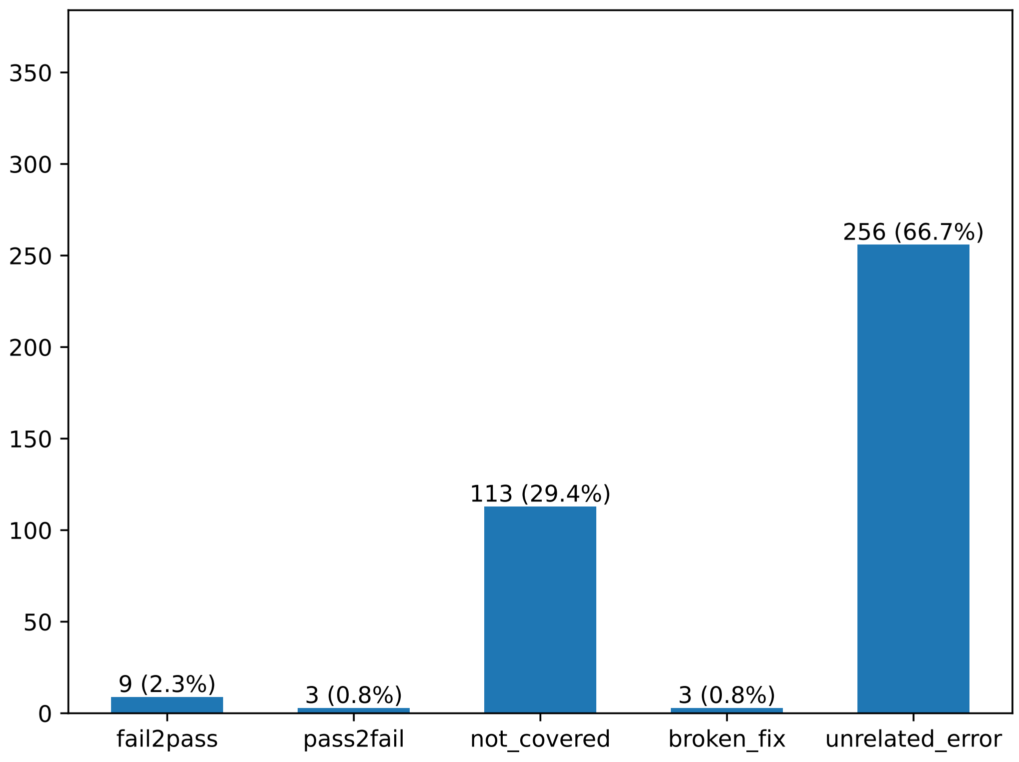

Figure 6.1: The number of commits for each mark.

Immediately obvious is the large number of commits being marked as having an unrelated error. This was mostly due to a major change in the team's build configuration, which impacted the evaluation results because test registration happens in the same file as the build configuration, and the evaluation script was not able to merge the old and new versions of that file.

Another important result is the large number of commits classified as `not_covered`. This was caused by only a relatively small number of tests existing, with most of them being very high-level system tests. For example, the team's decision-making package only has tests asserting that all ROS nodes of that package start correctly and no syntax errors appear in configuration files. Especially in the example of this decision-making package, there are various commits fixing the algorithms semantics, which of course were not covered by the simple test.

On the other hand, a combined number of 15 commits were identified in which execution of the tests would have produced valuable information like proving or disproving that a "fix" actually works. Six of these even were commits that had already been manually reviewed but still broke something.

Threats to validity

Primarily reservations concerning the construct validity of this evaluation exist. One such concern is that the team neither has an internal policy that enforces bug fixing commits to have “fix” in the commit message nor a policy that enforces that only bug fixing commits may have “fix” in the commit message. This fact may result in not all bug fixing commits being analyzed. On the other hand, commits might have been analyzed, which might not have been fixing any bugs which would count towards the `not_covered` mark. Another concern is the inability of the evaluation script to execute new tests with the old source code state while maintaining the old build configuration. As explained, this results in large numbers of unrelated errors. Another restriction of the evaluation method is that old software states might require old versions of dependencies. However, in the team’s package definitions, no dependency versions are explicitly specified, so the evaluation script may not have been able to provide correct versions to the system under test. This might have resulted in even more unrelated errors.

7 Discussion

In the following chapter, the evaluation results from chapter 6 are discussed. In section 7.1 a conclusion of this thesis is given while possible future work is suggested in section 7.2.

The results show that, in general, the team liked to use the developed solution and team members wanted to use it in the future out of their own accord. Based on this, it can be expected that more tests will be developed in the future, which thus further increases the stability and testability of the team's software. This result also highlights that the importance of good developer experience was adequately considered since the team gave positive feedback about their experiences when writing tests. Configuration and test execution also works through the *catkin* interface and, in doing so, successfully integrates into existing tools and workflows, which makes test execution and management seamless and easy. On the other hand, the participants only ever did write system tests, so nothing can be said about the developer experience when writing unit tests. This was also reflected in the relatively low number of faulty commits discovered during the quantitative analysis. Nonetheless, the system did discover a number of faulty commits which existed even though the team already uses an extensive review process. This shows how the developed test methodology is already able to provide valuable information to the team with minimal test authoring effort.

7.1 Conclusion

In this thesis, a testing methodology has been developed for the RoboCup team Hamburg Bit-Bots by researching and evaluating proven methods and solutions created in the software development industry and research. According to this research, a methodology was developed for the team's testing requirements. This methodology consists of the usage of the existing libraries and frameworks *rostest*, *nosetets* and *GoogleTest* which were generally found to be functional but also did not alone form a complete solution. Consequently, an additional package, *bitbots_test* has been developed that complements and abstracts those libraries to form a consistent framework in which the team can author its tests. This package has then be evaluated qualitatively and quantitatively. The results were generally found to fulfill all requirements, but some shortcomings were still observed. Additionally, the effectiveness of writing tests using *bitbots_test* should further be evaluated over a more extended time period and with a wider variety of implemented tests.

7.2 Future Work

In the future, more work could be done on the topic of testing the team's robotics software. Firstly, the team wishes also to test hardware control algorithms which require the corresponding tests to only execute when said hardware is connected. This use case could not be tested during this thesis because of the COVID-19 pandemic, but it should be done at a later date.

Also, the selected Python unit testing framework, *nosetest*, has discontinued its development. This is not yet a problem but might become one in the future, so an alternative should be evaluated.

Furthermore, the Jenkins abstraction library, *bitbots_jenkins_library* is currently rather specific to the team's setup in that it assumes certain hardware configurations. This should be improved in the future so that others can reuse it more easily. Similarly parts of *bitbots_test* should also be made more reusable by implementing non team-specific utilities and assertions directly in *rostest*. Since *rostest* is an open source project, an effort to do so will be made.

A more extensive evaluation could also be performed when more tests are written. It could be combined with monitoring the team's software development and the impact which the developed framework has on it.

Furthermore, *linting* is another method that aims to verify software correctness but by statically analyzing source code and then evaluating it according to a predefined static set of rules. However, it integrates into a development workflow similarly to testing. Its effectiveness could be analyzed in a similar manner in which testing was analyzed during this thesis.

Bibliography

Literature

[90] *IEEE Standard Glossary of Software Engineering Terminology.* Sept. 28, 1990. DOI: 10.1109/ieeestd.1990.101064.

[Boy00] Kenneth W. Boyer. “Test Process Improvement: A practical step-by-step guide to structured testing”. In: *ACM SIGSOFT Software Engineering Notes* 25.3 (May 2000), pp. 59–60. DOI: 10.1145/505863.505883.

[Ron01] Dana Ron. “Property testing”. In: *COMBINATORIAL OPTIMIZATION-DORDRECHT-* 9.2 (2001), pp. 597–643.

[Run06] P. Runeson. “A survey of unit testing practices”. In: *IEEE Software* 23.4 (July 2006), pp. 22–29. DOI: 10.1109/ms.2006.91.

[Qui+09] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software.* Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[Kha10] Mohd Ehmer Khan. “Different forms of software testing techniques for finding errors”. In: *International Journal of Computer Science Issues (IJCSI)* 7.3 (2010), p. 24.

[CH11] Koen Claessen and John Hughes. “QuickCheck: a lightweight tool for random testing of Haskell programs”. In: *Acm sigplan notices* 46.4 (2011), pp. 53–64. DOI: 10.1145/1988042.1988046.

[SBC12] Abhijit A Sawant, Pranit H Bari, and PM Chawan. “Software testing techniques and strategies”. In: *International Journal of Engineering Research and Applications (IJERA)* 2.3 (2012), pp. 980–986.

[DF14] Ermira Daka and Gordon Fraser. “A Survey on Unit Testing Practices and Problems”. In: (Nov. 2014). DOI: 10.1109/issre.2014.11.

[Pai+15] Ali Paikan et al. “A Generic Testing Framework for Test Driven Development of Robotic Systems”. In: *Modelling and Simulation for Autonomous Systems.* Springer International Publishing, 2015, pp. 216–225. DOI: 10.1007/978-3-319-22383-4_17.

[Hil+16] Michael Hilton et al. "Usage, costs, and benefits of continuous integration in open-source projects". In: *2016 31st IEEEACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2016, pp. 426–437.

[Bes17] Marc Bestmann. *Towards Using ROS in the RoboCup Humanoid Soccer League*. 2017.

[Afz18] Afsoon Afzal. "Quality assurance automation in autonomous systems". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. ACM Press, 2018. DOI: `10.1145/3236024.3275429`. URL: `https://www.researchgate.net/publication/328587601_Quality_assurance_automation_in_autonomous_systems`.

[Tim+18] Christopher Steven Timperley et al. "Crashing simulated planes is cheap: Can simulation detect robotics bugs early?" In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2018, pp. 331–342. DOI: `10.1109/ICST.2018.00040`.

[Afz+20a] Afsoon Afzal et al. "A Study on Challenges of Testing Robotic Systems". In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Oct. 2020. DOI: `10.1109/icst46399.2020.00020`.

[Afz+20b] Afsoon Afzal et al. "A Study on the Challenges of Using Robotics Simulators for Testing". In: *arXiv preprint arXiv:2004.07368* (2020). URL: `https://www.researchgate.net/publication/340683351_A_Study_on_the_Challenges_of_Using_Robotics_Simulators_for_Testing`.

Online Resources

[1] Martin Fowler. *Continuous Integration*. May 1, 2006. URL: `https://martinfowler.com/articles/continuousIntegration.html#AutomateTheBuild` (visited on 05/19/2021).

[2] *About status checks - GitHub Docs*. URL: `https://docs.github.com/en/github/collaborating-with-pull-requests/collaborating-on-repositories-with-code-quality-features/about-status-checks` (visited on 06/16/2021).

[3] *Agola*. URL: `https://agola.io/` (visited on 06/16/2021).

[4] *AWS CodeBuild – Fully Managed Build Service.* URL: https://aws.amazon.com/codebuild/ (visited on 06/16/2021).

[5] *Azure Pipelines | Microsoft Azure.* URL: https://azure.microsoft.com/en-us/services/devops/pipelines/ (visited on 06/17/2021).

[6] *Bamboo Continuous Integration and Deployment Build Server.* URL: https://www.atlassian.com/software/bamboo (visited on 06/16/2021).

[7] *Buddy: The DevOps Automation Platform.* URL: https://buddy.works/ (visited on 06/17/2021).

[8] *Buildbot.* URL: https://buildbot.net/ (visited on 06/17/2021).

[9] *Catkin Command Line Tools — catkin_tools 0.0.0 documentation.* URL: https://catkin-tools.readthedocs.io/en/latest/ (visited on 06/16/2021).

[10] *Cloud Build Serverless CI/CD Platform | Google Cloud.* URL: https://cloud.google.com/build (visited on 06/17/2021).

[11] *CloudBees CodeShip.* URL: https://docs.cloudbees.com/docs/cloudbees-codeship/latest/ (visited on 06/17/2021).

[12] *CMake.* URL: https://cmake.org/ (visited on 06/16/2021).

[13] *Concourse CI.* URL: https://concourse-ci.org/ (visited on 06/17/2021).

[14] *Continuous Integration & Delivery - Semaphore.* URL: https://semaphoreci.com/ (visited on 06/17/2021).

[15] *Continuous Integration and Delivery - CircleCI.* URL: https://circleci.com/ (visited on 06/17/2021).

[16] *Drone CI – Automate Software Testing and Delivery.* URL: https://www.drone.io/ (visited on 06/17/2021).

[17] *Empowering App Development for Developers | Docker.* URL: https://www.docker.com (visited on 06/23/2021).

[18] *Extending with Shared Libraries.* Jenkins documentation. URL: https://www.jenkins.io/doc/book/pipeline/shared-libraries/ (visited on 06/24/2021).

[19] *Features • GitHub Actions.* URL: https://github.com/features/actions (visited on 06/16/2021).

[20] *GitLab CI/CD | GitLab.* URL: https://docs.gitlab.com/ce/ci/ (visited on 06/17/2021).

[21] *GoogleTest User's Guide | GoogleTest.* URL: https://google.github.io/googletest/ (visited on 06/23/2021).

[22] *Intel® Neural Compute Stick 2.* URL: https://software.intel.com/content/www/us/en/develop/hardware/neural-compute-stick.html (visited on 06/29/2021).

[23] *Java Software | Oracle.* URL: https://www.oracle.com/java/ (visited on 06/16/2021).

[24] *Jenkins.* URL: https://www.jenkins.io/ (visited on 06/16/2021).

[25] *Licenses & Standards | Open Source Initiative.* URL: https://opensource.org/licenses (visited on 06/29/2021).

[26] *nosetests — nose 1.3.7 documentation.* URL: https://nose.readthedocs.io/en/latest/man.html (visited on 06/23/2021).

[27] *Open Robotics.* URL: https://www.openrobotics.org/ (visited on 06/16/2021).

[28] *Open Source Continuous Delivery and Release Automation Server | GoCD.* URL: https://www.gocd.org/ (visited on 06/17/2021).

[29] *pytest: helps you write better programs — pytest documentation.* URL: https://docs.pytest.org/en/latest/ (visited on 06/16/2021).

[30] *quickcheck - crates.io: Rust Package Registry.* URL: https://crates.io/crates/quickcheck (visited on 06/16/2021).

[31] *rostest - ROS Wiki.* URL: https://wiki.ros.org/rostest/ (visited on 06/23/2021).

[32] *Rust Programming Language.* URL: https://www.rust-lang.org/ (visited on 06/16/2021).

[33] *TeamCity: the Hassle-Free CI and CD Server by JetBrains.* URL: https://www.jetbrains.com/teamcity/ (visited on 06/17/2021).

[34] *The Go Programming Language.* URL: https://golang.org/ (visited on 06/16/2021).

[35] *Travis CI - Test and Deploy with Confidence.* URL: https://www.travis-ci.com/ (visited on 06/17/2021).

[36] *unittest — Unit testing framework — Python 3.9.5 documentation.* URL: https://docs.python.org/3/library/unittest.html (visited on 06/16/2021).

[37] *Welcome to Hypothesis! — Hypothesis 6.14.0 documentation.* URL: https://hypothesis.readthedocs.io/en/latest/index.html (visited on 06/16/2021).

[38] *Welcome to nose2 — nose2 0.6.0 documentation.* URL: https://docs.nose2.io/en/latest/ (visited on 06/16/2021).

[39] *Welcome to Python.org.* URL: `https://www.python.org/` (visited on 06/16/2021).

[40] *Xunit: output test results in xunit format — nose 1.3.7 documentation.* URL: `https://nose.readthedocs.io/en/latest/plugins/xunit.html` (visited on 06/23/2021).

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den ____________________: ____________________

Einstellung in die Bibliothek der Informatik

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den ____________________: ____________________