# Applying Software Static Analysis to ROS: The Case Study of the FASTEN European Project

Tiago Neto[1,2], Rafael Arrais[1,2], Armando Sousa[1,2], André Santos[2,3], and Germano Veiga[1,2]

[1] Faculty of Engineering of the University of Porto, Portugal,
[2] INESC TEC - INESC Technology and Science, Portugal,
[3] Universidade do Minho, Braga, Portugal,
tiago.f.neto@inesctec.pt

**Abstract.** Modern industry is shifting towards flexible, advanced robotic systems in order to meet the increasing demand for custom-made products with low manufacturing costs, and to promote a collaborative environment for humans and robots. As a consequence of this industrial revolution, some traditional, mechanical- and hardware-based safety mechanisms are discarded in favor of a safer, more dependable robot software. This work presents a case study of assessing and improving the internal quality of a European research mobile manipulator, operating in a real industrial environment, using modern static analysis tools geared for robotic software. Following an iterative approach, we managed to fix about 90% of the reported issues, resulting in code that is easier to use and maintain.

**Keywords:** Software Static Analysis, Safety, Mobile Manipulator, ROS

## 1 Introduction

The shifting of paradigm imposed by the ongoing Fourth Industrial Revolution is introducing a new set of constraints and opportunities for industrial enterprises. These constraints and opportunities are serving as a catalyst for the introduction of flexible, adaptable and collaborative human-robot hybrid systems which can enable even small and medium enterprises to adapt to paradigm changes in market demand, often characterized by increasing customization [2]. These systems are materializing as collaborative robotic solutions in industrial applications and as autonomous mobile robotics in sectors ranging from agriculture to intralogistics, operating in a dynamic and unstructured environments shared with humans.

Such advanced robotic systems, operating in cross-sectorial domains of activity, sensing and interacting with complex and unstructured environments require the integration and support of the technologies, models, and functional components that enable robotic operations. In this context, the safety of humans operating and interacting with potentially dangerous equipment is a core scientific and technological challenge. Thus, and to cope with market demand for

product customization or demanding field applications, contemporary robotics must drastically alter the safety assurance paradigm.

Traditionally, roboticists majorly relied on mechanical-based methodologies, such as physical barriers, to account for safety behaviour. However, as modern systems need to be flexible, adaptive and collaborative to adhere to the ongoing industrial revolution, software-based safety assurance mechanisms are emerging as a complement to traditional safety procedures. In addition, software-based safety assurance can play important social and psychological roles to foster the acceptance of robots in human-populated environments and to promote collaboration between humans and robots. This change affects the robotics ecosystem and calls for techniques to promote best software engineering practice guidelines for the development of safety-critical software, suitable for the robotics development environment.

In a clear contrast with the current necessities, particularly in the cutting edge of innovation efforts, this meticulous attention to software engineering guidelines and safety assurance of software-based components is often overlooked [5] due to the experimental nature of developments, the complexity of the systems, and the difficulties associated with validating the software-based safety mechanisms in physical hardware.

Over the last decade, frameworks such as the Robot Operating System (ROS) [3] have emerged as de facto standards for robotic software development, with an increasing presence in the industrial environment. ROS provides roboticists with abstractions and a vast amount of libraries that widely simplifies and speeds up the development of advanced robotic systems. However, these benefits come with a price, in particular, the intrinsic difficulty to fully assess and validate ROS-based software and external libraries in what regards their compliance with safety protocols or even guidelines for software engineering best practices.

The project_A (SAFER) project, in which this work is integrated, brings together the expertise of computer scientists, with a background on software system design and analysis, and experienced robot engineers, to overcome the aforementioned shortcomings of ROS-based software development. One of the project's main output is the High Assurance ROS (HAROS) tool, a static analyzer of ROS-based software, that can extract valuable information from the source code without the need for executing it (or even compiling it, in many cases). The application of this tool during the development process promotes compliance with software engineering best practices and can be a valuable tool to allow developers to assess the safety compliance of their software. Furthermore, by promoting the creation of better-structured source code, its readability, maintainability, and scalability are deeply improved, potentially resulting not only in increased safety compliance but also in long-term financial gains, as the produced source code is easier to work with.

In this paper, the application of the HAROS tool to a complete stack of ROS-based software powering a mobile manipulator operating in an industrial environment is explored, with the objective of assessing and iteratively improve the code quality. The remainder of the paper is organized as follows: Section 2

presents a conceptual overview of some of the discussed domains, as well as a brief state of the art of the subject; Section 3 presents a detailed description of the industrial utilization of the developed mobile manipulator, its hardware composition, and its software architecture; Section 4 highlights the principal scientific contribution of this research work, by presenting the methodology and results obtained from the application of the HAROS tool to guide ROS-based software development; and, finally, Section 5 draws some conclusions and points out some future work roadmap.

## 2 Related Work

A deciding factor in the adoption of robotic systems in real-world scenarios is related to the trust levels that humans have in their utilization. In order to fully promote the mass adoption of robotic systems in manufacturing, complying with the ongoing industrial revolution, users need to be fully confident in their operation. In what concerns these systems in a broader sense, trust can be defined as a combination of reliability, safety, security, privacy, and usability [7].

Static analysis techniques are one of many software engineering techniques that can elevate the quality of code, and thus also increase trustability in the developed system. This conceptually simple and time-efficient technique allows, since an early phase of development, the extraction of precious information from a program without running or even compiling it. Among the collected information, compliance of the code with given specifications, internal quality metrics and conformity with coding standards are amongst the most valuable metrics [6]. Static analysis tools evolved to be able to deal with industrial applications, containing millions of lines of code. In [1], the authors provide a comparative analysis of three of the most powerful and popular static analysis tools for industrial purposes, namely *PolySpace*, *Coverity* and *Klocwork*.

In the domain of robotics, ROS, an open-source tool-based framework that provides developers with a large set of libraries and abstractions to ease the difficult task of developing robotic software [3]. Since its introduction, ROS is increasingly being introduced in industrial applications. However, ROS does not impose strict development rules to ensure its safety. Due to the great diversity of ROS applications, there is no solution to completely analyse and verify ROS programs in a formal way and certify their safety to guarantee correct behaviour of robots.

As an alternative to the lack of intrinsic safety compliance mechanisms in ROS and the underlying difficulty to validate such compliance, software static analysis can yield valuable information about the behaviour of each of its subsystems and the interactions between them, thus allowing developers to preemptively verify if the source code is according to the requirements and, consequently and implicitly, improving its safety compliance capabilities [5].

Despite the potential of this technique, applying it to ROS is not so straightforward. As previously mentioned, ROS is very customizable, has a large number of primitives and can be written in several programming languages. This diver-

sity leads to an extremely complex and unfeasible ad hoc solution for an arbitrary ROS system. Nevertheless, for a more restricted set of ROS subsystems, and a bounded set of constraints, it could be achievable [5].

An example of a static analyser for ROS-based code is HAROS. HAROS is being developed having two fundamental ideas in mind: one is the integration with ROS specific settings, and the other is that it should not be restrictive, thus allowing the use of a wide range of static analysis techniques. The latter notion leads to HAROS allowing the integration and use of third-party analysis tools, as plug-ins [6]. This tool allows the fetching of ROS source code, its analysis and the compilation of a report in an automatic way. Therefore, it can be easily used, even by developers without extensive knowledge of ROS or static analysis techniques.

With HAROS, the user first chooses which packages should be analysed, and according to the required analysis, HAROS will dynamically load the adequate plug-ins. The properties that are analysed can be of two categories: rules or metrics. Rules report violations as individual issues, while metrics return a quantitative value, which can, in turn, result in a set of issues [6]. Once the configuration and analysis steps are concluded, the results are portrayed to the user in both a graphical form and by a list of issues, which can be filtered by their type. In its graphical form, the results are portrayed to the user in both a graphical form, and by a list of issues, which can be filtered by their type. In its graphical form, the results visually display the analyzed metrics, and, most importantly, the system-wide and intra-node architecture and properties.

On [5], the authors focused on interpreting the outputs of applying a static analysis provided by ROS on a set of popular and publicly available ROS packages. Collecting this kind of information is important to elucidate about less used or even misused features and is also useful for developers of static analysis tools to determinate which features are more relevant to be supported [5]. HAROS was also used by the authors of [4], to extract and analyze the architecture of a field robotic system for the agriculture domain at static time. This verification provides valuable information during the development phase, which was used to ensure that safety design rules were well implemented in the architecture of the studied robot, validating and improving the safety of the system [4].

In this work, HAROS is applied on an industrial robotic system not only with the purpose of validating this tool, but also, and more critically, to attempt to verify and improve the safety of the system and, indirectly, the maintainability of the source code, as it will be demonstrated in Section 4.

## 3 Case Study Description

The case study for the work was the H2020 Flexible and Autonomous Manufacturing Systems for Custom-Designed Products (FASTEN) project. This project aims to develop, demonstrate, validate, and disseminate a modular and integrated framework able to efficiently produce custom-designed products. In order to achieve that it integrates digital service/products manufacturing processes,

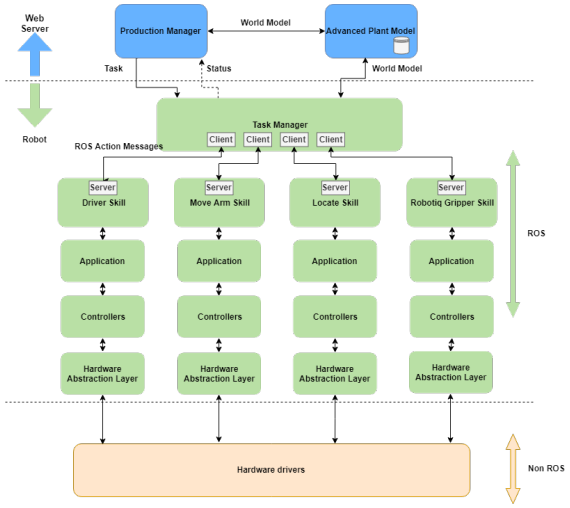


**Fig. 1.** High-level software architecture of the FASTEN robot system.

decentralized decision-making and data interchange tools. Thus, to achieve a fully connected and responsive manufacturing system, several technologies are being developed, as is the case of sophisticated self-learning, self-optimizing, flexible and collaborative advanced robotic systems. As proof of concept, a mobile manipulator, capable of assembling and transporting kits of aerospace parts is being developed. Currently, at the scenario, Embraer Portugal S.A. (Embraer Portugal S.A.), the industrial end-user of the project, stores the parts used for wing assembly in a Automated Warehouse System (AWS). The kitting operation, composed by the retrieval of components from the AWS is a repetitive, non-ergonomic and non-added-value task which can be automatized to improve performance and working conditions. Furthermore, by relying on an automatic solution to assemble kits, Embraer Portugal S.A. can further enhance the traceability of its intralogistics process.

For this, an automated solution is being developed (Fig. 2). It is composed by an Automated guided vehicle (AGV) with an omnidirectional traction configuration, fitted with a collaborative robotic manipulator. So, this mobile manipulator is capable of traversing the logistic warehouse in any direction and cooperate with human operators in the assembly of kits, increasing the automation level and freeing human operators for more added-value tasks.

The software architecture of this system is being developed with three main objectives in mind, that lead to three structural ideas. The first objective is to reduce the cost of adapting robot applications by promoting code re-usability. To achieve this, a skill-based robot programming approach was used. The second objective is to promote intuitive and flexible robot programming, achieved by task-level orchestration. The third objective is to support generic interoperability with manufacturing management systems and industrial equipment. As depicted


**Fig. 2.** FASTEN Mobile manipulator developed for application in an Embraer Portugal S.A. industrial plant.

in Fig. 1, this robotic system has a distributed architecture. In the server side implementation, there are two components, the Production Manager (PM) and the Advanced Plant Model (APM) [8], while on the robot side of the architecture, there are the skills and the Task Manager (TM). The APM keeps a near real-time model of the production environment. The PM is responsible to manage the production resources, control the execution of the production schedules and it is also responsible for monitoring the ongoing performance of the different production tasks.

On the robot, one of the most important components is the Task Manager (TM). The TM has two primary functions: it (i) provides integration between the robot and other modules on the system, like the APM or the PM, and (ii) is responsible for the orchestration of tasks, using the skills of the robot. On the TM there is a ROS Action Client for each skill and on each skill, there is a ROS Action server. This is due to the fact that skills are implemented using ROS Actions. The TM uses skills by defining a goal and sending it to the respective Action server. When the execution is completed it receives, from the skill Action server, the result and additional information about the outcome of the performed action.

For the H2020 FASTEN demonstrator, the robotic system has been instantiated with four different skills: (i) Move Arm Skill, (ii) Gripper Skill, (iii) Locate Skill, and (iv) Drive Skill. The Move Arm Skill is responsible for the movement of the robotic manipulator. The Gripper Skill is responsible for the actuation of the gripper. The Locate Skill is responsible for the recognition and localization of the parts that need to be handled. Finally, the Drive Skill is responsible for the movement of the robotic platform and ensuring that the movement is collision free. Each of these skills is organized in three different parts, which are the Application Layer, the Controllers Layer, and, finally, the Hardware Abstraction Layer. These three layers allow a goal received from the TM to be transmitted to the hardware drivers and then executed.

## 4 Software Quality Analysis

A software quality analysis was conducted on the ROS-based mobile manipulator software presented in the previous section. This software stack comprised the set of functional components, in the form of ROS source code and launch files, responsible for powering the FASTEN use case demonstrator. In total, 22 packages were analysed, from which 14 contained C++ source code, while the remaining contained Python source code or only ROS launch files. The C++ source code amounted to approximately 200,000 lines of code.

To conduct this analysis, the HAROS tool was used. After an initial overview analysis of the complete system, its source code issues were listed and grouped by category for each ROS package. The remainder of the analysis was iterative. This means that the source code issues and model inconsistencies discovered were addressed in several iterations. After each individual iteration, the obtained results were re-evaluated with the HAROS tool and the strategy for the next iteration was drawn. This iterative approach was elected due to the intrinsic difficulty to address all software issues in a single run, allowing developers to assess, in each iteration, if the proposed changes do not impose constraints on the integrity of the system. In addition, addressing all software problems in a single passage would most likely originate novel issues that would be hard to trace the origin of. Moreover, an iterative methodology was employed in order to promote the continuous integration paradigm.

The conducted analysis can be divided into two distinct phases. The Architecture Analysis, presented in subsection 4.1, allows developers to have the full-scale system-wide and intra-node overview of the system and assess if the developed architecture is according to the specifications. The Static Code Analysis, presented in subsection 4.2 refers to the reasoning on the source code of each software application that composes the system. This analysis allows developers to catch safety-critical issues, and assess if the code complies with normative standards and guidelines, thus empowering not only the safety of the whole robotic system but also the underlying code maintainability and scalability.

### 4.1 Architectural Analysis

The architecture analysis is the differentiator feature that separates the HAROS tool from the remaining static analysis tools. For this feature, it is necessary to inform HAROS which ROS launch files should be analysed. Then, with that information, HAROS extracts the ROS nodes that are being launched by that file and the arguments that are being passed during the launch. However, in its current version, HAROS is not capable of finding a node that is being launched conditionally.

As the FASTEN mobile manipulator development is adopting a methodology where the ROS launch file of each sub-system is conditional it was necessary to provide hints via a YAML configuration file required by HAROS. These hints provide HAROS with additional information about which ROS topics are subscribed or published by each ROS node that composes the system.

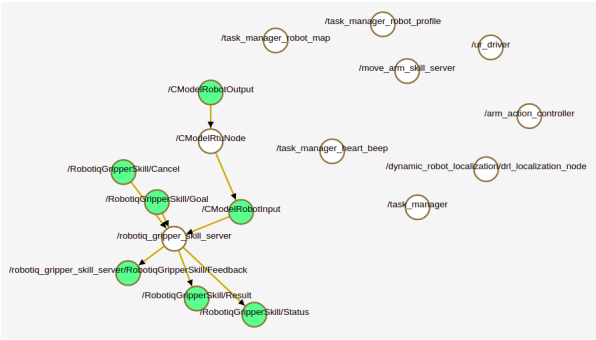


**Fig. 3.** Architectural analysis of the robotic system as displayed by the HAROS web-based visualization tool.

The visualization of the output of this architectural analysis in the HAROS user interface is depicted in Fig. 3. This visualization component provides a good insight into what is to be expected from the application ROS nodes. Nevertheless, since this extraction could not be automated and had to be provided by hints, the model extraction tool validity and correctness is questionable for the purposes of this case study.

### 4.2 Static Code Analysis

**Initial Analysis** This initial analysis contains the raw data collected using the HAROS tool. The issues were divided into 3 categories: Formatting, Code Standards, and Metrics. The first category, Formatting, encloses issues related to indention, whitespaces and the placement of braces. The second, the Code Standards, encloses issues related to the compliance with code standards, i.e. adhering to a specific style of programming or restricting oneself to a subset of the programming language. Finally, the Metrics, encloses issues related to internal quality code metrics, such as cyclomatic complexity or the maintainability index.

Since it was impossible and impractical to solve every issue with one run, the intervention process, guided by the issues reported by HAROS, was divided into several iterative steps. Furthermore, it was necessary to determine which issues would be tackled first. In order to elect the first issues to be tackled, a model, described by Equation 1 is proposed.

$$Score = K_1 \cdot Num + K_2 \cdot S + K_3 \cdot E; \tag{1}$$

This model attributes a score to each issue within a ROS package. The score is a weighted sum of the number of issues, $Num$, where $S$ represents the severity of the issue and $E$ represents and the effort to solve it. For this analysis, $S$ and $E$ were classified using a rank ranging from 1 (not severe, easiest to solve) to 3 (severe, hardest to solve). $K_1$ and $K_3$ were given the coefficient 1 while to $K_2$, which represented the severity, was given the coefficient 10. The biggest

coefficient weight was given to the severity so it could have a more pronounced impact on the total score of an issue.

The initial analysis of the source code resulted in the report of a total of 28,040 issues, as can be seen in detail on Table 1.

**First Iteration** For this first iteration, it was assumed that the code did not follow any code standard format since the code was developed by various development teams, and it also simplified the code format standard uniformization to be conducted. Analysing the results of the initial analysis, it is pretty clear that most of the issues are of the formatting type, as can be seen on Table 1, which means that they should be the first ones to be tackled. Since the code is vast it would be impractical and extremely time-consuming to correct all the formatting issues by hand. So, in order to tackle this kind of issues an automatic approach was taken. The chosen tool was the *Clang-Format* along with Visual Studio Code.

The *Clang-Format* was used to format the code accordingly to the Google C++ style guide. The decision to chose Google C++ style instead of ROS C++ Style was based on the fact that the portability of the majority of the source code to this style guide would be more straightforward. After the use of this tool, some additional adjustments had to be done by hand. This was necessary to ensure that the code still compiled. The adjustment done by hand were mostly related with include orders since the automatic tool rearranged the header files in such a way that compilation was not possible.

This first iteration allowed to eliminate 14 types of issues, from 67 in the initial analysis to 53 at the end of the first iteration. This was mostly because of the reduction of the Formatting issues from 24 to 10. On the total number of issues, it was registered a decrease of 22,686 issues. Even though the Formatting and Code Standard issues decreased, the Metric issues increased. The cause of this was the changes made to respect the line length that triggered an increase in the use of vertical lines. This increase originated a spike in the number of functions to have more than 40 lines of code, which, in its turn, triggered more Metric issues.

**Second Iteration** For the second iteration, one of the issues with a higher score was the *line length*. Since the automatic formatting did not solve this, the source code was manually analysed to understand the root of this issue. There were two explanations: (i) functions with long names could not be solved, and (ii) comments with section markers could not be automatically processed. Regardless, this could be solved by reducing the number of repeated characters without removing the code separation.

Another issue with a high count of occurrences was the *Non-const Reference Parameters*. This issue was caused by variables being passed by reference, but not using the keyword *const* as recommended by the Google C++ style guide. This issue has two possible solutions. The first is to use the keyword *const* if the variable does not need to be changed inside the function and the other, which

requires more effort, is to pass by a pointer and to change the code according to this demand. However, since the second solution was the one that needed to be applied more often, it was opted to leave the code as is, to avoid cross-package errors that could be hard to track. Also, this type of issue did not represent a safety threat.

The issues of the type *Integer types* were also among the issues with a higher count. These issues were mostly triggered by the use of the type *size_t*, but also by the use of the type *short* or *long*. The usage of the type *size_t* is allowed by Google C++ style guide when it is appropriate, which was the case for the totality of occurrences, and for that reason, it was not changed. When types such as *short* were being used, they were replaced by size specific types, such as *int16_t*.

In this iteration, issues with *whitespaces*, *copyright*, and *contructors* were tackled. The *copyright* issues were solved by adding a copyright statement to each file, while the *constructors* issues were addressed by making constructors with single argument explicit. Furthermore, issues related to *casting* were also solved during this iteration. However, at the end of the iteration, HAROS still identified 2 casting issues. Yet, while manually inspecting the code, it was found that these were not casting issues, but were, in fact, false positives.

Finally, in this iteration, the issues with the *floating point* were solved. These issues were caused by float point expressions that were expecting exact equality, which is not compliant with the MISRA C++ guidelines, deeming it unsafe. The solution for these issues was to rewrite the expressions in a way that did not test equality directly and that was compliant with the guidelines.

Overall, 2,498 issues were solved in this iteration, which reduced the total of issues to solve to 2,859 at the end of this iteration. The formatting issues decreased from 10 to 5 and the code standard issues from 34 to 32. However, the average severity and average effort to solve increased from 1.85 to 1.95 and from 1.68 to 1.83, respectively. This is justified by the fixing of more issues with lower severity and lower effort to solve. Nevertheless, this was also a successful iteration, since it led to a reduction of around 50% of issues reported in the previous iteration.

**Third Iteration** This third and final iteration focused on solving issues related to cyclomatic complexity, functions that were not thread safe and also analysed other issues to understand their causes.

Among the metrics, the cyclomatic complexity is the easiest to change and improve. Despite that, it does not mean that it is a simple issue to fix. Some functions with high cyclomatic complexity are impossible to do in a less complex way, as their purpose is to verify a set of conditions that can not be easily changed. Others are simply just too complex, and it is therefore very risky to change them without incurring in drastic changes to the behaviour of the software, as this code belongs to robotic software that is responsible for the implementation of very specialized and complex features, such as computer vision algorithms. Areas like this require some specialized expertise to alter those algorithms, which compli-

**Table 1.** Static code analysis results of the initial analysis and subsequent iterations.

| | | Types of Issues | Issues | Average Severity | Average Effort to Solve | Total Score |
|---|---|---|---|---|---|---|
| **Initial** | Formatting | 24 | 24511 | 1.61 | 1.47 | 34414 |
| | Code Standard | 34 | 3175 | | | |
| | Metric | 8 | 356 | | | |
| | Total | 66 | 28043 | | | |
| **First** | Formatting | 10 | 2327 | 1.85 | 1.68 | 10545 |
| | Code Standard | 34 | 2288 | | | |
| | Metric | 8 | 478 | | | |
| | Total | 52 | 5357 | | | |
| **Second** | Formatting | 5 | 253 | 1.95 | 1.83 | 7267 |
| | Code Standard | 32 | 2126 | | | |
| | Metric | 8 | 480 | | | |
| | Total | 45 | 2859 | | | |
| **Third** | Formatting | 5 | 253 | 1.93 | 1.90 | 6485 |
| | Code Standard | 30 | 1883 | | | |
| | Metric | 8 | 467 | | | |
| | Total | 43 | 2603 | | | |

cate the task of changing these algorithms. However, for some of these functions, it is possible to understand their purpose without deep knowledge of the area. For some of those, it is possible to achieve the same result using less complex ways. Thus, during this iteration, it was possible to reduce the cyclomatic complexity of functions with a cyclomatic complexity score as high as 17. Above that value, it was opted not to change them due to the high probability to introduce errors. For these more complex functions, it is recommended intervention from a development team with higher expertise in the domain.

In spite of this last iteration not being able to solve as many issues as the previous ones, most of the issues solved on this iteration were harder to solve. Most of the issues solved on this iteration were also more severe, which reflected on the decrease in the average severity. On this iteration, 256 issues were solved, which led to a decrease in the total number of issues from 2,859 to 2,603 at the end of these iterations (around 9%). In this iteration, 2 Code Standard issues were also eliminated, reducing the total type of issues to 43 and the Code Standard issues to 30.

## 5 Conclusion

Overall, the source code analysis allowed to solve 25,440 issues, which represents a reduction of 90% of issues from the initial analysis. Some of the fixed issues were deemed to be dangerous and could potentially compromise the run-time functioning of the mobile manipulator. As such, the alterations performed by this work undoubtedly allowed the improvement of the safety and maintainability of the source code, and, correspondingly, the FASTEN mobile manipulator operation in an industrial environment.

With this analysis, it was also clear that the introduced improvements could benefit the development process in the long run. Thus, the methodology described in the paper is being applied during nominal development procedures. As such, the FASTEN mobile manipulator development teams are actively using the proposed methodology and applying the HAROS tool in a continuous integration fashion, as to check for potential issues prior to any source code commit.

In the future, this methodology will be applied to other use cases, as an attempt to replicate the improvements in the domains of code maintainability and safety to other robotic systems.

## Acknowledgments

This work is financed by the ERDF European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project POCI-01-0145-FEDER-029583. The research leading to these results has also received funding from the European Unions Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 777096.

## References

1. P. Emanuelsson and U. Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.
2. H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann. Industry 4.0. *Business & information systems engineering*, 6(4):239–242, 2014.
3. M. Quigley, K. Conley, B. 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*, volume 3, page 5. Kobe, Japan, 2009.
4. A. Santos, A. Cunha, and N. Macedo. Static-time extraction and analysis of the ros computation graph. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 62–69. IEEE, 2019.
5. A. Santos, A. Cunha, N. Macedo, R. Arrais, and F. N. dos Santos. Mining the usage patterns of ros primitives. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3855–3860, Sep. 2017.
6. A. Santos, A. Cunha, N. Macedo, and C. Loureno. A framework for quality assessment of ros repositories. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4491–4496, Oct 2016.
7. L. Sha, S. Gopalakrishnan, X. Liu, and Q. Wang. Cyber-Physical Systems: A New Frontier. *2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (sutc 2008)*, pages 1–9, 2008.
8. C. Toscano, R. Arrais, and G. Veiga. Enhancement of industrial logistic systems with semantic 3d representations for mobile manipulators. In A. Ollero, A. Sanfeliu, L. Montano, N. Lau, and C. Cardeira, editors, *ROBOT 2017: Third Iberian Robotics Conference*, pages 617–628, Cham, 2018. Springer International Publishing.