Mining Repositories for Verification and Validation of ROS-based apps

Current Version: 1.1.0
Current Date: 20-07-2023

Investigators: Matei Schiopu, Ricardo Caldas, Thorsten Berger, Patrizio Pelliccione

The code used to mine the repositories is available online: https://github.com/SchiopuMatei/Chalmers-ROS-RM-FT-Datamining/blob/master/README.md.

1 Introduction

Charting ROS project architecture from a software engineering perspective is a task that requires both a thorough study of the state-of-the-art publications, and a broad view of the current methods and needs of ROS developer teams. A strong connection to the current project landscape is needed to define guidelines that may bring value to the field as a whole. In this paper we will showcase our approach to tackling this and establishing a link to the ROS ecosystem, as it is represented by the communities who actively develop it, using data mining.

The first part of our approach involves creating a dataset of projects by pooling public ROS projects from across GitHub, Bitbucket, and GitLab, and curating them to remove noise and low-relevance items, we establish the area from which we will extract exemplars for our guidelines. The principal issue in this undertaking is the vastness of the material we are trying to cover. Attempting to investigate each project individually by examining each file would take inordinate amounts of time, and require us to restrict the scope to a handful of projects which may not be representative of the actual ongoing trends in ROS development. Thus, we employ further data mining techniques to cover as many projects as possible and give each of them a chance to assert their trends within a certain development topic. We can search and extract large batches of code snapshots from the entire dataset by extracting keywords from the guideline topics from their description and methods. This enables us to quickly find trends in library and tool usage within the ROS environment. Once a usage trend is detected, the code of the project undertakes further examination and is then subject to a peer review process to determine the validity of the results and under which guidelines they may be classified.

2 Methodology

This section describes how we derived exemplars for the guidelines from online repositories in GitHub. Figure 1 depicts the process of deriving examples of patterns, tools, and techniques for facilitating and performing runtime verification and field-based testing of ROS-based applications.

2.1 Phase 1: Mining repositories

The mining project was initialized starting from the 'Mining guidelines for architecting robotics software' by Malavolta's replication package[1]. The main element used from the packages was the rosmap[1] implementation, a tool designed for dependency mapping, which allowed the gathering of an exhaustive list of all public ROS projects available on Github, as well as Bitbucket and Gitlab.

Python scripts from the replication package were implemented and modified. The 'explorer.py' script combines the separate sources into one monolithic link basket while also filtering out forked repos, duplicates, repos with low amounts of commits (less than 100), repos with low star count (less than 2 stars), and demo projects which have low relevance and no exposed primary source code, excluding any projects using the word 'demo' in the description or name of the project.

The final part of the script array is 'cloner.py' which takes the monolithic curated json file containing pointers to the projects, and the result of the previous script, and downloads them locally to a research server provided by RUB. The local setup allows for quick targeted searches inside the source files of mature projects, with a high degree of search-term flexibility, and ensures replication in the long term,

[1] https://github.com/jr-robotics/rosmap

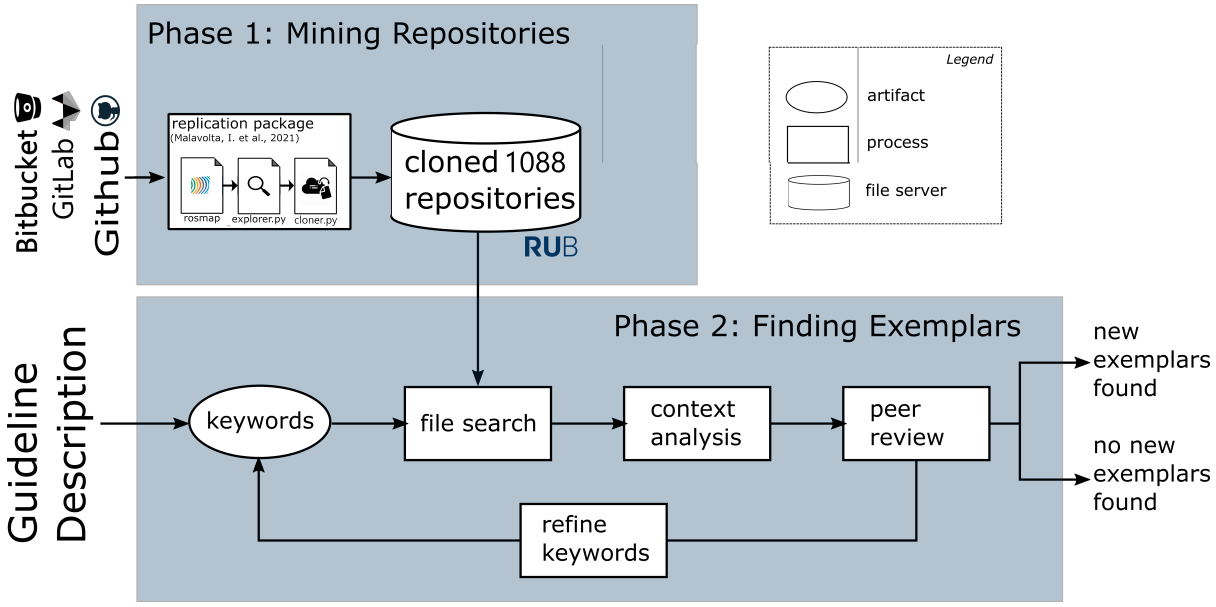

Figure 1: Overview of the activities to gather exemplars for the guidelines

solving the issues of projects transitioning from open-source to private repositories in the future or any repository deletions or migrations.

Table 1: Inclusion (I) and Exclusion (E) criteria to preliminary repository selection

ID	Criteria description
I1	Contains ROS python launch files for a standalone application or framework
I2	Contains code fit for deployment on a ROS system
I3	Contains test artifacts included in main code or as part of testing stack
I4	Contains quality assurance and mapping instruments for ROS systems
E1	Contains only simulations or simulation plugins (Webots, Gazebo)
E2	Contains course material or is part of a tutorial project
E3	Contains clearly deprecated or incomplete, nonfunctional code
E4	Contains only a demo of an unfinished concept or experimental code
E5	Contains clearly duplicate code of another included repository
E6	Contains only a reference to another library or repository
E7	Contains no form of documentation or instructions
E8	Contains descriptions and comments not written in English

Table 2: Inclusion (I) and Exclusion (E) criteria to peer review

ID	Criteria description
I1	Adequately fits the scope of a given guideline
I2	Contains scientific documentation like articles or papers supporting a guideline
E1	Contains only tangentially relevant content
E2	Contains inconclusive or otherwise vague documentation
E3	Project is found to have lacked maintenance for more than 10 years

Phase 1 results in a set of 938 repositories downloaded to a server in RUB where we can further query and analyse each repository.

2.2 Phase 2: Finding Exemplars

The curated repositories are then inspected by the research team. For the manual search, the UNIX 'grep' utility allowed for the efficient scanning of large amounts of files. The process would start with analyzing each guideline and research any provided exemplars to retrieve pointers and markers which indicate the exemplar is being utilized in repository code, and then search for these signs through the mass of downloaded repos. In addition, especially for guidelines missing exemplars, we would also search for exemplars by looking up keywords relevant to the guideline and examining code comments within the repos (i.e. LLVM LibFuzzer, found while looking for fuzzing mechanisms, used by PX4_____Firmware, a large opensource autopilot project).

Utilizing 'grep', researchers would map out usages, listing out which and how many projects are making use of an exemplar, and further inspecting the lines of code for matched results in order to ensure relevance and filter out false positives manually. The same code line inspecting process was used for discovering new exemplars utilizing keywords from interest areas and examining peripheral code from already-established exemplars.

The search method consisted of two steps: File Search and Code Examination. In order to accelerate the process of reviewing the code of the 938 repositories, an initial probe for usages is carried out during the File Search.
File search command example:

```
$ grep -Ril "keyword" --include=*.{cpp,c,py} --binary-files=without-match
```

The '-R' setting reads all files under each directory, recursively, following any symbolic links. The '-i' setting ignores case. The '-l' option swaps the output from matched text to the full paths of files where matches were found.

In order to speed up the search and target it towards more relevant areas we utilize'–include=*.cpp,c,py', which restricts it to certain file types, and '–binary-files=without-match', which ignores strictly binary files such as images, .tar archives, videos and other media. The processing of binary files provides no relevant content due to their non-human-readable nature, and slows down the search heavily due to the large size of the files.

For the case of domain-specific languages (DSLs), the use of exclusion filters allowed us to filter common programming language elements and focus on exotic examples with DSL implementations. (i.e. RML utilized by ROSMonitoring in .rml files, RCL ROSContractLanguage)

Example of an exclusion filter: –exclude=*.cpp,c,py,h,hpp
Code examination command:

```
$ grep -Ri -A 10 -B 10 "keyword" --include=*.{cpp,c,py}
\\ --binary-files=without-match
```

The '-A x' and '-B y' options convert the output from single matched lines to a contextual snapshot of the code, displaying 'x' lines of code above and 'y' lines of code below the match.

Following a successful file search, the matched files undergo a preliminary examination in the 'Code examination' phase. By viewing the matched lines, the file structure and projects to which they belong, and code snippets around the matched area, the examining researchers can discard any false positives and irrelevant findings according to the given criteria. The remaining projects can then be subject to deeper code examination, which is then brought up in peer-review sessions where the research team determines if a viable exemplar has been discovered for the corresponding guideline.

3 Results

We have inspected 15 guideline descriptions and found 29 new exemplars comprising code repositories, language specifications and ROS native and external libraries.

Official libraries come packaged with the ROS and ROS2 suite and are universally well-documented and well-maintained, with some receiving several major updates per year and multiple commits monthly. They also feature some of the highest usage and appearance rates within the scanned projects, being widely used. However, this is not always the case, as effective and sophisticated methods and libraries go entirely unused in many projects where their presence would have greatly benefited in improving the quality of the development process or the end product itself.

Unofficial libraries vary widely in quality and maintenance. Within the scope of the study we have taken quality control measures in the form of our acceptance criteria to ensure only active and fresh

exemplars are being considered. Even so, they are mostly surpassed by the official libraries when it comes to activity.

This is also a direct effect of the open-source nature of ROS. When an unofficial extension is developed and gains popularity, it is quickly endorsed and then assimilated into the official architecture, getting assigned constant maintainers even if the initial development team becomes inactive at some point in the future. Unofficial projects provide the basis for open-source development and are the main contributors to the furthering of the ROS toolkit, besides maintainers. They are often highly experimental and try to resolve issues within the topic or explore new development paradigms.

Standalone tools differ from the other categories in that they do not directly interface with ROS project code, and instead provide services by attaching externally. The interfacing can take the form of node subscribers, file analyzers, server API, different combinations of these methods, and many more. The usual objective of these exemplars is to provide value to the development environment in the form of debugging, visualization, and other methods. They ultimately provide indispensable aid to developer teams by speeding up the bugfixing process and giving accessible insights into the project as a whole, from what might have otherwise been gigabytes of unreadable logs and recordings.

Guideline	Keywords	Exemplars
C1. Identify timing constraints	”Autoware Perf”, ”AutowarePerf”, ”seams18”, ”seams” ”rtamt4ros” ”throttling” ”throttle”	ros-tooling/topic tools
C2. Identify security and privacy constraints	”SROS” ”ROSRV” ”encrypt”, ”encryption”, ”public-key”, ”private-key”	SROS2
C3. Identify safety constraints	”safeguard” ”safety check” ”safecheck”	No new exemplars found
CD1. Strive for ROS nodes with single responsibility	”bsn”, ”skiros2”, ”skiros” ”hmrs_mission_control”, ”hmrs”, ”mission_control”, ”mission control”, ”node”	No new exemplars found
CD2. Ensure global time monotonicity of events and states	”ros2-picas”, ”picas” ”mavros” ”timestamp” (in .msg files) ”SteadyTime” ”steady_clock”	ros::SteadyTime std::chrono::steady_clock
I1. Provide an API for querying and updating internal lifecycle	”rclc_lifecycle”, ”lifecycle_node”, ”rclcpp”, ”mode” def in .yaml files, api methods (”getState”, ”changeMode”)	No new exemplars found

I2. Provide an API for logging and filtering	”ROS_DEBUG”, ”ROS_INFO”, ”ROS_WARN”, ”ROS_ERROR”, ”ROS_FATAL”, ”rosrescue” ”rclpy.logging” ”rospy.logerr” ”rcutils” ”logging_macros.h”	Native ROS logging functions Rosrescue rclpy.logging rospy.logerr rcutils/logging_macros.h
I3. Provide an API for injecting faults in execution scenarios	”fuzzer”, ”fuzz” ”fault_injection” ”imfit”, ”im-fit” ”camfitool”, ”camfit”	LLVM LibFuzzer ROS rcutils library ”rcutils/testing/fault_injection.h” Software-Fault-Injection-Tool-IM-FIT camfitool
I4. Isolate components for testing	”throttle”, ”throttler” ”throttle_node” ”message throttle”	ros-tooling/topic_tools dheera/ros-synchronized-throttle UTNuclearRoboticsPublic/rosthrottle proboticks/dynamic_topic_tools

Guideline	Keywords	Exemplars
SDB1. Define properties using a logic-based language	Logic and temporal logic symbols ($\neg, \exists, \forall, \wedge, \vee, \perp$, etc.)	No new exemplars found
SDB2. Use Domain Specific Languages (DSLs) to specify properties	”property”, ”constraint”, ”precondition”, ”postcondition”, ”invariant”, ”assertion” Exclude: *.cpp,c,py,h,hpp	No new exemplars found
SDB3. Use languages and tools to scenario-based specification of test cases	”contract”, ”contract language” Exclude: *.cpp,c,py,h,hpp	ROS Contract language (RCL)
PE1. Understand the overhead acceptance criteria		No new exemplars found
PE2. Create models for runtime assessment		No new exemplars found
MTA1. Improve the robustness of the system by performing noise and fault injection	”fuzzer”, ”fuzz” ”fault_injection” ”imfit”, ”im-fit” ”camfitool”, ”camfit”	LLVM LibFuzzer ROS rcutils library ”rcutils/testing/fault_injection.h” Software-Fault-Injection-Tool-IM-FIT camfitool
MTA2. Exploit automation for test case generation, test case prioritization and selection, oracle and monitor generation		No new exemplars found

AR1. Perform postmortem analysis to diagnose non-passing test cases	”robot_localization”, ”aggregator” ”diagnostic” ”diagnostic monitor”	robot_localization, diagnostic_aggregator diagnostic_updater diagnostics_monitors
AR2. Use reliable tooling in order to manage field data	”mongodb_store”, ”warehouse_ros”	mongodb_store warehouse_ros_mongo warehouse_ros_sqlite
SE1. Use record-and-replay when performing exploratory field tests		No new exemplars found
SE2. No GUIs! Prioritize headless simulation		No new exemplars found

Table 4: This table defines and maps guidelines to exemplars found using the repository mining technique

4 Discussion

When searching for usage of provided exemplars, due to the nature of QA and development environment tools, many are attached externally to a project, without any code changes or additions to the projects themselves on their repositories. These consist mostly of tools installed directly on a ROS machine which and then attach to a process or run in parallel as a ROS add-on (i.e. SROS security package, which when added on a ROS machine it handles all the key exchanges and encryption on any connections that form with the machine). Regarding these exemplars, there is no reliable code-wise method to find out if they are utilized without an author explicitly stating in the project description or questioning the project teams on their development and testing practices and processes. These testing applications are likely found only in the development environment, if at all, and not outright in the final published source code repository.

References

[1] I. Malavolta, G. A. Lewis, B. Schmerl, P. Lago, and D. Garlan, “Mining guidelines for architecting robotics software,” *Journal of Systems and Software*, vol. 178, p. 110969, 2021.