Towards Rapid Composition with Confidence in Robotics Software

Neil A. Ernst
Computer Science
University of Victoria
Victoria, BC
nernst@uvic.ca

Rick Kazman
SEI/CMU and
University of Hawaii
Honolulu, HI
kazman@hawaii.edu

Philip Bianco
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA
pbianco@sei.cmu.edu

ABSTRACT

Robotics software is booming thanks in part to a rich and productive ecosystem around the Robot Operating System. We introduce a military effort to leverage the ROS ecosystem and reduce the challenges in building military robots, called ROS-M. We outline some of the work we have done on the ROS-M initiative, and explain our future directions in analyzing ROS code to balance between rapid adoption and confidence in the component.

KEYWORDS

robotics software, ROS, quality attribute requirements

ACM Reference Format:
Neil A. Ernst, Rick Kazman, and Philip Bianco. 2018. Towards Rapid Composition with Confidence in Robotics Software. In *RoSE'18: RoSE'18:IEEE/ACM 1st International Workshop on Robotics Software Engineering , May 28 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196558.3196567

1 INTRODUCTION

One of the pressing concerns in government and military acquisition is openness to enable 'rapid fielding'. Rapid fielding is a military term that means a focus on rapid capability delivery, e.g., being able to adopt software patches in a matter of days or hours, instead of weeks or months. In particular, recent trends have been to favour systems that have an underlying *open architecture*. Open architecture is a term of art meaning some combination of shared data standards, common infrastructure, interface standards, and so on. Government software acquisition is supporting open architectures to open up the space of possible vendors who respond to requests for proposal, and ultimately to make systems that are much more responsive to change. Past (and most current) systems were largely designed to favour the incumbent, using a contractor's proprietary suite of messaging, data sharing and other key interactions. This made it extremely difficult to change horses midstream (e.g. for non-performance), since potential new contractors had a very difficult time understanding or accessing the underlying code. It also made it very difficult to adopt new innovations (for example, in LIDAR processing). The contractor of record had to evaluate the new technology, then wrap it in a proprietary data exchange layer. In an ideal open architecture, the technical challenge of adoption, that is, making it work with the existing system, is greatly simplified.

In the robotics space, the Robot Operating System (ROS)[8] is an excellent example of an open architecture. The ROS-military (ROS-M) initiative aims to leverage ROS as an open architecture for military applications. However, a big challenge for government use of open architectures like ROS is to balance rapid composition with confidence that the component chosen will behave as expected. ROS's early history is in robotics research, where downside risks are relatively small. The military context is obviously quite different. This obviously includes security concerns, but our use of the word *confidence* is about "correct operation", and therefore also involves questions about performance, availability, maintainability, and other quality attributes. In this paper we describe some of the challenges facing ROS-M, show why it is an interesting example for robotics software, and explain the research we are doing on ROS-M.

2 THE ROS-M ECOSYSTEM

ROS-M is a community development initiative designed to meet the unique needs of military robots. The intent of ROS-M is to add more robust simulators, cyber assurance, and controlled code repositories to ROS. The concept is to leverage the existing rich ROS ecosystem, and in particular, the well-understood middleware and message passing layers. Military needs are different than commercial robotics companies and researchers, of course, so ROS-M adds ways to ensure the software is more reliable. ROS-M promotes code sharing and reuse, especially for components whose distribution is restricted due to national security or export control concerns.

A recent ROS-M working group [2] focused on code quality standards that ROS-M components should respect. For example, there is a range of maturity levels in components in the ROS ecosystem, ranging from Google-supported and field-tested libraries, to cutting-edge research releases with little to no documentation or testing. Integrating a component into a military-grade system requires a thorough understanding of that component's API specifications, gaps in completeness, and ability to meet certain quality attribute requirements. On the other hand, being too restrictive and demanding—for example, insisting on complete maturity model conformance— means narrowing the field of potential components

RoSE'18, May 28 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5760-9/18/05…$15.00
https://doi.org/10.1145/3196558.3196567

and increasing the time before a component can be fielded. ROS-M must therefore balance between complete confidence and rapid fielding. We are finding interesting research challenges abound in understanding this tradeoff.

3 COMPOSING ROS SYSTEMS

The tradeoff between confidence and rapid adoption is at its heart a *composition problem*. ROS systems are composed of many, relatively independent sub-systems, such as planners and vision processing elements [8]. The challenge is how to integrate a given component into the larger set of components. While there are numerous technical challenges to overcome (e.g. understanding the message format, and updating other components with new capability), we focus on the component assessment challenge. Solving this challenge means giving the integrator confidence that the candidate component will meet the system's key quality attribute requirements, without posing an undue risk to the overall system.

Typically defense software runs in tightly controlled environments where all components are directly managed by the system integrator or prime contractor. This is at odds with modern software development practices that increasingly rely on composition of components from various sources, leading to two problems: authority to deploy a system is slow, cumbersome, and hinders rapid system fielding/release/update [4]. Secondly, adopting modern software practices means accepting loss of some control; no one entity will control all components. This is particularly true in software ecosystems like ROS-M [10].

4 OUR APPROACH

Our scenario is an engineer working with a unmanned ground vehicle (UGV) using ROS-M. She wants to adopt a ROS package for doing SLAM (simultaneous localization and mapping) [9], and has (at least) three ROS 2.0 choices: Google's Cartographer[1], Gmapping[2], and MRPT's mrpt_rbpf_slam[3]. To choose one of these for her system, she needs to understand three things. First, she must know about its relevant quality attribute indicators (e.g., code maintainability, API documentation, performance, accuracy). Here, relevance means "with respect to her vehicle and task". Secondly, she needs to understand the project health indicators for that package (number of contributors, licence, number of recent commits, documentation status, package dependency chains). Project health will help her (and the maintenance team) understand how likely this component will be supported in the future, among other things. Finally, she needs to understand how, and where, the component could affect her system, in other words, how it will be integrated with what she already has.

Understanding how a component interacts with her system is necessary to appropriately mitigate any risks to the system's ability to fulfil its mission. In this case, we would need evidence for what type of data the SLAM package sends and expects; how errors are handled; and what CPU and network load it might impose. An example source of evidence would be the results of a network traffic analysis using a sample task in a lab setting, showing mean/peak data volumes.

Rather than focus on tools to collect information, we see the research challenges as:

(1) correctly and accurately identifying and modeling the set of components and interactions in a given system;
(2) identifying and measuring relevant robotics and operational environment *indicators* for system quality attributes;
(3) aggregating these indicators to support effective trade-off analysis.

Since documentation seldom represents ground truth, our approach is based on direct observations and analyses of the source code (where available) and the running software and related artifacts (e.g., file system changes).

5 METHODOLOGY AND PROGRESS TO DATE

Our approach is to produce a 'Consumer Reports' style matrix showing the capabilities of each component against a set of important indicators. Figs. 1 and 2 shows what such an output might resemble.

To create reports like this, we follow the following steps:

(1) Define a generic set of component indicators to discover. Start with existing quality attribute catalogs to determine what kinds of indicators are relevant to the scenario. Review observable resources for systems running on *nix (e.g., system process calls, network layers, heap memory, and file system) (bottom-up). This activity is independent of ROS.
(2) Identify a set of data collection mechanisms for each indicator. Find candidate measurement tools and approaches (e.g., profilers, static analysis tools, and manual code review).
(3) Identify tool inputs: source, binaries, commit history, issues, problem reports, KPPs.
(4) Apply analyses to each candidate component (e.g. use a component's internal test harness, run tools over collected project artifacts)
(5) Aggregate data: use expert input to elicit weighting criteria (e.g., peak load, design hotspots, vulnerability collection)
(6) Validate on open source corpus and with industry stakeholders.

We have begun by creating a project workbench based around a Gazebo simulation environment with a commercial UGV system, Husky, and the three SLAM components above (Cartographer, GMapping, and mrpt_rbpf_slam). Fig. 3 shows the details.

5.1 Preliminary Results

We identified the following project health indicators (Table 1) and quality attribute indicators (Table 2). Clearly the two candidates (the grey columns) are quite different, and pose some interesting dilemmas for the hypothetical integrator. For example, the code base for Cartographer is over twice as large, yet the code is updated frequently in comparison to Gmapping. With respect to our quality attribute indicators, some common metrics for maintainability suggest Gmapping is slightly simpler to maintain. A major challenge is to determine exactly what constitutes the component boundary:

[1] https://opensource.googleblog.com/2016/10/introducing-cartographer.html
[2] http://wiki.ros.org/gmapping
[3] https://www.mrpt.org/list-of-mrpt-apps/application-rbpf-slam/

Properties	Indicator	Measurement approach or tool (example)	Ease of collection
Community Health	# developers	code-maat	Simple, with project history
	Commit frequency	code-maat	Simple with repo access
	Bug frequency	code-maat	Simple with repo access
Codebase	LOC	cloc	Simple, needs source
	License	gh-licence	Simple
Team	Countries of origin	Linked-in/social metrics	Moderate
	Previous experience	Social metrics	Hard
(Others)	...		

Figure 1: Sample Project Health Indicators

Qualities	Indicator	Measurement approach or tool (example)	Ease of collection
Maintainability	# imported libraries	cvs_analy, sonatype	Simple
	Level of coupling	DV8	Simple
	Architecture flaws	DV8	Moderate
Performance	CPU load, peak	gprof	Moderate
	Disk access	gprof	Simple
	Memory use	valgrind	Moderate
Security	# CVE violations	fortify, coverity, SCALe	Moderate
(Others)	...		

Figure 2: Sample Project QA Indicators

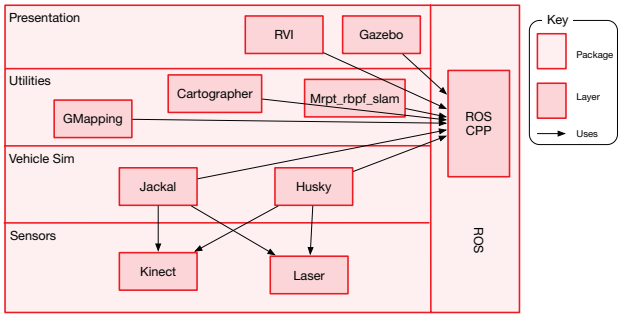

Figure 3: Current workbench

Gmapping, for example, has a ROS wrapper around a third-party library (along with numerous dependencies on core packages).

5.2 Validation

To validate our eventual conclusions (i.e., "choose component X"), we plan for the following. To test our **Tool Correctness** (i.e., internal validity), we will measure against known component measures (for example, third party benchmarking results) and aggregation functions. To ensure we are not overfitting to a particular type of component, we will use a train/test concept, where we 'train' our process on one component, then test it on new ROS component types to measure the bias and variance of our functions. To test if we **Increase Confidence** in adopting a particular component (i.e., by using the scoreboard), we will ask ROS experts for past problematic indicators and measures (e.g. use of an $O(n^2)$ algorithm in a component). We will then inject some of these problems into the appropriate components, and see if our scorecard flags over 90% of the injected problems. We also plan to conduct a stakeholder survey, and aim to show a doubling of confidence, e.g. from 2 to 4, on a 5 point Likert scale.

To measure if our scorecard leads to **Reduced Decision Time**, we will baseline current approaches, and assess if our scorecard result (post-setup) achieved 1 sigma faster than a developer acting without our tool. Finally, to assess **Operational Validity**, we will demonstrate the tool to our operational stakeholders and collect feedback.

6 DISCUSSION

Building an accurate scorecard relies on tools that correctly characterize the quality attribute characteristics of a given component, and a suitable testbed in which to deploy the component. While static and dynamic analysis tools are incomplete (e.g., the fact that a component is well behaved during profiling may not be indicative of its true behavior at run-time), the generated data will be a significant improvement over current analyses. Future work includes extending the automated analysis into automatically finding 'safe' components with little human interaction, in an autonomous software system. Potential future application could be to inform policy makers of what level of insight, analysis, and risk is expected with the use of external components.

A key research question is what aggregate function to use to create the final component score from the individual indicator scores. Possible choices include *equal weighting*: sum all indicators, normalize to 0..1; another choice is to normalize to industry baselines and categorize as high, medium, low (e.g., for SLOC). Our current approach is to give each indicator a weight w based on our model of priority. This model is derived from interviews and short analytical hierarchy process (AHP) prioritization exercises. We then normalize metrics as necessary for different denominators such as

Property	Indicator	Tool	Cartographer	Gmapping
Codebase	LOC	SLOCcount	17.5 SKLOC C++	7.6 SKLOC C++
	License	Manual	Apache 2	BSD-3
	Commit frequency	Code-Maat	87	3
Team	# developers	Code-Maat	13	1

Table 1: Project Health Indicators

Quality Attribute	Indicator	Tool	Cartographer	Gmapping
Performance	CPU usage	gprof	*in progress*	*in progress*
	% memory used	valgrind	*in progress*	*in progress*
Maintainability	Arch. flaws	DV8	0	2
	Level of coupling	DV8	17.1%	12%
	Total Dependencies	Understand	564	156

Table 2: Project Quality Attribute Indicators

SLOC, programming language (so that Python code is not penalized for verbosity, for example). Next, we create customizable templates, based on stakeholder interviews. An example aggregation template could be

$$(wM_1 * wM_2) + 2(wP_1 * log(wP_2)) + 3(wS_1)$$

Where M, P, S are Maintainability, Performance, Security indicator instances, adjusted by some weight w. This particular example weights Security three times as important as maintainability.

7 RELATED WORK

Other good examples of open architectures in the US government include the US Navy's Future Airborne Capability Environment (FACE[4]), and the Air Force's Open Mission Systems initiative. The ROS-Industrial initiative is similar to ROS-M but with a commercial focus. The complex/safety-critical systems space is awash in standards focusing on interoperability and safety.

Software composition research has its origins in component-based software engineering [7] and Open Source Software (OSS) and Commercial Off the Shelf (COTS) adoption [5]. It is important to leverage advances in those areas, while retaining a ROS flavour. We feel major differences include the shared set of middleware, wide range of component quality, and potential for run-time adjustment.

Currently, software systems used in military settings go through a risk assessment process before being given authority to operate. These risk-based approaches are an improvement over compliance checking, yet as Fabius notes, "[there is a] lack of supporting tools to help determine which safeguards are most appropriate [4]". Recent research has focused on shallow enumeration of all dependencies using build files [1], identifying versioning problems in dependencies [3], or identifying external dependencies from signatures [6]. IonChannel with the US National Geospatial Agency [1] simply enumerates dependencies. These approaches do not discriminate among different types of dependencies and fail to capture others that are relevant to risk analysis.

[4]https://www.opengroup.us/face/

8 CONCLUSION

We have presented ROS-M, an initiative to leverage the open architecture of the ROS ecosystem to improve adoption time for military robotics. We explained ROS-M, and then introduced our research approach for evaluating possible components for adoption into an existing system. The adoption decision must balance component capabilities against possible risks. This decision can be supported by understanding of how a component scores on a set of aggregated quality indicators.

REFERENCES

[1] Sebastian Benthall, Travis Pinney, JC Herz, and Kit Plummer. "An Ecological Approach to Software Supply Chain Risk Management". In: *15th Python in Science Conference*. 2016.

[2] Jonathan Chu. *Army Robotics in the Military*. 2017. URL: https://insights.sei.cmu.edu/sei_blog/2017/06/army-robotics-in-the-military.html (visited on June 12, 2017).

[3] J. Dietrich, K. Jezek, and P. Brada. "What Java developers know about compatibility, and why this matters". In: *Empirical Software Engineering* (2016), p. 1371. DOI: 10.1007/s10664-015-9389-1.

[4] Jennifer Fabius and Richard Graubart. *Beyond Compliance—Addressing the Political, Cultural and Technical Dimensions of Applying the Risk Management Framework*. Tech. rep. PR-14-3551. MITRE, 2014. URL: https://www.mitre.org/sites/default/files/publications/pr-14-3551-beyond-compliance-applying-risk-management-framework.pdf.

[5] Ian Gorton, Anna Liu, and Paul Brebner. "Rigorous evaluation of COTS middleware technology". In: *Computer* 36.3 (2003), pp. 50–55.

[6] Takashi Ishio, Raula Gaikovina Kula, Tetsuya Kanda, Daniel M. German, and Katsuro Inoue. "Software Ingredients: Detection of Third-party Component Reuse in Java Software Release". In: *Proceedings of the International Working Conference on Mining Software Repositories*. 2016, pp. 339–350. DOI: 10.1145/2901739.2901773.

[7] Wojtek Kozaczynski and Grady Booch. "Component-based software engineering". In: *IEEE software* 15.5 (1998), p. 34.

[8] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[9] Søren Riisgaard and Morten Rufus Blas. *SLAM for Dummies: A Tutorial Approach to Simultaneous Localization and Mapping*. Tech. rep. MIT, 2005. URL: https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-412j-cognitive-robotics-spring-2005/projects/1aslam_blas_repo.pdf.

[10] B. Sadowski. *Shaping the Future: Army Robotics and Autonomous Systems*. Tech. rep. National Defense Industry Association, 2016. URL: http://www.dtic.mil/ndia/2016GRCCE/Saowski.pdf.