ROS-Industrial Quality-Assured
Robot Software Components - ROSIN

Quality Assurance Process and Community Management in ROS

D3.1

Grant Agreement No.	732287
Start date of Project	1 January 2017
Duration of the Project	48 months
Deliverable Number	D3.1
Deliverable Leader	ITU
Dissemination Level	PU
Status	1
Submission Date	01/09/2017
Author	**Yvonne Dittrich, ITU (ydi@itu.dk)** Claus Brabrand, ITU (brabrand@itu.dk) Gijs van der Hoorn, TUD (g.a.vanderhoorn@tudelft.nl) Jon Azpiazu, TEC (jon.azpiazu@tecnalia.com) Iñigo Martinez, TEC (inigo.martinez@tecnalia.com) Nicolas Limpert, FHA (nicolas.limpert@alumni.fh-aachen.de) Jonathan Hechtbauer, IPA (jonathan.hechtbauer@ipa.fraunhofer.de) Jon Tjerngren, ABB (jon.tjerngren@se.abb.com) Adam Alami, ITU (adaa@itu.dk) Andrzej Wąsowski, ITU (wasowski@itu.dk)

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 732287.

This page has been intentionally left blank

Modification Control

Version #	Date	Author	Organisation
1.0	01-09-17	**Yvonne Dittrich,** Claus Brabrand, Gijs van der Hoorn, Jon Azpiazu, Iñigo Martinez, Nicolas Limpert, Jonathan Hechtbauer, Jon Tjerngren, Adam Alami, and Andrzej Wasowski	ITU, TUD, Tecnalia, FHA, IPA, and ABB

Release Approval

Name	Role	Date
A. Wasowski	WP3 Leader	31-08-2017
C.Hernandez Corbato	WP2 Leader/ Coordinator	01-09-2017
A. Ferrein	WP4 Leader	31-08-2017
M. Bornignon	WP5 Leader	01-09-2017
M. Wisse	WP6 Leader	30-08-2017

History of Changes

Section, page number	Change made

Executive Summary

The Deliverable 3.1 “Quality Assurance Process and Community Management in ROS” explore the current state of Quality Assurance (QA) in the ROS and ROS-Industrial open source communities and understand the challenges with regards to the quality of robot applications based on ROS and containing ROS software. The task is addressed along three avenues: the analysis of the wiki and other material provided by the of the ROS and ROS-Industrial communities; interviews with different community members; and the analysis of bugs and their correction in ROS. The analytical results are complemented with initial interventions improving the continuous integration infrastructure and starting a quality hub, a web site that is meant to promote QA in the community.

The chapter 2 reports the investigated ROS and ROS-Industrial current quality assurance (QA) practices and processes based on the ROS and ROS-Industrial wiki and infrastructure. Some industry-wide accepted QA practices and processes, like well-defined development process, defect management, code review, continuous integration and testing and knowledge sharing, have already been embraced and adopted especially in the evolution of core packages.

However, our analysis suggest that QA practices are experiencing implementation and execution challenges. Some of the key issues are highlighted below: e.g. standards are not consistent across the community development roles; documentation is outdated; inconsistency in the QA practices across the various development streams (i.e. Core, Drivers and Reusable packages).

The interview study reported in chapter 3 confirms and complements this analysis: It reports how different developers – core developers and maintainers, a developer of drivers and application developers perceive the QA challenges of developing ROS, respectively, developing robot applications based on ROS; QA is not consistently handled by all maintainers; the community is unclear about what are the currently practiced standards; there is no onboarding and knowledge transfer process in place for new members; the complexity of robotics development is not considered in the current implementation of QA processes and tools. Especially, there is a lack of ownership of QA in the ROS and ROS-Industrial communities.

Chapter 4 on "ROS Quality Issues" describes the results of a quantitative analysis of 177 known bugs harvested from six different ROS systems and a few *confidential* systems. The results show that, currently, most ROS bugs eventually get fixed; i.e., very few bugs remain unfixed indefinitely. The results, however, also show that one in five bugs in ROS are detected at runtime, often by users, which is not good for the reputation of ROS.

The results indicate that a CI (Continuous Integration) service simply compiling and building ROS projects automatically ought to catch a number of errors. Since one out of five errors occur as a result of software evolution, the CI service should take care in re-checking all code that interacts with recently modified code. Our results indicate that a wide range of bugs could be caught by extending the CI service to run a collection of different kinds of code-scanning tools. About 36% bugs would require difficult higher-level development techniques. A number of domain specific languages (DSL) are involved in bugs, which would require development of dedicated tools. Finally, a number of cross-language bugs that occur as a result of inconsistencies between the languages used. Again, these would require dedicated tools to be built.

Chapter 5 describes two "early interventions" in the current ROS quality assurance processes and tools. The first contribution is the extension of the ROS wiki and buildfarm to more clearly display

the results of unit tests and continuous integration for all ROS packages that have been registered with the ROS buildfarm. Instead of only showing whether a package has enabled testing, status indicators on the wiki show the actual results of tests, which provides a much better insight into the status - and quality - of software components. The second contribution is the creation of the ROS Quality Hub: a 'home' for QA for the ROS and ROS-Industrial communities. As a start, it is populated with a database of patterns documenting QA practices and tools for ROS components. The ROS Quality Hub will be hosted on a publicly accessible website that will be disseminated among ROS and ROS-Industrial developers. The material published on the quality hub will be further developed throughout the runtime of ROSIN.

Chapter 6 concludes the deliverable by outlining a number of possible directions of how to address the identified challenges: clarifying QA processes; making QA practices and tools easily available; making the quality of community contributed packages and drivers visible; supporting knowledge sharing and on-boarding among contributors; and developing and implementing code scanning tools.

Table of Contents

1. Introduction

1.1 Purpose of this document

The task of this deliverable is to explore the current state of Quality Assurance (QA) in ROS and ROS-Industrial and the challenges with regards to the quality of robot applications based on ROS and containing ROS software.

We decided to address this task along three avenues: the analysis of bugs and their correction in ROS; interviews with different community members; and the analysis of the wiki and other material provided by the of the ROS and ROSIN communities.

The results are documented in three different ways:

1. This report represents the results of the analysis in textual form with the purpose to identify challenges and prepare possible ways to improve QA in ROS and ROS-I.
2. A proof-of-concept implementation of tighter interfacing of ROS CI (Continuous Integration) services with ROS wiki, to better incentivize use of CI and to better announce availability of the CI services in the community. This includes contributions to upstream projects.
3. The ROS Quality hub: A website that is meant to be continuously developed throughout the project and provide QA relevant material for the community. We start with QA patterns implemented by the ROS community for driver and application developers with pointers to develop quality software and, at the same time, explains how the ROS core is maintained to be of high quality. The web site can be found at http://rosin-project.github.io/quality-hub.

The results of this first delivery are meant to inform the decision on what to focus on when working with the improvement of both the software quality and the quality assurance during the remainder of the project.

Before presenting the results of the work below, the following two sections introduce ROS as a software product and give a short overview over the history of ROS. Both underpin the following sections.

1.2 ROS as a Software Product

The ROS core needs to be regarded as a generic software product, a middleware that is not functional in itself. A robot application based on it requires both, specific application code and drivers making specific hardware available through the ROS middleware. As with many software products the different parts of the software are not necessarily developed by the same developers or organisation (Dittrich, 2014). A company developing specific robot applications e.g. uses the ROS core maintained by the ROS community, uses maybe a driver provided and maintained by the company providing the robot to be used, and might choose other re-usable modules developed by third parties, e.g. to compute the movement of the robot's arm.

The core provides basic functionalities, that implement the core design idea to regard robotic system as sets of connected sensors and actuators that pass messages. For each concrete robot and each sub system like cameras and manipulators a driver needs to be developed. The driver of a robot provides a way to control the hardware of the robots: the sensors and actuators. This can be combined with descriptions of the geometry and kinematic structure. Based on the core and the relevant drivers, an application is developed that programs the robot to implement certain actions either based on fixed routines or in response to input from cameras or other sensors.

Based on the analysis of the community and the research produced here we have decided for chapter 2 and 3 to distinguish three modes of development of and with ROS. Each development requires different measures for quality assurance and face a different set of challenges, as will be further developed in the chapters 2 and 3:

- **Core development:** Here we can identify three roles: contributors of bug-fixes or patches, developers, and maintainers of the modules - often the developer of a module would also act as a maintainer, that however is not a necessity. The quality of the code is subject to negotiation between the maintainer on one side and the contributor on the other side. New developments that affect a wider range of modules or set borders for future development are subject to a REP (ROS Enhancement Proposal) that has a detailed process including reviews and deliberation by at least the core members of the community, but often also by subject experts.
- **Driver development and development of reusable packages** is the second category. Though drivers of hardware are also packaged as reusable components, they are distinguished from other re-usable components as they interface with hardware and require a different kind of quality assurance, namely to test against the hardware. Further, robotics related expertise is needed to design the driver to allow application developers to make best use of the robot’s capabilities.
- **Application development** uses the ROS core plus hardware plus drivers and other reusable packages to program a robot to implement a specific task. The challenge here is the complexity of the resulting software and the interaction of the robot with the concrete environment. Erroneous behavior can point to faults in the application itself, but also to errors in core modules or drivers or to problems due to unanticipated usage (by the author(s) of the reused drivers and other components).

Though the development or contribution to core modules, application development and the development of drivers are clearly distinguishable development tasks, they are often done by the same developer and can take place in an intertwined fashion: Both in the context of research and in industrial applications. The development of an application might require to adjust the hardware - e.g. by adding an additional camera - and with that add a new driver and adjust the geometrical model of the robot. A research group developing innovative algorithms that allow robots to cooperate in a safe fashion might at some point find a bug in the ROS core and decide to correct it by submitting a patch to the code repository. Part of the own application might eventually become open-source as well, to be used by other research groups. Most of the core maintainers participate in application development as well.

1.3 History and Background of ROS

The initial version of ROS was developed in the mid 2000s as Open Source Software at Stanford University (ROS.org, 2017). In 2007, the Willow Garage, a robotics start-up focusing on non-military robots, started to contribute to the development of ROS to the extent that some describe it as Willow Garage ‘taking over’ the ROS project (willowgarage.com, 2017). Willow Garage in turn was a core founder of the Open Source Robotics Foundation (OSRF) (osrfoundation.org, 2017), a non-profit organisation dedicated to promote Open Source Robotics. Today, most of the maintainers of the core modules of ROS and Gazebo, the robot simulator based on ROS, are employed at the OSRF.

Whereas the core modules are developed and maintained by a small group of close knit developers, the tools, drivers for robots and specialised hardware, and modules for specialised tasks, like path finders, are developed, maintained and distributed by both companies and research groups in a decentralized manner. That means each author is free to distribute his packages.

Currently the OSRF is working on ROS 2.0, based on a massive redesign changing part of the technology stack and improving the platform independency of the basic modules. (See section 3 for more information.)

ROS-Industrial or ROS-I has been founded in 2012 to promote the use of ROS in industry. Besides the original US based organisation, a European consortium and an Asia-Pacific consortium have been created (rosindustrial.org, 2017). The ROS-Industrial organisation hosts software that extends the ROS core with functionality that is especially relevant for industrial and manufacturing related (robot) applications. It further hosts a number of drivers for industrial robots and other automation hardware.

1.4 Outline of the Report

The remainder of the report is structured as follows:

Chapter 2 presents the result of the analysis with ROS community members. Here we focused on interviewing members expert with different kinds of development of and with ROS. We discuss the distinction of different kinds of developments within the ROS ecosystem and connect that to different QA challenges and the respective tools and techniques applied to address these challenges.

Chapter 3 provides an analysis of the ROS wiki and website regarding QA as well as an analysis of the QA related tools like the use of git, issue trackers, continuous integration and testing.

Chapter 4 then presents an analysis of historical bugs in the ROS components. We investigated a representative set of ROS packages to determine the types and numbers of bugs that are common in ROS and ROS-Industrial development. This gave us an orthogonal, problem-based perspective (as opposed to community member-based perspective) on QA practice and challenges in ROS. The results of this analysis are discussed and directions for future analysis and tool development proposed.

Chapter 5 provides an overview over the changes developed for the continuous integration infrastructure and an overview over the website we developed.

Chapter 6 presents a summary and points out possible avenues on how to improve QA in ROS and ROS-I.

2. The ROS Community Quality Assurance

2.1 Introduction to chapter

ROS is open-source software. That implies that the development and evolution of the software and its quality is not controlled by one company or organisation, but is subject to a community based negotiation. This has implications on the quality assurance: Different from a company's internal software development, quality assurance (QA) processes cannot be decided by management, and developers cannot be ordered to comply with these processes.

That does, though, not mean that the quality of the resulting software is necessary lower. open-source software is used more frequently, which implies that defects that are not caught by the tests of the developer will show up quicker and will be corrected. Further, open-source communities developed what Scacchi (2002) calls 'informalities', practices of developing software based on the community infrastructure of tools and communication channels, common rules or habits, some codified in written and unwritten rules, and the joint interest in the continuation of the software development.

These common practices though are not set in stone, but are negotiated and renegotiated based on evolving interests and observed needs. Sigfridsson (2010) e.g. shows how an open-source community explicitly discuss how to adapt their development to attract new developers hand help them to contribute to the development. The community processes are adapted accordingly.

This section aims to map out the QA practices, starting with the analysis of the ROS development infrastructure and the part of the ROS wiki detailing the development process and QA measures that should be applied. We complement the document analysis with the interviews that are also the basis for the analysis of quality challenges and remedies from the perspective of community members in the previous section.

The goal is to better understand how QA today takes place in the ROS community in order to provide directions for future action in order to increase the quality to support industrial strength applications.

The next section shortly details the methods applied. The then following section presents the results.

2.2 Method

In order to document the QA practices in the community, this section analyses the ROS wiki and the setup of the development environment. The document and tool analysis is triangulated with the analysis of the respective parts of the interviews that are the basis for the identification of challenges and useful support by community members. Further, the preparation as well as the resulting text is reviewed by project members who are themselves part of the community. This can be regarded as 'member checking' which is part of the quality assurance in qualitative research.

2.3 Results

The results of the analysis are structured as follows: The next section gives a historical background of the ROS software and community. Thereafter, we present the ROS community QA practices and link to the respective pages, sites and tools of the ROS wiki and infrastructure. The last section will identify issues, especially the difference between what is presented on the web site and real practices.

The presentation of the ROS community QA practices presented here are based on the analysis of the ROS documentation, web site and the development environment provided by and for the community. One of the first observations, which is further discussed below, was that the current structure of the wiki does not support orientation regarding the QA practices and the support provided by the community.

The team therefore decided to describe the QA practices in form of ‘practice patterns’ – low ceremony method descriptions detailing the context, the problem and the solution as main parts of the description (Dittrich, 2016). The patterns serve both as a presentation of the practices and are meant to be published to the community as a way to provide an entry point to the relevant pages and tools of the ROS community site. Though the documented practices are applied and supported by tools, their application depends on the specific context. Whereas the practices described for the core development are implemented due to the community of maintainers of the core modules, the application of patterns for package or driver development and application development depends on the individual developer and the organisation.

The discussion or the practices is structured in line with the identified developer personas in depth discussed in section 3: These personas are based on the initial analysis of the ROS web site and initial discussion with the team members who also are ROS community members.

2.3.1 Core development

The core of ROS has been developed and continues to be maintained by a few close knit persons working at the OSRF. The OSRF also supports the community by hosting and maintaining a build farm that allows for continuous integration as part of the github supported development process.

The ROS-Industrial core and other packages are maintained by a more distributed group of maintainers, who nonetheless aim at keeping common high quality standards.

QA of the core modules takes place along two lines: The day-to-day issue tracking and community development process that is steered through the github setup. Patches are developed, submitted, reviewed and tested before being integrated in the newest distribution and eventually backported. Bigger changes and additions are managed through REPs, ROS Enhancement Proposals. The process for such a proposal is described in a meta-REP. The list of accepted REPs represents one of the most important sources to understand the design rationale of the ROS core modules. Below these two levels of core development QA are presented and discussed.

REPs: ROS Enhancement Proposals

The core of the strategic decision taking is defined and documented as REPs, Ros Enhancement Proposals. REP 0 (http://www.ros.org/reps/rep-0000.html) lists all REPs, both the valid ones and the abandoned ones. The numbering of the REPs in the following follows this list and the respective REPs can be accessed through the above-provided link.

REPs 1, 2 10 and 12 together define the REP process, scope, voting guidelines, and a template for

REPs. Though the community is promoting sharing of functionality, changes to the ROS core are deliberated with greater care: As the core packages of the ROS distributions are defined in a REP, the inclusion of a new package into the distribution core has to follow the REP process.

REPs can also be informational, defining standards and describing design rationales across the whole core, e.g. REP 103 defines the "Standard Units of Measure and Coordinate Conventions". As all REPs are saved, whether they are active, implemented or withdrawn, they provide a concise summary of core decisions that have shaped and are shaping the development. The latest REP - 147 "A Standard interface for Aerial Vehicles" - has been created in May 2016 and has not yet been made active.

ROS-Industrial applies a similar process for proposing and deliberating changes to its core modules and agree on common standards and interfaces (ROS-Industrial GitHub repository, 2017).

The REP process has been adapted from the Python community. According to interviewee GH, the discussion of REPs is not as active as similar discussions in the Python community. As a reason he suggests that many community members might not feel competent enough to contribute to the discussion.

Issue tracking and maintenance routines

Bugs and enhancement proposals are handled through the Github tools for distributed development: Packages in ROS and ROS-Industrial have a github repository which includes an issue tracker. The issue tracker is the main communication channel between the community members using ROS and running into an error and the developers resp. maintainers of a package. The issue tracker is then used by the maintainer to further explore the issue or question.

As in all open-source projects, everybody in the community is invited to contribute to the development. This is done by submitting a pull request to the repository hosting a package and the owner of this repository. The pull request often refers to an issue previously reported in the issue tracker. The process of submitting a pull request is further described in the pattern 'Submitting a Pull Request'. Changes to the package that are related to an issue refer to this issue in the pull request on github.

Pattern 1: Submit a pull request

Name

Submitting patches to core packages through maintainers

Context

An application developer discovers a bug in a core ROS package (e.g. roscpp), corrects it and tests the patch on his own computer. The fix should be merged into the main code base, but that should be done in a controlled manner.

Problem

How to ensure quality of the proposed changes to the core – and of the software as a whole after integration of those changes – and at the same time encourage newcomers and members of the ROS community to participate in the development?

Forces

Striving for quality

Everybody and especially maintainers are interested in maintaining the quality of core packages, as the introduction of new bugs or regressions or reducing maintainability will affect all current and future users of ROS. (Perceived) usability and stability have a direct influence on the reputation of ROS – especially in industrial settings where quality of the core packages is often considered paramount.

Guarding development (policies)

ROS development is governed by a number of policies. One example is that development always targets the latest release, with patches being backported to older and *long term stability* (LTS) releases whenever this is considered desirable or required. Another example would be maintaining portability of code in order to remain platform independent as much as possible. Not every newcomer is aware of these policies.

Involve community members in development

As every Open-Source Software project, ROS lives through the enthusiasm of its community members. But this also requires that community members can participate in the development, learn about the design and code structure and can earn merits in order to take on more responsibility in future. Contributing to core packages is an important step in the onboarding of new community members.

Educate community members

To successfully contribute, newcomers need to learn how to write high quality software in-line with ROS' best practices. These principles can best be taught based on actual development rather than only publishing them on the wiki and expecting newcomers to read, understand and apply them before writing and submitting their patch.

Maintaining efficiency

Blocking contributions from being merged into the main code base for too long can be detrimental, both to the engagement of the submitter, as well as to the chances that they will get merged (as a patch may have been written for an older version of the software, increasing the effort required to make it compatible with the current state). As such, Pull Request reviews and iterations should be efficient, with minor (cosmetic) issues not holding up the process.

Solution

Community members as *Submitters* submit a change through a *Maintainer*. The Maintainer should guard the quality of both the contributions and the result of merging the contribution with mainline by making use of their understanding of ROS and of the automated quality assurance tooling.

Stakeholders

A **Maintainer** is either part of the core ROS development team, or a well reputed community member who has taken on the responsibility for a (number of) core packages. The Maintainer 'owns' the respective repository on a ROS Github organisation. His interest and task is to make sure that changes to this repository adhere to the coding standards, that the development guidelines and policies are followed and that introduction of changes by the community does not diverge from the overall design of the software components affected (see also MaintenanceGuide on the ROS wiki).

A second interest is to involve a growing number of community members in the development and to educate newcomers about ROS development guidelines and policies, so that they eventually will be able to not only develop better patches but also might be able to take on more responsibilities.

A **Submitter** is every other community member who contributes for instance a bug fix to a core package. These could be developers using ROS to develop applications and encounter a problem or could also be developers of drivers or maintainers of other core packages. The submitter creates a Pull Request containing the patch and – ideally – a unit and / or regression test for the new or affected functionality.

Tools involved

Github is set up to require approved code reviews of Pull Requests by maintainers before such PRs can be merged into mainline.

The **ROS Buildfarm** runs **Continuous Integration** tests on all PRs submitted against core repositories, and Github is configured on core repositories to require successful test runs before PRs can be merged.

Maintainers can execute components locally to do **runtime or acceptance testing** when appropriate (for instance because an automated integration test is not available or possible due to hardware requirements).

Example

A user of the roscpp client library identifies a bug in queue management (wrong order of operations) that results in messages being lost and submits a PR to remedy this after finding the cause and testing it locally on his own machine (https://github.com/ros/ros_comm/pull/1054).

The maintainers of roscpp now need to merge this patch, after making sure it is safe to do so.

Example resolved

The pull request is reviewed, updated and finally merged by two maintainers of roscpp, which are referred to as MA and MB respectively. The chain of events in the review, polishing and final merge of the PR (and related PRs) is as follows:

- user identified problem in a core package (ros_comm/actionlib)
- he diagnoses it, writes a patch and tests that locally
- he then submits a Pull Request against the indigo-devel branch (second-to-last LTS)
- MA identifies some small omissions in the submitted patch and proposes some fixes
- MA creates a new PR based on that of the submitter, as the original PR did not target the correct branch (policy: PRs should target latest ROS release). Submitter's PR is closed in favour of that one. The new PR also includes a regression test contributed by MA.
- MB reviews the replacement PR by MA (policy: many-eyes) and waits for the Continuous Integration server to complete the tests.
- MB merges the replacement PR into the latest development branch.

At this point maintainer B decides to *backport* the merged fix to other development branches for older ROS releases. He will do that at a later time together with other fixes for which backporting is desirable.

Links

List of core modules

The list of packages which are considered part of the core are documented in REP-142, sections *ROS Core* and *ROS Base*.

Best practices

- For developing ROS libraries and core components see the ROS Developer's Guide.
- For maintaining and releasing ROS libraries and core components see the ROS Maintenance Guide.

Consequences

There must be at least one maintainer per core package (but preferably more).

Maintainers need to have access to the necessary tools, both locally and remote (CI output of ROS Buildfarm).

It becomes possible to add more Q&A tooling to the buildfarm to more easily enforce Q&A process / best practices (make it less subjective).

Known Uses

Many open-source systems have assigned the responsibility for the quality of core modules to so called maintainers, among them the Linux project: https://github.com/torvalds/linux/blob/master/MAINTAINERS

Related patterns

Continuous Integration Testing

Test automation.

As the pattern describes, the maintainer of the package reviews the submitted code and reviews the results of the regression tests on the buildfarm that is triggered by the pull request. He provides feedback to the contributor. The contributor is expected to improve the code according to the feedback, especially to make sure that it passes the regression tests run by the Continuous Integration infrastructure. The pattern 'Accept a pull request' describes the situation from the perspective of the maintainer and provides an overview of the responsibilities of package maintainers both for core packages and for other packages available through the ROS site.

Pattern 2: Accept a pull request

Name

Accepting a Pull Request

Context

A user has identified an issue in a package, diagnosed it, wrote a fix and now submits a pull request. The maintainers of the package want to merge the contribution, but they need to make sure that that happens in a controlled and predictable manner.

Problem

Merging in of any (external) contribution presents a risk: specific bugs may be fixed or functionality enhanced, but at the same time new bugs may be introduced or repositories may diverge architecturally.

Following predefined procedures can help, as they reduce the chances of making mistakes during the review process, but which aspects of pull requests need to be checked, and how can maintainers cooperate during the review?

Forces

Controlled development

Well-defined processes for contribution will increase the probability of avoiding messing up the main code base. Which for example otherwise could result in components that are difficult to understand and use. This could be prevented by checking that any used code conventions and styles are followed.

Striving for quality

Everybody and especially maintainers are interested in maintaining the quality of core packages, as the introduction of new bugs or regressions or reducing maintainability will affect all current and future users of ROS. (Perceived) usability and stability have a direct influence on the reputation of ROS – especially in industrial settings where quality of the core packages is often considered paramount.

Guarding development (policies)

ROS development is governed by a number of policies. One example is that development always targets the latest release, with patches being backported to older and *long term stability* (LTS) releases whenever this is considered desirable or required. Another example would be maintaining portability of code in order to remain platform independent as much as possible. Not every newcomer is aware of these policies.

Solution

The following is an example workflow for reviewing pull requests.

General, high-level checks

1. Make sure the PR does not introduce regressions:
    a. If a CI infrastructure is available: check its status
    b. Otherwise run tests manually / locally: this could include human-in-the-loop tests if necessary (fi when testing requires significant human-machine interfacing)
2. Check proposed changes for adherence to conventions:

a. ROS REPs: as far as applicable
b. ROS code style: automated if available (clang-format) or manually
c. ROS naming conventions (packages, nodes, topics, services, actions, coordinate frames, etc)
d. Repository/package specific conventions
e. Design/architectural: would acceptance of the changes cause the overall design of the package to significantly diverge from its current structure (both static and dynamic)?

If the pull request is a bug fix:

1. Check that either a new test is included or that an existing one is extended or adapted that proves that the issue is fixed

If the pull request introduces new functionality:

1. Check that a test is included that covers the new functionality

Detailed checks

If the pull request is a bug fix:

1. Do the proposed changes actually fix the reported issue?
2. Is the fix generic enough, or does it only work for the submitter?
3. Does the fix not conflict with other uses of the code (ie: those that might not be immediately apparent to the submitter)?
4. Is there a less invasive, more efficient, easier or more maintainable solution that would be an equivalent fix?

If the pull request introduces new functionality:

1. Is the proposed functionality actually a new feature?
2. Does the proposed functionality do what is claimed?
3. Is the new feature generic enough, or does it only address a use-case of the contributor?
4. Is there a less invasive, more efficient, easier or more maintainable implementation that would result in the same enhancement?

Accepting the contribution

Pull requests should only be accepted and merged if the above checks have all been completed and the review has been performed by at least two maintainers.

Links

http://wiki.ros.org/Industrial/Tutorials/IndustrialPullRequestReview

http://wiki.ros.org/MaintenanceGuide

http://moveit.ros.org/documentation/contributing/pullrequests

Consequences

Policies and standards for reviewing pull requests must be made available to maintainers.

Maintainers must be aware of the policies regarding pull request review.

All pull requests will have been reviewed by at least two maintainers.

The repository must have been setup to run CI on pull requests.

Contributions must be accompanied by sufficient rationale as to why they are to be included (and as to why the change introduced).

Known Uses

All pull requests against ROS repositories maintained by the OSRF use a similar - albeit implicit - review policy.

The ROS-Industrial community also has a similar policy for reviewing contributions.

Another example is the maintainers policy of the MoveIt community.

Related patterns

Standards and Patterns (Driver Developer)

Submit a patch (Component Developer)

Submitting patches to core packages through maintainers

CI with public infrastructure (App. Developer)

Best Practices

The role and tasks of maintainers in ROS is specified on http://wiki.ros.org/MaintenanceGuide. The ROS-Industrial wiki details the tasks of a maintainer as follows:

"Maintainers perform the day to day (release to release) tasks that ensure existing packages continue to build and are available as binaries and/or source. Typical tasks include:

- Keeping documentation up-to-date (minor changes)
- Answering user questions
- Reporting bugs and performing minor fixes
- Reviewing/accepting minor pull requests (PRs)
- Updating and releasing new package versions

The typical maintainer commitment is about 4 hours/week on average for mature repositories. Less mature repositories may take more effort early on."

According to our interviewees, the criteria guiding the review of a pull request on ROS are not specified clearly and are not documented; they can differ from maintainer to maintainer. According to our interviewees, the unclarity of the criteria can lead to frustration between maintainer and contributor. anyhow

The Python style guide - together with an extension for how to make Python modules available for the ROS build system - is part of REP 8. Other style guides are listed in the developer's guide on the wiki. Best practices, conventions and the like are listed in two places on the ROS wiki. Other websites providing orientation on how to write ROS applications and contribute to ROS core can be found in the tutorials and on the Developer's Guide page:

- http://wiki.ros.org/ROS/Patterns/Conventions presents naming conventions for packages but also when using ROS, e.g. how to name topics & services (communication channels in a ROS application) when developing an application.
- http://wiki.ros.org/BestPractices lists a number of best practices mainly related to how to best make use of ROS and to avoid errors.
- http://wiki.ros.org/DevelopersGuide .This page is part of a set of pages that describe the

QA process (http://wiki.ros.org/QAProcess). According to our interviewees, these pages are not well maintained any more and are hardly used. They seem to not be used as a reference for the code review process of maintainers.

ROS-Industrial has additional information on how a pull-request should be handled: http://wiki.ros.org/Industrial/Tutorials/IndustrialPullRequestReview. Each pull request to the ros-industrial and ros-industrial-consortium repositories is reviewed by at least one other ROS-Industrial committer. Pull requests need to be small enough to make reviews possible (ie: authors should avoid introducing too many changes in a single pull request). The page though is kind of hidden among the tutorials, and it does not make the criteria to be applied in the review explicit. The page on the ROS-Industrial development process (http://wiki.ros.org/Industrial/DevProcess) mentions a list of style guides and also points to the continuous integration as a means to check for broken references.

Continuous Integration

In order to guard the development against regressions - unintended introduction of bugs in old code when developing a patch to address a different bug - the OSRF provides an infrastructure for continuously building and testing especially the core packages. The continuous integration mechanism is connected to the Github repositories so that contributors can easily build and test their contribution before submitting a pull request. The continuous integration with this public infrastructure is described in the respective pattern.

Pattern 3: Continuous integration with the public infrastructure

Name

CI with public infrastructure (App. Developer)

Context

Running tests should be automated, and should preferably be done after each change to the code. In addition, (proposed) changes should be checked to make sure they don't introduce new bugs or break already existing functionality. Such regression testing should also be automated and should ideally also be run after introduction of changes.

Finally, development of ROS applications is often done in a collaborative or federated way, introducing the need to share test results and metrics between collaborators in an efficient way.

Problem

Setting up Continuous Integration for a code repository can be daunting, as not only does the CI service itself need to be configured correctly, but the repositories to be tested will need to be properly set up as well. Such configuration is in addition to the tests that need to also be present.

And in order for results to be viewable by all collaborators, a shared infrastructure will be needed, which further increases setup and maintenance overhead.

Forces

Producing Quality Code

Developing quality code is a process that can be facilitated by the use of the appropriate tools. CI is one of such tools, and a cornerstone in quality assurance procedures, providing automated ways of ensuring reproducibility, regressions tests, deployability and so on.

Striving for Quality

Everybody and especially maintainers are interested in maintaining the quality of (core) packages, as the introduction of new bugs or regressions or reducing maintainability will affect all current and future users of ROS. (Perceived) usability and stability have a direct influence on the reputation of software – especially in industrial settings where quality of packages is often considered paramount.

Increasing trust

The use of Continuous Integration to continuously test and analyse software components and guard them against regressions increases trust in those components: a high coverage (and growing) test suite with a long history of succeeding tests are a good indication of the quality of the software.

Testing also increases trust and confidence of developers, as such continuous testing functions as a 'safety net' or 'canary' that warns developers whenever changes to be introduced would cause regressions and / or unintended side-effects.

Solution

Re-using a publicly available Continuous Integration infrastructure significantly reduces the effort required to introduce unit and regression testing into a project. Hosting code on publicly accessible repositories further reduces the administrative burden, leaving just the configuration of the repositories to be tested.

Within the ROS and ROS-Industrial communities several ready-to-use CI setups are provided. An overview is given on the Continuous Integration page on the ROS wiki.

ROS Buildfarm

The ROS buildfarm can be configured to run CI for user repositories. This includes both CI for every change committed to a repository (called *Development Tests*) as well as CI for specific Pull Requests (*Pull Request testing*, only GitHub supported right now).

Configuring development tests for a repository is documented in REP-143. In order to add pull request testing, refer to the buildfarm/Pull request testing page on the ROS wiki.

Travis CI

In all cases, the Travis CI service needs to be enabled for the repositories that should be tested. The Travis Documentation shows how this can be done. After this, testing setup can either be done manually, or by using the industrial_ci package provided by the ROS-Industrial project.

Vanilla Travis

Again, consult the Travis Documentation for how to create the CI configuration. This includes the tools used, software that needs to be present in order to build the software and run the test, combinations of OS and versions of the software to test.

In order to test ROS and ROS-Industrial packages, additional configuration is required, including adding the ROS package repositories, installing ROS, creating a workspace, configuring the workspace, resolving and installing all dependencies of the packages under test, building the workspace and finally running tests and gathering the results.

There is no official documentation in ROS on how to do this, but felixduvallet/ros-travis-integration is one example repository that shows how this may be done.

With industrial_ci

To exploit the fact that many ROS packages are tested in similar ways, and thus the fact that the configuration of the repositories that host them is also similar, the ROS-Industrial project has made the industrial_ci package available. In cases where the template configuration that it provides can be reused, this greatly simplifies Travis setup for a particular repository.

The Quick Start documentation lists the necessary steps. More complicated setups (such as those needing special build or test dependencies) are covered in the detailed documentation.

Links

- REP-143
- REP-141
- Indexing Your ROS Repository for Documentation Generation
- ROS wiki: CIs
- ROS wiki: buildfarm/Pull request testing
- ROS wiki: regression_tests/Development Tests

Consequences

Contributors will be notified of test results as soon as the CI service has completed running the tests.

Developers will need to keep test suites up-to-date and meaningful: the CI service only automates running the tests, gathering the results and reporting on them. It does not create any tests itself. As tests are also code, this means increased maintenance for maintainers.

Maintainers and developers will need to guard against a false sense of security: a 'green badge' from the CI service does not necessarily mean that all is ok. Low coverage from test suites may leave defects undiscovered.

Everyone, not just the developer or maintainer, has access to build and test results for registered public repositories.

Related patterns

Continuous Integration with private repositories

Integrate tests in catkin

Pre-release Testing

Regression Testing

Submitting patches to core packages through maintainers

This together with the maintenance routines, the continuous integration infrastructure represents the backbone of code quality assurance in the core development of ROS. Continuous integration though is only as powerful as the tests that can be run on the code. The last related QA practice that both supports the maintenance routines and complements the continuous integration practices is the infrastructure for testing individual routines but also components in the interaction in a running robot system.

Unit test and ROStest

Many of today's programming languages and development environments provide the possibility to define *unit tests* that are executed automatically when a new version of the program is built. In order to test not only the individual procedures, but also to allow to test the interaction between different components, the *ROStest* framework has been developed to support *integration testing* (http://wiki.ros.org/ROStest). Pattern 4 describes how to work with regression tests in the ROS continuous integration environment. Pattern 5 shows how to integrate unit tests and ROStests in the build so that they can be run as part of the continuous integration environment.

Pattern 4: Regression test (Unit test)

Name

Regression Testing

Context

A developer fixes a bug in a core ROS package (e.g. roscpp), corrects it and tests the patch on his own computer. The developer wants the bug not to be reintroduced later. The pattern is beneficial for component and application developers as well, but it is mostly used by core developers (core components are easier to test, and it is more important to test them). Any ROS project (including application and component process) that decide to use continuous integration, should seriously consider also using regression testing.

Problem

The problem is that often bugs are (re-)introduced by developers not knowing the context sufficiently well, and bugs keep returning the project (they are known as regression bugs), because the rationale for a certain decision in code is forgotten. The problem is to prevent future developers from reintroducing the bug.

Forces

Avoiding Reintroduction of Problems

Using tests for past bugs influences quality in the long term. It is a well accepted practice in many open-source projects. Writing regressions tests, you contribute to long term quality of the ROS ecosystem. You enable maintenance of code, and better automated discovery of inconsistencies.

Documenting Design and Requirements

Regression tests document your coding decisions, and communicate to other developers automatically about their violation. Thus tests become documentation for your code.

Enable Others to Contribute to ROS

It is very difficult for new external developers to contribute to your components. When they make changes to code, they are often doing it in the blind, driven by a lot of guesswork. By providing a harness of regression tests, you help them in the task. They get immediate feedback for their changes. It becomes easier to contribute to a project.

Amplifying Value of Continuous Integration

Regression tests, along with normal scenario-based requirements tests contribute to overall body of automated tests for your component. This increases effectiveness of the build system and of continuous integration (CI). Your component is better tested against evolution of other APIs that it depends on (CI servers will tell you better and more precisely what problems develop in your code).

Development Cost

Regression testing comes at a cost: you need to develop a test, which sometimes may be difficult or costly. Sometimes it might also be nontrivial, as the test should be automatic.

Maintenance Cost

Regression tests need to be maintained. When the design of the component changes, a lot of tests become invalidated (for instance no longer compile, or throw runtime exceptions related to the API design). These tests fail not only because the redesign re-introduced bugs but also because they need to be updated to the new design. Occasionally, with bigger redesigns, old regression tests should be dropped.

Solution

In order to avoid a bug to be reintroduced, you should write a test that fails if the bug is reintroduced. Often this test is best to be written before the bug is fixed. Once you understood the bug, write a small unit tests that exhibits the problem (fails). Then see this failure go away, once the bug is fixed.

It is best to write this test at the lowest possible level, where the problem is exhibited. If the problem is local in a library function, it is beneficial to write the test at the API level of this library. If the problem involves communication on a ros-topic, it is probably best written at the ROS node level. The reason for this is two-fold. First, lower-level tests are more efficient, involving less ROS infrastructure, and thus more efficient to execute. Fast execution of tests is beneficial both offline (on your machine) and in continuous integration. Second, lower-level tests localize the problem better, so when they fail, it is easier for new developers to diagnose what is going on.

The ROS project provides several standard processes and solutions for regression testing. It also (at least formally) requires unit tests for code review. Regression tests are suitable unit tests for bug fixes.

Stakeholders

A **Submitter** is the programmer willing to submit a bug fix to a ROS repository of a package.

A **Maintainer** is the programmer taking care of the package.

A **Code Reviewer** is another ROS developer who together with the maintainer will be reviewing the code (very often, for simpler packages, or for less formalized projects this is the same person as the maintainer).

Tools involved

For testing Python code at library level (at the Python API level) ROS projects should use Python's *unittest* framework. See http://pythontesting.net/framework/unittest/unittest-introduction/. For testing C++ code at the library level (at the C++ API level) Google test framework *gtest* should be used. See https://github.com/google/googletest. For testing at the ROS node level, involving ROS as a communication middleware, *rostest* is used together with unittest or gtest. See http://wiki.ros.org/rostest. This applies both to single node tests, and tests that require integrating several nodes (technically known as integration tests, not unit tests).

It is key that the tests are not only automatic, but are integrated in the project scripts, so that they are run by the build and test infrastructure, whenever the project is being tested. To run the tests, you will need catkin/roslaunch integration (see the pattern *Integrate tests in Catkin* and http://wiki.ros.org/rostest/Writing). This may involve introducing a build dependency on rostest in package.xml and including a launcher for the test in the test file.

Test nodes should be introduced using the <test> tag into launch files. The rostest tool interprets these tags to start the nodes responsible for running node-level tests. (See http://wiki.ros.org/roslaunch/XML/test)

Regarding the submission please refer to the pattern *Submit a patch* where git infrastructure for submission is discussed.

Example

Fix race condition that lead to miss first message #1054. This particular example involves a Python unittest (without rostest).

Example resolved

- Submitter identified problem in a core package (ros_comm/actionlib)
- Wrote a patch and tested locally using the new regression test.
- In this particular case, another developer submitted a regression test exhibiting the problem
- Submitted Pull Request against indigo-devel (second-to-last LTS). Importantly the pull request included both the test and the submission, so the both the fixing patch and the test are used to explain the problem (in the example this is a race due to wrong locking).
- maintainer #1 identifies small issues with submitted patch, proposes fixes. In particular he requests that the regression test is incorporated into the patch.
- Maintainer #2 reviews the new PR by maintainer #1 and waits for the Continuous Integration server to complete the testrun.

- Eventually (on another branch, see https://github.com/ros/ros_comm/pull/1058) the fix for the problem is merged together with the test. The test remains active for future modifications of locks in this package.

Sometimes regression tests will involve large amounts of data (for instance ROS bags storing data from the failing execution). It is possible to make tests conditional, so that they are not called if this data is not available. This allows the other developers on your package to avoid running expensive tests, if they don't wish it.

Links

http://wiki.ros.org/UnitTesting

http://wiki.ros.org/unittest

http://wiki.ros.org/gtest

Consequences

There must be at least one maintainer per core package (but preferably more).

Maintainers need to have access to the necessary tools, both locally and remote (CI output of ROS Buildfarm).

It becomes possible to add more Q&A tooling to the buildfarm to more easily enforce Q&A process / best practices (make it less subjective).

Regression testing benefits community developers as well (not only maintainers, and not only core package developers).

Known Uses

Many open-source systems are using regression testing, and regression testing has been popularized a long time ago. A good example of an open-source project with regression tests is WebKit: https://webkit.org/regression-testing/, a web-browser engine used by many applications, including by Apple's Safari browser. Another project is KDE, a popular desktop manager for Unix-like systems: https://community.kde.org/Guidelines_and_HOWTOs/UnitTests. Both projects use CMake as their build manager, which is a similarity they share with ROS.

Related patterns

Continuous Integration Testing

Submit a patch.

Integrate tests in catkin

Pattern 5: Integrating tests in the build (catkin)

Name

Integrate tests in catkin

Context

A robot application is not only developed but the software is subject to change either as part of the development process or as part of later maintenance and evolution. Thus, crucial behavior of the software can get affected and in worst case fail or yield wrong outputs.

Problem

In many cases this error will not arise right away and at the time that a developer or an user comes across this bug it will be difficult to track down its source. Even if the developer defined certain tests for the software, it is likely that they are not executed on a regular basis.

Example

A team develops for example a library for path planning of a robot. One of the developer wants to clean up the code and decides to use matrixes calculations instead of simple equations. However, he mismatches two entries of a matrix, or forgets a minus at some position. Afterwards, he pushes his changes to the common repository of the library and moves on to his next task.

Forces

Strive for quality

In order to courage other parties to work with the provided software, it should meet preparations to sustain a flawless code.

Fail fast

Development becomes much more expensive if bugs are not detected at an early stage.

Solution

The whole application needs to be tested regularly to catch these regressions. Thus, one has to define explicit tests for the crucial behaviour of the program. Common methods for this purpose are unit tests and rostests (an extension to *roslaunch*, which enables testing across multiple nodes).

In order to run tests for a package globally they can be defined in the build system of ROS named catkin. Afterwards, the catkin tools offer to run all tests of the ROS workspace either at once or individually. This makes it more convenient to run tests on a regularly basis. Furthermore, they can be integrated into Continuous Integration to run the tests for example for every push to the Github repository.

Preparations

The howto documentation of catkin [1] describes the steps of integrating tests extensively. First of all, it recommends to configuration all steps related to testing conditionally. In this way larger data files, that might for example be required for Replay testing, do not need to be downloaded if the user does not intend to do testing.

The document further covers the integration of the following types of tests:

- Configuring gtest for C++
- Configuring Python nose tests
- Configuring rostest

Execution

One can either run all test at once with the command

```
$ catkin_make run_tests
```

or an individual test (here of type gtest) for the package example package by

```
$ catkin_make run_tests_examplepackage_gtest_exampletest
```

Some further information can be found in [1, Running unit tests].

Stakeholders

- The developers of the package
- The community members

Tools involved

- catkin
- gtest (cpp)
- nose (Python)

- rostest

Example resolved

In the mentioned scenario of the library for path planning, it is useful to have a test that verifies that the mechanics of the kinematic functions remain flawless. This can be achieved by defining a test that inputs an arbitrary (but reachable) cartesian goal position into the inverse kinematic to calculate the required robot’s joint states and subsequently, input this values to the forward kinematic to again get a pose in cartesian space. Finally, one can compare the original and the calculated pose to see if the functions are working correctly. The test will fail, if the difference is higher than a defined threshold, which is accounting numerical errors. If this test is include in catkin and the Continuous Integration system, it can be executed every time a developer pushes his patches to the library.

Links

[1] Configuring and running unit tests

[2] Conceptual overview of catkin

[3] Rostest

Consequences

There is an initial effort to develop a suitable set of test cases for the application and as best practice a Continuous Integration is required to run the test on a regular basis.

Related Patterns

- Continuous Integration with the public infrastructure
- Continuous Integration with private repositories
- Regression Testing (unit tests)
- Replay testing

Unit test and ROStest are used for regression tests of the ROS core in order to make sure that changes do not break existing functionality. They are part of everyday maintenance and the test suites are updated together with the code.

2.3.2 Reusable package, tool and driver

Whereas the bar for changes and contributions to the core modules is quite high, the community is very embracing when it comes to reusable packages, tools and drivers. Whereas the quality challenges for developing a reusable package (e.g. to steer calibration) or a driver (whether for a full robot or for specific hardware such as a camera) are quite different, they are treated similarly from the community side. The ROS and ROS-Industrial communities support the release of reusable packages by offering the possibility to provide the component through a ROS prebuilt infrastructure: registering the reusable component for release through the ROS buildfarm makes it easier for users to find and install them. Official ROS packages have an automatically created wiki page, which shows basic package information such as the link to the source code and that the CI is configured to run for this repository. (See also section 5.) In addition, one can increase the usability by filling the page with further documentation and tutorials. Releasing the package through the buildfarm requires that the developer commits to act as a maintainer of the package. The process of releasing a reusable module is described in more detail in Pattern 6.

Initial Quality Assurance

Before the release of a reusable package or driver, the package should be tested thoroughly. This can be done with manual testing or through simulation. Pattern 7 describes the ROS and ROS-Industrial state-of-the-art testing of reusable modules and drivers. What is important here is to formulate as much of the pre-release testing as possible using programming language specific test frameworks and tools.

For drivers for full industrial robots, model-in-the-loop testing using vendor provided simulators and hardware-in-the-loop testing are specifically recommended: Simulation-based tests provide the necessary security for application developers as failures will happen in simulation and not on real hardware. Such testing is further described in pattern 8.

Pattern 6: Release a reusable module

Name

Release a reusable module (e.g. a driver)

Context

As part of the development, a model of general interest has been developed, e.g. a driver for a specific camera, or a new path finder. The development team wants to make this module available for others in an easy manner. The community wants to have this software available, but also wants to assure that it is tested for the specified ROS distribution and optional provides documentation.

Problem

If a package is provided in the form of source code to interested end-users, the installation can reveal difficulties, since the user has to take care of installing required dependencies by himself. Then again, it takes quite some effort to create an own independent Debian package. Such a package could be used by interested community members, however, people would have a hard time to search for it and they might be insecure of the quality of the software. One last point is that there are different Debian-based distribution as well as different computer architectures and thus, one has to build several Debian packages.

Forces

Strive for quality

In order to encourage other parties to work with the provided software, it should meet preparations to ensure a proper compilation and installation.

Community reputation

It is a feather in the hat to have authored and maintain an “official” ROS package, which is hosted in a central place.

Community benefit

The ROS community is an open-source community. The idea is to benefit from each others’ development and to share results.

Solution

Releasing the reusable component to the ROS building repository makes it available to the public. Official ROS packages have an automatically created wiki page, which shows basic package information, the link to the source code and the status of continuous integration. In addition, one can increase the usability by filling the page with further documentation and tutorials.

Another very important benefit is that this process makes the software available as an pre-compiled Debian package for APT (Advanced Package Tool). This makes the managing of the package on Linux distributions very easy for an end-user, by providing functionalities as search, install, update and remove. The ROS build farm will automatically build binaries for different Debian-based distribution as well as different computer architectures. As an example the Ubuntu Xenial packages will be build for amd64, i386 and armhf.

Moreover, if any build-dependency of the software is updated, the build farm will automatically rebuild and check all of the package’s binaries.

In order to assure the quality of officially released packages, the process requires a certain procedure which is described in the following. Be aware that you have to release the package for every ROS distribution individually.

0) Pre-release testing

Best practice is to perform Pre-release testing to assure that dependencies are declared and installation commands are set-up in the correct way [3].

1) Create a release repository

One has to create a new Github repository, which should be named the same as the package but denoted as release. For now it stays empty for, but later the software's source code will be copied into this location via the bloom tool.

Within this repository one can optional grant additional developer writing rights and thus to release patches.

2) Configure the package's release track with bloom

The package can now be released to the ROS build farm with bloom. Once you run the build automation tool, it will guide you through all necessary steps. It can be installed by the APT package python-bloom and started by the following command.

Execution

```
$ bloom-release --rosdistro <ros_distro> --track <ros_distro> repository_name --edit
```

Explanation

- <ros_distro>: This is the name of the ROS distribution, e.g. kinetic.
- repository_name: This is the name of the new ROS package
- --edit: This is important for a first time release to create a new track

What is a track?

"bloom is designed to allow the release of the same package for different ROS distributions and versions in the same release repository. To facilitate this, bloom uses release "tracks" to maintain configurations for different release processes." [1]

Steps to perform with bloom

a) The tool will inform that the specified repository_name is not yet listed in the distribution file and ask for the link to the Github release repository.
b) Specify a desired repository name.
c) Insert the link to the package's development/upstream (not release) repository, which contains the source file of the software.
d) In the next step select the provided repository type. Available options are git, svn, hg or hosted tar.
- Version can be left as default
- Release Tag can be left as default

e) Specify the right branch of the development/upstream repository.
f) Specify the ROS Distribution
- Rest can be left as default

g) Provide your Github credentials

See [2] for a more detailed overview and [1] for even more details on bloom.

Subsequently, bloom will prepare your release repository and create a pull request for the distribution.yaml of the selected ROS distribution.

3) Wait for ROS build farm

"Once your pull request has been submitted then you will have to wait for one of the ROS developers to merge your request (this usually happens fairly quickly). Then after 24-48 hours your package should be built by the build farm and released into the building repository. Packages built are periodically synchronized over to the shadow-fixed and public repositories, so it might take some time (weeks) before your package has made it into the public ROS debian repositories." [1]

Stakeholder

- The author of the package to be released

- The community members

Tools involved

- Github
- Bloom
- Ros build farm

Links

[1] Bloom: First time release

[2] ROS: Releasing a package

[3] How to build and install target with catkin

Consequences

- At least one developer must commit to be the maintainer of the released package.
- A member or the ROS team must review the pull requests for the distribution. yaml and the ROS build farm must periodically build packages from this yaml lists.

Related Patterns

- Continuous Integration with the public infrastructure
- Pre-release testing

Pattern 7: Pre-Release Testing

Name

Pre-release Testing

Context

A developer wants to contribute a piece of code (a patch) to a ROS component that is tested using continuous integration (the build farm).

Problem

The build farm will install the component in a clean environment, on standard distribution components, attempt to build it and run the tests. This environment is typically quite different from the one that the developer used to code the component, that tends to include a lot of local customizations for convenience, legacy, for interaction with his other ROS projects, etc. For this reason, the installation, the build, or the tests on the build farm will often fail. To make it worse, it can take several hours before the developer will receive the information from the build farm. The information might come when he is already working on a different issue, and it requires an expensive context switch to fix the component and re-submit to the build farm (or another CI infrastructure). The problem is thus to shorten the modify-test-debug loop, but involving the setup that is as close as possible to the continuous integration conditions.

Another problem is being able to to run the tests, before the package release is made. A package release, that causes tests to fail on the build farm is very problematic, because usually many other dependent packages will be affected, and their maintainers might be notified. So failures of released packages in the build farm runs should be avoided.

Pre-release tests make sense to be run, even if there is no automated tests in the component. The setup will still check for compilation issues (errors find by compilers) and installation issues. These traditionally are a large fraction of issues in ROS components, so just building and installing the component on the build farm (and then also in a pre-release testing environment) can detect many problems.

Forces

Development Speed / Release Speed

Pre-release testing allows to speed up testing, fixing, and releasing components, as one does not have to wait for the external build farm to report on test results.

Complexity

Running pre-release tests is fairly complex, requires using docker components. However the process has been streamlined with external tools, and is still more efficient than relying on external CI servers.

Solution

The solution is to run pre-release tests, which are realized using the docker component technology and a set of scripts that mimic possibly closely the conditions of the external build farm, in a local docker environment. Typically they will be able to detect the same problems as the build farm.

Unlike the build farm tests, pre-release tests should be run manually. ROS streamlines running the pre-release testing process. You need to

1. Install the prerequisites (mostly docker, the build farm scripts, and catkin). See http://wiki.ros.org/regression_tests#How_do_I_setup_my_system_to_run_a_prerelease.3F
2. Use http://prerelease.ros.org/ to synthesize an invocation command for the pre-release testing of your component. This interface allows you to configure the set of components that should be available on the test system (you can also use non-standard release components, which is often needed in testing bleeding edge development).
3. The synthesized command will be several lines long. You just need to run it.

Stakeholders

A **Submitter** is the programmer releasing the component.

Tools involved

Docker (for creating a virtual test environment), along with build farm scripts, and catkin usually installed (http://wiki.ros.org/regression_tests#Prerelease_Tests). Of course, all other build and test dependencies will also be used.

The online tool http://prerelease.ros.org helps you by synthesizing the run command for the configuration for your build.

Links

http://wiki.ros.org/regression_tests#Prerelease_Tests

http://prerelease.ros.org

Consequences

Your component needs to be packaged as a proper ROS package (with all the meta-data).

A 64-bit machine is required to run docker.

You can avoid/reduce build farm errors after release.

Related patterns

Continuous Integration Testing

Integrate tests in catkin

Pattern 8: Model-in-the-loop (MIL) Testing with Specialized Robot Simulators

Name

Model-in-the-loop (MIL) Testing with Specialized Robot Simulators

Context

Developers wants to perform tests and verifications of how different kinds of components operate together with specialized robot simulators. The components could for example contain control algorithms or network communication. And the specialized robot simulators could for example come from ABB, Fanuc, Kuka etc.

Problem

How to test and debug robot applications and drivers before running the software on a real robot that could both result in damage of the robot and damage of the environment including potentially human casualty.

Forces

Rapid development

MIL testing allows for performing tests relatively quickly. It is not necessary to have access to real equipment, which might be limited in number and availability. This can then for example be used to verify that component still work as intended after new updates as well as testing current components against new simulator versions.

Quality

Depending on the test cases one should decide on the expected behaviors before performing the tests. E.g. what should happen if there is communication failure (if such parts are simulated), what should happened if only partial user input is used or is the simulated robot following the specified path or trajectory. This will allow the developers to detect possible issues and achieve components with higher quality.

Familiarization

MIL testing can help beginners to get used to working with robot software without danger, before potentially starting to work with real robots.

Solution

If you have access to a simulator of the targeted robot, you can use it to test the software on a simulation model of the robot (Model-in-the-Loop testing). To be able to make the best assessments of the results, you need to familiarise yourself with the usage of the robot simulator system that is being used for the tests.

Then the process for running the MIL tests could be:

1. Set up the different test cases.

The focus of MIL tests is the (mal-)functioning of the software rather than the hardware. Test cases should contain both success scenarios and anticipatable error scenarios. The reaction of the software to changes in the environment the robot is operating in might be a subject of tests as well.

Define suitable success criteria and how to measure them. What logs would you need in a malfunction situation to analyse the error?

Prepare (program) the simulation setup including what data to log. Besides the model of the robot the test scenarios can contain specific arrangements of the environment.

2. Run the tests.

Deploy the software to the simulated robot and run the test scenarios.

3. Evaluate the results.

If the simulator has a visual interface, a first observation would be whether it shows the expected behavior. However, logs and the measurements for the success criteria should be analysed as well.

4. Report/fix any detected issues.

The error report needs to contain the information about the setup of the test, the measurements of the success criteria, and the log files.

5. Rerun the tests if needed.

Related Patterns

Regression Testing

Consequences

By performing MIL testing continuously then it should be possible to achieve higher quality components.

You need a simulator. The simulator has to allow to define, save and run test scenarios including variation of the environment the robot is operating in.

The simulator used in the testing should have been validated before trusting the results. (If the simulator is from the robot manufacturer this should not be an issue.)

If the success criteria can be evaluated algorithmically, MIL test can be part of a regression test set up.

Limits: simulation is always ideal, does not always capture all the physical aspects (both of the robot and of the environment.

Issue Tracking and Maintenance

Each reusable component published through the ROS build repository ideally has a wiki page with a link to the GitHub issue tracker of the respective repository. This issue tracker allows users of the package to report bugs and propose package specific enhancements.

Users are also invited to contribute to published packages. The contribution should be formulated as a pull request referring to an issue in the issue list (a bug report or an enhancement request). For further detail refer to pattern 2 above.

For the role and tasks of maintainers in ROS and ROS-Industrial see above.

The work as a maintainer is further specified above in pattern 2.

Continuous Integration

The continuous integration frameworks provided by ROS and ROS-Industrial (pattern 3) also allow maintainers of individual packages to continuously integrate and test their packages upon changes and merging of pull requests. It would typically be the maintainer of a package who not only cares for the quality of the software, but also for the quality of the test suite that is assuring the quality of packages. This means that the test suite is expected to increase over time due to additional features being added and tested, pull requests and the definition of (regression) tests that proof a bug and its correction.

Unit test and ROStest

Continuous integration facilitates the use of unit tests and the ROStest framework, also for testing reusable components and drivers. Please refer to patterns 4 and 5 for further detail.

For driver development one of the core challenges is the correct interpretation of ROS messages by the hardware and the correct translation of hardware output into ROS messages. Another core challenge, depending on the hardware in question, can be safety concerns. For example if an unexpected event occurs, then a camera driver might be more forgiving compared to a robot driver. Especially if the robot driver is for motion control, since motions can potentially cause harm. In this case robustness is a high priority issue which can mitigated with tests covering several potential cases. Unit test and ROStest can also be applied to higher level functionality especially in cooperative development and when maintaining reusable software.

Best practices, Tutorials, Standards and Q&A

To support community developers, the ROS and ROS-Industrial communities have developed a number of best practices and tutorials. The most relevant for the development of reusable components are:

- http://wiki.ros.org/BestPractices lists a number of best practices mainly related to how to best make use of ROS and how to avoid common errors.
- http://wiki.ros.org/ROS/Tutorials provides a listing of the introductory and intermediate level tutorials available on ROS basics, some additional choice topics (such as navigation, data visualisation and coordinate systems) and links to external resources. The page seems to not be too well maintained though, with only sporadic edits over the past year.
- http://www.ros.org/reps/rep-0000.html is the entry point to the ROS Enhancement Proposal documents described earlier. Relevant ROS REPs include: 8, 9, 128, 132, 135 and 136.
- http://wiki.ros.org/ROS/Patterns

Pattern 9 further details the role and usage of best practices.

Pattern 9: Best Practices

Name

Best practices

Context

Several contexts can be envisioned here:

- An application developer wants to develop components that could be used in other applications. The reuse could be done by him, his team or eventually external collaborator.

- A team of Developers want to agree on programming policy to ease their collaborative work.

- A Developer desires to produce code of quality by applying agreed best practices.

Problem

How to develop a ROS code so that its reuse in other contexts is facilitated ?

How to make sure that other developers can easily embed such development in other applications, but also contribute to its maintenance and evolution along time in a fluent way?

Forces

Strive for quality

By taking into consideration defined programming policies, guide styles and programming patterns, developers are taking on board the experiences collected by the Community along time for producing high quality code, in terms of readability, re-usability or extensibility among others.

Note that the *community* terms refers here not only to the ROS Developers, but also to the community conducting research on software quality.

Increase the efficiency in developing new applications

Even though the conception of applications through a collection of nodes is already a step towards the design of reusable code, a particular care has to be given to the communication interface for maximizing the reuse potentiality.

In that line, a well-described node (including documentation), which interface focuses on the core functionalities provided, and designed to be less dependent to the current application it is being developed for, is more likely to be reused in other contexts.

Facilitate the collaboration and community contribution

By following and / or establishing coding and architecture policies, developers can reduce the subjectivity of their implementation, and ease the readability of their development.

Such readability is necessary for facilitating the improvement and extension of the code by other Developers, for the benefit of all.

Finally, and in line with the previous point, the quality of the node interface (in term of documentation and interaction means) is also crucial for maximizing its reuse by the community.

Solution

Look for best practices

Generally speaking, a key in developing code intended to be understood and reused by other Developers or Integrator is to reduce as much as possible the subjectivity in the implementation choices.

Nevertheless all Developers do not have the same background and experience in software programing, robot programing or ROS. And even if they would have, there would still be a need for agreeing common practices for handling any problem.

This is why it is a good practice (if not mandatory) to look for *best practices* during the development of any component or application.

Unfortunately, there is not a unique place hosting all the good practices, and it still required to dive into several sources to find the relevant information we are looking for.

The main sources of best practices are the following:

- ROS Wiki
- ROS Answers
- ROS Discourse
- ROS-users mailing list
- ROSCon programs
- Team or developer best practices.

The ROS Wiki is a tremendous source of information, including for finding best practices. Yet, the page ROS Best Practices provides useful pointers on best practices in ROS, as a set of *“statements of how best to achieve common tasks with ROS”*.

It mentions the existence of the ROS Enhancement Proposal(REP) which can be used to register best practices. It lists existing best practices, and also mentions the open points that would deserve a best practice description.

The wiki section on ROS USe Patterns and Best Practices also gathers best practices organized through abstract design patterns that are Conventions, Workspaces, Modularity, Communication, Parametrization, Logging and Robot Modeling.

The wiki section ROS developer´s guide is a good starting point for getting used to the common practices for developing components to be shared with the community.

Among other links, we can mention the pointers towards ROS Programming styles (for C++, python and javascript), and the description of the Quality Assurance Process to be followed by a package to be included into the ROS eco-system.

Considering that the people are likely to have already considered a question we have, it is relevant having a look at the ROS Answers section of ROS. In particular, the tag best_practices can be looked for for getting answers related to best practices. See here the list of related questions.

Note that ROS Discourse is intended to be the successor of ROS Answers, and progressively this engine will replace the former one.

Finally, bet practices can be searched as well through the information provided by individuals or Developer teams.

Several presentations done at ROSCON (2017, 2016 , 2015) provide good insights on best practices that could be used. Some developers groups give access to the best practices they gathered and follow, like:

- The Autonomous Systems Lab of ETH Zurich
- The Artificial Intelligence and Robotics Laboratory from Milano, which provides also a generic ROS node template that can help to define new nodes.

Consider Automation tools

Several high-level tools are existing for automating the creation of nodes and packages.

The advantage of such automation tools is that it reduces the code production by using hidden code templates or skeleton.

If the proposed templates fits the Developer needs, then the Developer can focus on the core added value of a component, and let the automation tool prepare the rest of the architecture.

ROSLab

ROSLab is a High-level Programming Language for Robotic Applications.

The code is maintained by the Precise Lab, but has not been updated since two years.

Examples of use are available at Nicola Bezzo´s website.

ROSLab is a high-level programming language based on blocks and links dragged on a java workspace which generates the skeleton code for robotic applications involving different types of robots.

Components can be connected though their communication interface.

Once generated, the Developer can implement the core code, having all the package and node skeleton automatically created.

The code generation is done using the ROSGen component, which is implemented in Coq.

ROSLab seem to be an interesting solution for roboticist Developers with limited knowledge in programming language and ROS, and want to quickly develop applications.

ROSMOD

ROSMOD is a work from the Vanderbilt University.

As it can be seen on the github account, the development is still active, and an online demo is available here.

A extensive documentation is also accessible on github.

ROSMOD is a Robot Operating System Model-driven development tool suite, providing graphical tools for rapid prototyping and deploying large-scale applications.

It follows a component-based approach structure, and is said to be a refinement of the ROS component model.

The ROSMODE tool-suite is intended to reduce the amount of time and effort they spend installing, configuring, and maintaining applications.

Nevertheless, it requires agreeing with the proposed model of component, that is slightly different from the traditional ROS one, which may be acknowledged only by advanced Developers.

Bride

Bride is on the main outcomes of the European project Brics.

Bride stands for *BRICS Integrated Development Environment*.

It allows for a definition of component interface and behavior using an abstract representation. enabling automatic model validation and code generation, where appropriate. It also provides a clear separation in between framework-specific (like the component interface) and the framework-independent code (like the core component computations).

Bride is integrated as an Eclipse plugin, in which the Developer can graphically design nodes together with its communication interface with the ROS world.

The development is following the spirit of Component-Based Software Engineering, targeting quality, technical and functional reusability (see that paper).

Considering that a software component is defined to be a *unit of composition with contractually specified interfaces and explicit context dependencies only*, brics stresses in its implementation the clear distinction in between the interface and the implementation of the component functionalities.

From the definition of the interface, through the Eclipse plugin, or directly through a xml description, brics prepares the ROS node structure, defines the communication tools, and prepares in a distinct file the skeleton of the code to be filled by the user.

The concepts followed by brics seem to be of major importance for developing stable components with clear interfaces.

Unfortunately, the developments have been stopped two years ago and are sticked to ROS indigo.

Furthermore, the life pattern implemented in the generated ROS core module can not be adapted to Developer needs, and would require good skills in java and eclipse plugin for being adapted to specific needs.

Rosnode 2.0 node life-cycle

In addition to the relevance of defining well the interface a node provides to the user, it is also crucial making clear what is the life-cycle of the node once launched.

Even though part of it can be deduced from the interface definition, critical aspects may not be easily inferred from such description, such as when the node computation starts, can we and should we stop and resume the node activity during the application, …

A particular care is placed on such aspect in the design of ROS2, essentially in the concept of managed nodes.

The inheritance mechanism is used to associate to nodes a common life-cycle, with pre-defined interaction mechanisms. This way the monitoring of the deployment, initialization, pausing and resuming of any node launched is made easier since all managed nodes follow the same policy.

Managed nodes execute following a known state machine which state indicates whether the node is *unconfigured* (just instanciated), *inactive* (configured but not running), *active* (performing its computation) or *finalized* (before destruction).

The implementation of a managed nodes requires implementing the different transitions from one state to another.

The advantage of such model is that, if well-spread in all nodes, the monitoring of an application is eased since all nodes follow a common methodology and can thus be triggered or consulted on their status in a common way.

The use of a component is thus less dependent on the implementation proposed by the Developer.

This definitely ease the collaboration in the community as well as the reuse of components in other applications.

Even though ROS2.0 is not yet deployed, such mechanism and philosophy is a good practice to consider when developing nodes, even in the current ROS structure.

Related patterns

- Submit a patch
- Regression testing (unit tests)
- Code review

- Accepting a pull request
- Standards and patterns

In order to be in line with the ROS and ROS-Industrial core packages, reusable components and drivers should use the ROS and ROS-Industrial standards for metrics, measurements, naming and the like. The most relevant ones for the development of drivers and reusable packages are:

- http://wiki.ros.org/ROS/Patterns/Conventions presents naming conventions for packages but also when using ROS, e.g. how to name communication channels when developing an application.
- Relevant ROS REPs: 3, 103, 104, 105, 107, 117, 118, 120, 122, 127, 138, 140, 144 and 147.
- http://wiki.ros.org/Industrial/Tutorials/SuggestedPackageLayoutNewRepositories
- http://wiki.ros.org/Industrial/Tutorials/WorkingWithRosIndustrialRobotSupportPackages#Naming

Other questions can be posed either to the maintainers of a specific package via the issue tracker or to the ROS Q&A. If these channels are used the developer is advised to first seek relevant issue lists and the Q&A. The Q&A can be found under https://answers.ros.org. To pose a question requires to sign up as a user.

2.3.3 Support for QA for Application Development

For users of ROS and ROS-Industrial, the main QA is to determine whether the robot running the application is doing what it is supposed to do. Many of the methods, techniques and tools are the same as to determine the correctness of the ros core and the reusable packages. They are though used slightly different in the application development. Below the main techniques and tools are highlighted.

Continuous Integration and Industrial CI

The continuous integration environments are of course also a help for developers. Already the information, whether an application under development can be built supports incremental development and decreases the effort to build and test everything in the end. ROS-Industrial has developed the *industrial_ci* package, which is a set of scripts and configurations that lowers the threshold for developers of ROS-powered packages to integrate these into a Continuous Integration pipeline. The standard use of industrial_ci however, is defined by hosting the ROS packages in GitHub, and using Travis CI to run the CI tests. Pattern 10 details the setup and usage of this pipeline relying on other tools that can be set-up privately. This is specially interesting for companies developing code or applications based on ROS, that cannot distribute publicly their source code.

Pattern 10: Continuous Integration with private repositories

Name

Continuous Integration with private repositories

Context

An application developer wants to use a continuous integration service as part of his/her development process in order to parallel to the development check that the application can be integrated and deployed and allow for regression tests. The CI shall be run in a separate server, and run automatically builds and tests defined in the code.

Problem

There are several options available to use CI, as Travis or the ROS buildfarm. However Travis does not support privately hosted repositories (only GitHub at the moment), and setting up and running a private installation of buildfarm is overcomplicated.

Forces

Producing quality code

Developing quality code is a process that can be facilitated by the use of the appropriate tools. CI is one of such tools, and a cornerstone in quality assurance procedures, providing automated ways of ensuring reproducibility, regressions tests, deployability and so on.

Make ROS business friendly

The use of ROS and ROS-I should be extended beyond the academic community, into industrial applications and industrial actors using and contributing to it. These actors usually have stricter IP sharing rules, and by giving them the proper tools such as CI, we facilitate their involvement into the ROS community.

Facilitate the collaboration and contribution to the community

Code contributed to the ROS repository is expected to comply with certain rules regarding its quality. By the use of CI as part of the development process, compliance with such requirements is facilitated, which fosters further contributions.

Solution

We provide instructions here on how to set up a system/process based on the use of Jenkins + industrial_ci + GitLab. The rationale to choose such a combination, instead of other available alternatives such as buildfarm, Travis and others is:

- support for private repositories and in-house installation
- based on open-source tools
- easy installation and setup time
- low threshold for usage, easy integration of new components into the process
- extensible
- GitLab is a well-known repository manager; however the instructions can be easily adapted and applied to other infrastructures

Components of the solution

Jenkins

Jenkins is an open-source automation server written in Java, that runs in servlet containers such as Apache Tomcat. It is mainly targeting facilitating the automation of continuous integration aspects: it offers integration with many version control tools, such as CVS, Subversion, Git, or Mercurial (can be extended to others through the use of plugins), and offers different ways of triggering builds, such as commits in the control version system or scheduling with a cron-like mechanism. It also provides web-based reporting capabilities for the results of the builds (interface with the user is mainly web-based).

Jenkins supports scalability through a "master/slave" mode, where the workload of building projects are delegated to multiple "slave" nodes, allowing a single Jenkins installation to host a large number of projects, or to provide different environments needed for builds/tests.

Jenkins' functionality can also be extended by the addition of plugins.

industrial_ci

industrial_ci is a set of bash scripts that can be used to check that a ROS package builds and installs without issues.

If unit or system tests are defined, it can also run them. In order to ensure reproducibility, the builds are run in empty Docker containers, in which the dependencies specified are installed.

Stakeholders

Installer / SysAdmin

Link to *Jenkins and industrial_ci: Quick manual*; Section *For installer / administration*

ROS application developer

Link to *Jenkins and industrial_ci: Quick manual*; Section *For developers*

Links

- industrial_ci: ROS wiki; GitHub repository
- industrial_ci documentation
- Jenkins
- Jenkins plugins
- rosdep
- REP 126
- rosinstall file format
- The ROS build farm - what it can do for me pdf; video
- buildfarm web / ros_buildfarm2
- ROS and CI

Related patterns

- Integrating tests in the build
- MIL testing
- Best Practices
- Regression Testing
- Pre-release testing

Unit test and ROStest

Continuous integration opens up to use unit tests and the ROStest framework also for the testing of reusable components and drivers. Please, refer to the patterns 4 and 5 for further detail.

An additional tool to define test cases is the possibility to record message streams of a real or simulated ROS applications. That way, e.g. a recorded data of a camera can be used to define a test case for image recognition. Pattern 11 further elaborates this possibility.

Pattern 11: Replay testing

Name

Replay testing

Context

As for every software, it is required for applications that process data to have a continuous evaluation of patches, which are introduced by members of the developer team. Moreover, development is more efficient, if the application is constantly fed with test data, resembling the intended environment as close as possible.

Problem

An application that processes data can only be tested sufficiently if it actually processes data. However, resource requirements are too high to repeat an event multiple times, because a real application requires designated hardware and a certain environment; a simulation requires high processing power and application dependencies.

Example

An algorithm that processes laser scans for line detection, which is used as groundwork for localization, is developed. In order to test the code the team needs scanning data and thus, they have to reserve a scanning device, a mobile robot and require an appropriate experimental area.

Forces

Strive for quality

In order to encourage other parties to work with the provided software, it should meet preparations to sustain a flawless code.

Fail fast

Development becomes much more expensive if bugs are not detected at an early stage.

Save resources

Regression test using real hardware or simulation is too expensive.

Solution

The ROS-activities of either a real or a simulated event can be recorded to be later replayed. In this way, one can efficiently develop and test applications that have to process data. For this purpose, ROS provides a tool named rosbag [1]. Most common is to work with the tool from command-line, which offers functionalities like for example record, play and info and are described comprehensively in tutorial [1a] and overview [1b]. Moreover, the plugin rqt_bag [2] offers the option to use most of the commands from a GUI.
It is worth to highlight the option to record only selected topics and the one to filter already existing bag-files. The later one is very powerful and one can for example also remove only a certain transformation between two frames from the /tf topic [3].
The bag file can be finally used for testing by referencing to it within a rostest-launch file. Furthermore, there is the possibility to conditionally for testing download bag-files from the web. This is useful if they are too large and thus, would overload the source repository [4].

Procedure

1. Define test scenario
2. Define required data
3. Record the data either from real hardware or simulation
4. Implement the test with the recorded data
5. Include test to Continuous Integration

Stakeholders

- The developers of the package
- The community members

Tools involved

- rosbag
- rqt_bag

Example resolved

In order to save resources, the team runs the experiment of scanning an environment once and record the data. The laser is mounted on a mobile robot, which has a very precise localization. The team also record the transform

(tf) of the robot's pose relative to the starting position, since this information is valuable to evaluate the performance of their new scanning-based localization.
For visualizing the robot, the team decides to publish a tf from the result of their novel algorithm, which estimates the robot's pose relative to the starting origin. However, this transformation will conflict with the recorded tf, because they use the same frame names. In order to solve this issue, they filter only the conflicting tf of the recorded data and save it as a new file. Apart from that they keep all other transformations, since they still require the relative pose of the laser to the mobile robot.
Furthermore, the team establish a rostest for Continuous Integration that makes use of the recorded data. It will run the novel algorithm on a particular subset of the data and finally, check if the estimated pose is within a certain threshold of an ideal value.

Links

[1] rosbag
[1a] rosbag tutorial
[1b] Command-line functionalities
[2] rqt_bag
[3] Remove a certain tf frame
[4] Download test data

Consequences

The developer team has to define certain scenarios to test their application and afterwards, perform them either in simulation or in real world to acquire required data.
The recorded files will increase the source repository if the mentioned corrective action is not taken.
Replay testing as such does not offer the possibility of a feedback loop. Influencing the test scenario, like e.g. making decisions on navigation, will make recorded data of the environment like e.g. laser scans unusable.

Related Patterns

- Continuous Integration with the public infrastructure
- Continuous Integration with private repositories
- Regression Testing (unit tests)
- Integrate tests in catkin

Simulation

A powerful tool for developing a robot application is to simulate the hardware and let the application interact with the simulation. Pattern 8 describes Model-in-the-Loop testing with a vendor specific simulator. The advantage here is that the vendor guarantees that the simulator for all practical purposes behaves like the real robot. Pattern 12 describes how to use vendor specific simulators for application testing.

Pattern 12: Model-in-the-Loop Testing with Specialized Simulator (Applications)

Name

Model-in-the-loop (MIL) Testing with Specialized Robot Simulators

Context

Developers wants to perform tests and verifications of how different kinds of components operate together with specialized robot simulators. The components could for example contain control algorithms or network communication. And the specialized robot simulators could for example come from ABB, Fanuc, Kuka etc.

Problem

How to test and debug robot applications and drivers before running the software on a real robot that could both result in damage of the robot and damage of the environment including potentially human casualty.

Forces

Rapid development

MIL testing allows for performing tests relatively quickly. It is not necessary to have access to real equipment, which might be limited in number and availability. This can then for example be used to verify that component still work as intended after new updates as well as testing current components against new simulator versions.

Quality

Depending on the test cases one should decide on the expected behaviors before performing the tests. E.g. what should happen if there is communication failure (if such parts are simulated), what should happened if only partial user input is used or is the simulated robot following the specified path or trajectory. This will allow the developers to detect possible issues and achieve components with higher quality.

Familiarization

MIL testing can help beginners to get used to working with robot software without danger, before potentially starting to work with real robots.

Solution

If you have access to a simulator of the targeted robot, you can use it to test the software on a simulation model of the robot (Model-in-the-Loop testing). To be able to make the best assessments of the results, you need to familiarise yourself with the usage of the robot simulator system that is being used for the tests.

Then the process for running the MIL tests could be:

1. Set up the different test cases.

The focus of MIL tests is the (mal-)functioning of the software rather than the hardware. Test cases should contain both success scenarios and anticipatable error scenarios. The reaction of the software to changes in the environment the robot is operating in might be a subject of tests as well.

Define suitable success criteria and how to measure them. What logs would you need in a malfunction situation to analyse the error?

Prepare (program) the simulation setup including what data to log. Besides the model of the robot the test scenarios can contain specific arrangements of the environment.

2. Run the tests.

Deploy the software to the simulated robot and run the test scenarios.

3. Evaluate the results.

If the simulator has a visual interface, a first observation would be whether it shows the expected behavior. However, logs and the measurements for the success criteria should be analysed as well.

4. Report/fix any detected issues.

The error report needs to contain the information about the setup of the test, the measurements of the success criteria, and the log files.

5. Rerun the tests if needed.

Links

regression tests

Consequences

By performing MIL testing continuously then it should be possible to achieve higher quality components.

You need a simulator. The simulator has to allow to define, save and run test scenarios including variation of the environment the robot is operating in.

The simulator used in the testing should have been validated before trusting the results. (If the simulator is from the robot manufacturer this should not be an issue.)

If the success criteria can be evaluated algorithmically, MIL test can be part of a regression test set up.

Limits: simulation is always ideal, does not always capture all the physical aspects (both of the robot and of the environment.

In order to be able to also simulate robots that do not come with a bespoke simulator, Gazebo has been developed: Gazebo allows to model a robot and then interface to it from a ROS application. Pattern 13 describes how to use Gazebo for Model-in-the-loop testing.

Pattern 13: Model-in-the-Loop Testing with Gazebo

Name

MIL Testing with Gazebo (not for drivers)

Context

When developing robotic applications, performing experiments in the real hardware (the robot) is expensive: usually it is a resource to be shared, running new code in a safe manner is challenging, setting the experimental setup requires quite some work, or it is difficult to debug. It is a good practice to use a model-in-the-loop approach to test new algorithms and tune parameters in a simulated robot and environment before testing on real hardware.

Problem

MIL testing can be adopted as part of the development process, but doing the tests and preparing the environment should not take much effort in order to be effective. The similarity between the results of a simulation and the outcome of the execution in real hardware is also a prerequisite.

Forces

Accelerate development

The preparation of a proper real testing environment and running experiments in a real robot is quite time consuming. Artifacts related to real hardware also makes experiments difficult to reproduce and to debug. By the use of MIL testing the process can be accelerated.

Safety

New algorithms or modified parameters can pose safety issues for the integrity of the robot and the operators.

Quality

By facilitating the execution of experiments and the reproducibility, more exhaustive tests can be performed which result in more reliable and robust results.

Solution

The solution presented here is not an automatic process (which will be explored in a future pattern); it is not either suitable for testing new drivers, as in these cases the exact behavior and dynamics of the sensors or actuators tested can not be reliably reproduced.

The solution presented here involves using a simulator, and we rely on Gazebo due to the excellent integration with ROS; a key element for the MIL process to be useful is that switching between the real hardware and the model does not require changes in the algorithmic aspects which are tested. Ideally the switching could be done just by rerouting the commands and sensor information flow.

This is the process to run MIL tests using the Gazebo simulator:

- Setup the simulation environment
- The Gazebo Tutorials explain how to create a simulated environment where the tests will run.
- An important step consists on generating the SDF files on which Gazebo relies instead of the URDF descriptions used in ROS. While URDF is the standardized way of representing a robot model in ROS, it can only specify the kinematic and dynamic properties of a single robot; the SDF format can hold additional information which is required to run the simulation.
- Instead of create a robot model from scratch, Gazebo already provides a Gazebo Model Database of commercial robots available to be used.
- Sensors can also be simulated in Gazebo, including their noise characteristics
- In order for the ROS nodes/infrastructure to be able to communicate with the Gazebo simulator, the specific ROS plug-in has to be used.
- Once the simulation is up and running and communicating with the ROS nodes, write tests that send goals, e.g. in case of a path planner, a client that produces goals poses and sends them to the appropriate server. A challenging aspects is that these tests should aim at covering all possible cases.
- Run the tests and use the Gazebo GUI to check for the performance, e.g. no collisions or unwanted behaviors.
- Use RViz to debug the robot system.

Additionally:

- In some cases, tests can be automated, e.g. in the path planner example, a client can be automatically checking for collisions, and after a certain timeout after sending the goal, check that the right pose is achieved.
- Gazebo is split into a server (where the simulation is run) and a client (the GUI). In case of automated tests, only the server part can be used to use resources in an appropriate manner.
- There are some tools that allow this process to be run in a fully automated way and as part of a continuous process. For example, the Automated Test Framework ATF is a testing framework written for ROS which supports executing integration and system tests, running benchmarks and monitor the code behaviour over time. The ATF provides basic building blocks for easy integration of the tests into your application. Furthermore the ATF provides everything to automate the execution and analysis of tests as well as a graphical web-based frontend to visualize the results.

Links

- Gazebo Tutorials
- Gazebo Model Database
- Adding sensor noise in Gazebo
- ROS pluging
- ATF
- Paper: Simulation Environment for Mobile Robots Testing Using ROS and Gazebo

Related patterns

- Integrating tests in the build (catkin)
- MIL testing using robot simulate
- Replay testing without feedback loop (ros bags)
- Continuous Integration
- Best practices

- MIL Testing with specialized simulator
- Hardware in the loop testing with CI

Best practices, Tutorials, Standards and Q&A

As already described above, the ROS and the ROS-Industrial communities have developed a number of best practices and tutorials. The most relevant for the development of robot applications are:

- http://wiki.ros.org/BestPractices and lists a number of best practices on how to how to best make use of ROS and avoid errors.
- http://wiki.ros.org/ROS/Tutorials provide the most comprehensive repository of making use of ROS and contributing to it on the ROS wiki.
- http://www.ros.org/reps/rep-0000.html is the entry point to the ROS Enhancement Proposal documents described earlier. Relevant ROS REPs include: 8, 9, 128, 132, 135 and 136.
- http://wiki.ros.org/ROS/Patterns

In order to be in line with the ROS and ROS-Industrial cores, reusable components and drivers should use the ROS and ROS-Industrial standards for metrics, measurements, naming and the like. The most relevant ones for the development of robot applications are:

- http://wiki.ros.org/ROS/Patterns/Conventions presents naming conventions for packages but also when using ROS, e.g. how to name communication channels when developing an application.
- Relevant ROS REPs: 3, 103, 104, 105, 107, 117, 118, 120, 122, 127, 138, 140, 144 and 147.
- http://wiki.ros.org/Industrial/Tutorials/SuggestedPackageLayoutNewRepositories
- http://wiki.ros.org/Industrial/Tutorials/WorkingWithRosIndustrialRobotSupportPackages#Naming

Other questions can be posed either to the maintainers of a specific package via the issue tracker or to the ROS Q&A. If these channels are used the developer is advised to first seek relevant issue lists and the Q&A. The Q&A can be found under https://answers.ros.org. To pose a question requires to sign up as a user.

Table 2.1: Type of Development, QA measures and Patterns

Type of Development	**QA measures**	**Patterns**
Core development	REPs: Ros Enhancement Proposals	
	Issue tracking and maintenance	Pattern 1: Submit a patch Pattern 2: Accept a patch
	Continuous Integration	Pattern 3: Continuous integration with the public infrastructure
	Unit test and ROStest	Pattern 4: Regression test (Unit test) Pattern 5: Integrating tests in the build (catkin)

Reusable package, tool and driver	Initial Quality Assurance	Pattern 6: Release a reusable module Pattern 7: Pre-Release Testing Pattern 8: Model-in-the-loop (MIL) Testing with Specialized Robot Simulators (drivers)
	Issue tracking and maintenance	Pattern 1: Submit a patch Pattern 2: Accept a patch
	Continuous Integration	Pattern 3: Continuous integration with the public infrastructure
	Unit Test and ROStest	Pattern 4: Regression test (Unit test) Pattern 5: Integrating tests in the build (catkin)
	Best practices, Tutorials, Standards and Q&A	Pattern 9: Best Practices
Application development	Continuous Integration	Pattern 3: Continuous integration with the public infrastructure Pattern 10: Continuous Integration with private repositories
	Unit Test and ROStest	Pattern 4: Regression test (Unit test) Pattern 5: Integrating tests in the build (catkin) Pattern 11: Replay testing
	Simulation	Pattern 12: Model-in-the-Loop Testing with Specialized Simulator (Applications) Pattern 13: Model-in-the-Loop Testing with Gazebo
	Best practices, Tutorials, Standards and Q&A	Pattern 9: Best Practices

2.4 Sub-Conclusion: Issues with the QA practices

This chapter has presented insights into quality aspects of ROS and ROS-Industrial communities gained through analysis of the ROS resources for the community. These insights can be used as the starting point for further investigations regarding quality in the ROS community, along with the implementation of quality improvement strategies. The analysis indicates that the community retains some of software engineering and industry practices and processes. However, there are challenges in the implementation and execution of these practices and processes as the next section will further detail.

Software engineering quality models suggest two areas of practices, quality assurance and quality control. Quality assurance's center of attention is processes and procedures. Quality control is the validation and verification of the product through a well-defined testing process and tools.

- *Software Quality Assurance* is a set of activities for ensuring quality in software engineering processes (that ultimately should result in quality software products). The activities establish and evaluate the processes that produce products (Dobbins and Buck 1983).
- *Software Quality Control* is a set of activities for ensuring quality in software products. The

activities focus on identifying and resolving defects in the actual products produced (Dobbins and Buck 1983).

Although some practices and processes have been adopted, we found that many of the ROS quality assurance activities are still evolving and experiencing challenges in the implementation and execution. The tables below summarize these challenges for each QA practice and process segmented by ROS specific development roles (i.e. Core, Reusable packages, etc.):

2.4.1 Core Development

Table 2.2: QA practices adopted by the community and the corresponding challenges, core development.

Software engineering practices	**Area of practice**	**Community form of adoption**	**Challenges**	**Discussion and recommendations**
Well-defined development process	Quality assurance	ROS Enhancement Proposals (REPs)	Community members do not feel obliged to contribute into REPs reviews. This challenge is of cultural characteristics to the community social particularities.	This behavior and the underlying social construct need to be understood further and corrective measures should be recommended and implemented in collaboration with the community.
Defects management process and tool	Quality assurance	Issue tracking and maintenance	The adoption and implementation of this best practice seems to lack procedural rigor. 1. New enhancements and defects fixing seem to be regarded equal. There seems to be no urgency to address defects as commonly practiced in commercial environments. 2. Prioritization of defects vis-à-vis new enhancements does not seem to take place in the process. For example, if a defect is found in a run-time environment (i.e. operational robot), should it take priority over new enhancements? 3. No dedicated resources for testing, as per the industry practices. The community relies on users and contributors to report defects.	The current defects management process should be documented and analyzed. In collaboration with the community, area of improvements should be identified and implemented.

Code review	Quality assurance	Code peer review	There is a code standard. However, code standards are not consistent across the community development roles (i.e. Core, Reusable packages). 1. There seem to be some ambiguity around code standards. The community is unclear about what are the currently practiced standards. 2. Not enforced and universally accepted.	1. Define and standardize a universal code standards for the community. 2. Review, update and promote current code standards. 3. Work with the community to identify, introduce and promote additional standards.
Continuous Integration	Quality control	Continuous integration with the public infrastructure	1. The current Continuous Integration implementation has limited scope and coverage. Continuous Integration is currently used to make sure that there are no unresolved dependencies. The idea is to broaden the usage and include functional tests and regression test. 2. Limited coverage of the current automated testing suite.	Software engineering practices recommends early defects detection. The current continuous integration process should be documented and analyzed. In collaboration with the community, area of improvements should be identified and implemented.
Unit testing	Quality control	Unit test and Rostest	1. Test coverage is inadequate. There have been instances where use cases have challenged the code. 2. Lack of infrastructure to support end to end testing.	Increase the test coverage and provide an end to end infrastructure for testing. However, this requires additional resources. The ROS community capacity is already strained. The body of literature seems to suggest that open-source software evolution and community growth are codependent.
Knowledge sharing	Other	Best practices, Tutorials, and Q&A	1. Outdated documentation (i.e. Standards, guidelines, and process documentation) 2. There no onboarding and knowledge transfer process in place. 3. The current content is informal there is no governance process for knowledge management.	1. Update and extend the current documented knowledge. 2. Propose a knowledge sharing and collaboration strategy for the community. This should include a process to capture, maintain and extend knowledge and a platform to host knowledge sharing and collaboration.

Hardware-in-the-loop testing (HILT)	Quality control	Hardware-in-the-loop testing	Informal implementation of the process.	Work with the core development community to identify their requirements for HILT.

2.4.2 Reusable packages, tool and driver

Table 2.3: QA practices adopted by the community and the corresponding challenges, reusable packages, tool and driver.

Software engineering practices	**Area of practice**	**Community form of adoption**	**Challenges**	**Discussion and recommendations**
Defects management process and tool	Quality assurance	Issue tracking and maintenance	The adoption of this practice in the reusable packages and drivers' community is informal. The process is not an integral part of the community practice.	A universal defect management process for the ROS community should be defined and implemented.
Continuous Integration (CI)	Quality control	Continuous integration with the public infrastructure	Inconsistent use of CI across the various development life cycle segments of ROS.	Formalize and standardize the CI process across the various ROS development life cycle.
Unit testing	Quality control	Unit test and Rostest	Informal process. The infrastructure and tools are available but not enforced. It's up to the developer to decide whether to use them.	Define and formalize the process of unit testing across the various development life cycle of ROS.
Knowledge sharing	Other	Best practices, Tutorials, and Q&A	1. Outdated documentation (i.e. Standards, guidelines, and process documentation) 2. There no onboarding and knowledge transfer process in place.	1. Update and extend the current documented knowledge. 2. Propose a knowledge sharing and collaboration strategy for the community. This should include a process to capture, maintain and extend knowledge and a platform to host knowledge sharing and collaboration.
Model-in-the-loop Testing	Quality control	Gazebo, Rviz and hardware simulators by the robot manufacturers	Complexity of set-up, especially as part of the regression test, can be a challenge.	Widely accepted practice. Promote the practice by better communicating set up and test strategies
Hardware-in-the-loop testing	Quality control	Hardware-in-the-loop testing	Complexity of set up: requires dedicated hardware in a protected environment.	Widely accepted practice. Promote the practice by better communicating test strategies

2.4.3 Application development

Table 2.4: QA practices adopted by the community and the corresponding challenges, application development.

Software engineering practices	**Area of practice**	**Community form of adoption**	**Challenges**	**Discussion and recommendations**
Continuous Integration	Quality control	Continuous integration with the public infrastructure	Inconsistent use of CI across the various development life cycle segments of ROS.	Formalize and standardize the CI process across the various ROS development life cycle.
Unit testing	Quality control	Unit test and Rostest	The infrastructure is available, but, it is left to the company building the robot discretion to decide whether to use it or not.	
Knowledge sharing	Other	Best practices, Tutorials, and Q&A	1. Outdated documentation (i.e. Standards, guidelines, and process documentation) 2. There no onboarding and knowledge transfer process in place.	1. Update and extend the current documented knowledge. 2. Propose a knowledge sharing and collaboration strategy for the community. This should include a process to capture, maintain and extend knowledge and a platform to host knowledge sharing and collaboration.
Model-in-the-loop Testing	Quality control	Gazebo, Rviz and hardware simulators by the robot manufacturers.	Complexity of set-up, especially as part of the regression test, can be a challenge.	Widely accepted practice. Promote the practice by better communicating set up and test strategies

The next chapter further explores how the above discussed QA and QC practices are implemented and what community members perceive as challenges and remedies of those.

3. ROS-Industrial community

3.1 Introduction to the chapter

The previous chapter presents and discusses the quality assurance and quality control processes, methods, processes and tools that the community applies respectively provides for its members. But how are these means applied? Are they resulting in a high quality software product and applications? What are the quality challenges? And what to community members regard as remedies? In order to address these questions, we have performed four in-depth interviews with members of the community experienced with the different modes of development. These interviews are analysed in the following sections.

3.2 Methods

The purpose of the interviews is to systematically map out what is perceived as quality challenges for different kinds of development. To this end we interviewed four community members contributing to and using ROS in different ways.

The interviews and their analysis followed guidelines for qualitative empirical research (Robson, 2011). The interviews were prepared by an interview guideline that has been based on an initial analysis of the ROS web site and through discussions with ROS and ROS-Industrial consortium members. The interviews have been transcribed and analysed using qualitative analysis techniques: in a first round, themes - also called codes - were identified. These themes were then used to code the interviews. During the coding, the coding scheme, that is the list of themes used for coding, has been extended. Below the analysis is presented in sections 3.3 to 3.6. Section 3.7 sums up the findings.

In order to establish trustworthiness in the results, we triangulated the three interviews. The analysis was done by two researchers (researcher triangulation). Further we asked the interviewees to check and comment on the analysis (member checking). The bug analysis and the analysis of the related ROS community web sites and tools provide the possibility for further triangulation.

3.2.1 The Interviewees

The Interviewees present a convenience sample: Mainly other members of the ROSIN project were interviewed who either used ROS or contributed to ROS in different roles. Further interviewees were recruited through this network.

We interviewed two core developers and maintainers of core modules: one responsible for the ROS core, the other acting as a technical lead for the ROS-Industrial community and maintainer of central modules there. The other interviewees are one driver developer who is about to publish a driver for an industrial robot and an experienced robot application developer. We here shortly present their respective background in order to allow the reader to appreciate their expertise. We are aware that for readers who also are members of the ROS community, this might challenge their anonymity. The interviewees have agreed to the presentation below.

Core Developer A: One of the interviewees is a core developer and maintainer of the majority of the ROS core modules. He developed an interest in robotics from a RoboCup project while at the university. His PhD was about efficient middleware for multiple cooperating autonomous agents. He now works at the Open Source Robotics Foundation (OSRF). Besides being a maintainer for the ROS core he has developed the build farm and works as well on Gazebo, a physics and dynamics simulator for ROS, application development and is one of the developers of ROS 2.

Core Developer B: The second core developer and maintainer is a technical lead in the ROS-Industrial consortium. His master thesis was about component based systems. ROS being a component based system, he also studied ROS. After his master thesis, he developed robot applications at a university spin-off and later developed a ROS driver for an industrial robot. This led to him to joining ROS-Industrial. He is now involved in technical design and decisions for ROS-Industrial. He maintains a few packages, like a motion planning framework, in addition to a number of his own packages.

Driver Developer: The interviewee works with a major developer of industrial robots and has developed a ROS driver for several of the company's robots. He holds a M.Sc. degree in Electrical Engineering from Linkoping University in Sweden. He learnt about ROS from colleagues. As a beginner, he used mainly tutorials to grasp ROS but he had already knowledge about programming from his university studies.

Application Developer: The interviewed application developer holds a M. Sc. degree in computer graphics from Spain where he also specialised in computer graphics. He worked with robots in his PhD studies. This gave him a transition to the robotics sphere. At that time, ROS was not in existence and there was no well-structured way to develop various components for robot applications. The team started to use Player, which can be seen as a predecessor of ROS. Therefore, the move to work in ROS, another Robot middleware, came naturally. He now has ten years work experience in ROS in several application development projects.

3.3 The status of ROS and ROS-I

This section focusses on the aspects of ROS and ROS-I community: How do the community members and developers evaluate the advantages and the quality and status of the open-source software is presented in subsections 3.3.1 and 3.3.2. 3.3.3 and 3.3.4 present the activities of the ROS-Industrial community and the motivation for the development of ROS2 respectively.

3.3.1 What makes ROS and ROS-I attractive

For the core developer, one clear advantage of ROS over a proprietary Robot Operating System or one written by the individual research community is that Open Source ensures exposure and usage, and that in turn ensures survival and longevity of the software. In academia, there is otherwise a low chance for software to survive outside the group who developed it.

As another advantage, he mentions the set of tools provided for ROS like the build farm and scripts, the ROStest infrastructure, the simulator etc. (Please refer to chapter 2 for a discussion of these QA means.) For a comprehensive discussion of the list of the success of ROS is to a large degree based on the whole ecosystem fitting together and providing these nifty tools. The whole set of tools work together but still the user has the freedom to decide which of them to use.

This is also highlighted by application developers: ROS as well as ROS-Industrial supports quality in application development through the continuous integration and deployment system. Although this approach is not used to test in the development process, it is used as a check for dependencies. Before the Continuous Integration Infrastructure, the dependencies were managed manually by hand but now the billboard had been introduced.

The ROS community further provides a number of tutorials. Also here, the community continuously contributes, e.g. on the ROS Discourse forum. (ROS Discourse is a platform where people have discussions about topics concerning ROS.) There are people uploading videos that describe certain aspect such as debugging which is a great learning resource with good content for those who want to dig deeper. 'answers.ros.org' is the central place for ROS users to discuss questions and find or get answers.

For industrial users, it is important that they can retain the copyright for their application. This is one of the advantages of using ROS. Users decide whether they want to share their developments because the licenses that are attached to shared code do not enforce to open-source derived software. There are other opportunities when using ROS. Free applications and modules from the open-source community are an attraction.

The members of the community who contribute a package or a driver or use ROS in industrial applications highlight additional motivation: For one of the interviewees, the most positive aspect of work of driver development for ROS is that of reusability and that others can contribute to the driver and the applications. This is much more satisfying than developing something for own use only.

The interviewed driver and application developers highlight the friendly community as a motivation to get involved and stay engaged: The community engages through several forums: There is user support through the ROS Answers forum and this allow users to ask questions about a given component. The ROS Users mailing list is primarily for announcements. Discussions about packages or new packages, technological development are directed to the ROS Discourse site.

Over time, members get to know each other's names and experience and are able to differentiate between a novice and a guru. The ROS community uses the concept of 'karma' to indicate reputation in the community: A member earns 'karma' depending on how many positive recognitions he received for the quality of the contributions he made on the forum. Karma-points are a form of measurement that distinguish contributors. A contributor with a great number of karma earns respect when he asks questions compared to a contributor without any karma who might be ignored.

Summing up the section: Many of the motivation to choose ROS and to contribute to it are rooted in the fact that ROS is an open-source software and is used by more than one company or research group. The resources provided by the community not only form of the software product itself, but also as tools, advice and support is clearly perceived as an advantage.

3.3.2 ROS Quality

According to one of the interviewed core developers, the wide exposure is also one of the main contributions to the quality of the ROS core modules. Many projects implicitly help with use cases in order to develop and debug ROS and Gazebo.

The core developer and maintainer is by and large happy with the quality of ROS: Any part of the code will have bugs, but the rate at which bugs are discovered depends on how often the code path is exercised and in how many scenarios it is used. He judges that the core packages are probably less likely to have remarkable problems, as they are widely used. Exceptions might become visible when using core packages in an unexpected way: then there are chances that one runs into bugs. He cites a use case as an example, where a mobile robot drops out of the wireless network frequently and needs to reconnect again, which leads to the system running out of file descriptors - something that would not be a problem with a more stable network.

The ROS core developer estimates that ROS core and base parts have a good test coverage, as it has been used and maintained for a long time. This though depends on the kind of package. Different packages have different needs for testing. For example, the ROS module managing publishing and subscribing to messages and timers are all covered largely. The command line tools might have low or no coverage at all, as they are not crucial.

The regular build for the core part includes also higher-level packages. The testing of these packages can be regarded as integration test. Unit tests on this level are not specific for a single package anymore. “However, there is no defined infrastructure for a system test where one can say, fire up a Docker container, install the latest compiler, debian packages, or run some of the tutorials.“

Before the official release, changed packages are placed in a shadow release - a staging area - for a few days or up to a couple of weeks, to be used by community members. That way there is a high chance that errors get caught before it goes into the public repository that the majority of the community uses. Things that do not work and do not have an automated test are often caught by the community before it hits the public space.

There are however use cases that challenge the software beyond what has been anticipated. Those cases cannot be covered by tests.

Both ROS and ROS-Industrial work with coding guidelines and naming conventions. ROS-Industrial seems to enforce these guidelines more strictly. ROS-Industrial extends the ROS guidelines to fit with industrial robotics. E.g. industrial settings have clearly defined product names and services as compared to service robots which is still a comparatively recent field. This has repercussions on the number drivers and but also allow to make use of the industrial naming conventions.

To assure backward compatibility the tick-tock model is used, when changes break old APIs: Members are informed beforehand. They are also informed about the backward compatibility measures, what will be eliminated in the next version.

The Continuous Integration environments are mentioned by application developers as one of the important quality assurance measures.

Though they indicate concrete points for improvement, that are further discussed below the interviewees evaluate the quality of the core modules as high. Further, the core developers and maintainers are interested in providing good support for quality assurance for contributors of reusable packages and drivers and for application developers.

3.3.3 Activities of the ROS-Industrial community

Though ROS-I is a sub-community of ROS, the software provided through the ROS repository is complementary to the ROS software: ROS provides the basic packages providing the core of a robot operating system and a number of additional packages, developed both by industrial and non-industrial community members.

ROS-Industrial focuses on complementing ROS with specific packages relevant for industrial applications. For example, ROS-Industrial provides models and drivers for series of industrial robots. Other examples are modules for the calibration of robots, which is a basic requirement in industrial settings but less of a challenge for other types of robots, and algorithmic functionality for processing sensor data from industrial sensor hardware, such as laser scanners. A “[…I]n ROS-Industrial we basically say okay if we use ROS, it's all compatible, there is no difference basically at least at that level. But we focus primarily on industrial robotics and service robotics, those kind of things, in [...] industrial contexts.”

The ROS-Industrial community also contributes to the ROS community: E.g. a visualisation tool developed by a ROS-Industrial developer definitely satisfies users in ROS and ROS-Industrial. However, calibration routines for laser scanners that cost approximately € 60,000 might not be interesting for people in academia due the cost of the laser scanner.

At the time of the interview, Spring 2017, the majority of components in ROS-Industrial was open-source with a minimal amount that were not yet open-source.

Components developed outside the ROS-Industrial consortium can be ‘adopted’ by it. E.g. A manufacturer provided motion interface e.g., for Comau, Mitsubishi MXT, or ABB robots is superior to what could be developed by others, as hardware manufacturers know their products best. Algorithmic functionality might also be taken on if it has a clear industrial purpose such as for example object recognition or collision avoidance.

As described in chapter 2, both projects apply a similar community process based on Pull Requests submitted to the repositories hosting the packages. There is a two-level contribution process: changes to the core are subject to community deliberation, whereas additional package, like a path planner or hardware driver are welcome because those are typically orthogonal to what is already provided, and thus has a limited chance of introducing regressions or compatibility problems. If a contributor wants to make his package available through the ROS wiki and build farm, he or she has to follow a more rigorously defined release process.

3.3.4 ROS and ROS2

The developers at the OSRF are currently about to finalise ROS2. According to the OSRF developer, ROS2 was started to address some of the fundamental technical design issues that showed up over time in ROS1, sometimes due to new technical development or changes in the use cases for robots.

Improving these problems would probably have meant significant breakages in existing code. E.g. package discovery in ROS uses the file system, which is very slow. Normally, tickets are received about packages that cannot be found because they are not crawled. This issue cannot be fixed in ROS1. The number of such flaws in ROS1 had raised the question whether changes should be applied in the existing ROS when rolling out a new version and breaking applications of users or whether better to make a fresh start. The OSRF decided to develop and implement ROS 2. When developments on ROS2 is complete, it will be up to each individual user to decide whether to migrate their code to the new version. However, ROS2 systems will still coexist with the older version.

Conceptually, ROS2 is very similar to ROS1 with the same kind of concepts for building nodes, communication, publishing, subscribing, request and response. Two major changes are that ROS2 uses a C-based interface for the messaging core based on the DDS (Data Distribution Service) standard, which makes it easier for community members to develop interfaces to other languages. Now, community members have already written a C# API for ROS2 without support from the OSRF developers and admittedly with not much documentation at that level. They managed to look at the existing implementation in C++ and Python and came up with something similar in C#.

Further, the ambition is that ROS2 will support more platforms than ROS1. And that sometimes leads to surprises: For example, the type sub-system relied heavily on runtime resolution of identifiers. The Windows linker will not take the concept of linking without having all symbols resolved at build time rather than relying on them being resolved at execution time. It has been valuable to have tests on all platforms early to catch design flaws. Tests are run on all the three major platforms and if people come up with additional platforms, it should be straightforward to add support for them as well.

Comparing the motivation to develop ROS2 with the lifecycles of other software products, as e.g. discussed in the article "Software Engineering Beyond the Project" by Dittrich (2014), we can see that the development of ROS2 is an expected step in the evolution of a software product: the extension of the product with new features and the changes in the application domain stress the original technical design and result in a major technical re-development without challenging the continuity of central design decisions or the commitment of the company or community developing the core.

3.4 Quality in the community development process

Quality assurance in open-source projects takes place as part of the community development process. The quality related aspects of the community development process are described in section 2 of the deliverable. Here we present the view of the participants on the quality of this process.

At the top level, changes are discussed based on submitted ROS Enhancement Proposals (REPs). The past REPs present the community's agreement about certain aspects. These agreements are about technical design decisions and shared standards as well as about procedures.

According to the interviewed application developer, REPs provides important information to the ROS community. The features described in the REPs are basic but essential. E.g. one of the REPs provides information about encoding images and different encodings that can be used.

With respect to ROS, the OSRF plays a key role as it provides the majority of the maintainers for ROS. ROS core and other basic packages are maintained to a large extend by a team working at the OSRF (approximately ninety percent according to our interviewees). The ROS core with about sixty to seventy packages is maintained by core OSRF developers. A few specific parts like for example the LISP-interface, which is also part of the core, is maintained by an external person. The ROS base, on top of ROS core, is also maintained by employees of the ORSF. The RViz tool and robot_model package is maintained by OSFR as well.

According to the interviewed core developer and maintainer from the OSRF, the ROS community members actively participate in debugging and evolving the core by submitting issues as well as patches in the form of Pull Requests. However, only a few maintainers come from outside the OSRF. This means that the base system is under pressure as issues and Pull Requests are not as frequently processed as one could wish for. Spending time on maintaining already existing functionality rather than developing new software is already a toll on funding for the OSRF because funders are more interested in supporting new discoveries and innovations than maintaining what already exists.

The process around the submission, review, and acceptance of a pull request are detailed in section 2.

As some errors only show over time, it is very valuable that members of the community are willing to test the patches and run them in a real environment. Clearpath, a company developing mobile robots, recently took the initiative to review bigger patches and deploy them on their robots together with their own applications.

According to the interviewed application developer, the community plays an important role when planning to develop a specific functionality, like a driver for some hardware accessory. Community members might engage in the discussions on the design and the implementation.

The following subsections further discuss how the interviewees perceive how the ROS community cares for quality in the core development and supports contributors of reusable packages and applications in the development of quality software.

3.4.1 Communication between maintainers and developers

Issue Tracking and Pull Requests

The main communication between maintainers and developers is through the issue trackers and the pull request process. Due to the lack of resources, the maintainers' work in ROS seems to focus mainly on bug fixes and review of pull requests, not so much on new development. The interviewed core developer mentions that often the answer to a change request or a new feature proposal is that the originator best start to develop the feature him or herself. Bigger changes though often require a more in-depth discussion between contributor and maintainer. In the best case, the contact is established early so that the contributor can get timely feedback on the design in order to avoid spoiling significant own time as well as the maintainer's effort. Changes that break API might have wider impacts for the users of ROS. Contributor are asked to justify breaking API. For some changes, alternative ideas should be evaluated, but there is seldom time for this.

The received patches are of different quality: There are extreme cases, where a maintainer may get a patch that has never been compiled or run. In other situations, someone makes a pull request and matches the exact coding style of the the existing code and even argues that the code presented uses the style of the original code. Comments about errors, what changes have been made and suggestions are essential. If the new code is accompanied by test cases extending the regression tests for the module, the maintainer is even happier.

What is finally merged is a result of an often extensive dialogue between contributor and maintainer addressing coding style, structure, and design. Sometimes the proposed patch clashes with design decisions and rationales for other parts of ROS. This dialogue takes time. With relatively few team members compared to in the era of Willow Garage maintainers sometimes take shortcuts and fix the issue and merge the patch or, in severe cases, ignore and do not spent time on a pull request.

The Wiki

A second communication channel are the wiki pages: For the core developer and maintainer, the wiki pages serve as a reference point for old and new employees of the OSRF as well as ROS contributors and users. It is also a checkpoint for information on updates for new features. A contributor may notice something and say , "hey there's this new feature you added a year ago but wiki page does not mention it at all. " In such cases, people are always encouraged to update the wiki pages and if there is a ticket, they need to reference a change so that people are aware. Maintainers usually browse the wiki pages. The wiki is where changelogs are also shown. Developers proactively mention the wiki page as point of reference for contributors to view, review, comment and iterate on it.

Also for application developers, the users of ROS, the wiki is a central source of information. It is regarded as one of the reason why ROS has been so attractive. Although the wiki is not always up-to-date and missing some information, the information that is currently available is important. It is easy to navigate and find links to various information. Best practices published on wiki is also an inspiration to solve development problems. When running into problems, the wiki is used as an entry point to get solutions. However, the interviewee finds that at times coincidence plays a role when looking for a solution needed due to the (lacking) structure of the wiki. Also he experiences that the quality of different parts of the wiki varies.

Community Fora

Both the interviewed driver developer and the application developer highlight the importance of the different fora the community provides.

The driver developer finds that the open-source community in ROS is helpful.

The Question and Answer forum (Q&A) provides access to earlier discussions and example solutions and best practice can be accessed.

An interesting aspect of the ROS community is the karma system, expressing the merits of the community member. A community member earns karm depending on how many votes or likes his or her question or answer receives. A contributor with a great number of karma earns respects when he asks questions compared to a contributor without any karma who may be ignored for instance. By the term respect, the interviewee meant replies are received for any questions posted on the forum depending on previous contributions.

3.4.2 Pull request reviews and peer reviews

At the core of the QA in open-source-software is the review of patches respectively pull requests by the maintainers of the software.

Also in ROS, the maintainer evaluates pull requests and makes sure that they can be built and do not introduce regressions. However, both core developers and maintainers emphasise that they also look at the quality of the code with respect to coding standards and readability, cross-platform compatibility and overall design rationale. The ROS core developer emphasised that the amount of time spent on writing code is smaller compared to the time involved in at a later point in time understanding and maintaining the code.

Often it takes several iterations until the code is good enough to be merged.

Also in the application development, code reviews and other QA technique are practiced. Code is hereby evaluated line by line. In the company of the interviewed application developer, code review is done when as peer review between developers working together on a project.

In the context of re-usable package or driver development, sometimes packages are released to the outside without any defined or established process within the organisation. This might lead to low quality contributions and harm the organisation's reputation. According to the interviewee, at least a review process should be done before a contribution is released. An established process how to release software to the Open Source community does not exist.

3.4.3 Importance of Continuous Integration

All interviewees emphasize the importance of continuous integration and testing. On the core development side, pull requests need to pass all existing tests in order to make sure that no regressions are introduced. The code necessary for this infrastructure is treated as part of the core source code.

The core developers interviewed both propose to make better use of the build infrastructure: In ROS 2.0 code linters and static analysis tools are already applied.

From an application development perspective, the ROS-Industrial build infrastructure is used to make sure that there are no missing dependencies. One of the difficulties for industrial application development is that the ROS buildfarm requires to move code to the public realm. ROS-Industrial has developed an infrastructure that allows to use continuous integration on proprietary servers. The interviewee's company is in the process of integrating continuous integration in a better way in the proprietary development.

3.4.4 Package release and maintenance by community members

Open-source projects benefit from community members contributing additional functionality. This is not different for ROS and ROS-Industrial.

Whereas the review and maintenance process for the core software is rather strict, the ROS project runs a more accommodating policy with respect to package releases by community members. Only when the packages are made available via the ROS buildfarm a minimum of documentation is required.

With respect to ROS-Industrial, the process is a bit stricter. Here, the quality of the maintenance process can be subject of discussion between the technical lead and the maintainer of a project.

3.4.5 Summary: Quality in the community process

The interviews show that the development and maintenance processes described by the REP and wiki pages on maintenance are by and large applied. The maintainers of the core modules are very aware of their responsibility with respect to quality assurance and control. This quality work though can become quite cumbersome.

With respect to community provided functionality outside the core, less rigor is applied in order to encourage contributions. This in turn implies that the quality assurance of packages and drivers by community members depends on the quality assurance and control applied by the respective developing individual or organisation.

Below we analyse the quality challenges and remedies the interviewees see for their respective development practices.

3.5 Quality Challenges

In the interviews, we discussed general challenges and quality challenges. Challenges identified were specific for the different kinds of developments. However in some cases, similar factors were identified.

3.5.1 Challenges from the core developers' point of view

The challenges mentioned by the core developers focus on the QA resources and the core development process.

Lack of overview for new contributors.

For a newcomer to the community, it is not easy to find the relevant pages describing the quality criteria for submitted code and the deliberation and QA process for both changes to specific packages and changes related to ROS as such.

Package specific changes are discussed on the issue tracker of the related package.

The REP process is used for new proposals and discussions that are of a more fundamental nature, where several developers need to agree. This could be a detailed specification of an API e.g. a XML file which determines the exact structure of an XML. Another example is a REP that defines which platforms will be supported by a ROS distribution. Usually an idea is proposed by one person or a group sharing an idea. The authors use the REP both to get feedback and to come to a common agreement on how to implement the idea. However, not everything that maybe should be subject to a REP is documented in a REP. The interviewee mentions as an example that the launch files, that specify how processes are launched in ROS, do not have a REP.

The set of pages that are detailing required code quality criteria for contributions, best practices and standards to be used are not located at one place: Best practice pages, tutorials, accepted REPS might be relevant and all have their own place. A set of pages detailing a QA process that differs from both the pull request and the REP process is described as outdated and date back to Willow Garage times. From the author's own investigation, the impression of a company internal QA process can be confirmed. Both interviewees acknowledge that it might be difficult for newcomers to understand what QA to follow and how to follow it.

Heterogeneous quality criteria of maintainers

There is no one set of criteria for maintainers to address when reviewing pull requests. A maintainer has the freedom to make decisions on a package. Each package has its own level of scrutiny. But this may compromise quality assurance standards because the new option being added may not work. The interviewee highlights that though this is not optimal, it is not as severe as e.g. a regression or a breaking of APIs.

The criteria for pull requests and what is expected of a maintainer though is communicated among OSRF employees as part of the internal on-boarding process. (See below.) There is therefore a tacit common understanding of quality in the ROS core.

Maintenance effort

The speed of fixing bugs depends highly on the community and the time a maintainer can spend on reviewing tickets. ROS is not a funded project at the OSRF and 'there is no money in maintaining ROS'. Often maintenance work for ROS is done on overtime.

However, funded projects use ROS: 'This is a great reason to spend time on the code, fix it and make improvements.' Due to the lack of time, adherence to quality standards may not always be carried through: “I think certainly due to to very constrained time, it happens more often that someone will take a pull request [and] review it and accept it and later find out that he should have probably reviewed some parts more carefully [...]”

The interviewee finds it sometimes difficult to review and answer pull requests within a reasonable time frame, that means within a few days or weeks, which in turn might frustrate the contributors. “There is a high chance that if a contributor does not get response within three months, he may never get back and yet this could be a potential contributor.”

Balancing time spent supporting the open-source community with time needed for commercially relevant projects has also been taken up but the ROS-Industrial developer:

He emphasises that the open-source concept involves aligning the company goals with that of the community. This had been the case with Willow Garage. But also today, industry supports the Open Source idea. “Toyota for instance up to now supports them [OSFR] in their quest for automated vehicle products development. That helps but is not the same as hundreds of employees all doing the same thing basically and open sourcing everything in the process.” There is a need for shift in mindset about open-source ideas as well as the need for financial resources in order to increase participation and reap the overall benefits that comes with open-source paradigm. Several projects by ROS-Industrial are aimed at that goal.

Lack of Maintainers

Getting maintainers that will continuously spend time and prioritize to contribute to ROS is deemed challenging. ROS has not yet figured out a good way to involve more people outside the OSRF. This is also a challenge for ROS-Industrial: “And it's unfortunate, but I am happy to hear that we are not the only ones (ROS-Industrial) that have this problem. It's like I am almost hearing myself say these things because we have exactly the same problem.” Lack of maintainers is regarded as one of the biggest quality challenges for the future. Especially with ROS 2 being launched soon, the available time for ROS 1 will decrease.

Unmaintained packages

Normally, if a package is part of a release, it is required that there is a maintainer taking care of issues and defects, and makes sure that the package works with the changes decided for the future releases. However, this is not always the case. In some cases, a package is needed by the community, and though the situation is not optimal it is after minor adaptations released without an active maintainer.

Errors that only show after some time of deployment

As already mentioned above, there is currently no defined process that helps to discover defects that only show up when the software is deployed over an extended timespan.

3.5.2 Challenges from the driver developer’s point of view

Connection to Hardware

Driver development is focused on development of hardware drivers that enhance communication between ROS and robots. Concerns are mainly the interface between ROS and the actual robot.

The interviewed driver developer emphasises that working with the robot in terms of motion and control is not an easy task. The communication between the robot and ROS app. e.g. instructing the robot to move is challenging and needs to be tested and verified with real hardware involved.

Quality of architectural design

Also, the level of granularity of the interface is a design decision that requires experience with robot application. Normally, the first design is not always flexible and may execute only a pre-calculated track. The latest version of the driver developed by the interviewee now provides more flexibility; the interface is on a different level with more possible configurations added and this renders its usability in multiple ways. With experience and time, a driver will go through several redesign cycles.

Lack of established corporate processes to quality assure open-source contributions

The interviewee representing driver development and the interviewee representing application development both are employed in for profit companies that work with proprietary robot and application development. Both emphasise the lack of established organisational processes to quality assure releases to the open-source community. Companies are reluctant, as with the publicly available software the reputation of the organisation is at stake.

3.5.3 Challenges from the application development point of view

The QA challenges from an application development point of view focus more on the development process of the individual application, rather than on community and maintenance processes.

Complexity

Robot application are complex systems consisting of often unique hardware arrangements and several layers of software: Hardware drivers, the middleware that coordinates the heterogeneous hardware, algorithmic packages, e.g. to compute a movement path for the robot, and finally the application code. Debugging here becomes a problem

According to our application developer interviewee, the available tools support the debugging, but robots are complex systems consisting of subsystems that interact with each other where these tools are limited when trying to identify why a robot’s behavior is not as expected.

Such breakdowns may be attributed to a developer’s code issues in ROS, in other modules or even hardware problems. A recurrent problem are unannounced updates of ROS. ROS Distribution updates are well announced identifying a set of versioned packages. However smaller updates from packages within a distribution are more difficult to track; and sometimes there are changes in packages from one distribution to the next, that cause changes in the final behavior which are difficult to be foreseen.

Hardware and software often co-evolve during a project. This often requires to reconsider the application code when the hardware is changing. Depending on the design of the application code, this can be a challenge.

Complexity as a challenge is confirmed by the interviewed core developer and maintainer, who has substantial experience with developing software as well. He states that the biggest challenge for robotics applications as for any application of a similar make-up is the high degree of complexity: Finding out how to show confidently that a whole application or a big system at the end of the day satisfies quality requirements is daunting. He refers to autonomous driving as an example. This is especially the case for the autonomous car industry given the demands by the government and the people that their system is safe and free from errors. Checks can be done at every level with good tools or by using model driven development. Good coding and testing practices are contributing to QA as well. But proving the overall system quality becomes a huge challenge. The bottom line is that it is difficult to quantify quality of complex systems.

Selecting the right module

'You need to pick your poison' is described as one of the core challenges of application development: As the ROS community is an active one there are often many alternative modules to choose from, e.g. when looking for a navigation and planning system. The first problem is to find what is available. Some domains are better organised and e.g. have a wiki page that describes the different approaches and links to the individual projects. A second is the quality of the individual packages. This has to be evaluated by looking at the project and code itself. "You really have to pick your pieces for your application by hand and for each of them you have to judge, is it well enough documented, [is there] quality assurance and [is it] available on the platforms I need that… And that is really a piece by piece decision which is really tedious if you do [this for] like a bigger application."

ROS does not have a voting system that allows the community to evaluate and indicate the quality of available modules.

Interdisciplinary domain

Several interviewees address the interdisciplinary nature of the robotics field as a challenge:

Different disciplinary educational backgrounds such as software, mechanical, and electronic engineers are all critical to produce quality robots. For example, knowledge of Software engineering helps transition and translate ideas into algorithms that provides quality services. The percentage of each discipline involved in robotics depends on the use case.

The interviewed application developer states that the robotic sphere is a complex one and involves several experts for both hard and software to communicate with one another in an effective and efficient way. Programming is viewed as a tool. This means, some practices that come natural for a software developer such as functional or regression test might not be applied during the development process in a robotic team. The interviewee sees this as a big disadvantage.

Further, ROS itself becomes with a specific terminology and rationale which might make it difficult to understand with a disciplinary background, "for example, if you come from a computer science background, you have heard a lot about [...] producers and subscribers. But then suddenly in ROS, they start talking about topics and [..] it takes some time to really understand that it is exactly the same but with different name." Variations in naming creates many ambiguity and a developer has to take time grasping many variants of the same concept.

User Interfaces

The interviewed application developer highlights the design of user interfaces as a central challenge for application development: How can a complex system like a robot be represented so that the operator who is a domain expert can steer a robot. It is difficult for many robot developers to put themselves in the position of a potential user. This is especially important when the robot needs to be operated in harsh environments, e.g. to inspect equipment in the oil and gas industry.

3.6 Remedies: Existing ones and proposals

The interviewees were all asked for possible remedies and different approaches were discussed. In the following the proposed remedies are presented for each of the interviewed roles. Some of the proposals overlap. This is left on purpose to make visible what measure in which way is of advantage for different kinds of development.

3.6.1 Remedies from the core developers' point of view

The core developers and maintainers interviewed mainly focused on how to address the discussed challenges.

Clarifying code quality and QA standards

Both core developers and maintainers of ROS and ROS-Industrial mention the need to explicate and clarify code quality and QA standards. The OSRF developer proposes to use the REP mechanism, which would allow the community members to vote.

The ROS-Industrial maintainer highlights the need to align activities between ROS-Industrial and ROS to ensure that good work practices geared towards quality embraced by ROS-Industrial are also influencing the ROS core.

Onboarding of new maintainers

To bring new employees up to speed, the OSRF applies internal onboarding: experienced ROS developers are asked to mentor newcomers. The learning curve for new employees is quite high due lot of information to comprehend within a short time. New employees are given a big picture of what goes on in ROS and the expectations regarding review involved in maintenance.

New maintainers would typically take over one or two packages from ROS that are situated 'on a higher level' and do not carry as many dependencies. This allows the new employees to get used to maintaining packages, getting tickets from contributors, giving response and so on. That way new staff gets a feeling for working in an open-source environment.

ROS tries to encourage people outside the OSRF to become maintainers. This has been done through ROSCon where companies were asked if they were interested in helping out to review and merge pending requests. There has been good response. It's done in a way that company A reviews and tests pull requests of Company B and B does the same for A. This turned out to be a win-win situation for both companies: Though both spend time and resources, they both got their changes merged. However, getting people to be involved for longer time has been a challenge.

There are currently no systematic onboarding activities by the community for new ROS core contributors or maintainers from outside the OSRF.

Improving CI and Buildfarm with Static Analysis and Linters

For ROS and ROS-I developers and maintainers the build infrastructures are a daily work tool and are therefore not discussed in greater detail. In the interviews possibilities to improve them are discussed.

E.g. to improve the coding style for ROS2, the developers added linters to the build process that not only check code style but also implemented some static analysis. For instance, when one forgets to initialize a variable, to deallocate memory not used any more, or a declared variable is not used. These are defects that one easily misses during a manual pull request review.

An automatic software check for defects makes the interaction between maintainers and contributors easier. Practices such as static code scanning etc. are expected to uplift the standards and quality of systems developed in ROS.

3.6.2 Remedies from the driver developer's point of view

The driver developer reported on what helped him most in terms of quality assurance and control when developing his driver. Part of it though is not a community practice, but corporate practice that could be adapted in the community.

Documentation

Both, the driver and the application developer interviewed, emphasise that getting involved with ROS implies a steep learning curve. Training takes the form of tutorials, practice and' trial and error'. Initial knowledge or experience in programming helps in catching up. As a first task, an existing driver was updated.

The tutorial page has been very helpful, especially tutorials providing introduction for developers. The page is easy to follow, short and concise.

Best practices on the wiki page have been followed, especially, parts from thee ROS-Industrial Consortium that hint on how to write programs and document them. Good documentation will make it easier for others to understand what is written so that it can be reused.

Code review

The interviewee's company, a major developer for industrial robots, does a lot of quality assurance, the experience of quality assurance in both the robots and in programing is positive.

The Team Foundation Server for Microsoft is used for version control. There are also code reviews where a reviewer does approvals before the code is committed. In addition, there are those who make sure that what is completed is in the delivery system as part of quality assurance practices.

The buildfarm and continuous integration

The ROS buildfarm has not been used a lot in the driver development, as the driver is not yet public. The currently developed infrastructure for ROS-Industrial that allows to keep the proprietary code on premises is currently used to increase the test coverage.

Continuous testing

As far as possible, a nightly build and test scheme is applied. Tests are run partly with the help of a proprietary robot simulator. (See also related pattern in section 2.)

Testing has challenges from a development perspective. Communications and the robot motions caused by it are usually the main objectives. In order to control the robot, MoveIt has been used.

3.6.3 Remedies from the application development point of view

Documentation

Tutorials are highlighted by the core developer and maintainer interviewed: ROS has many tutorials on the lower level. Though these tutorials are not always perfectly maintained, following the tutorial helps to get things working though with some effort.

High level tutorials pave the way for more complicated tasks, and are a valuable resource for an application developer to build more complex applications.

Quality indicators for modules

One of the core developers interviewed is also developing robot applications. He discusses the possibility to develop quality indicators or rating for modules contributed by the community. Currently, there are no indicators that help to evaluate modules from a quality point of view. The core developer interviewed reports that there were discussions about having a system that allows users to vote on a package e.g. by giving a certain number of stars to a package for the code, for the quality or for the documentation. It could be good to have a central place where one could see which of the ten alternative packages are ranked high and why.

Continuous Integration

For the application developer interviewed, the Continuous Integration environment provided by ROS-Industrial is used and has proven useful. Continuous Integration is currently mainly used to make sure that there are no unresolved dependencies. The idea is to broaden the usage and include functional tests and regression test.

Testing support

Testing is regarded as critical in Robotics. There is need for an overall system tests. This can be done with automatic simulation after launching a push. However, the effort in setting up a test and maybe collect test data and script the test is perceived as rather cumbersome. The interviewee observed that, "… you are developing your application or you are developing your driver and you want to spend as little time writing or preparing the things for the test as possible." A developer working on a new component normally has more focus on the component rather than setting up tests and gather data based on simulation. Adoption and frequent use of the continuous integration and testing workflow may help to address this attitude. Automated testing would be a way forward and will save time.

Debugging support

Application debugging basically takes place using simulation and visualisation. Simulation is done to check for collision and unexpected occurrences. Visualisation uses the RViz-tool. Rviz allows visualization of what the robot is seeing, thinking and doing. RViz allows a developer to produce visual output from a robot and these outputs can be controlled. However, it is difficult to automate visual output when processing and producing images.

3.7 Summary and Conclusions

Based on the interviews, the analysis of both the community quality assurance in chapter 2 and the interviews and their analysis has been distinguished in three tiers of development: Application development, driver and re-usable package development and ROS core development.

The interviews clearly show that the quality challenges differ. The table below provides an overview over the challenges and remedies discussed in the interviews. This list serves as input to the conclusion in chapter 6.

Table 3.1 Overview over quality challenges and remedies.

Kind of development	**Challenges**	**Remedies**
Core development	● Lack of overview for new contributors ● Heterogeneous quality criteria of maintainers ● Maintenance effort ● Lack of maintainers ● Unmaintained packages ● Errors that only show after some time of deployment	● Clarifying code and quality standards ● Onboarding support for new maintainers ● Improving CI and Build Farm with static analysis and linters
Driver developers and developers of reusable packages	● Connection to hardware ● Quality of architectural design ● Lack of established corporate processes to quality assure open-source contributions	● Documentation ● Code review ● The build farm and continuous integration ● Continuous testing
Application Development	● Complexity ● Selecting the right module ● Interdisciplinary Domain ● User Interfaces	● Documentation ● Quality indicators for modules ● Continuous integration ● Testing support ● Debugging support

The analysis further provided a rich picture of how the community's QA processes and practices were implemented in the day to day work or maintainers, providers of drivers and re-usable packages and application developers.

These results will be further discussed in the conclusion chapter 6.

4. ROS Quality Issues

4.1 Introduction

One of the objectives of the ROSIN project is to raise the overall quality of ROS robotics software. We aim to contribute to this goal by developing code scanners that continuously and automatically analyze ROS software and detect as well as report programming errors and quality issues in the code. In order to figure out what kind of analysis tools are needed, we first need to understand what kind of errors ROS developers often make and what kind of code quality issues they often have. To this end, we systematically harvested a couple of hundred real documented bugs from a representative collection of ROS code repositories. We then analyzed this collection in order to figure out which kinds of analysis tools we need to build to support the developers. The analysis identified a number of important observations that we will guide our subsequent development of code scanning tools in the ROSIN project.

4.2 Method

In order to make sure that any observations and conclusions that would come out of our bug analysis would be valid in general and not just applicable and biased towards a particular narrow set of systems, we identified a collection of qualitatively different subject systems. We deliberately chose systems with which the ROSIN partners had first-hand experience. To further widen the scope of systems, we invited two external partners from the University of Minho, Portugal and Carnegie Mellon University (CMU), USA to join the efforts. They each contributed with a system that they had experience with and also had experience analyzing bugs from. Finally, our two industrial partners (ABB and Tecnalia) contributed with bugs harvested from their confidential systems. For those closed-source systems, we obviously cannot disclose as many details as from the other open-source systems; some of the information for these bugs will thus be marked confidential. The following figure gives an overview of the subject systems chosen for the ROS bug study and the sizes of their repositories (as of August 2017):

subject system	category	size	C++	C	Python	XML
Kobuki	ROS application	3.2 MB	2,768 LOC	0 LOC	2,268 LOC	614 LOC
Mavros	ROS application	1.7 MB	10,775 LOC	50 LOC	2,213 LOC	279 LOC
Universal Robot	ROS-Industrial	24.0 MB	1,418 LOC	0 LOC	1,733 LOC	407 LOC
Motoman	ROS-Industrial	24.0 MB	5,805 LOC	3,216 LOC	84 LOC	752 LOC
Turtlebot	ROS application	17.0 MB	136 LOC	0 LOC	133 LOC	417 LOC
Care-O-bot	ROS application	39.0 MB	40,482 LOC	272 LOC	9,979 LOC	22,276 LOC
Confidential	Closed source	N/A	N/A	N/A	N/A	N/A

4.2.1 Description of the subject systems

The **Kobuki** and **Turtlebot** repositories both provide complete stacks of packages that integrate the Kobuki and Turtlebot hardware platforms with ROS. These platforms are both implementations of the ROS platform interfaces as described in REP-119: Specification for TurtleBot Compatible Platforms (http://www.ros.org/reps/rep-0119.html), providing a common base for ROS users to build their own robots on-top of. Types of packages included range from low-level hardware drivers interfacing with servo motors and power systems to complete (example) applications for automatic docking and charging, autonomous navigation and dynamic leader-follower behaviours. Packages are mostly written in C++ and Python.

The **Universal Robots** and **Motoman** repositories provide packages for the integration of those industrial robot (controller) platforms with ROS and ROS-Industrial. Packages provided include low-level interfaces to the motion controllers of the robot, sensors and I/O interfaces, as well as higher-level declarative description packages that provide information on robot geometry, kinematic and dynamic properties, motion planning configurations and motion planning plugins. Both repositories also include programs that are to be executed on the industrial robot controller itself, and which will collaborate with their ROS counterparts. Packages are mostly written in C++, Python, and C.

MAVROS is slightly different from the other repositories in that it implements a bridge between ROS and the MAVLink protocol used for communicating with the autopilot computers of unmanned vehicles (air, ground, water). The MAVROS repository provides various tools and plugins that allow almost transparent bridging between a MAVLink enabled autopilot and a ROS application, exposing as much of the data gathered and processed by these systems to the ROS node graph. The repository contains only the bridging nodes, and explicitly leave the modelling of vehicle geometry, kinematics and dynamics to other packages. Packages are mostly written in C++ and Python.

The **Care-O-bot** is an autonomous service robot created by Fraunhofer IPA and uses ROS as its main control system. Nearly all robot behaviour is implemented in ROS nodes, and a significant part of it is open-source and made available through the GitHub repository that was analysed in this report. Package types provided include low-level interfaces to motor controllers and sensors, human-machine interfaces including speech, face, emotion and gesture recognition, collision free path planning, manipulation planners, object recognition and localisation components. Configuration packages for simulation of the entire robot are also included, as well as packages describing the robot's geometry, kinematics and dynamics. Packages are mostly written in C++ and Python.

4.2.2 Bug Harvesting

We decided to base the bugs harvesting on the online issue trackers associated with each of the projects. An issue tracker (or issue tracking system) is an online repository that maintains and manages a list of so-called issues related to a project (including bugs reported by users and/or developers). Using bugs from the issue tracker has two advantages. Bugs harvested from the issue tracker are guaranteed to be sufficiently interesting to be made into "an issue" and to refer to errors in the online code repository. Taking the bugs from the issue tracker thus helps with both generalizability and reproducibility of the results. (Note that the confidential systems from ABB and Tecnalia did not use a publicly available issue tracker which made it harder to identify as many historically documented bugs as in the other systems.)

We then followed the following methodology: First, we identified real documented bugs from the issue tracker of our subject systems. Second, we analyzed and explained each of the identified ROS bugs. Third and finally, we reflected on the aggregated data (the bug collection) to make observations and draw conclusions. We will now elaborate on each of the three steps.

4.2.2.1 Step 1: Identifying the ROS bugs

For each subject system, we went through the issues on the corresponding issue tracker. We recorded issues corresponding to bugs and assigned to each bug identified a hash code comprising the first seven hexadecimal digits of the bug report; e.g., 22e4e4f. This led to the identification of N=177 bugs distributed onto the subject systems as detailed in the following figure which shows the number of bugs and issues (as of August 2017) from each of the subject systems:

#bugs	subject system	#issues
57	Kobuki	325
40	Mavros	623
25	Universal Robot	158
22	Motoman	78
12	Turtlebot	170
11	Care-O-bot	182
10	*Confidential*	N/A

4.2.2.2 Step 2: Analyzing the ROS bugs

For each of the bugs identified in the previous step, we analyzed the bug. This second step requires considerably more effort than the first in the sense that, for each ROS bug identified, we manually analyze the issue, the patch fix, and the actual code to build an understanding of the bug. We give each bug a description, and mark it up by filling out a number of predefined fields as exemplified in Figure 4.1 which shows the bug record for bug #22e4e4f. The fields report on relevant information about the bug report and how it was fixed. Each bug is classified according to the CWE (Common Weakness Enumeration) which is a taxonomy of numbered software weaknesses and vulnerabilities. We follow CWE whenever possible. However, since CWE is mainly concerned with security, we had to extend it with a few additional types of bugs, including type errors, incorrect uses of APIs, among others. We also record a number of other relevant information about each bug.

4.2.2.3 Step 3: Reflecting on the ROS bugs

Finally, we reflect on the set of collected data and arrive at a number of observations and conclusions (see below).

- bug #22e4e4f -

title: Image from the Kinect v2 is flipped

description:

The Kinect v2 is a camera providing images and depth data. The driver Freenect2
is used to access the images from that device. By default, the images provided
by the camera + Freenect2 driver are "mirrored" or flipped by the vertical axis,
which is not standard and/or expected by the application using the images.

classification: CWE-137 Representation Errors #PHYSICAL

keywords: camera | driver | image | representation | format

system: confidential

severity: error

links: https://github.com/OpenKinect/libfreenect2 |
http://docs.opencv.org/2.4.13.2/modules/core/doc/operations_on_arrays.html?highlight=flip#cv2.flip |
http://www.ros.org/reps/rep-0118.html | http://www.ros.org/reps/rep-0103.html

bug:

phase: runtime-operation

specificity: robotics-specific

architectural-location: application-specific code

application: SLAM

task: SLAM

subsystem: driver

package: robdream/kintinuous/kintinuous_ros

languages: C++

detected-by: developer

reported-by: member developer

issue: https://git.code.tecnalia.com/robdream/kintinuous/issues/9

time-reported: 2017-03-13 (17:05)

reproducibility: always

trace: N/A

fix:

repo: https://git.code.tecnalia.com/robdream/kintinuous/commit/22e4e4f44ed3ead6f9d2d863fcd9d6d677e771f3

hash: 22e4e4f44ed3ead6f9d2d863fcd9d6d677e771f3

pull-request: N/A

fix-in: kintinuous_src/utils/Freenect2Reader.cpp

languages: C++

time: 2017-03-22 (11:46)

Figure 4.1: Bug record (bug #22e4e4f).

4.3. Results

In our analysis, we will focus on the collective properties of the bugs from all projects as a whole as opposed to studying properties of bugs from individual projects. We will go through a number of dimensions starting with the classification of the bugs (i.e., bug type).

4.3.1 Classification (bug type)

Figure 4.2 shows a comprehensive overview of the bug types in our bug collection (according to their classification field from the bug records). We see, for instance, that there are seven bugs with classification CWE-628 "Function Call with Incorrectly Specified Arguments" in our collection; two bugs in Kobuki and five bugs in mAvros. Another seven bugs are attributed to the "Use of Obsolete Functions" (CWE-477) which happens in four Kobuki, one Universal Robot, one Motoman, and one Care-O-Bot bug.

As expected, we see a broad range of different kinds of functional errors: 88 in total. These bugs span from errors involving the intended behavior of the robots, to memory management errors, all the way to arithmetic errors.

> OBSERVATION: Finding the functional errors is likely to take a *wide collection of different techniques and tools*. It appear to be worthwhile to add a wide collection of different code scanners to the continuous integration service.

Interestingly, we see a whopping 59 dependency errors that pertain to inconsistencies between files; i.e., inter-file errors. The presence of many dependency errors indicates that a Continuous Integration (CI) service would work well if it simply attempts to build the ROS projects. The collection also contains 17 compiler errors within a file; i.e., intra-file errors. These could also be caught at compile-time by the CI service if it simply invoked the compiler on all the files.

> OBSERVATION: A Continuous Integration service that simply *builds* and *compiles* the projects ought to work well. On our sample, it would catch about two out of five bugs.

The bug collection contains six concurrency errors which are significantly harder to detect because it involve reasoning about and correlating concurrent activities of multiple threads. Finally, there are seven miscellaneous errors which indicate sub-optimal issues in the code.

Σ	Bug type (classification)	CWE	K	A	U	M	T	C	X
88	**FUNCTIONAL ERRORS:**		**34**	**20**	**10**	**12**	**1**	**1**	**10**
7	Wrong Behavior	-	3		1	2		1	
7	Function Call with Incorrectly Specified Arguments	628	2	5					
5	Dangerous Behavior	-	1		2	2			
6	Use of Obsolete Functions	477	4		1	1			
4	Wrong Robot Model	-	2		2				
4	Incorrect Calculation	682	4						
4	Encoding Error	172			1	3			
3	Wrong Numeric Constant	-				3			
3	Representation Errors	-		2					1
3	Improper Input Validation	20		2					1
3	Incorrect Data Parsing	-							3
3	Exception Handling	-	3						
3	Improper Resource Shutdown or Release	404	2						1
2	Improper Control of a Resource Through its Lifetime	664							2
2	Memory Management	-	1		1				
2	Loop with Unreachable Exit Condition	835	1	1					
2	Expected Behavior Violation	440			1				1
2	Return of Wrong Status Code	393	1	1					
2	Off-by-One Error	193		1	1				
21	*other …*	-	10	8		1	1		1
59	**DEPENDENCY ERRORS:**		**15**	**8**	**11**	**6**	**11**	**8**	
15	Missing Dependency	-	2	3		2	3	5	
8	Runtime Dependency	-			8				
5	Missing Installation Dependency	-		1	1	2		1	
5	Interface Incompatibility	-	3				2		
4	Missing Include	-	2			1		1	
3	OS-library Incompatibility	-		1			2		
3	Wrong Remappings (application configuration)	-	2				1		
3	Use of Global Names	-					2	1	
2	Missing Build Dependency	-			2				
2	Circular Dependencies	-	1				1		
2	Linking Error	-	1			1			
7	*other …*	-	4	3					
17	**COMPILER ERRORS:**		**3**	**7**	**4**	**2**		**1**	
3	Missing Initialization of a Variable	456		3					
3	Incorrect Type Conversion or Cast	704	1	1		1			
3	Syntax Errors	-	1	1	1				
3	Undeclared Identifier	-		1	1	1			
2	Type Clash	-			2				
2	Wrong Number of Function Arguments	685		1				1	
1	*other…*	-	1						
6	**CONCURRENCY ERRORS:**		**3**	**3**					
6	Improper Synchronization (race condition)	362	3	3					
7	**MISCELLANEOUS ERRORS:**		**2**	**2**		**2**		**1**	
3	Inconsistent Naming	-	1			1		1	
2	Inefficiency Errors	-	1	1					
2	Dead Code	-		1		1			

Figure 4.2: The kinds of bugs in our bug collection. The gray lines designate categories with sub-totals. [K = Kobuki, A = mAvros, U = Uni. Robots, M = Motoman, T = Turtle, C = Care-O-bot, X = *confidential*]

4.3.2 Evolution

A number of the bugs appear to be caused by evolution. Such bugs occur when one part of ROS is evolved while others, that depend on or otherwise interact with it, are not. The following table gives an overview of the 33 bugs (19%) that appear to be caused by evolution. We see that errors that are caused by evolution appears to plays a role in all the projects sampled. These bugs could be caught by a CI service running a consistency checker to see if modifications to the source code (evolution) caused any existing code that it interacts with to break.

Σ	Bug type (classification)	CWE	K	A	U	M	T	C	X
33	**ERRORS caused by EVOLUTION:**		**13**	**5**	**5**	**3**	**3**	**3**	**1**
6	Use of Obsolete Functions	477	4		1	1			
5	Interface Incompatibility	-	3				2		
2	Missing Installation Dependency	-				1		1	
2	Undeclared Identifier	-			1	1			
2	Type Error	-			2				
2	Wrong Number of Function Arguments	685		1				1	
2	Function Call with Incorrectly Specified Arguments	628	2						
12	*other….*	-	4	4	1		1	1	1

QUALITATIVE: Looking deeper into the individual bugs in question, we see that they all have a common structure. One part of the program is changed—functionality is either added, removed, or modified—which then interacts and causes problems in some other part of the code, such as crashes, incompatibilities, or unintended behavior either at compile-, build-, or run-time. Bug #332f09f, for instance, adds functionality: new features in newer versions of the Robot Web Service (RWS) did not work with older system parts and is inappropriately reported as a communication problem at runtime; bug #27e7db9 removes functionality: when "mavconn" was removed from Mavros, the install destination for the "mavconn" headers were not appropriately updated, causing certain builds to break at build-time; and bug #263650d modifies functionality: the name "gps" was changed to "global_position", but this renaming was, for instance, not reflected in the function "_find_gps_topic()" which is a problem that appears at compile-time.

OBSERVATION: About one in five of the bugs appear to be *caused by evolution*. Every time functionality is added, removed, or modified in some part of ROS, it could be worthwhile to have the CI service check all components that interact with it.

4.3.3 Robotics-specific errors (with *physical* manifestation)

We also see a number of robotics-specific bugs that manifest themselves in physical reality. These are bugs where the programming causes some un-intended effect in the (physical) behavior of the robot or in (virtual) simulation. The following table shows 27 bugs (15%) that have to do with physical reality. Again, we see representation from all projects sampled. Intercepting these errors are harder as it often requires knowledge and modelling of the physical characteristics and use of the robot and correlating this with the implementation.

Σ	Bug type (classification)	CWE	K	A	U	M	T	C	X
27	**ERRORS that have PHYSICAL manifestation:**		**10**	**1**	**6**	**7**	**1**	**1**	**1**
7	Wrong Behavior	-	3		1	2		1	
5	Dangerous Behavior	-	1		2	2			
4	Incorrect Calculation	682	4						
4	Wrong Robot Model	-	2		2				
3	Wrong Numeric Constant	-				3			
2	Representation Errors	-		1					1
2	*other…*	-			1		1		

QUALITATIVE: Analyzing the individual bug reports, we see a lot of errors involving ***unintended actuation*** (i.e., *output* for the robot); in particular, ***motion***. We see ***no motion*** (e.g., bug #fc95a19, where the robot ceases to move, because the motion planner exceeded a driver's internally hardwired velocity limit of two radians per second); ***incorrect motion*** (e.g., bug #1c141a5, where the robot was supposed to move forwards but ended up moving backwards instead, due to accumulated rounding errors caused by the use of the wrong data type); ***too slow motion*** (e.g., bug # af7946f, where the robot would move very slowly and also turn in the wrong direction, due to an arithmetic calculation error); ***too fast motion*** (e.g., bug #ad906f0, where the robot would even do wheelies when instructed to move after being idle, due to lack of acceleration smoothing), or even ***dangerous motion*** (e.g., bugs #3f260cb and #b1b6fcb, where the robot could move after stop, respectively, pause).

Besides motion actuators, there is a single example involving another kind of actuator, namely **sound** (e.g., bug #5a44ead, which would not emit a success beep sound, because of premature termination).

There are also a number of bugs involving ***sensors*** (i.e., *input* to the robot). For instance, ***no sensory data*** (bug #f01d952, where no images were received from the camera, due to a missing runtime dependency); or ***wrong sensory data*** (bug #2f647af, which would give wrong odometry position estimates, because of an inappropriately reset offset; or bug #22e4e4f, for which the input from the camera is mirrored, due to not adhering to the conventions associated with the representation of the vertical axis).

Finally, a number of bugs involve ***simulation*** (i.e., *virtual* rather than *physical* reality). They range from ***no simulation*** (e.g., bug #4ea5ea7, which did not visualize the robot, due to a declarative modelling error); to ***wrong simulation*** (e.g., bug #493e3f9, which causes the robot to bounce back and forth in simulation, due to a wrong placement of the robot's center of gravity).

OBSERVATION: About one in six of the bugs have *physical manifestation*. There appears to be potential in developing code scanners targeting these kinds of bugs. (Presumably, off-the-shelf checkers will not be helpful as they require knowledge of physical reality.)

4.3.4 Resource Management Errors

An interesting category of errors can be labelled as resource management errors. These are errors where some resource is mis-manipulated in some inappropriate way. Our bug collection contains

18 such bugs (10%). This kind of error occurs in some of the projects sampled.

Σ	Bug type (classification)	CWE	K	A	U	M	T	C	X
18	**RESOURCE MANAGEMENT ERRORS:**		**9**	**6**					**3**
6	Improper Synchronization (race condition)	362	3	3					
3	Missing Initialization of a Variable			3					
3	Improper Resource Shutdown or Release		2						1
2	Improper Control of a Resource Through its Lifetime								2
4	*other …*		4						

QUALITATIVE: This category of bugs mis-manipulate various kinds of resources: For instance, ***streams*** (e.g., bug #2688e7a, where a communication stream is not reset after an error, as it should be; and hence cannot be reused); ***messages*** (e.g., bug #fcf9cd9, where a failure to initialize an attribute in a message causes incorrect timestamps, since they are not appropriately updated); ***network resources*** (e.g., both bugs #c5dc9de and 62a38a9, fail to properly release network resources upon receiving a Ctrl-C signal); ***pointers*** (e.g., bug #, where pointers will be freed/destructed twice due to circular references); and ***locks*** (e.g., bug #1f01916, which fails to acquire a lock before closing a TCP connection).

This bugs are interesting as they lead to difficult runtime problems, while existing techniques for automatic finding resource manipulation bugs [Abal et al., 2017 = Abal, I., Brabrand, C., Wasowski, A., "Effective Bug Finding in C Programs with Shape-and-Effect Abstractions", VMCAI 2017] might be applicable to detect them statically.

OBSERVATION: About one in ten bugs are *resource management errors*. This bug category could benefit from recent development in bug finding targeting precisely such errors.

4.3.5 Type Checking (Python)

Many of the errors appear to be type checking errors many of which occur because Python is not statically typed:

Σ	Bug type (classification)	CWE	K	A	U	M	T	C	X
14	**TYPE ERRORS:**		**2**	**6**	**3**	**2**		**1**	
3	Missing Initialization of a Variable	456		3					
3	Incorrect Type Conversion or Cast	704	1	1		1			
3	Undeclared Identifier	-		1	1	1			
2	Type Clash	-			2				
2	Wrong Number of Function Arguments	685		1				1	
1	*other…*	-	1						

However, optional static type checkers exist for Python; e.g., mypy (http://mypy-lang.org/).

OBSERVATION: About one in twelve bugs are *type errors*. It could be worthwhile to have the CI service run a Python type checker on the Python files.

4.3.6 Bug longevity (aka, bug lifespan)

The following table shows statistics about bug reporting versus fixing:

#	%	Reporting vs Fixing
135	76%	Both reported and fixed
35	20%	Not reported, just fixed
5	**3%**	**Reported, but never fixed**
2	1%	*unspecified*

We see that more than three out of four of the bugs (76%) are both reported and fixed. About a fifth of the bugs were fixed, but, for some reason, never reported as a bug (issue) in the first place. (Presumably, this is simply because developers who identify and fix a bug do not want to bother with bureaucratically creating an issue on the issue tracker.) Interestingly, only five bugs in the entire collection (only 3%) are reported as an issue, but, in fact, never fixed. However, three of these bugs were reported as issues only recently (within the last couple of months), so presumably the developers did not have enough time (yet?) to fix them. Only two of the bugs were reported a long time ago and not fixed. Bug #89145c4 is a "Dangerous behavior" bug from Universal Robots that was reported in October 2016. It has never been fixed despite having rather many supporters on the issue tracker. The bug itself involves problems with a potential "self-collision" when the angle of the joint gets close to +pi or -pi (i.e., ±3.1415). Bug #4ea5ea7 is a "Wrong Robot Model" error from Kobuki that was reported all the way back in December 2012 and for some reason never fixed. An error caused the robot not to be displayed at all in simulation. The bug appears to never have been fixed despite being reported such a long time ago.

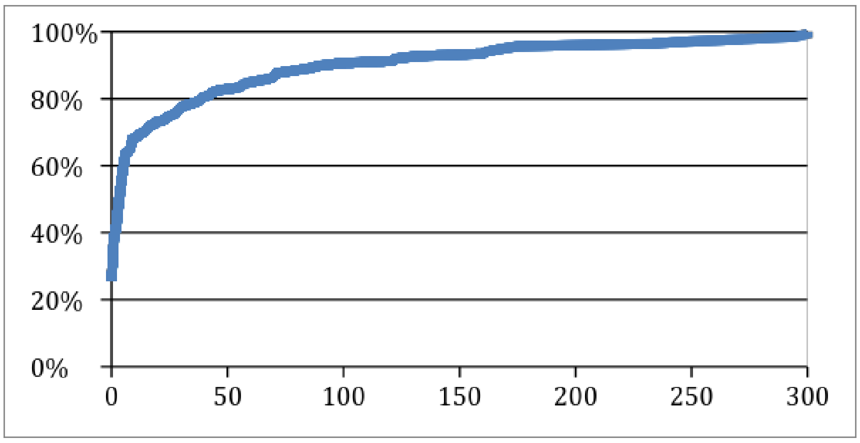
> OBSERVATION: Most bugs that are reported as issues are eventually fixed (almost no bugs are reported as issues and remain unfixed indefinitely).

For the 135 bugs that were both reported and fixed, we can calculate the bug longevity (aka, lifespan); i.e., how much time it took to fix the bugs:

Σ%	Bug longevity
25%	fixed same day
38%	fixed within 1 day
41%	fixed within 2 days
48%	fixed within 3 days
63%	fixed within 1 week
77%	fixed within 1 month
84%	fixed within 2 months
90%	fixed within 3 months
96%	fixed within 6 months
99%	fixed within 1 year
100%	fixed within 2 years

(Note that the 25% of bugs that are same day fixes is probably an upper bound estimate. Presumably, it also includes bugs that are found-and-fixed by a developer on different days, but only registered on the remote GIT repository later; i.e., where the report and fix issues are created on the same day.)

The following figure visualizes the data by plotting the accumulated percentage of bugs fixed (along the y-axis) as a function of how much time it took to fix the bug (along the x-axis):



OBSERVATION: Among the bugs that were both reported-*and*-fixed, about ¼ of the bugs are fixed on the *same day*, ½ *within three days*, ¾ *within a month*, and almost all bugs are fixed *within a year*.

4.3.7 Severity

The following figure shows the contents of our bug database for the severity field:

#	%	Severity
128	**72%**	**Error**
20	11%	Bad smell
17	10%	Minor issue
10	6%	Warning
2	1%	Bad style[1]

We observe that most bugs in the collection do, in fact, constitute significant issues:

OBSERVATION: Almost three out of four bugs in the collection are, in fact, *errors*. The rest constitute *lesser issues*. This report predominantly focusses on the significant errors.

[1] Usually known as "convention violations" in programming, but this term has a special meaning within ROS.

4.3.8 Phase

If we look at the phase field, we see, as expected, that most bugs (68%) occur at runtime:

#	%	Phase
92	52%	runtime-operation
29	16%	runtime-initialization
22	**12%**	**build-time**
20	11%	compile-time
14	**8%**	**deployment-time**

However, we also observe that a remarkable number of bugs involve build-time and deployment-time. In fact, if we do a simple text-based keyword search for the terms "package.xml" and "CMake" in the bug records, we get the following:

#	%	Involving / Mentioning
28	15%	Involving "**package.xml**"
23	14%	Involving "**CMake**"
51	29%	TOTAL

(Note that Package XML specifies meta-data, what dependencies are, versus CMake or CMakeList, which states what it should do with the build system (https://answers.ros.org/question/217475).) This leads us to the following observation:

OBSERVATION: About one in five bugs constitute *build-time* or *deployment-time* errors; and almost one out of every three bugs appear to involve the *build system* or *package meta-data*, referencing either "CMake" or "package.xml".

4.3.9 Subsystem

124 out of 177 of the bugs (70%) designate a particular sub-system:

#	%	Sub-system
68	**38%**	**Driver**
55	31%	*unspecified*
15	8%	Specific application component
6	3%	Simulation plug-in
5	3%	Motion

(Additionally, there are a number of subsystems mentioned in only a few bugs.)

OBSERVATION: It seems that the developers working with *drivers* (which are not ROS core developers) need more help with finding and fixing bugs. ROSIN should prioritize this.

4.3.10 Languages (involved in the bug and fix)

113 out of 177 bugs (64%) designate a language of the *bug;* and 167 out of 177 bugs (94%) mention languages for the *fix*. The following table gives the statistics for the *bug* and *fix* languages:

#	%	Bug language
58	33%	**C++**
19	11%	**Python**
8	5%	**CMake**
7	4%	*unspecified XML*
6	3%	**Package XML**
4	2%	**Launch XML**
3	2%	**Xacro**

#	%	Fix Language
80	45%	**C++**
22	12%	**Package XML**
19	11%	**Python**
14	8%	**CMake**
13	7%	**Xacro**
10	6%	**Launch XML**
9	5%	*unspecified XML*

(A number of languages are implicated in only a few bugs and fixes; e.g., YAML, TXT, URScript, ROS Message, C, STL, udev rules, and shell.) Only three bugs occur in multiple languages: #eed104d (which occurs in Package XML and CMake), #e05c71 (XML and CMake), and #599c588 (Python and C++). For bug and fix languages, we generally see the same six languages showing up:

OBSERVATION: Most bugs occur in and are fixed in the following languages: **C++**, **Python**, **CMake**, **Package XML**, **Launch XML**, and **Xacro**. It could be worthwhile to focus the code scanning efforts on these six languages along with the interplay between them.

4.3.11 Multiple languages

If we look deeper at how many languages are involved in a fix, we get the following data:

#	%	#Languages fixed
147	83%	Fixed in 1 language
16	**9%**	**Fixed in 2 languages**
1	**1%**	**Fixed in 3 languages**
3	**2%**	**Fixed in 4 languages**
10	6%	*unspecified*

We see that many bugs are fixed in multiple languages. In fact, three bugs are fixed in four languages; for instance, bug #2f647 fixes the four languages: C++, YAML, Launch XML, and Package XML; bug #65e7ee6 fixes XML, CMake, YAML, and C++; and bug #e05c71a fixes Ros Msgs, ROS services, Package XML, and CMake. The fix of one bug (#b5f0943) involves three languages, namely: C++, CMake, and Package XML. In total, 20 of the 177 bugs (11%) involve fixing more than one language:

> OBSERVATION: One in nine bugs appear to be fixed in more than one language. It could be worthwhile to investigate cross-language dependency closure bugs that occur because of inconsistencies between C++, Python, CMake, Package XML, and Launch XML.

4.3.12 Language categories

Let us now split the languages into two language categories: imperative programming languages (e.g., C++ and Python) vs declarative modelling languages (e.g., CMake and Package XML). This gives rise to the following table:

#	#bug	#fix	Language categories:
191 (62%)	**84 (72%)**	107 (55%)	Imperative programming languages
119 (38%)	32 (28%)	**87 (45%)**	Declarative modelling languages

This points out an interesting trend: most bugs (72%) appear to occur in imperative programming languages. Of course, this could possibly be due to developers (and users) being more "focussed" on imperative programming languages than on declarative modelling languages and hence mis-attribute the cause of a bug in their analysis and bug report. However, quite a number of bugs, 45% in fact, are fixed in declarative modelling languages:

> OBSERVATION: Even though almost three out of four bugs occur in *imperative programming languages*, almost half are fixed in *declarative modelling languages*. It could be worthwhile to look into code scanners for declarative modelling languages.

4.3.13 Domain-Specific Languages

The bugs mention a number of different Domain-Specific Languages (DSL's). For example, bug #377d7be involves communicating with the Motoman controller through a series of codes that have to follow the language rules (instructions and code have to obey some specific order). The ROS driver code for Motoman issues these codes in the wrong order. In another case, the STL files (3D modeling and mesh files) are stored with a wrong prefix (bug #eadbcb8). A similar problem is seen in bug #0829607. An interesting variation can be seen in bug #778c1ac, where a Launch XML file violates the schema, because the location of parameters have been changed in the schema.

The bugs in domain-specific languages are interesting from the language research perspective. Some of them can be addressed by incorporating schema checkers in the build system, especially for languages following standard technology stacks such as XML, JSON, or YAML (launch files, xacro files, and urd files and package files belong to this group). Some other of these bugs are much more difficult to handle, and require a more elaborate action than simple process improvement. Especially the problem of synthesizing correct scripts is a difficult research challenge in software language engineering. We will make examples of this bugs available for fellow researchers in this field, hoping to reduce this kind of problems long-term.

> OBSERVATION: There are many Domain-Specific Languages (DSL's) involved in the bugs; in particular: CMake and Package XML, but also: Launch XML and Xacro. For these languages, we may have to develop ROS-domain-specific code scanners. It could also be worthwhile to have the CI service with schema checkers for the XML-based DSL's such as Package XML, Launch XML, and Xacro.

4.3.14 Detected by

The following table gives statistics on how the bugs were detected:

#	%	Detected by:	During:	How:
117	66%	Developer	Development	Manually
24	14%	Runtime	Usage	Manually
13	7%	Compiler	Development	Automatically
11	6%	User	Usage	Manually
7	4%	Build system	Development	Automatically

(Additionally, two bugs were detected by a code scanning tool and one bug was detected by a so-called "Xacro Execution Engine".) If we consider that bugs detected by the developer, compiler, or build system are detected during development then they collectively account for 77% of the bugs vs 20% bugs detected during use. Bug #a794de9 was detected by an assertion violation (included under "compiler" above). Bug #fa64ec6 was detected by the Xacro Execution Engine. Of course, some developers may, in fact, be using tools to find the bugs, so the number of bugs detected by tools is presumably a lower bound.

> OBSERVATION: Four out of five bugs are detected *during development* (by the *developer*, *compiler*, or *analysis tools*). The remaining one in five bugs are detected *during usage* (by users or at runtime). This is not good for the reputation and the perception of ROS among its users.

We further observe (the last column in the table above) that 86% have been detected manually or at runtime, so in a rather inefficient process. These bugs are possibly most interesting from the ROSIN project perspective, as they represent opportunities for automation and development of further quality assurance support for the developers. We have qualitatively analyzed about half of these bugs grouping them into seven groups:

1. Dependency closure bugs (missing dependency between source code, build system, packaging system, and ros names, for instance topic names or message types) [30%]
2. Resource manipulation bugs (lock, socket, file manipulation, and accessing critical resources in concurrent programs) [6%]
3. Dynamic typing problems in python (problems that could have been detected at compile-time, should python be a statically typed language) [8%]

4. Coding style bug (a large category of problems that can be detected by pattern matching on syntax, ignoring error codes from functions that may fail, not handling exceptions, using C API instead of C++ API, using deprecated API, bad code smells) [9%]
5. Language processing errors (synthesis of broken scripts, syntactic mistakes in domain-specific formats, changes to schemas of description formats) [11%]
6. Physical bugs involving communication with hardware or interaction with physical environment. [13%]
7. Functional errors. [23%]

The percentages in each line capture how big part of the analyzed sample falls into this category. Groups 1-4 can likely be addressed automatically using relatively simple solutions, or adapting solutions. Some bugs in group 5 are difficult for most of the time, and require a low TRL research project to address (this topic is of growing interest in the language engineering community). Some others in these group can be addressed (see the section on domain-specific languages above). Bugs in group 6-7 likely cannot be detected in a general manner. They require tests or other specifications of correctness.

OBSERVATION: It appears then that about half (53%) of the problems could potentially be detected automatically using technology that is within reach. About 36% is difficult and requires higher level development techniques such as extensive automatic testing, formal methods, or specification-based model-driven development.

An attentive reader, will observe that many of the above groups have been observed in the other statistics in this document. In fact many examples of concrete bugs used in this document to exemplify the other statistics have been identified in the process of analyzing the manual bugs.

4.3.15 Reported by

The following table gives information on who (or what) reported the bugs:

#	%	Reported by:	By:
83	47%	Developer	Developer
60	34%	Contributor	Developer
21	12%	User	User
2	1%	Tool	Tool
11	6%	*unspecified*	N/A

Note that the perspective might be skewed by the fact that robotics is characterized by "pipeline development" in the sense that a developer can be simultaneously a provider with respect to one project and a consumer in another. Two of the bugs were automatically reported by tools: Bugs #f8d175b and #fd6b589 were detected and automatically reported by the ROS build system.

OBSERVATION: Four out of five bugs are reported by *developers*. About one out of seven bugs are reported by *users* which is bad for the reputation of ROS. Finally, only about one percent of bugs are automatically reported by by *tools*, so there seems to be potential in having the CI service issue bug reports automatically.

4.4 Conclusions

Overall, ROS is in decent shape when it comes to getting bugs fixed:

- Most bugs that are reported as issues are eventually fixed (almost no bugs are reported as issues and remain unfixed indefinitely).
- Among the bugs that were both reported-and-fixed, about ¼ of the bugs are fixed on the same day, ½ within three days, ¾ within a month, and almost all bugs are fixed within a year.

However, many bugs are unfortunately found or reported by users, which is bad for the reputation and the perception of ROS among its users:

- One out of five bugs are detected during usage (either by users or at runtime).
- One out of seven bugs are reported by users.

There seems to be good potential in having the CI service issue bug reports automatically (as only about one percent of bugs are currently automatically reported by by tools). There are a number of concrete recommendations for what to have the CI service run:

- A CI service that simply builds and compiles the projects ought to work well (as one out of five bugs constitute build-time or deployment-time errors).
- It appear to be worthwhile to add a wide collection of different code scanners to the continuous integration service (as it will take different tools to intercept different kinds of errors).
- It could be worthwhile to have the CI service run a Python type checker on the Python files (as about one in twelve bugs are type errors).
- Every time functionality is added, removed, or modified in some part of ROS, it could be worthwhile to have the CI service check all components that interact with it. (About one in five of the bugs appear to be caused by evolution.)
- It could also be worthwhile to have the CI service with schema checkers for the XML-based DSL's such as Package XML, Launch XML, and Xacro.

It appears then that about half (53%) of the problems could potentially be detected automatically using technology that is within reach. There are a number of concrete recommendations regarding the development and use of code scanners and analyzers:

- About one in ten bugs are resource management errors. This bug category could benefit from recent development in bug finding targeting precisely such errors.
- It seems that the developers working with drivers (which are not ROS core developers) need more help with finding and fixing bugs. ROSIN should prioritize this.
- Most bugs occur in and are fixed in the following languages: C++, Python, CMake, Package XML, Launch XML, and Xacro. It could be worthwhile to focus the code scanning efforts on these six languages along with the interplay between them.

Finding the functional errors is likely to take a wide collection of different techniques and tools. This is likely outside of ROSIN's scope.

Finally, there are some more challenging and robotics-specific errors that may be worth considering:

- About one in six of the bugs have physical manifestation. There appears to be potential in developing code scanners targeting these kinds of bugs. (Presumably, off-the-shelf checkers will not be helpful as they require knowledge of physical reality.)
- About 36% is difficult and requires higher level development techniques such as extensive automatic testing, formal methods, or specification-based model-driven development.
- There are many Domain-Specific Languages (DSL's) involved in the bugs; in particular: CMake and Package XML, but also: Launch XML and Xacro. For these languages, we may have to develop ROS-domain-specific code scanners.
- It could be worthwhile to investigate cross-language dependency closure bugs that occur because of inconsistencies between C++, Python, CMake, Package XML, and Launch XML (as one in nine bugs appear to be fixed in more than one language).
- It could be worthwhile to look into code scanners for declarative modelling languages (because, even though almost three out of four bugs occur in imperative programming languages, almost half are fixed in declarative modelling languages).

5. Early Interventions: QA Tools & the QA Hub

Following the plan for Task 3.2 we have performed the first interventions into ROS community by improving the usefulness and visibility of the Continuous Integration services (Section 5.1) and initiating the quality assurance hub for developers entering the community (Section 5.2). This actions were performed partly to address the original description of work (T3.2), but also to shed some light on our future modus of operandi, giving light weight examples of future engagements with the community: providing education material, contributing to existing QA tools, and creating a communication center (the hub) for discussing and learning about quality issues. Of course, after the initial phase of ROSIN all these are merely at a nucleus stage, but way more is planned. In the future periods of ROSIN we intend to build new QA tools, attend personal meetings with developers, and develop the hub further. We will also constantly think of new innovative ways of engaging with the ROS and ROS-I community, quality issues.

5.1 Modernizing, Tailoring, and Scaling up the Continuous Integration Service

This part of the deliverable is primarily documenting the efforts of Task 3.1.

5.1.1 Build Farm and ROS Wiki

As we have seen in Chapter 4, a large number of quality issues in ROS can be addressed by spreading the use of continuous integration. According to the study presented there, up to 40% bugs could have been detected, if carefully exploiting the already available benefits of CI.

The ROS project already has an official build farm hosted by the OSRF and based on Jenkins. This ROS BuildFarm is integrated with ROS wiki such that components that are using CI receive a suitable badge on the front page. Presently the wiki shows a green "Continuous Integration" Badge that is displayed for projects that have jobs running on the OSRF cloud:

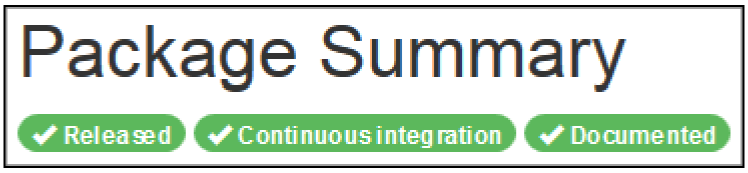


Fig. 5.1 Standard ROS Wiki CI badge (in the center)

This badge has two purposes. First, it indicates to component users that the maintainers of this component are using continuous integration (among other things), which likely means that they care about quality assurance and the quality of the package is probably better than average. Second, it incentivizes the maintainers to use CI, as they know that they will be rewarded with the badge on their ROS Wiki page, which increases their reputation.

Besides the badge there is a list Jenkins jobs in the side panel, with information about passing/failing build jobs, and passing/failing tests. However this list is not easily accessible, as it needs to be manually unfolded. It also appears too detailed for a casual user coming to the wiki.

The ROSIN project recognizes the importance of the BuildFarm, ROS Wiki, and the badges as a form of gamification quality assurance in open-source development. Yet, we chose to reinforce the effect of the top level **Continuous Integration** badge even further, to inform not only about the project using CI, but also the extent to which it succeeds in using it. This can be done by replacing the badge with information about the success rate of build and test jobs. The old badge (above) is green as soon as a component uses CI. We decided to show whether CI is used properly: whether build jobs fail, pass, in which proportion, and highlight the badge in red, if the failure rate is high. This should incentivize package owners to follow CI results more closely, and, even more importantly, should inform the package users about benefits of CI better, so that they might be inclined to use it themselves.

The following figure shows how the implemented badge looks for a component. The white cross in front of “Continuous Integration” and the red color of the badge belong to the aggregated status which in this example is set to “unstable” by the buildfarm. Other statuses are “stable” (leading to a checkmark instead of a cross and a green badge) or “other” (resulting in a dash and a yellow badge). Build status aggregation is done by the buildfarm and visualised by the wiki. The numbers to the right indicate that there is a total of 11 builds of which 10 succeeded. Previously accessing this information required 3 clicks and thus the way CI is used by the component was invisible for most visitors. The user had to first visit the ROS Wiki, open the list of Jenkins Jobs on the right and select the particular build.

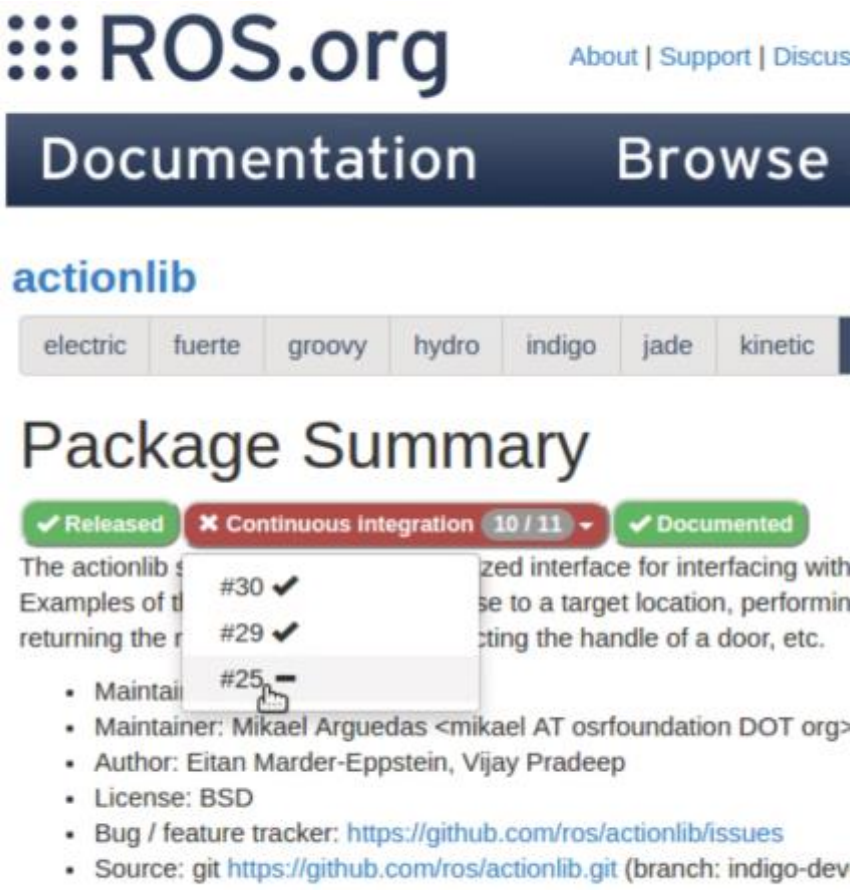
When the new badge is clicked a drop-down list is open with a summary of jobs handled by the buildfarm where each number stands for a job ID. The user can click on each job which links to the job’s overview on the buildfarm. Just like for the badge itself, the list shows a checkmark, dash or a cross. The status is derived from the success / failure rate of tests for that particular job ID. In this example Job #30 ran a set of tests without any failures whereas job #25 had some failures but also some successful test runs.

Previously this information was very hard to access as the user had to look for the particular build information in “Jenkins Jobs” on the wiki page, decide which build information is of interest (either source / binary including the operating system selection) and distinguish between jenkin’s information pattern of success or failure of jobs. The user also had to find a way to get information about success or failure of tests for the job ID.

No special action is required from the package owner to use this service. It just suffices that they start using the buildfarm. All of the package headers are generated using the information in manifest files, as with the previous badge.

Besides making the CI infrastructure (even) more visible we settled to expand the CI possibilities by incorporating extra scanning service that provides feedback about quality of code without requiring to write tests or specification. We chose to use the HAROS project (Santos et al. 2016), which has been funded in a parallel European project via the European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme. HAROS computes a number of standard quality metrics, and compliance reports for ROS components, generating a web report. This report is easy to directly integrate into the ROS Wiki of the component.

A developer wishing to have HAROS reports generated for his component can include haros as a test dependency. The build farm will install it and execute automatically. We decided to settle on this opt-in approach after communication with OSRF, which would prefer to stage the introduction of such practices, and align them with other efforts in extending the use of linting in ROS.



ROS.org

About | Support | Discus

Documentation Browse

actionlib

electric | fuerte | groovy | hydro | indigo | jade | kinetic

Package Summary

✓ Released ✗ Continuous integration 10 / 11 ✓ Documented

#30 ✓
#29 ✓
#25 −

The actionlib zed interface for interfacing with
Examples of t se to a target location, performin
returning the r ting the handle of a door, etc.

- Maintai
- Maintainer: Mikael Arguedas <mikael AT osrfoundation DOT org>
- Author: Eitan Marder-Eppstein, Vijay Pradeep
- License: BSD
- Bug / feature tracker: https://github.com/ros/actionlib/issues
- Source: git https://github.com/ros/actionlib.git (branch: indigo-dev

Fig. 5.2 The expanded interactive continuous integration badge developed in ROSIN.

Implementation: the implementation of the above extensions required modifying the ROS wiki and agreeing on a communication interface between the build farm and the Wiki. A simple YAML format has been selected. Instead of gathering the information from the buildfarm each time a user visits the wiki page, the YAML file is generated once by the buildfarm per build run. As already handled for doc-job information generation this YAML file is used to generate the website by the ROS wiki python scripts. he presentation processing happens on the client side. This way the ROS Wiki service remains responsive.

The HAROS integration has been implemented with an opt-in assumption. We packaged HAROS as a ROS component and allow including it as a test dependency for the projects. This has been modeled after the roslint project (wiki.ros.org/roslint 2017). Previously HAROS was a standalone tool, largely independent of ROS, able to scan projects in a local directory. We repackaged HAROS as a ROS build tool.

Deployment: the intention of ROSIN is to contribute as much of the project's outcomes to the community. To sustain the longevity of the results, we prefer to include our developments into existing projects and hand over their maintenance to the respective project maintainers, instead of setting up parallel infrastructure that will only be maintained for the duration of the project. For example, we chose not to create our own version of HAROS, but contributed necessary integration changes upstream to the maintainer of the project (github.com/git-afsantos/haros/pull/7, 2017; github.com/git-afsantos/haros/pull/8). Similarly, we are in dialog with OSRF regarding the ROS Wiki changes.

Development of the additions to the wiki was done on a private instance of the ROS buildfarm (rosin-build-master.3me.tudelft.nl 2017) that is identical to the public buildfarm that is operated by the OSRF, but only accessible to ROSIN project partners. This was done for two reasons. First, the public buildfarm is a crucial piece of infrastructure that has high availability requirements as it is responsible for all the Q&A as well as all day-to-day software build output of the entire ROS project. Second, by running a separate instance for ROSIN, we are free to experiment with whatever new techniques the project sees fit (be it code scanning tools, reporting utilities or wiki enhancements) without interfering with day-to-day operations or having to coordinate with the OSRF.

Deployment of the wiki enhancements as described in the previous sections is delayed until the OSRF has completed their planned upgrade of the buildfarm infrastructure and supporting software. Besides upgrading to a more recent version of Ubuntu, this upgrade also includes significant changes to the scripts that generate jobs - such as the ones extended by us to enable the wiki enhancements - and it was therefore concluded that synchronising our contribution to this upgrade schedule would be more efficient than trying to keep up with changes in the upgraded scripts.

5.1.2 Industrial CI

The ROS-Industrial organization has developed *industrial_ci*, which is a set of scripts and configurations to be used as part of the Continuous Integration process. Originally it was designed and developed to be used with repositories hosted in GitHub, and taking advantage of the excellent integration of Travis CI and GitHub. However, it can be easily used with other infrastructure such as the combination of GitLab and Jenkins. *Industrial_ci* offers a good combination of power and simplicity, where the jobs can be easily customized by changing a set of variables. The objectives of *industrial_ci* are the same as for the main build farm:

- Check if the ROS package builds correctly (e.g. compilation problems, dependency resolution)
- If tests are defined, run them and ensure that they pass successfully
- Check that the package builds correctly into install space, not only build space
- If there are downstream packages defined (packages that depend on the one being built) , run also the tests in those packages

The *industrial_ci*, however, has several advantages over the main OSRF build farm. As opposed to the complexity of getting a package accepted in the official buildfarm infrastructure, *industrial_ci* focuses on simplicity, highly reducing the effort for a developer to have a simple CI process running in his package. It might be considered an interesting alternative for industrial users of ROS, who seek simplicity and/or work with other code repositories, including closed source repositories.

Pattern 10 presents a simple case and guide on how to setup a private infrastructure to run *industrial_ci* in an automated way. This can be useful for users not wanting to rely on the combination of GitHub and Travis CI, but specially for cases where the source code needs to be kept private, and the process run in a private environment. An audience for this case can be companies developing based on ROS, but willing to keep parts of their developments private. Even if the guide shows how to do the setup using Jenkins and GitLab, it can be easily extended to other combinations. The guide is also included in the appendix of this deliverable.

5.2 ROS Quality Hub

The vision for the ROS Quality Assurance Hub is to create a website which is an entry point for professional industrial users of ROS that want to learn about applying standard engineering processes with ROS. Creating such a website, and gaining the reputation from the community is, of course, very difficult, and will take a long time. What we are presenting in this deliverable is a starting point, which is a collection of process descriptions for existing processes in ROS. This is primarily based on the community patterns described earlier in Table 2.1. Besides this it contains other relevant data from this deliverable (for instance the data about the studied bug collection).

Our vision for this website is to grow it along several dimensions (note that these are possibilities, as the actual development will largely be shaped in dialog with the community):

- New pattern descriptions for industrial-related quality assurance processes that are not currently used in ROS and ROS-I (for instance architectural design principles, the patterns for the role of a professional tester, or release engineering and devops)
- Articles about exemplifying industrial activities and how-tos with ROS and ROS-I.
- Benchmarks for researchers that can use ROS to test their methods and tools for quality assurance (and eventually contribute to ROS).

We intend to disseminate the information about the quality assurance hub via social media, ROS community channels, and at developer meetups and conferences.

We send the developers to the online resource at http://rosin-project.github.io/quality-hub. However, the patterns and the materials are the same as presented above in this deliverable.

6. Conclusion: Possible directions for solutions

This chapter summarises the results of the different chapters and lists a number of possible directions for future development. The points raised here are meant as a starting point for both a discussion within the ROSIN project and with the ROS and ROS-Industrial communities. The list below therefore will cover a wider range than what can possibly be achieved in the remainder of the project.

The conclusions do not differentiate between ROS and ROS-Industrial: ROS-Industrial is a sub-community of ROS. However, the software available from the ROS-Industrial repositories is complementary to the ROS software. As the quality of the final application is depending on the quality of all software modules part of it, the quality of industrial ROS applications is depending on the quality of the core and of the re-usable packages. QA of the ROS core is therefore as relevant. The part of QA focusing on re-usable package and driver development and application development are formulated from an industrial application point of view. Experimental and research robots might e.g. not be as interested in systematic testing as industrial application developers.

Though the quality of the ROS core is of importance for all ROS applications and therefore for the whole ROS community, the interest to improve QA and to work with improved tools and processes for non-core packages, drivers and applications might be more wide-spread in the ROS-industrial community than in other parts of the community. For the future work, the promotion of QA needs to clarify what techniques, tools and processes shall be developed for the ROS-Industrial sub-community and which should target ROS as a whole.

The Quality Hub:
Clarifying Quality Assurance and Quality Control processes in ROS and ROS-Industrial

Chapter 2 showed that ROS and ROS-Industrial already today apply quality assurance and quality control in the core development and provide a number of quality control tools for the community and also for application developers. The conclusions show how the identified processes and tools correspond to state of the art quality assurance and quality control practices.

For good reasons QA and QC for core development is further developed than for the development of reusable non-core packages and drivers and for application development. The quality of ROS and ROS-Industrial stands and falls with the quality of the core. The quality of contributions of non-core packages is the responsibility of the respective developers and maintainers. Each application developer is responsible for the quality of their own application. Here the community provides tools, but leaves it to the individual or the organisation how to use them.

However, the information about QA and QC on the wiki is fragmented and it contains partly contradictory descriptions of the development processes (REPs and the Quality Assurance pages on the ROS wiki). The major part of the latter pages is regarded as outdated by core community members. A clear description of *current* QA practices in the ROS and ROS-Industrial community would already help to develop trust in the quality of both.

Further, the clarification of quality control tools like rostest, the CI infrastructures and simulators for example and how to embed them into corporate development processes will allow companies to establish quality assurance and control for their corporate development processes.

The development of QA-patterns that take a quality problem as a starting point and detail practices and tools to address them is meant as the core for the Quality Hub described in the last chapter, where the tools, techniques, methods and processes developed by the ROSIN project and the community can be made available in one place. The material provided should be organised according to the kind of development process and persona addressed by the QA support.

The results of Chapter 3 fall into 4 categories: Specific support for each of the three development practices. However, we also see the need to develop a shared understanding of QA and QC among at least the industrial sub-community in order for both to further develop the trust in the ROS open-source software and to support companies to make use of the QA and QC tools provided.

Making Quality Assurance and Quality Control practices easily available:
Support for high quality application development:

One of the core problems becoming visible during the development of Chapter 2 and the interview analysis of Chapter 3 is the difficulty to both get access to and to start to apply QA and QC practices. The above-mentioned quality hub tries to address part of this need, as it both collects information about QA and QC as well as communicates it in a problem and solution oriented form.

Increasing the usability of the tools further promotes accessibility. The ROS and ROS-Industrial communities do already provide a number of tools and infrastructures, ranging from simulation (Gazebo) and visualisation of program deployment (RViz), to CI infrastructures and test tools and environments (ROStest). And many of them are very well appreciated by application developers. However, some of the tools are cumbersome to use.

In order to invite new industrial application developers into the community, the tools should support the integration of public, that means open-source code and proprietary code. Specific patterns can support companies to include the respective tools into their corporate quality assurance and control practices.

Making Quality Visible:
Support for non-core package and driver development and usage.

The better availability of QA practices and supportive tools and infrastructures will also help developers and maintainers of reusable non-core packages and drivers.

The interviewed industrial developers both mention the reluctance of companies put their names to published packages: Low quality of the open-sourced package might reflect poorly on the company. To support companies to contribute to the ROS open source project in a responsible way, a set of good practices for pre-release quality assurance and maintenance of the published packages might make it easier for companies to decide for open-sourcing reusable parts of their applications and at the same time benefit from the quality increase the external usage of the package will likely result in.

In order to not only offer the possibility but to also promote the usage of quality assurance measures, several of the interviewees proposed to visualise the quality of these packages in some way. The extension of the industrial_ci infrastructure described in Chapter 5 is a first step into this direction: Here the test results of the (regression) test suite of the package is made visible. However other information like last update, state of maintenance, coding guidelines and standards to be applied etc. are used by expert application developers to judge a package. This information though is not easily available for application developers who are not open-source savy. A system that reuses the Karma system from the ROS Answers forum could add an element of gamification, where providers of packages and drivers are motivated to e.g. gain quality badges for different quality aspects for their package.

Interviewees mentioned that several sub-communities interested in certain aspects of robotics, e.g. navigation, developed pages that evaluate and compare packages in their area of interest. The development of such pages by domain experts could be encouraged. Such pages could be made available through the quality hub.

Tutorials should include QA and QC as standard parts.

Supporting contributors and maintainers to take care of ROS' quality:
Support for the evolution of ROS Core.

Common maintenance procedures and quality criteria need to be developed respectively explicated. Though the REP process is a common denominator of how to address changes that affect more than one specific module, much of the day-to-day ROS evolution and maintenance is communicated as tacit knowledge and not available to community members outside the core groups of ROS and ROS-Industrial maintainers. This hinders community members to contribute and it hinders community members to take over responsibility for the maintenance of core modules: What is the role of the maintainer? How do maintainers triage and prioritise items on the issue tracker? What are the criteria they apply when reviewing a pull request? How am I expected to document my patch? The maintainers do not have a common understanding of their role and do not apply common quality criteria. It can therefore be quite challenging for new contributors to understand what is expected of them.

Both maintainers interviewed proposed and appreciated the usage of linters and static analysis tools to encourage coding styles and avoid easily recognisable errors. Such tools so far mainly have been applied in the development of ROS2. Including similar tools into the ROS1 CI infrastructure would allow maintainers to focus on semantic issues of the code as other quality issues are taked care of by the CI infrastructure.

ROS-Industrial already took initiative to invite newcomers to the community to participate as developers or maintainers in certain areas. A similar 'job list' for the ROS core is missing. Members of the core community and maintainers only become visible as experts on the ROS Answers list, their role and responsibilities clarified. Such openes could invite experts from companies to join the maintenance and evolution of the jointly used and appreciated software.

Both ROS and ROS-Industrial both lack a clear onboarding procedure for new maintainers: Currently, the burden on the core maintainers is huge and the survival of ROS depends on a low number of dedicated individuals. The interviewed core developer clearly indicated the shortcomings of the current situation. The above-mentioned measures would already make visible how e.g. members from companies that use ROS and have an interest in maintaining a high quality ROS core could contribute to the maintenance and evolution of ROS. However, the evolution from a ROS user and ROS contributor to a ROS maintainer can be supported through more explicit measures: Dedicated 'best practices', a forum where new maintainers can raise issues they do not yet have enough knowledge of to address themselves and mentoring of new maintainers are just some examples of onboarding activities the community could introduce. The initial additional effort would quickly pay off.

Talking about Quality:
Developing a ROS community quality culture

For an outsider, the current community appears to be stratified along the identified kinds of development. There does not seem to be a shared understanding of quality for robot applications. This can in part be due to the heterogeneity of the community consisting of a range of different members from enthusiastic students and research groups to industrial robot manufacturers and industrial robot application developers. Additionally different disciplinary rationales and quality traditions contribute to the heterogeneity. However, robots become more and more important in our society and we rely on them behaving correct both in industrial settings (Industry 4.0) and in our everyday life. The quality of the software driving the robot will be of utmost importance for the acceptance of robots in society.

In an open-source project it is impossible to enforce one quality standard, neither based on the tradition of one of the involved disciplines nor of one part of the community. However, by providing a place to discuss quality issues and to share methods, tools and techniques, the community can be supported to develop quality criteria and standards adequate for the requirements of its members.

WP3 of the ROSIN project can contribute to kick start this process.

Development and use of code scanning tools

Our investigation of bug reports in ROS indicates that ROS is already a mature, active, and well organized open-source project, at least from the QA-effectiveness perspective. Most bugs that are reported as issues are eventually fixed (almost no remain unfixed indefinitely: among the reported bugs ¼ of the bugs are fixed on the same day, ½ within three days, ¾ within a month, and almost all bugs are fixed within a year). Yet, this does not mean that ROS has no quality problems. Many bugs are unfortunately still found or reported by users, which is bad for the reputation and the external perception of ROS: One out of five bugs are detected during usage (either by users or at runtime). One out of seven bugs are reported by users. One of the goals of ROSIN is to improve the situation by developing and introducing new static checking tools (and fostering use of the existing ones), in order to enable the project's developers to identify more problems before they hit the user.

The data confirms that expansion and popularization of CI services should decrease the overall bug density in ROS. About 20% bugs that are reported are detected at build-time (the main strength of a CI service is detecting such bugs). It appear to be worthwhile to add a wide collection of different code scanners to the continuous integration services (as it will take different tools to intercept different kinds of errors). The ROS2 project is already moving in this direction and we intend to collaborate on OSRF to use their interfaces to that end. In particular we consider building the following scanners:

- It could be worthwhile to have the CI service run a Python type checker on the Python files (as about one in twelve bugs are type errors). Suitability of existing tools, such as mypy (http://mypy-lang.org/) for this purpose will be evaluated.
- It could also be worthwhile to have the CI service with schema checkers for the XML-based DSL's such as Package XML, Launch XML, and Xacro. Existing schema checkers exist for XML and YAML-based formats. Furthermore, It could be worthwhile to look into code scanners for declarative modelling languages (because, even though almost three out of four bugs occur in imperative programming languages, almost half are fixed in declarative modelling

languages).

About half of the bugs we studied could potentially be detected automatically using static checking technology that is within reach. There are a number of concrete recommendations regarding the development and use of code scanners and analyzers:

- About one in ten bugs are resource management errors. This bug category could benefit from recent development in bug finding targeting precisely such errors. There is experience in the ROSIN consortium in building these kinds of checkers.
- It seems that the developers working with drivers (which are not ROS core developers) need more help with finding and fixing bugs. ROSIN should prioritize this. The tools could be primarily tuned and evaluated for drivers.
- Most bugs occur in and are fixed in the following languages: C++, Python, CMake, Package XML, Launch XML, and Xacro. It could be worthwhile to focus the code scanning efforts on these six languages along with the interplay between them. Especially tracking external interfaces and cross-file references, and reporting dangling references statically might help to fix a large part of the bugs. It could be worthwhile to investigate cross-language dependency closure bugs that occur because of inconsistencies between C++, Python, CMake, Package XML, and Launch XML.

Finally, there are some more challenging and robotics-specific errors to consider. These bugs appear to require functional correctness specifications so they cannot be addressed with pure code scanning.

ROS and ROS2

As discussed in Chapter 3, the development of ROS2 is a normal evolutionary development step in the life cycle of a well-established and appreciated software product: As in the life cycle of other software products (Dittrich 2014), the development of new features and changes both with respect to developing implementation technologies and requirements, caused by evolution of the domain, challenge the technical design and lead eventually to the decision to start the endeavour of an architectural and technical redesign.

For the near future, we expect that ROS and ROS2 will co-exist, opening up for the owners of non-core packages and application developers to choose whether to port their software to ROS2 or whether to remain with the original ROS. In addition to the renewal of the technical base, new quality assurance and control tools have been deployed. ROS2 already now uses linters as part of its continuous integration environment.

We therefore recommend to follow the developments in the developing ROS2 sub-community, and where possible, to take the new technical base into account. Core ROS developers are involved in ROS2 development as well; involving them in the community development part of WP3 will help to adapt the applicable results of ROSIN in ROS2 in parallel with their adoption in ROS.

References

Dittrich, Y. (2014). Software engineering beyond the project–Sustaining software ecosystems. *Information and Software Technology*, *56*(11), 1436-1456.

Dittrich, Y. (2016). What does it mean to use a method? Towards a practice theory for software engineering. *Information and Software Technology*, 70, 220-231.

Dobbins, J. A., & Buck, R. D. (1983). Software Quality Control and Assurance. *IFAC Proceedings Volumes*, *16*(18), 127-136.

Wasowski, A. Pull request 7 to HAROS git repository, retrieved from http://github.com/git-afsantos/haros/pull/7

Wasowski, A. Pull request 8 to HAROS git repository, retrieved from http://github.com/git-afsantos/haros/pull/8

mypy description. Retrieved from http://mypy-lang.org/

Open Source Robotics Foundation. Retrieved from https://www.osrfoundation.org/

Robson, C. (2011). Real world research 3 rd Ed. *UK: Wiley*.

REP-199: Specification for TurtleBot Compatible Platforms. Retrieved from http://www.ros.org/reps/rep-0119.html

ROS Answers. Retrieved from https://answers.ros.org/question/217475

ROS History. Retrieved from http://www.ros.org/history/

ROS-Industrial GitHub Repository. Retrieved from https://github.com/ros-industrial/

Private ROS build farm. Retrieved from http://rosin-build-master.3me.tudelft.nl

Roslint Package description. Retrieved from http://wiki.ros.org/roslint

ROS.org Wiki. Retrieved from http://wiki.ros.org/

ROS REPS. Retrieved from http://www.ros.org/reps/

Santos, A., Cunha, A., Macedo, N., and Lourenco, C. (2016). A framework for quality assessment of ROS repositories. In 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 4491-4496. IEEE.

Scacchi, W. (2002). Understanding the requirements for developing open source software systems. *IEE Proceedings-Software*, *149*(1), 24-39.

Sigfridsson, A. (2010). The purposeful adaptation of practice: an empirical study of distributed software development.

Willow Garage website. Retrieved from http://www.willowgarage.com/

Appendix A
ROS Quality Issues - Additional data

This appendix contains additional background information collected during the investigation of historical bugs in the ROS and ROS-I packages. It also has information about the process, and the participants.

## Task:

51 out of 177 bugs mention a task:

#	%	Task
126	71%	*unspecified*
16	9%	Simulation
8	5%	Manipulation
6	3%	Motion
6	3%	Diagnostics

(Additionally, 1 SLAM[2] and 1 visualization, 2 Auto Docking, 2 Communication, 2 Planning, 2 Teleoperation, 2 Vision, and 3 Testing.)

Number of files involved in a fix:

167 out of 177 bugs in the collection are fixed:

#	%	#Files fixed
97	55%	Fixes 1 file
32	18%	Fixes 2 files
11	6%	Fixes 3 files
6	3%	Fixes 4 files
12	7%	Fixes 5-10 files
7	4%	Fixes 11+ files
10	6%	*unspecified*

OBSERVATION: Four out of five bugs are fixed in three files or less.

Outliers: bug #65e7ee6 fixes 29 files and bug #3236783 fixed 158 files (because it entailed a "complete update of a driver package"). Fixes in multiple files are common as ROS is often distributed with lots of different packages with dependencies among them.

[2] SLAM (Simultaneous Localization And Mapping).

People:

Here is an overview of the persons involved in harvesting the bugs:

Name	Email	Partner	Organization
Jon Tjerngren	jon.tjerngren@se.abb.com	ABB	ABB, Sweden
Chris Timperley	ct584@york.ac.uk	CMU	Carnegie Mellon University, USA
Patrick Wiesen	wiesen@fh-aachen.de	FHA	Aachen Uni. of Applied Sciences, Germany
Jonathan Hechtbauer	jonathan.hechtbauer@ipa.fraunhofer.de	FHG	Fraunhofer Inst. (IPA), Stuttgart, Germany
Andrzej Wasowski	wasowski@itu.dk	ITU	IT University of Copenhagen, Denmark
Claus Brabrand	brabrand@itu.dk	ITU	IT University of Copenhagen, Denmark
Jon Azpiazu Lozano	jon.azpiazu@tecnalia.com	TEC	Tecnalia, Spain
André Santos	contact.andre.santos@gmail.com	UMINHO	University of Minho, Braga, Portugal

Subject systems:

An overview of the systems analysed:

bugs	system	category	size	#issues	URL
57	Kobuki	ROS application	3.2 MB	325	http://github.com/yujinrobot/kobuki
40	Mavros	ROS application	1.7 MB	623	http://github.com/mavlink/mavros
25	Universal Robot	ROS-Industrial	24.0 MB	158	http://github.com/ros-industrial/universal_robot
22	Motoman	ROS-Industrial	24.0 MB	78	http://github.com/ros-industrial/motoman
12	Turtlebot	ROS application	17.0 MB	170	http://github.com/turtlebot/turtlebot
11	Care-O-bot	ROS application	39.0 MB	182	http://github.com/ipa320[3]
10	*Confidential*	*N/A*	*N/A*	*N/A*	ABB & TEC confidential systems

[3] Comprises: cob_robots (58 issues), cob_control (44i), cob_driver (48i), cob_android (7i), and cob_command_tools (25i).

Appendix B

ROS Quality Issues - Bug Taxonomy Description

This document explains how to systematically capture and record a bug. It details what information should be captured and in what format. Note that it will not always be possible to perfectly fill in all fields for all bugs, but please try to fill in as many fields as realistically possible. In cases where the information is not available (or you cannot obtain it without spending excessive amounts of time), please leave the field empty. In cases where the information is not applicable, please put "N/A" (Not Applicable).

title:

Concise textual one-line summary of the bug intended for domain non-experts (typically 10 words or less).

description:

Textual description of the bug (typically 5-10 lines). Try to write this so that a domain non-expert software developer will be able to understand what this bug is about. This often involves writing about the cause of the bug (what was the underlying problem and what needed fixing) as well as the effect of the bug (how did the bug manifest itself), including whatever else is relevant in order to have a rough idea of the bug. (The text is indented five spaces.)

classification:

The Common Weakness Enumeration (CWE) is a community-developed list of common software security weaknesses.[4] It is essentially a taxonomy of errors (weaknesses). We will try to use this taxonomy to classify the errors in terms of their effect; e.g.: "CWE-476: NULL Pointer Dereference". Since CWE is primarily concerned with security, it is sometimes necessary to add your own new categories (without a number identifier) not covered by the taxonomy; e.g., dependency errors, type errors, and robotics-specific weaknesses. In many cases, multiple categories are applicable. You should then pick the most appropriate category.

keywords:

This is a "|"-separated list of keywords; e.g.,: "xacro | gazebo | urdf | driver".

system:

This is an identifier for the system at hand (e.g., kobuki, motoman, mavros). If it's a *confidential* system, then just put the value "confidential".

severity:

This is a subjective assessment of the severity of the bug in terms of its overall effect. Its value should be one of the following options: error, warning, convention-violation, bad-smell, or minor-

[4] https://cwe.mitre.org/

issue.

Most problems will be *errors*. All categories refer to code, so convention-violation is a coding convention violation (similarly, for *bad-smells*).

links:

A list of links to additional information relevant for understanding the bug.

BUG

The following fields relate to the manifestation of the bug itself which is also sometimes referred to as the effect of the bug (as opposed to the underlying cause of the bug involving how it was fixed which is covered later; cf. "FIX" below).

phase:

In what phase of the software life cycle did the error occur? Please pick the most appropriate among the following predefined options (each is accompanied by a elaborative description):

- build-time (for errors that are reported by the build/make tools that compose the source code *prior* to compilation)
- compile-time (for errors that are reported by the compiler itself)
- deployment-time (for errors that occur *after* compilation and *before* the program is run; often when some deployment or installation scripts are run. This also includes "installation-time")?
- runtime-initialization (for errors that occur when the software is run and being initialized). [Note that this including both "virtual" simulation and "real" hardware.]
- runtime-operation (for errors that occur when the software is run on normal operation *after* having been initialized). [Note that this including both "virtual" simulation and "real" hardware.]

specificity:

This is an open textual field about how this bug generalizes; whether it is a general software issue applicable to many or most software projects or whether it is a general robotics issue or something completely specific applicable only to the ROS or ROS-I projects. Please pick the most appropriate among the following options:

- general-issue (similar issues are to be expected in many or most software projects). Heuristic: An issue is general if a single tool solving this issue could plausibly solve it for a broad class of software projects.
- robotics-specific (similar issues are to be expected in other robotics platforms).
- ROS-specific (quite idiosyncratic as to how ROS or ROS-I is built). Heuristic: If a special ROS tool is needed to solve this issue, then it is probably ROS-specific.
- application-specific (specific to an application).

architectural location:

Where did the bug occur? Please pick one of the following two options:

- application-specific code (did the bug occur in an application)
- platform code (did the bug occur in ROS or ROS-I platform itself)

application:

In case you answered "application-specific" above, please specify what the application was:

- textual description of robot application (e.g., "picker robot"); "N/A" for platform code.

Leave empty if the application is unknown or classified.

tasks:

Please pick the task in which the bug occurred:

- perception (including Object Detection, Collision Detection)
- localization
- planning (including path, trajectory, collision)
- manipulation (including actuation and motion)
- human-robot interaction
- simulation (including visualization)
- diagnostics
- slam (simultaneous localization and mapping)
- N/A

subsystem:

In which subsystem (part of the organizational/architectural structure of the project) did the bug occur:

- motion
- driver
- core component
- generic task component (e.g, a planner)
- specific application component (e.g, auto docking)
- ...

packages:

This is a "|"-separated list of packages involved. Each entry should specify the *project*, the *repository*, and the *package* involved; e.g.:

ros-industrial/universal_robot/ur_bringup | ros-industrial/universal_robot/ur_description

languages:

A "|"-separated list of the languages involved in the manifestation of the bug. "N/A" if the error is not explicitly reported by the language infrastructure. Let's also try to normalize the languages: python, cmake, C++, package.xml, launch XML, msg, srv, xacro, urdf. Avoid a generic XML tag (all files in ROS have some known schema, and let's try to narrow it down when writing). Also the language should be N/A if the bug is not reported by the language infrastructure (so if the error is in package.xml but a C compiler fails then the language is "C" here, not package.xml. The latter is listed under the fix. If the error is not reported by a language infrastructure, but for instance wrong behavior is discovered in simulation, then do not put a language in). For this reason it should be fairly unusual to have more than one language listed here.

detected-by:

How was the bug detected? Please pick the most appropriate among the following predefined list:

- build system (the bug was detected by the build system *prior* to compilation)
- compiler (the bug was detected by the compiler itself)
- code scanning tool (the bug was detected by Q&A code scanning tools)
- assertions (the bug was detected by assertions statements in the code; e.g., "assert(x>0);")
- runtime detection (the bug was detected at runtime; e.g., an exception was thrown)
- runtime crash (the bug caused the system to crash at runtime and cease functioning)
- testing violation (the bug was detected by violating a test case)
- developer (the bug was detected by a developer of the system)

- user (the bug was detected by a user of the system)

reported-by:

How was the bug reported? Please pick the most appropriate among the following predefined list:

- guest user (the bug was reported by a guest user)
- contributor (the bug was reported by a developer/contributor)
- member developer (the bug was reported by a member developer)
- automatic (the bug was reported by an automatic test/analyze QA service, incl CI)
- unreported (the bug was not reported; e.g., because it was fixed directly without any reports)

issue:

This is a URI reference to an issue tracker entry (e.g., GitHub or StackOverflow). This will obviously be empty if the bug is unreported. But even for some reported bugs there will be no issue created (bugs can be reported through other channels). Add reports through other channels under links (above).

time-reported:

The time the bug was reported (“unreported” if it was not reported); e.g.: “2017-12-31 (23:59)”.

reproducibility:

- {always,sometimes,rare}

trace:

For runtime bugs, this is a trace (call stack/sequence of function calls) to the bug. “N/A” for bugs not involving runtime (e.g., type errors or build-system bugs).

FIX

The following fields relate to the *fixing* of the bug which is also sometimes referred to as the *cause* of the bug (as opposed to the underlying *effect* of the bug which was covered above under “BUG”).

repository:

URI reference to repository where the bug was fixed.

hash:

The hexadecimal hash code of the commit that fixed the bug. (With pull requests there are two commits; one is in the local branch from which the pull request is created, the other is the actual merging commit on the mainline; we should try to report the latter as this is what is more easily accessible long term; other branches may be deleted.)

pull-request:

URI for pull request that fixed the bug.

fix-in:

A list of the files that were updated when fixing the bug.

languages:

A “|”-separated list of the languages involved in fixing the bug. (See list of conventions for naming languages under bug/language.)

time:

The time the bug was fixed (“unfixed” if it was not fixed); e.g.: “2017-12-31 (23:59)”.

Appendix C

Jenkins and industrial_ci: Quick manual

Jenkins and industrial_ci: Quick manual

Iñigo Martínez Jon Azpiazu

July 13, 2017

Contents

1 Overview

Software quality has been for a long time a goal for software engineering. It reflects how well a software complies with or conforms to a given requirements or specifications.

In order to help with this task **Continuos Integration (CI)** had been conceived as a practice of merging frequently all developer working copies to a shared mainline. The last years several **CI** tools have flourished: Jenkins, Travis CI, Buildbot.

These tools help in *automate the building process*, *make the build self-testing* and also *automate the deployment.*

In our use case, we are focused on creating a coding environment for **ROS developers** in companies where a *private infrastructure* is needed. GitLab does offer a community edition which helps on setting up a private repositories, being the natural option with respect to other best known options like GitHub. Although *GitHub* offers a plethora of services, is not possible to set up a private service. *Jenkins* is also a good option as a *CI* service, as setting up a private service is possible.

industrial_ci fits perfectly when trying to get a higher level of software quality when working with *ROS*, as it contains *CI* configuration that any *ROS-powered* packages can commonly use.

2 Introduction

This is the proposal for a quick reference manual for working with Jenkins + industrial_ci.

The objectives of the manual are:

1. a quick reference for the administrator / sysadmin
2. a quick reference for a developer that wants to put his package into Jenkins + industrial_ci
3. material for the ROSIN deliverable

3 For installer / administration

3.1 Installing Jenkins

The following reference is based on *Ubuntu* operative system.

In order to install jenkins we must add the official repository. To use the repository, we must add the proper key:

```
wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key
    add -
```

Then we must add the following repository entry to sources:

```
echo "deb https://pkg.jenkins.io/debian binary/" > /etc/apt/sources.list.
    d/jenkins.list
```

Last versions of *Jenkins* are working over *Java 8*. Although it is supported on later *Ubuntu* versions and the following steps are not necessary, there is no official support for it in *Ubuntu 14.04*, we will add an external repository for it:

```
sudo add-apt-repository ppa:openjdk-r/ppa
sudo apt-get update
sudo apt-get install openjdk-8-jdk
sudo update-alternatives --config java
sudo update-alternatives --config javac
```

We must update the local package index and then finally install *Jenkins*:

```
sudo apt-get update
sudo apt-get install jenkins
```

The installation procedure will start *Jenkins*. We can confirm that this is the case by querying its status:

```
service jenkins status
```

We should see a message similar to the following one:

```
Jenkins Automation Server is running with the pid 46427
```

We can open our web browser and use the following address to access our new *Jenkins* instance.

```
http://server-host:8080
```

3.2 Jenkins configuration

The first time we access our *Jenkins* instance, it will execute an authentication procedure. It will ask us to introduce the key written in the following file:

```
cat /var/lib/jenkins/secrets/initialAdminPassword
947526feafef4006a1ee478f5fc7f6d3
```

If we do this properly, we can follow a wizard to configure *Jenkins* properly.

3.3 Jenkins slaves

Jenkins does support distributed builds, where the workload of building projects are delegated to *slave* nodes. We will take advantage of this feature.

First of all, we have to setup a proper user and directory to host *Jenkins* data in the slaves.

```
adduser --system --group --home=/var/lib/jenkins-slave --no-create-home
    --disabled-password --quiet --shell /bin/bash jenkins-slave
```

This command does add a new *user* and *group* to our *OS* named **jenkins-slave**.

```
install -d -o jenkins-slave -g jenkins-slave /var/lib/jenkins-slave
install -d -m 700 -o jenkins-slave -g jenkins-slave /var/lib/jenkins-
    slave/.ssh
```

These commands will copy files and set attributes to *jenkins-slave* user in order to be used properly as a *Jenkins slave*.

Jenkins slaves use SSH as the network protocol. We will copy the *SSH* key from the master instance to the authorized keys file.

```
cat id_rsa.pub >> /var/lib/jenkins-slave/.ssh/authorized_keys
```

Once the node is properly setup, we will add this information to the *Jenkins* instance. We will move to the **Manage Nodes** option in **Manage Jenkins**, and choose **New Node** option.

The **Remote root directory** must be: */var/lib/jenkins-slave.*

Change the **Launch method** to *Launch slave agents via SSH* and choose the proper **Host** name and **Credentials**.

3.4 Jenkins plugins

Jenkins is an extensible *Continuous Integration* tool. Its features can be extended by using plugins. Although, *Jenkins* comes with a lot of extensions by default, the following list of plugins should also be installed:

- **Email Extension Plugin** which allows notifications by using Email.
- **embeddable-build-status** that provides a graphical icon which can be embedded providing build status.
- **Git plugin** to allow *Git* repositories.
- **GitLab Authentication plugin** which helps with the authentication step when accessing a *Git* repository hosted in a private *GitLab* instance.
- **SSH Slaves plugin** for integration with slaves using *SSH.*
- **Yaml Axis plugin** for matrix project axis creation and exclusion plugin using *yaml* file.

Once these plugins are installed, a proper environment will be available to create *ROS* software with improved quality.

3.5 Authentication with GitLab

GitLab is a *Git* code repository platform available as free software. It can be flawlessly integrated with *Jenkins* by using the *GitLab* plugin and setting it up in the following way:

- Move to *GitLab* web interface.
- Click on the *configuration menu* and then **Settings**.
- Go to **Account** menu. You will see a **Private token** you can use, or click on **Reset private token** to generate a new one. Copy it.
- Move to *Jenkins* settings menu.
- Go to **GitLab Account Settings**
- Enter *GitLab's URL* on the **Endpoint URL**.
- Copy the *private token* we copied before into the **Private token** field.

Jenkins will be able to authenticate properly when accessing repositories under the chosen *URL.* The building step will happen without any user interaction, which will allow to test the last changes in the repositories and notify developers upon any broken changes.

4 For developers

The steps to have a ROS package tested by *industrial_ci* are as follows:

1. Make sure you have a properly defined ROS package that compiles locally.
2. Specify the dependencies of the ROS package (see 4.1).
3. If required, generate the file with the build matrix and add it to the repository (see 4.2).
4. Create a Jenkins job that runs the script and performs the tests (see 4.3)
5. Optionally configure Jenkins to run when the repository is updated (see 3.5) and show the status in GitLab (see 4.5).

4.1 Specifying dependencies

4.1.1 ROS dependencies

If the package dependencies are ROS packages or either libraries *supported* by ROS, you can specify the dependencies in the *"package.xml"* file. Those dependencies will be automatically installed using rosdep. *rosdep* will then use a mapping between the names of the libraries as specified by ROS, and the names of the libraries as installed by the specific distribution. This mapping is stored in YAML format.

4.1.2 Private ROS packages

This case deals with:

1. The dependency is not available to be installed using *rosdep* (otherwise refer to Section 4.1.1)
2. The dependency is a ROS package
3. The dependency is available as source code
4. The dependency is reachable as an URL (e.g. git, subversion)

If this is the case, create a *".rosinstall"* file in the package's root folder, and specify the dependency name, route and optionally version in the following manner:

```
- git:
    local-name: manipulation_dreambed
    uri: 'https://github.com/kth-ros-pkg/manipulation_dreambed.git'
    version: master
- svn:
    local-name: dummy_robot_info
    uri: http://svn.dummy.com/scm/svn/rospkg/indigo/trunk/src/
        dummy_robot_info
```

The ".rosinstall" file can be created manually following the format defined in REP-126 and rosinstall documentation, or it can be generated using some of the available tools to manage the catkin workspaces such as wstool.

To handle this type of dependency, the variable *UPSTREAM_WORKSPACE* must be set to "file" in industrial_ci. This can be done either in the YAML file for the multi-axis project (Section 4.2), or in the bash script executed (Section 4.3).

4.1.3 Other dependencies

This section deals with dependencies not covered by Sections 4.1.1 or 4.1.2.

To install a dependency, that can be either binary or available as source code, we can define a bash script that will be run just before the build is started. This script can either install binary dependencies (e.g. using apt-get), or download code and compile it. The script to be run is specified to *industrial_ci* by using the *BEFORE_SCRIPT* environment variable.

Example installing CUDA:

```
#!/bin/bash

cd /tmp
wget http://developer.download.nvidia.com/compute/cuda/repos/
    ubuntu1404/x86_64/cuda-repo-ubuntu1404_8.0.61-1_amd64.deb
sudo dpkg -i cuda-repo-ubuntu1404_8.0.61-1_amd64.deb
sudo apt-get update
sudo apt-get -qq install --no-install-recommends -y cuda-toolkit-8-0
export PATH=/usr/local/cuda-8.0/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-8.0/lib64${LD_LIBRARY_PATH:+:
    ${LD_LIBRARY_PATH}}
echo 'export PATH=/usr/local/cuda-8.0/bin${PATH:+:${PATH}}' >> /root
    /.bashrc
echo 'export LD_LIBRARY_PATH=/usr/local/cuda-8.0/lib64${
    LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}' >> /root/.bashrc
```

Example upgrading CMake from sources:

```
#!/bin/bash

cd /tmp
wget https://cmake.org/files/v3.5/cmake-3.5.0.tar.gz
tar xvfz cmake-3.5.0.tar.gz > /dev/null
cd cmake-3.5.0
./bootstrap
make
sudo make install
```

4.2 Specifying the build matrix

The build matrix plugin is intended to define several different configurations for a build job, so that when the build job is started, Jenkins launches one job for each configuration. A typical scenario of interest that will be described here is to launch build jobs for different ROS distributions.

Note that in order to be able to use a multi-axis build job, the Jenkins job must be defined as a "Multi-configuration project" (see Section 4.3 for details).

In order to use the multi-axis functionality to build for several ROS distributions, a YAML file is created at the root of the repository. The file can be typically named ".jenkins_axis.yaml", although the name is not relevant; just make sure that it is correctly referenced in the Jenkins job (see Section 4.3):

```
ROS_DISTRO:
   - indigo
   - jade
   - kinetic
```

If required, more variables can be defined here. As an example referred in Section 4.1.2

```
UPSTREAM_WORKSPACE: file
```

4.3 Setup the Jenkins job

These are the steps to create a Jenkins job:

- From the Jenkins dashboard (in the web interface), click on "New item"
- Enter a unique name for the item, which is descriptive enough
- Select "Multi-configuration project" and click "Ok". The options in the new menu will depend upon the installed plugins so we will proceed to fill the details
- If old jobs should be discarded and only a set of them maintained, "Discard old builds" should be checked and a strategy that fits the needs chosen
- In the "Source Code Management" section, select "Git" and enter the repository URL. Make sure the proper credentials to access the repository are selected (see Section 3.5 for details on how to set the credentials). Select also the branch or branches to be built.
- The strategy regarding what triggers the build is also important. It can be enabled in the "Build Triggers" section by checking "Poll SCM" and the text box should also be filled. An example would be "H H(0-7) * * *" which will check the repository between "12:00 AM" and "7:59 AM" and will run the build if any change is detected.
- In the "Configuration Matrix" section, click on the "Add axis" drop-down, select the "Yaml axis" option
- In the "Axis yaml file" field, enter ".jenkins_axis.yaml" or the filename specified in 4.2
- In the "Axis name" field, enter "ROS_DISTRO"

- In the “Build” section, click the “Add build step” drop-down, and select “Execute shell”
- Enter the following code in the “Command” box:

```
git clone https://github.com/ros-industrial/industrial_ci.git .
    ci_config
bash -xe .ci_config/ci.sh
```

  Additional variables can be defined in the shell script; as an example referred in Section 4.1.2:

  `export UPSTREAM_WORKSPACE=file`

- (Optional) In the “Post-build Actions” section, click the “Add post-build action” drop-down, and select “E-mail notification”. Select the relevant options.
- (Recommended) In the “Post-build Actions” section, click the “Add post-build action” drop-down, and select “Delete workspace when build is done”, which will delete the workspace used by the build freeing its storage space.

4.4 Integration with Gitlab

There are two main ways of triggering a Jenkins build job. First to run the build job with a certain periodicity, which can be configured when setting the Jenkins job (as showed in Section 4.3. Second, to be triggered when certain events at the repository happen (typically a push). Here we will tackle the second case, using Gitlab as an example.

The easiest way to integrate Gitlab or other SCMs (source code managers) is by using web-hooks. For each job, there are certain functionalities that can be accessed using web-hooks. For example:

`http://<server-name>:<port>/job/<job_name>/polling`

Accessing that URL will trigger a polling of the code in the SCM, and a build job in case the code has changed since the last run.

To enable the web-hook in Gitlab, enter the Project and click on “Settings” → “Integration”, and in the URL field enter the address formatted following the example above.

Troubleshoot

If the web-hook does not work, Jenkins’ protection to CSRF attacks (Cross site request forgery) shall be disabled. To do so, get into the Jenkins dashboard, and navigate into “Manage Jenkins” → “Configure Global Security”, and make sure the option “Prevent Cross Site Request Forgery exploits” is disabled.

4.5 Showing status in Gitlab

When navigating different SCMs such as GitHub or GitLab, it is common to include files with documentation information, that are rendered and presented to the user. A common feature is to include information about the build status in such files. GitLab supports Markdown and AsciiDoc.

To include the badge information using Markdown format, assuming that the package is well documented and contains a "readme.md" file, we can add the building status within it by adding the following lines:

```
[![](http://<server>:<port>/buildStatus/icon?job=<job_name>)](http://<
    server>:<port>/job/<job_name>/)
```

To include the badge information using AsciiDoc format, create a file with the ".adoc" extension, and use the following format:

```
image:http://<server>:<port>/buildStatus/icon?job=<job_name>[http://<
    server>:<port>/buildStatus/icon?job=<job_name>]
```

4.6 Troubleshooting

4.6.1 Running industrial_ci locally

Before pushing the code into the repository, it might be useful to first run the CI locally, to make sure that not trivial failure cases are committed. All the industrial_ci process (except for the multi-axis) can be run locally by following the instructions from the official documentation.

4.6.2 Inspecting the docker container

After each build job, the Docker container created to run the build is destroyed - remember that in order to assure reproducibility, the build jobs are run starting from an empty Docker image in which the dependencies are installed. In order to debug certain problems, it might be interesting to keep the Docker container and inspect it.

Currently industrial_ci does not support keeping the Docker containers after the build. The instructions here assume that the user is running industrial_ci locally (see Section 4.6.1 for details on how to do that). The way to keep the container is to edit the industrial_ci code, the file named "docker.sh" and comment out the command around the line 91:

```
docker rm "$cid" > /dev/null
```

After that, re-run the industrial_ci script, e.g.:

```
$ rosrun industrial_ci run_ci ROS_DISTRO=indigo UPSTREAM_WORKSPACE=file
```

Now the container created to run the build job should not have been removed. Check its ID:

```
$ docker ps -a
CONTAINER ID        IMAGE                 COMMAND                CREATED
              STATUS                     PORTS              NAMES
0d7e0ac3cd96        industrial-ci/trusty "/bin/bash /root/i..." About a
    minute ago Exited (1) About a minute ago
    kind_mcnulty
```

In order to change the container's entry point, and start and interactive session, the Docker container must be first committed into a Docker image:

```
docker commit $CONTAINER_ID test_image
```

And then start the image with a different entry point:

```
docker run -it test_image /bin/bash‘
```

You should get now a bash prompt that can be used to inspect and debug the problems.