A Framework for Quality Assessment of ROS Repositories*

André Santos, Alcino Cunha, Nuno Macedo and Cláudio Lourenço
HASLab — High-Assurance Software Laboratory
INESC TEC & Universidade do Minho, Braga, Portugal

***Abstract*— Robots are being increasingly used in safety-critical contexts, such as transportation and health. The need for flexible behavior in these contexts, due to human interaction factors or unstructured operating environments, led to a transition from hardware- to software-based safety mechanisms in robotic systems, whose reliability and quality is imperative to guarantee. Source code static analysis is a key component in formal software verification. It consists on inspecting code, often using automated tools, to determine a set of relevant properties that are known to influence the occurrence of defects in the final product. This paper presents *HAROS*, a generic, plug-in-driven, framework to evaluate code quality, through static analysis, in the context of the Robot Operating System (ROS), one of the most widely used robotic middleware. This tool (equipped with plug-ins for computing metrics and conformance to coding standards) was applied to several publicly available ROS repositories, whose results are also reported in the paper, thus providing a first overview of the internal quality of the software being developed in this community.**

I. Introduction

In the next decades, service robotics is expected to expand significantly and to be deployed in complex, unstructured environments, often requiring close human-robot interaction. This proximity to humans increases the chances of catastrophic consequences due to robot malfunction, including human injury or even loss of life.

Traditionally, safety is ensured by confining robots to controlled environments, through electronic safeguards or by physical barriers, but such mechanisms are not flexible enough to support the desired interaction and cooperation with humans. This need for more flexibility led to a gradual transition from hardware- to software-based safety mechanisms in robotic systems, whose reliability and quality is imperative to guarantee. Unfortunately, most robotic software comes from developers not proficient in software engineering techniques, which results in products with highly variable quality. In fact, a recent literature review [1] concludes that most developers rely on ad-hoc or bespoke techniques to verify safety, making it hard to assess their effectiveness.

Among the various software engineering techniques that promote the quality of the final product, source code *static analysis* is one of the most widely adopted, since it is able to (usually automatically) extract valuable information about a program without actually executing it. This information includes *internal quality metrics* – which have shown to predict defects upon release [2] – or conformance with *coding standards* – widely enforced in safety-critical contexts to improve reliability [3]. Such techniques are simple and time-efficient, applicable from early development stages, and moreover, are easily understandable by developers not accustomed with more advanced formal methods.

Developing a robot requires the integration of many complex subsystems, such as perception, planning, reasoning, navigation, and manipulation. To ease the development of robotic software systems and conceal the underlying heterogeneity, a number of *middleware* architectures have been developed [4]. This is essential in an area where modularity, re-usability and portability are inherent to the development process, as the physical components of the robot are replaced or control algorithms are re-used from other application contexts. The *Robot Operating System* (ROS) [5] is one such architecture that has been increasingly adopted by robotic software developers and has reached a high level of maturity. This growing community currently stands on tens of thousands of users worldwide, with projects as varied as autonomous vehicles and humanoid, surgical and industrial robots. Unfortunately the quality of the software is just as varied, as expected in such open communities, which renders the automatic extraction of software quality indicators even more relevant.

ROS was designed to be a thin, multi-lingual, open source framework, consisting of the interoperability of many small tools and components, and based on a peer-to-peer architecture. This modularity allows users to determine to what extent they need ROS's functionalities, and allows ROS to interoperate with other robotic frameworks. Moreover, ROS promotes a policy of publicly sharing projects as GitHub repositories through a centralized distribution file. For ROS Indigo Igloo, the latest long-term support release of ROS, the distribution file features about 700 repositories. This provides a unique opportunity not only to perform static analysis on the most relevant ROS projects, but also collect *process metrics* associated with the evolution of the repositories, which some studies actually indicate to be more effective in predicting defects [6].

Our main contribution in this paper is to provide an infrastructure to promote the quality of ROS software. Instead of proposing a self-contained framework targeting a fixed set of static analyses, we instead propose an infrastructure

*This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme within project «POCI-01-0145-FEDER-006961», and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) as part of project UID/EEA/50014/2013.

that can be easily extended to perform varied analyses. These techniques can be incorporated without specific ROS knowledge and then applied by users without any need for static analysis expertise. The resulting framework – *HAROS* – together with the incorporated plug-ins, is built to provide the regular ROS developer with valuable information without the need to learn complex techniques: as long as the repository is indexed, and a specific analysis is available through a plug-in, analyzing its code is an automatic process. As a proof-of-concept, plug-ins for the extraction of code and process metrics, and violations to coding standards have already been developed. These are then applied to a set of relevant ROS repositories, serving not only to evaluate the framework, but also to provide a snapshot of the current internal quality of software being developed in the ROS community.

The remainder of the paper unfolds as follows. Section II presents ROS as well as static analysis techniques that can be applied in its context, while Section III explores previous work on this topic. Section IV then presents HAROS along with the currently accompanying plug-ins. Section V presents the results of applying the tool to a number of ROS repositories, painting a landscape of the area. Finally, Section VI draws conclusions and points directions to future work.

II. ROS and Software Quality

The *Robot Operating System (ROS)* [5] is an open-source framework that provides hardware and operating system abstractions, as well as libraries and tools for the development of robotic systems software. It can also be seen as a middleware aiming to address the heterogeneity of these systems, by allowing the use of multiple programming languages, high-level modularity and code reuse.

A typical ROS system is composed by a set of *nodes* (processes executing different tasks) connected under a peer-to-peer topology. A special node, called *Master*, provides name registration, lookup services, and a key-value store for regular nodes to use. The communication between nodes follows mainly a publish-subscribe architecture pattern, but point-to-point communication through *remote procedure calls* is also supported.

ROS components, such as nodes and message descriptions, are organized into *packages* – the smallest build and release units. A ROS system aggregates several packages, typically developed by different people in varied contexts and using different programming languages. Since packages are logical components that depend on other components, they can in general be classified as *core packages*, *libraries*, *drivers* and *applications* depending on their role in the system. The core packages comprise the ROS tools and libraries, and make a minimal ROS installation. Libraries and drivers are developed by the community, the former providing general utility and the latter encapsulating hardware components. They can then be used as components in application packages, that represent a final robotic system – the high-level controller of a robot with some logic or instructions to execute (e.g., making a robot move in circles). The source code in a ROS package is written in general-purpose languages, being C++ and Python the most widely used among the community.

The ROS community uses Git repositories for its source code, hosting them freely on GitHub. Distribution repositories host *distribution files* for the various ROS releases – they contain information about all packages featured in a release, such as the addresses of the repositories, the current version of each package and the supported platforms for that release. GitHub is thus a central service for the ROS community.

Static analysis of source code [7] consists on extracting information from the source code with no need to compile or execute it. The kind of properties that can be verified range from simple conformity checks, such as variable initialization, to more complex properties, such as functional behavior of the program. This allows for problems to be detected early in the software development life cycle, which would otherwise go unnoticed into later stages or even into production.

Software quality metrics [8] assess the internal quality of software by means of concrete values – the idea is to have a number, calculated through static analysis, that indicates how well, or how bad, a software system conforms with a property. The values obtained can provide different types of feedback. For instance, they can say how complex or maintainable a system is, or even help to predict faults in the software product [2]. In order to monitor and control the overall quality of the software, thresholds may be defined over the measured metrics.

In the context of safety critical systems (e.g., automotive or space industries), enforcement of *coding standards* is the norm [9]. These consist mainly of rules or guidelines aiming to increase the robustness of software systems. They impose style guidelines and limit the use of certain programming languages' features to avoid common mistakes and achieve a safer subset of the language. Violations to these rules are detected through static analysis techniques.

The ROS community proposes two mechanisms to promote the quality of software components: a set of quality metrics thresholds [10] and a ROS C++ style guide [11] based on Google's C++ Style Guide [12]. This is still work in progress and the rules are non-strict, in the sense that they provide guidelines, instead of literal rules – most of them address stylistic concerns, such as naming conventions, and code formatting. It could be interesting to explore how ROS packages fare under more strict coding standards for C++, like MISRA C++ [13] and JSF AV C++ [14] which are widely and successfully used in the automotive and air vehicle industries.

Although there are plenty of static analysis tools capable of carrying the tasks described above, they do not consider the specificities of ROS (for instance, its package architecture), or if they do, they are too restrictive w.r.t. the analysis they perform, as is the case of the ROS package *roslint*[1] which encapsulates Google's *cpplint* in order to assess the compliance of ROS code to the style guide. This lack of suitable tools presents an opportunity to contribute to the ROS community, with an unified ROS-specific infrastructure

[1] `http://wiki.ros.org/roslint`

that allows for different static analysis tools to be plugged in and to promote better development practices which will lead to more robust robotic software.

III. Related Work

Research on the quality of robotic software systems is scarce, and, in general, does not target ROS systems. Cortesi et al. [15] motivate the use of static analysis techniques in robotic software. Four techniques are presented and explored, but concrete solutions on how to adapt them to the development of robotics are not proposed. Ingibergsson et al. [1] review the safety certification practices applied in the development of software for field robots. They conclude that most of the time ad-hoc or bespoke methodologies are used to assess the safety of robotic systems, disregarding the recommendations of existing robotic standards, such as ISO 13482 (personal robots) and ISO 10218 (industrial robots). Reichardt et al. [16] advocate instead that the development framework should itself provide quality promoting mechanisms. They propose a new framework for the development of robotic software with a focus on quality, in particular on maintainability, by providing components that ensure quality and enforcing development guidelines. Such components may be integrated as ROS nodes.

The collection of metrics through static analysis, and the prediction of their impact on the overall quality of the software is a very active research topic (see for instance [2] for a review). More closely related to our work, Ray et al. [17] collect several process metrics from public GitHub repositories and try to infer how these affect the quality of the projects and how this is related with the chosen programming language.

The idea to aggregate quality metrics in a plug-in-driven framework is not new. For instance, SonarQube[2] is an open source code quality management platform that allows teams to manage, track and improve the quality of their projects. However, such generic tools are not able to exploit some of the particularities of ROS (like the automatic retrieval of source code from the repositories) nor present them in a manner suitable to the regular ROS developer (taking into consideration the particular architecture employed by ROS).

IV. The HAROS Framework

It is often accepted that static analysis techniques, like the measurement of non-trivial code metrics, are effective only when assisted by tools and automation. There are plenty of such tools, but considering their limitations or inability to fit seamlessly into a ROS development environment, we developed a new ROS-centered static analysis tool, HAROS[3], with two main priorities: (i) it should be seamlessly integrated, considering specific settings of a ROS system, and (ii) it should have a broad focus, not restricting itself to particular static analysis techniques (for instance, it should support the extraction of code metrics and compliance with coding standards, but also more advanced analysis techniques). The best way of achieving (ii) is by allowing the integration of third-party analysis tools, encapsulated as HAROS plug-ins. In this perspective, the main features of this tool are:

1) source code *fetching* of indexed ROS packages;
2) easy integration of plug-ins for static *analysis*;
3) interactive graphic *reports* of the results mirroring the ROS architecture.

The use of plug-ins gears the tool towards providing static analysis benefits to the regular ROS developer, with minimal effort and required knowledge. Automating source code fetching, analysis and report production makes the tool both easy to use and to integrate in the current development process. The generation of graphic models to report quality issues also eases the use of the tool, rendering manageable what can otherwise be an overwhelming amount of information. The plug-in system makes the tool extensible, flexible, and adaptable to specific requirements and programming languages by selecting appropriate plug-ins.

Regarding the architecture of the framework, we split its core into two components that work almost independently: a configurable component that fetches and feeds the source code to the different static *analysis* plug-ins; and a component for rendering a *visualization* of the analysis results. These components are backed by a local database that the framework uses to store an index of all known source code, properties and analysis reports between sessions. The following subsections present the core components and their interaction with other components, as well as some proof-of-concept plug-ins.

A. Analysis Component

The analysis component, implemented in Python, is the main component of the tool. It runs as a console program, and it is responsible for everything but data presentation. This includes managing the source code repositories, running static analysis plug-ins, and keeping the local database updated. Its general workflow is phased in startup operations, and then the *update*, *analysis* and *export* stages, as depicted in Fig. 1.

During startup operations the tool parses user arguments and configurations. These control which of the succeeding stages will be executed, and the extent of their functionality. For instance, users can explicitly disable all operations that require a network connection. The configurations determine the list of plug-ins that should be dynamically loaded for later execution, and the output format for the analysis results.

In the first stage of execution, the tool updates its database and local copy of the source code repositories. It does so using a ROS distribution file and a *filter* file that defines the set of packages that should be analyzed. Also during this stage, a *properties* file is provided to the tool to determine which properties are expected to be analyzed and reported by the plug-ins. These properties are either *rules* or *metrics* – the former reporting violations and the latter a quantitative result – and are declared with an identifier, a description, and a set of *tags*. The tags are user-defined labels that serve mainly as a way to categorize, filter and sort rules. These files are currently made by hand, following the human-readable YAML syntax. They should be automatically provided by plug-ins in a future version.

[2] http://www.sonarqube.org/

[3] https://github.com/git-afsantos/haros

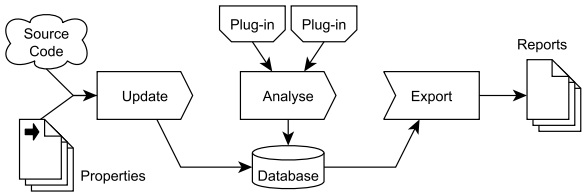

Fig. 1. Simplified workflow of the developed tool.

The tool verifies the source code in the analysis stage, relying on dynamically loaded plug-ins to look for occurrences of the defined properties. Plug-ins have access to an interface, provided by the framework, that abstracts the internal database and data structures, while allowing them to register occurrences in the form of rule violations or metric values. The framework validates whether the occurrences refer to existing properties, packages or files; further validation is left as future work – e.g., what action to take when multiple plug-ins agree (or disagree). Although plug-ins may be full-fledged analysis tools, they are expected to act as a bridge between free analysis tools and HAROS, wrapping the capabilities of third-party tools. A consequence of separating property declaration from the proper analysis is that the data structures for rules and metrics do not need to hold any information on how they are verified or measured. Another result of this plug-in model is that the source code of the tool itself needs not change to accommodate new rules, metrics or programming languages. Only the plug-ins and the configuration files loaded on startup need be adapted.

The final stage of execution exports report files with data from the database, as a way to interoperate with other tools. In particular, this functionality is used to interoperate with the visualization component of HAROS presented in the next section. The reports include the set of considered properties, a summary of the selected packages, and a detailed analysis for each package. The summary contains general details about each package, while the analysis files contain information about each measured property in a package, as detailed as possible (e.g., measured values, source file and line number). All exported files are under the JSON or CSV formats.

B. Graphic Component

From the exported data, this component, implemented in HTML and JavaScript, builds a diagram – a directed graph where each node represents a package, and each edge represents a package dependency. It then applies a color scheme in which darker nodes have more reported violations. Fig. 2 shows a graph, as rendered by the application. The side menu allows inspection of all rule violations. However, inspecting code metric values is still a work in progress.

The tags associated to each rule allow the user to filter the reported violations, so that a subset can be hidden or emphasized (e.g., show only a certain quality standard). The component adjusts node colors to the filters in place. This coloring system is relative: the darkest node just represents the package with most rule violations in the visible graph.

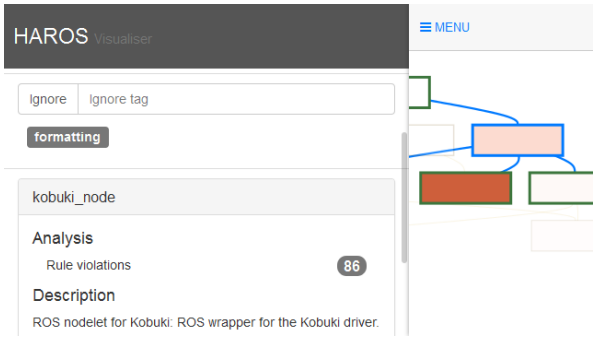

Fig. 2. A package graph rendered by the tool, with a selected node.

Regarding user interaction, a user can zoom and pan the graph (zooming in reveals node names), as well as select specific packages to inspect and highlight. Package inspection shows the user which rules were violated, along with all the registered information about each violation. This detailed inspection is also subject to the tag filters. One possible enhancement for future versions is to allow inspection of particular files, classes or functions. Fig. 2 depicts the effects of node selection on the graph and on the collapsible side menu. Additionally, users can focus the graph on a node of their choice, reducing the visible graph to that node and its neighbours. Clearing the focus renders the complete graph again.

C. Currently Implemented Plug-ins

As a proof-of-concept, a series of plug-ins were developed for HAROS, mainly encapsulating existing third-party tools. The focus was on collecting metrics and rule violations that would primarily fit the ROS quality metrics thresholds and ROS C++ Style Guide (Section II). These are summarized in Table I.

The set of collected metrics includes common metrics, such as the number of lines of code and comments, the average cyclomatic complexity [18], the Maintainability Index [19], and some other metrics proposed in [20]. The complete set of metrics includes half of those defined in ROS quality metrics and other metrics that the ROS quality model does not include.

As for coding rules, unrelated to metrics, we support the verification of about 120 rules from various C++ coding standards – a result of wrapping the *roslint* and *Cppcheck*[4] tools as plug-ins – 90 of which come from the combined Google and ROS C++ coding standards. Other C++ standards, such as HIC++, MISRA C++ and JSF AV C++, were also covered, although to a much smaller extent, of about 30 rules (when excluding overlaps).

Since quality models essentially define thresholds over the metrics, quality metrics plug-ins report both the measured values and threshold violations. Such is the case with the ROS quality thresholds (RQV) and others from which it drew inspiration (listed in [10]). This has two advantages: firstly, it results in an uniform report, since all violations, on metrics or

[4] `http://cppcheck.sourceforge.net/`

TABLE I

SUMMARY OF COLLECTED METRICS

	property	description
rules	RQV	violations to the ROS quality thresholds
	NQV	violations to the NASA SATC quality thresholds
	HQV	violations to the HIS quality thresholds
	KQV	violations to the KTH quality thresholds
	AQV	violations to the University of Akureyri quality thresholds
	RCV	violations to the ROS C++ style guide
	GCV	violations to the Google C++ style guide
	HCV	violations to the HIC++ coding standard
	JCV	violations to the JSF AV C++ coding standard
	MCV	violations to the MISRA C++ coding standard
source metrics	LOC	number of lines of code
	LOCom	number of lines of comments
	%Com	comment to code ratio
	CC	average cyclomatic complexity
	MI	maintainability index
	CBO	coupling between objects
	WMC	weighted methods in class
	MAC	methods available in class
	#Dep	number of depending packages
	#RQV	number of ROS quality model violations
	#RCV	number of C++ coding violations
process metrics	#Rev	number of commits
	#DC	number of distinct contributors
	#OI	number of open issues
	#CI	number of closed issues

otherwise, are considered equal (although these can be filtered in the visualizations); secondly, the number of violations to the ROS quality thresholds and coding standard constitute other metrics for consideration (#RQV and #RCV).

A plug-in was also developed to collect process metrics from the GitHub repositories, including contributors to the repositories, number of commits and number of repository issues.

V. OVERVIEW OF THE ROS CORPUS

Although HAROS was developed with the goal of providing ROS developers with relevant information regarding their repositories, the public nature of the ROS distribution policy allowed us to apply the tool to a series of ROS repositories considered relevant. The result is a snapshot of the current state of the ROS corpus. Additionally, this also serves as evaluation for the proposed framework.

Specifically, we hope to shed some light regarding the following questions:

- Are the quality thresholds and coding standards followed by the developers?
- What are the most common violations to those rules?
- What is the relation between the internal code quality of a package and its process metrics?
- What is the relation between the internal code quality of a package and its role in a ROS system?

The repositories were selected based on their popularity and level of activity in GitHub. Concretely, the repositories for 11 ROS robots that contained at least 100 commits, and development branches for ROS Hydro Medusa or newer were collected. Examples include the PR2[5] and TurtleBot[6], that have been used in public events and research. This yielded nearly 50 very heterogeneous repositories – a total of 180 packages that contain C++ source code.

We deliberately left out the ROS core packages from this process, in order to focus on the products (robotic systems) developed by the ROS community, the target of the proposed framework. Furthermore, we believe that a reliable quality assessment of the ROS core packages demands a greater range of techniques and more refinement than what we can muster at the moment. The selected packages were categorized according to their role in a ROS system, as follows:

1) drivers, hardware interfaces and other low-level code;
2) libraries and hardware-independent utility code;
3) applications that depend on the previous items.

As we are still unaware of any simple, deterministic criteria to classify packages, this categorization was made manually, based on the traits that each package exhibited the most.

A descriptive overview of the analyzed packages can be seen in Table II, which shows that most packages have tens of violations to the ROS quality model thresholds, but these numbers scale to the thousands when regarding the ROS C++ coding standard. The few packages with no violations to the thresholds are very small, with fewer than 300 lines of code. The most violated thresholds are the minimum comment ratio (the code is not documented enough), the maximum CBO (the classes are heavily coupled), and the maximum CC (the functions have too many decisions). On the other hand, the less violated thresholds are the maximum WMC (class methods are not too complex) and the maximum MAC (the classes do not have too many methods), both below a dozen violations. These observations manifest consistently across all package roles, but, perhaps surprisingly, the majority of the violations regarding insufficient documentation occurs in library code. A possible explanation is that, sometimes, documentation is stored separately from the source code (e.g., ROS Wiki, or tutorials).

Although looking at raw data is valuable in identifying problematic targets, it does not provide the whole picture. We calculated the Pearson correlation coefficients between the measured metrics, in order to try to identify patterns and influences. Unfortunately, the correlation coefficients are very low, in general, and the correlations are not surprising. For instance, the lines of comments increase as the lines of code increase, but so does the number of violations (#RQV and #RCV), which are also correlated to each other. Some worthy mentions are the correlations between the MI and the comment ratio (well documented code is more maintainable), and between the WMC and the MAC (the more methods a class has, the more complex it tends to be).

In our setting, the metrics show that applications and drivers tend to have more developers, commits and raised issues than library code, suggesting that these packages may be more faulty. Indeed, this code is, at least, more complex (CC) and tightly coupled (CBO). However, libraries exhibit greater reported violation figures. These packages are less ROS-oriented, and so these rules (e.g., formatting) may be less of a concern to the developers. Or it could be that libraries

[5] http://wiki.ros.org/Robots/PR2

[6] http://wiki.ros.org/Robots/TurtleBot

TABLE II

DESCRIPTIVE PACKAGE ANALYSIS

	Median	Standard Deviation	Min.	Max.	Threshold
Role	1	0.71	1	3	—
LOC	461	3779.17	20	36353	—
LOCom	270.50	1322.57	0	14503	—
%Com	0.52	0.53	0	3.85	$0.2 \leq x < \infty$
CC	3.92	5.03	0	47.48	$1 \leq x \leq 15$
MI	73.13	17.58	0	99.97	$0 \leq x \leq 100$
CBO	1.70	0.60	0	3.20	$0 \leq x \leq 5$
WMC	4.67	3.76	1	28.36	$1 \leq x \leq 100$
MAC	2.00	2.66	0	21.20	$1 \leq x \leq 20$
#Dep	1	5.94	0	41	—
#RQV	3	26.80	0	333	—
#RCV	387	5384.63	10	61712	—
#Rev	292	572.71	16	2706	—
#DC	9	10.03	1	49	—
#OI	6	9.93	0	41	—
#CI	50	91.67	1	343	—

are simply more static, and less fun to work on, than other packages. Overall, the various quality indicators are not very conclusive, and an accurate quality assessment may depend on a prioritization of said indicators.

VI. CONCLUSION

This paper presented HAROS, a plug-in-driven framework for the automatic static source code analysis of ROS repositories. HAROS's primary focus is to assist in uncovering potential faults and defects in ROS systems, by employing diverse analysis techniques, but requiring as little effort and prior knowledge from the ROS developer as possible. Hence the importance of incorporating the specificities of ROS systems and presenting the analysis reports in a user-friendly way, without neglecting detail. This framework also presents an opportunity for new static analysis tools to emerge, or for existing tools to be reused as plug-ins.

In order to evaluate the tool, we analyzed a set of ROS repositories, consisting of 11 robots and containing some of the most popular and iconic robots of this community, such as the PR2 and TurtleBot. Consequently, this analysis provides a snapshot and an overview of the quality of the ROS body of work. Although we focused on gathering quality metrics and violations to coding standards for manually categorized packages, our results clearly suggest that, both at global and role levels, there are many occurrences of overly complex and insufficiently documented code, and compliance with standards is not yet a strong concern of the community.

The extraction of quality metrics and violations to coding standards is but a first step towards our goals of promoting the quality of ROS software. Concretely, we are currently working on the formal verification of functional properties and exploring model-based techniques. Regarding HAROS itself, there are many improvements underway. These include typical performance optimizations and enhancements to the user interface, but also an overhaul to the current data structures, to better accommodate new analysis techniques. We intend to allow interoperation between plug-ins, to integrate the tool with the *catkin* build system, and to track package quality evolution over time. Finally, concerning our analysis results, we hope yet to achieve a deterministic and automatic system to categorize ROS packages, perhaps with finer grained roles. We look forward to revisit this quality assessment of the ROS corpus with more accurate and conclusive results.

REFERENCES

[1] J. Ingibergsson, U. Schultz, and M. Kuhrmann, "On the use of safety certification practices in autonomous field robot software development: A systematic mapping study," in *PROFES'15*, ser. LNCS, vol. 9459. Springer, 2015.

[2] D. Radjenovic, M. Hericko, R. Torkar, and A. Zivkovic, "Software fault prediction metrics: A systematic literature review," *Information & Software Technology*, vol. 55, no. 8, pp. 1397–1418, 2013.

[3] G. J. Holzmann, "Mars code," *Communications of the ACM*, vol. 57, no. 2, pp. 64–73, 2014.

[4] A. Elkady and T. Sobh, "Robotics middleware: A comprehensive literature survey and attribute-based bibliography," *Journal of Robotics*, vol. 2012, pp. 959 013:1–959 013:15, 2012.

[5] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: An open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009. [Online]. Available: https://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf

[6] F. Rahman and P. T. Devanbu, "How, and why, process metrics are better," in *ICSE'13*. IEEE/ACM, 2013, pp. 432–441.

[7] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.

[8] C. Kaner and W. P. Bond, "Software engineering metrics: What do they measure and how do we know?" in *Metrics'04*, 2004.

[9] A. Goforth, "The role and impact of software coding standards on system integrity," in *I@A'13*. AIAA, 2013.

[10] Open Source Robotics Foundation, "ROS code quality," http://wiki.ros.org/code_quality, 2013, [Online; accessed 14-October-2015].

[11] ——, "ROS C++ style guide," http://wiki.ros.org/CppStyleGuide, 2014, [Online; accessed 14-October-2015].

[12] Google, "Google C++ style guide," http://google-styleguide.googlecode.com/svn/trunk/cppguide.html, 2014, [Online; accessed 14-October-2015].

[13] C. Tapp, "An introduction to MISRA C++," *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, vol. 1, no. 1, pp. 265–268, 2009.

[14] Lockheed Martin Corporation, "Joint Strike Fighter Air Vehicle C++ coding standard for the system development and demonstration program," Tech. Rep. 2RDU00001 Rev C, December 2005.

[15] A. Cortesi, P. Ferrara, and N. Chaki, "Static analysis techniques for robotics software verification," in *ISR'13*. IEEE, 2013, pp. 1–6.

[16] M. Reichardt, T. Föhst, and K. Berns, "On software quality-motivated design of a real-time framework for complex robot control systems," *ECEASST*, vol. 60, 2013.

[17] B. Ray, D. Posnett, V. Filkov, and P. T. Devanbu, "A large scale study of programming languages and code quality in GitHub," in *FSE'14*. ACM, 2014, pp. 155–165.

[18] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[19] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," in *ICSM'92*. IEEE, 1992, pp. 337–344.

[20] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.