Deliverable 2: Safety Verification Techniques

Abstract

This document will encompass the functional safety considerations regarding the first VORTEX demonstrator. This will allow for a centralized view of the techniques and developments for verifying safety properties in ROS, and guide the design of the developed components.

Chapter 1

Introduction

1.1 Structure

Chapter 2 presents an overview of safety verification techniques for ROS applications. Section 2.1 defined scope of this overview, which is effectively presented in Section 2.2. Chapter 3 will present the strategy to be employed for the analysis of the demonstrator. Chapter 4 finally applies that strategy to the demonstrator, building on the requirements specified in document D1.

Chapter 2

State-of-the-art

2.1 Context

The architecture of a ROS application is defined by its *Computation Graph*. It consists of nodes, programmed in general-purpose programming languages (typically C++ or Python), that communicate through message passing (usually through a publish/subscribe paradigm, but also supporting remote procedure calls). This architecture is determined by launch files that specify which nodes are launched, with each parameters, and may remap communication topics. In this context, ROS analyses, and in particular those for safety verification, can be divided into two main axes:

Intra-node analysis Techniques that focus on the behavior of individual ROS nodes. Since nodes are programmed in general-purpose languages, such analyses must be as powerful as those tailored for general-purpose software. Thus, they are usually adapted to the ROS domain from state-of-the-art techniques.

System-wide analysis Techniques that focus on the behavior of the whole system, including the launched configurations and the interaction of the nodes. These techniques are usually tailored specifically for the ROS domain, as they must consider the particular languages and primitives used in the ROS environment.

Scope This document aims to report ROS-specific approaches to promote the safety of software components. We aim at techniques that minimally disrupt the standard ROS software development process. Thus, techniques that require a complete overhaul of this process (e.g., developing in a formal specification language that will then communicate with other ROS components) are outside the scope of this report. The application of general-purpose verification techniques to the ROS domain (e.g., state-of-the-art analyses for C++) are also outside the scope of this report, and are the target of a parallel deliverable (D[?]).

2.2 Approaches to ROS verification

2.2.1 Quality in the ROS Development Process

The open nature of ROS has not yet led to the development of adequate mechanisms to ensure dependability, resulting in packages with varied quality and hindering the future certification and marketability of ROS robots. Initiatives like the ROS Industrial consortium[1] and the H2020 ROSIN project[2] are pushing to promote the quality and reliability of ROS code[3]. Nonetheless, currently this still amounts to the proposal of a set of guidelines for ROS software development, which include unit testing (for C++, based on Google's gtest), documentation, code style guides (for C++, derived from Google's C++ style guide) and thresholds for internal quality metrics (regarding complexity and maintainability). The community is working on providing automatic feedback regarding these for indexed packages during continuous integration. Studies [21, 18] show that the adoption of style guides and concerns with internal quality are still not a major concern of the community.

2.2.2 Formal Approaches to ROS

Internal software quality While not allowing the direct verification of safety properties, identifying faulty patterns promotes the internal code quality of a software product and reduces the chances of encountering problems later in the development process. These techniques are mostly unaware of the configuration of the Computation Graph.

There is some ROS-specific work on this area, including tools that automate the detection of violations. Santos et al. [21] proposed HAROS, a static analysis tool tailored for C++ ROS packages to measure and report quality metrics (like complexity metrics) and violations to coding standards

[1]`https://rosindustrial.org`
[2]`http://rosin-project.eu`
[3]`http://wiki.ros.org/Quality`

(including the C++ ROS style guide). The framework is plug-in based, allowing users to encode their own rules. It is currently being integrated in the continuous integration process for indexed ROS packages. Pichler et al. [18] also proposed a tool, rosmap to inspect the dependency graph and the repository quality of ROS packages, having it applied to a large set of (indexed and non-indexed) ROS packages. Here the notion of quality is mainly based on repository information (like stars, issues or contributors). Come et al. [4] propose a technique, associated with the tool Pangolin, to detect faulty programs through the specification of patterns in first-order temporal logic that query the AST of C++ ROS code. The patterns are user-specified but the authors explore a set of patterns useful for ROS code.

Software verification Software verification techniques are used to verify concrete software products, rather than software models (although these can be extracted as an intermediate step for the subsequent analyses). Static source code analysis, in particular, act directly on the developed code, imposing minimal disruption on the development process. Cortesi et al. [5] explored the potential of using standard static analysis techniques in ROS, identifying the lack of standardization, the abstraction gap and the complexity of the components as the main challenges. Extracting the ROS Computation Graph in static time is itself challenging, as it emerges from the calls to ROS primitives in general-purpose code. Since these techniques actually analyze the node code, they would be well suited for intra-node analysis, but they are also required to infer the Computation Graph back from code.

Existing approaches of this kind usually address specific classes of problems and are still mostly concerned with reverse engineering parts of the architecture. They mostly focus on ROS applications developed in C++. Purundare et al. [19] propose a technique for the analysis of the dependency between ROS nodes. The C++ ROS code is statically analyzed for calls to ROS communication primitives, and under which conditions they are called. These can then be used to identify which components of the ROS application are affected by changes in particular nodes. Sharma et al. [23] propose a similar approach but focused on changes to the rate of message publication of nodes, resulting in more accurate report of affected nodes. The technique also statically analyses C++ ROS code, identifying nodes that are sensible to the rate of incoming data. Ore et al. [17, 16] propose a tool, PhrikyUnits, to detect inconsistencies between physical units. They identify standard libraries where known physical units are used, and then analyze ROS C++ code where such values are manipulated for possible inconsistencies. This work has also been extended to support probabilistic inference [12]. In a topic relevant to our application domain, Muscedere et al. [14] proposed one such static analysis technique for automotive software developed in ROS. The focus here is on analyzing feature interactions, and a tool is proposed to extract relevant information from C++ ROS code, over which patterns for feature interaction symptoms are tested. These patterns are user-specified, but the authors suggest some patterns relevant for the automotive domain. Santos et al. [22] have more recently proposed a meta-model for the ROS Computation Graph and a general approach for its extraction from ROS C++ applications. This model in then amenable to be inspected and analyzed, in particular through a query language that allows the developer to detect undesirable issues. The technique is integrated in HAROS.

There have been some studies on attempting to verify safety properties of ROS applications using theorem provers, namely Coq by Anand and Knepper [2], and model checkers, namely SPIN by Webster et al. [25] and UPPAAL by Halder et al. [7]. However, these were mainly exploratory studies based on manual codifications of the system under study in the target language.

Model-based approaches An alternative approach to software development is to design an application in a higher-level language, more amenable to verification, from which low-level code can be generated. This report, in particular, focuses on such frameworks whose target language is ROS-based. This kind of approaches do nonetheless affect the development process, and often require expertise on formal methods. The classes of supported properties is dependent on the expressiveness of the modeling language, which is often at a high-level of abstraction.

Hochgeschwender et al. [8] propose a DSL for model-based development, over which OCL properties can be specified and verified. Variability is a key feature of the DSL, and the elements of the model map to ROS nodes that may or not be launched in particular configurations; ROS launch files are then generated by the framework. Meng et al. [13] propose another such approach, ROSGen, where ROS glue code in C is generated using Coq. This allows safety properties to be (interactively) proved for the generated code. Wong et al. [26] propose the synthesis of verified controllers, in the shape of finite state machines, into ROS nodes that integrate the remaining ROS application. Wang et al. [24] propose to model ROS applications in UPAAAL, over which timed temporal logic properties can be verified. A translation into C++ ROS code is then proposed.

Runtime monitoring Runtime monitors are a popular approach in robotic systems, and some ROS-specific approaches have been proposed. Although these can not guarantee the correct behavior of the system, they can guarantee the avoidance of catastrophic failures. As they act externally to nodes, they are better suited to handle inter-node properties.

Adam et al. [1] propose a DSL for the high-level specification of safety rules over a provided system model, from which (Python) ROS monitors are then generated. Rules can be specified over nodes and topics, and can have temporal conditions. With a similar goal, Huang et al. [9] also propose a generator of a C++ ROS monitor nodes from a formal specification language. Jiang et al. [11] propose an approach where the invariants to be monitored are inferred from the traces of successful runs, rather than specified by the developers. The templates for invariant inference are tailored for the robotic domain, and the user can then assign recovery actions when the generated monitors detect an issue.

Testing Little work has been done on advanced testing techniques tailored for ROS. In particular, integration testing of the complete ROS application (in contrast to unit testing) is still challenging.

Bihlmaier and Wörn [3] proposed to bring modern testing techniques into ROS by incorporating simulated environments into the process. Ernits et al. extend this approach for integration testing following a model-based testing approach [6]. The expected behavior (focusing on localization and navigation) is modeled in UPPAAL, which is then tested against Python ROS applications. Santos et al. [20] propose a property-based testing approach where test scripts are automatically generated and executed from the specification of properties over the whole ROS system.

Chapter 3

Safety Verification Strategy

This chapter reports on the strategy defined for the safety verification of ROS applications in the context of the demonstrator. Alternative approaches, better-suited for different classes of problems, are identified when relevant.

System-wide analysis The techniques for analyzing system-wide properties of ROS applications identified in our study focused on detecting particular classes of issues. More general safety analyses will require a formalism that is sufficiently expressive and flexible to encode complex architectures with variability, the behavior of the individual nodes in an adequate level of abstraction, and interesting system-wide properties. Moreover, it should ideally be accompanied by automated analysis tools to ease V&V tasks. In this demonstrator we propose to use Alloy [10] and its Analyzer for this purpose, a formal specification language based on first-order relational logic backed by an automated model finder. Alloy has proven to be well-suited for this kind of analysis (e.g., in the verification of dependability cases [15]). The main limitation that we envision from this formalism is its limited support for real-time properties. Other tools, like UPPAAL based on timed automata, could be used for that purpose, but would lack the flexibility needed to encode architectures with the expected level of complexity.

[TBD, which classes of system-wide properties do we want to address?]

We propose to specify an Alloy model that encodes the general architecture and behavior of ROS applications, which could then be instantiated for particular ROS applications like that of the demonstrator. The animation functionalities of the Analyzer can be used in a first stage to validate the design, and the verification functionalities in later stages to actually check safety or liveness properties.

Intra-node analysis Our study has not identified ROS-specific techniques for analyzing the behavior of individual nodes. Thus, for the context of this demonstrator, we advocate their verification using state-of-the-art techniques for general-purpose source code (in particular C++, the language used in the demonstrator). These may include static analysis techniques for critical nodes (reported in a parallel deliverable, D[?]) or advanced testing techniques for less-critical nodes. Runtime monitors can additionally be deployed to guarantee the safety of non-verified components.

The behavior of the nodes must nonetheless be specified for two purposes. First, as the functional properties that such general-purpose analysis will verify. Second, they are used as assumptions for the analysis of the system-wide properties. [In the former context, what do we want to verify, what's the level of detail?] In the latter, a higher-level version of the properties is to be expected, addressing the API of the nodes (message publication/reception level).

Strategy implementation From the identified frameworks, HAROS proved to be the one best integrated in the ROS development process. Its plug-in based architecture will allow us to introduce new analyses (or wrap existing ones) as needed, which HAROS may apply to the project and report feedback in continuous integration. The extraction of the Computation Graph from ROS source code also allows the deployment of analysis plug-ins at the architectural level, as expected for the system-wide analysis. A simple DSL for the specification of architectures (launched nodes and communication channels) is provided if the code is not fully available, enabling the validation of requirements during the design phases. Another DSL is provided for the specification of the behavior of ROS applications at the level of message publication/reception level (currently used for property-based testing) which we expect to explore both for the specification of the node behavior and the desirable system-wide properties.

Concretely, for the system-wide analysis of the system, we envision the implementation of the following strategy. During system design (Figure 3.1):

- formalize the architecture of the demonstrator in the DSL provided by HAROS
- if relevant, define style rules over such architectures that HAROS will check
- formalize the behavior of the individual nodes of the demonstrator in the DSL provided by HAROS

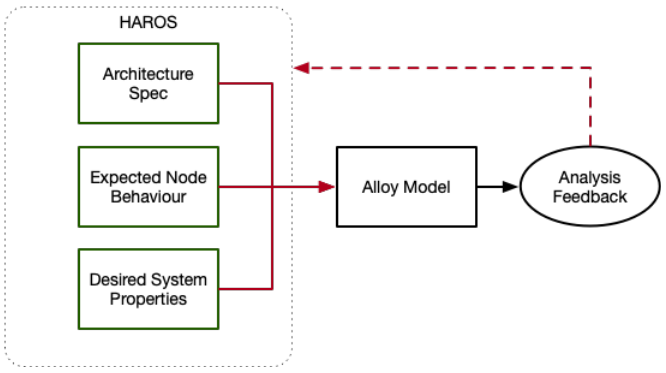

Figure 3.1: Verification strategy at design time (in red artifacts missing from the strategy, in green artifacts specific to each product).

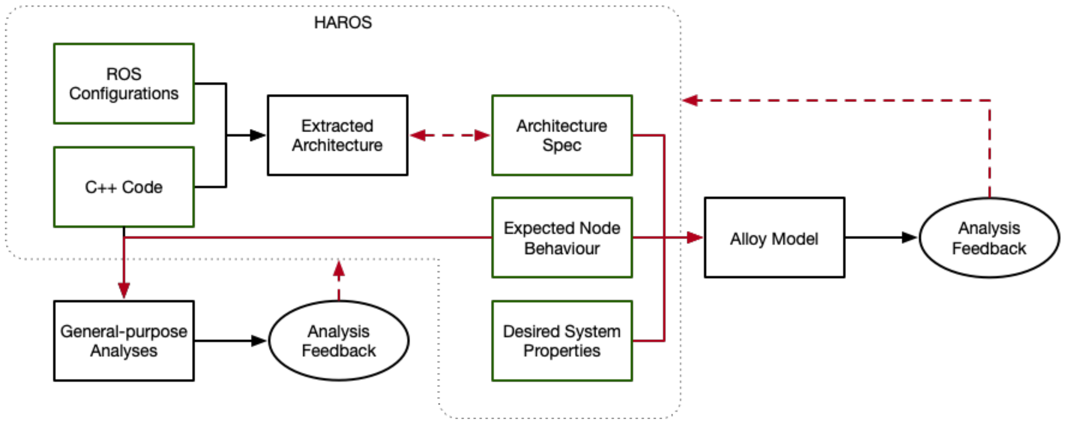

Figure 3.2: Verification strategy at development time (in red artifacts missing from the strategy, in green artifacts specific to each product).

- formalize the desirable system-wide properties of the demonstrator in the DSL provided by HAROS
- translate these 3 artifacts into an Alloy model, which can automatically check whether the properties hold for the design.

The system-wide properties will hold for a ROS application if the implemented architecture matches the one specified, and the implemented nodes act according to the specified behavior. Thus, during system development (Figure 3.2):

- extract the architecture directly from the source code and configuration files
- validate the extracted architecture against the specification

- check whether implemented nodes follow the specification using general-purpose formal analysis techniques.

The need for extensions to both the architecture and behavioral DSLs may be identified and addressed during implementation of this process. The formalization of ROS applications in Alloy, and the subsequent translation of the DSLs into it, will need to be developed as well. [TBD: which intra-node analysis will be employed, and how to integrate them with the node specs?]

Chapter 4

Design of ROS Demo #1

This section reports on the application of the strategy defined in Chapter 3 to the demonstrator.

4.1 Use Case

4.2 System Formalization

This section presents an overview of the developed formalization of the architecture of the ROS application and the behavior of the individual nodes, building up on the requirements specified in document D1. The system design was formalized using 2 DSLs supported by HAROS. An Alloy model for the behavior of ROS applications was developed for its analysis (Figure 4.1) [To do: roughly explain], which is extended by the translation of the specifications of the demonstrator. This allows for its validation and subsequent verification (addressed in the following sections). The complete artifacts are available elsewhere ([point to repo]).

Architecture Figure 4.2 depicts the formalization of the two possible configurations in the simple architectural DSL provided by HAROS. [currently, just one, offline from the kitti dataset, the live one is still under specification] A snippet of its encoding into Alloy is depicted in Figure 4.3 [To do: roughly explain].

Node behavior Figure 4.4 depicts part of the formalization of the node behavior. A snippet of its encoding into Alloy is depicted in Figure 4.5 [To do: roughly explain]. At this stage, the model can be simulated to inspect possible runs. A prefix of one such run is depicted in Figure 4.6.

```
abstract sig Node {
  subscribes : set Topic,
  advertises : set Topic,
  var inbox : set Message,
  var outbox : set Message
}
abstract sig Topic {}
sig Message {
  topic : one Topic,
  value : one Data,
  stamp : one Stamp
}
sig Data, Stamp {}

fact Messages { always {
  all n : Node {}
    n.inbox.topic in n.subscribes
    n.outbox.topic in n.advertises
  }
  all m : Message | m in Node.outbox implies
    all n : subscribes.(m.topic) | after eventually m in n.inbox
  all m : Message | m in Node.inbox implies
    some n : advertises.(m.topic) | before once m in n.outbox
  ...
  }
}
```

Figure 4.1: Snippet of the base Alloy model for ROS applications.

4.3 Safety Properties

4.4 Safety Verification

4.5 Optimizations and Extensions

```
dataset:
  /input:
    advertise:
      kitti/velo/pointcloud:
        sensor_msgs/PointCloud2
      kitti/oxts/imu:
        sensor_msgs/Imu
      kitti/oxts/gps/fix:
        sensor_msgs/NavSatFix
      kitti/oxts/gps/vel:
        geometry_msgs/TwistStamped
  /processor:
    advertise:
      processed/tracklets_marker:
        visualization_msgs/MarkerArray
      processed/pointcloud:
        sensor_msgs/PointCloud2
      processed/imu:
        sensor_msgs/Imu
      processed/gps/fix:
        sensor_msgs/NavSatFix
      processed/gps/vel:
        geometry_msgs/TwistStamped
    subscribe:
      kitti/velo/pointcloud:
        sensor_msgs/PointCloud2
      kitti/oxts/imu:
        sensor_msgs/Imu
      kitti/oxts/gps/fix:
        sensor_msgs/NavSatFix
      kitti/oxts/gps/vel:
        geometry_msgs/TwistStamped
  /compressor:
    advertise:
      compressor/frames:
        ????
    subscribe:
      processed/tracklets_marker:
        visualization_msgs/MarkerArray
      processed/pointcloud:
        sensor_msgs/PointCloud2
      processed/imu:
        sensor_msgs/Imu
      processed/gps/fix:
        sensor_msgs/NavSatFix
      processed/gps/vel:
        geometry_msgs/TwistStamped
  /output:
    subscribe:
      compressor/frames:
        ????
```

```
live:
  /input:
    advertise:
      ...
      ...
      ...
  /tracklets:
    advertise:
      ...
  /processor:
    advertise:
      ...
      ...
      ...
    subscribe:
      ...
      ...
      ...
  /compressor:
    advertise:
      ...
    subscribe:
      ...
      ...
      ...
      ...
  /output:
    subscribe:
```

Figure 4.2: Architecture formalization under two alternative configurations.

```
one sig KittiPC, KittiImu, KittiFix, KittiVel, ProcessedPC, ProcessedImu,
  ProcessedFix, ProcessedVel, ProcessedTracklet, Compressed extends Topic {}

one sig Input extends Node {} {
  subscribes = none
  advertises = KittiPC+KittiImu+KittiFix+KittiVel
}

one sig Processor extends Node {} {
  subscribes = KittiPC+KittiImu+KittiFix+KittiVel
  advertises = ProcessedPC+ProcessedImu+ProcessedFix+ProcessedVel+ProcessedTracklet
}

one sig Compressor extends Node {} {
  subscribes = ProcessedPC+ProcessedImu+ProcessedFix+ProcessedVel+ProcessedTracklet
  advertises = Compressed
}

one sig Output extends Node {} {
  subscribes = Compressed
  advertises = none
}
```

Figure 4.3: Snippet of the generated Alloy model for the architecture of the `dataset` configuration.

```
/input:
    - must publish on kitti/velo/pointcloud at 10 hz
    - must publish on kitti/oxts/imu at 10 hz
    - must publish on kitti/oxts/gps/fix at 10 hz
    - must publish on kitti/oxts/gps/vel at 10 hz
/processor:
    - must publish on processed/pointcloud up to 1s after
        sync receive on kitti/velo/pointcloud,
                     on kitti/oxts/imu,
                     {linear.x in -5 to 30} on kitti/oxts/gps/vel
                     {latitude in -90 to 90, longitude in -180 to 180} on kitti/oxts/gps/fix,
    - must publish on processed/imu up to 1s after
        sync receive on kitti/velo/pointcloud,
                     on kitti/oxts/imu,
                     {linear.x in -5 to 30} on kitti/oxts/gps/vel
                     {latitude in -90 to 90, longitude in -180 to 180} on kitti/oxts/gps/fix,
    - must publish on processed/gps/fix up to 1s after
        sync receive on kitti/velo/pointcloud,
                     on kitti/oxts/imu,
                     {linear.x in -5 to 30} on kitti/oxts/gps/vel
                     {latitude in -90 to 90, longitude in -180 to 180} on kitti/oxts/gps/fix,
    - must publish on processed/gps/vel up to 1s after
        sync receive on kitti/velo/pointcloud,
                     on kitti/oxts/imu,
                     {linear.x in -5 to 30} on kitti/oxts/gps/vel
                     {latitude in -90 to 90, longitude in -180 to 180} on kitti/oxts/gps/fix,
    - must publish on processed/tracklets_marker up to 1s after
        sync receive on kitti/velo/pointcloud,
                     on kitti/oxts/imu,
                     {linear.x in -5 to 30} on kitti/oxts/gps/vel
                     {latitude in -90 to 90, longitude in -180 to 180} on kitti/oxts/gps/fix,
    - only publish {linear.x in -5 to 30} on processed/gps/vel
    - only publish {latitude in -90 to 90, longitude in -180 to 180} on processed/gps/fix
/compressor:
    - must publish on compressed/frame up to 1s after
        sync receive on processed/pointcloud, on processed/imu,
                     on processed/gps/fix, on processed/gps/vel,
                     on processed/tracklets_marker
```

Figure 4.4: Node behavior formalization.

```
one sig GoodVel in KittiVel+ProcessedVel {}
one sig GoodFix in KittiFix+ProcessedFix {}

fact Processor_Behavior {
  always {
    all m : Processor.outbox | m.topic in ProcessedPC implies
      once (some n1,n2,n3,n4 : Processor.inbox {
        n1.topic = KittiPC and n1.value = m.value
        n2.topic = KittiFix and n2.stamp = n1.stamp
        n3.topic = KittiVel and n3.stamp = n1.stamp
        n4.topic = KittiImu and n4.stamp = n1.stamp
      })
    ...
    all m : Processor.outbox | m.topic in ProcessedFix implies m.topic in GoodFix
    all m : Processor.outbox | m.topic in ProcessedVel implies m.topic in GoodVel
  }
}

fact Compressor_Behavior {
  always all m : Compressor.outbox |
    once (some n : Compressor.inbox | m.value = n.value)
}
```

Figure 4.5: Snippet of the generated Alloy model for the architecture of the `dataset` configuration.

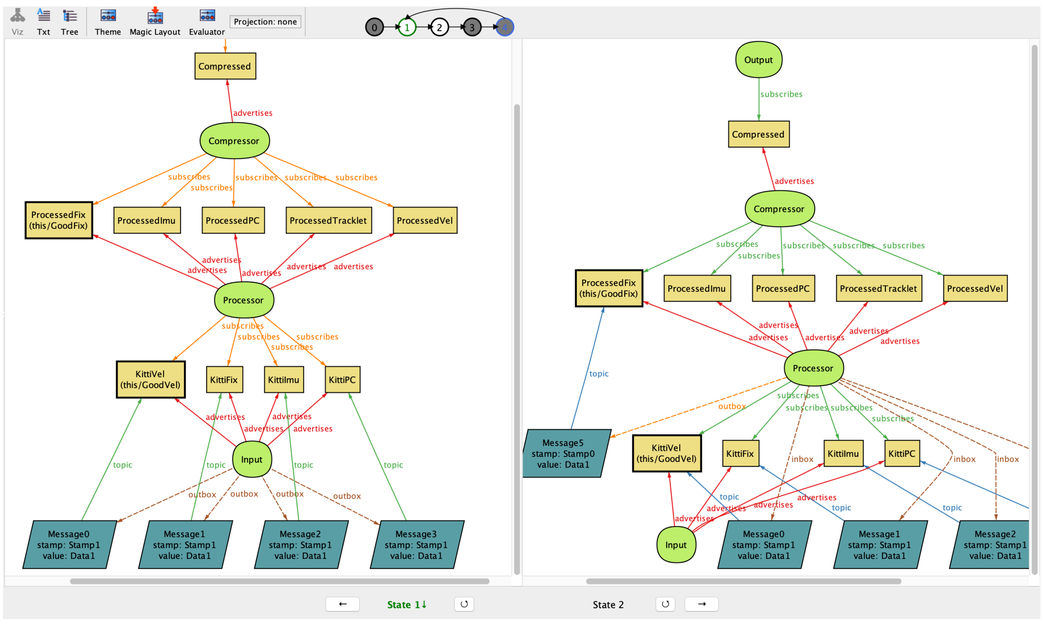
Viz
Txt
Tree
Theme
Magic Layout
Evaluator
Projection: none
Compressed
advertises
Compressor
subscribes
ProcessedFix (this/GoodFix)
ProcessedImu
ProcessedPC
ProcessedTracklet
ProcessedVel
Processor
KittiVel (this/GoodVel)
KittiFix
KittiImu
KittiPC
Input
topic
outbox
Message0 stamp: Stamp1 value: Data1
Message1 stamp: Stamp1 value: Data1
Message2 stamp: Stamp1 value: Data1
Message3 stamp: Stamp1 value: Data1
Output
inbox
Message5 stamp: Stamp0 value: Data1
Message0 stamp: Stamp1 value: Data1
Message1 stamp: Stamp1 value: Data1
Message2 stamp: Stamp1 value: Data1
State 1
State 2

Figure 4.6: Example simulation run.

Bibliography

[1] Sorin Adam, Morten Larsen, Kjeld Jensen, and Ulrik Pagh Schultz. Towards rule-based dynamic safety monitoring for mobile robots. In *SIMPAR*, volume 8810 of *LNCS*, pages 207–218. Springer, 2014.

[2] Abhishek Anand and Ross A. Knepper. ROSCoq: Robots powered by constructive reals. In *ITP*, volume 9236 of *LNCS*, pages 34–50. Springer, 2015.

[3] Andreas Bihlmaier and Heinz Wörn. Robot unit testing. In *SIMPAR*, volume 8810 of *LNCS*, pages 255–266. Springer, 2014.

[4] David Come, Julien Brunel, and David Doose. Improving code quality in ROS packages using a temporal extension of first-order logic. In *IRC*, pages 1–8. IEEE, 2018.

[5] Agostino Cortesi, Pietro Ferrara, and Nabendu Chaki. Static analysis techniques for robotics software verification. In *ISR*, pages 1–6. IEEE, 2013.

[6] Juhan P. Ernits, Evelin Halling, Gert Kanter, and Jüri Vain. Model-based integration testing of ROS packages: A mobile robot case study. In *ECMR*, pages 1–7. IEEE, 2015.

[7] Raju Halder, José Proença, Nuno Macedo, and André Santos. Formal verification of ROS-based robotic applications using timed-automata. In *FormaliSE@ICSE*, pages 44–50. IEEE, 2017.

[8] Nico Hochgeschwender, Luca Gherardi, Azamat Shakhimardanov, Gerhard K. Kraetzschmar, Davide Brugali, and Herman Bruyninckx. A model-based approach to software deployment in robotics. In *IROS*, pages 3907–3914. IEEE, 2013.

[9] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon M. Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu. ROSRV: runtime verification for robots. In *RV*, volume 8734 of *LNCS*, pages 247–254. Springer, 2014.

[10] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT, 2nd edition, 2012.

[11] Hengle Jiang, Sebastian G. Elbaum, and Carrick Detweiler. Inferring and monitoring invariants in robotic systems. *Auton. Robots*, 41(4):1027–1046, 2017.

[12] Sayali Kate, John-Paul Ore, Xiangyu Zhang, Sebastian G. Elbaum, and Zhaogui Xu. Phys: Probabilistic physical unit assignment and inconsistency detection. In *ESEC/SIGSOFT FSE*, pages 563–573. ACM, 2018.

[13] Wenrui Meng, Junkil Park, Oleg Sokolsky, Stephanie Weirich, and Insup Lee. Verified ROS-based deployment of platform-independent control systems. In *NFM*, volume 9058 of *LNCS*, pages 248–262. Springer, 2015.

[14] Bryan J. Muscedere, Robert Hackman, Davood Anbarnam, Joanne M. Atlee, Ian J. Davis, and Michael W. Godfrey. Detecting feature-interaction symptoms in automotive software using lightweight analysis. In *SANER*, pages 175–185. IEEE, 2019.

[15] Joseph P. Near, Aleksandar Milicevic, Eunsuk Kang, and Daniel Jackson. A lightweight code analysis and its role in evaluation of a dependability case. In *ICSE*, pages 31–40. ACM, 2011.

[16] John-Paul Ore, Carrick Detweiler, and Sebastian G. Elbaum. Lightweight detection of physical unit inconsistencies without program annotations. In *ISSTA*, pages 341–351. ACM, 2017.

[17] John-Paul Ore, Carrick Detweiler, and Sebastian G. Elbaum. Phriky-Units: A lightweight, annotation-free physical unit inconsistency detection tool. In *ISSTA*, pages 352–355. ACM, 2017.

[18] Marc Pichler, Bernhard Dieber, and Martin Pinzger. Can I depend on you? Mapping the dependency and quality landscape of ROS packages. In *IRC*. IEEE, 2019. to appear.

[19] Rahul Purandare, Javier Darsie, Sebastian G. Elbaum, and Matthew B. Dwyer. Extracting conditional component dependence for distributed robotic systems. In *IROS*, pages 1533–1540. IEEE, 2012.

[20] André Santos, Alcino Cunha, and Nuno Macedo. Property-based testing for the Robot Operating System. In *A-TEST@ESEC/SIGSOFT FSE*, pages 56–62. ACM, 2018.

[21] André Santos, Alcino Cunha, Nuno Macedo, and Cláudio Lourenço. A framework for quality assessment of ROS repositories. In *IROS*, pages 4491–4496. IEEE, 2016.

[22] André Santos, Alcino Cunha, and Nuno Macedo. Static-time extraction and analysis of the ROS computation graph. In *IRC*. IEEE, 2019. to appear.

[23] Nishant Sharma, Sebastian G. Elbaum, and Carrick Detweiler. Rate impact analysis in robotic systems. In *ICRA*, pages 2089–2096. IEEE, 2017.

[24] Rui Wang, Yong Guan, Houbing Song, Xinxin Li, Xiaojuan Li, Zhiping Shi, and Xiaoyu Song. A formal model-based design method for robotic systems. *IEEE Systems Journal*, 13(1):1096–1107, 2019. URL: `https://doi.org/10.1109/JSYST.2018.2867285`, `doi:10.1109/JSYST.2018.2867285`.

[25] Matt Webster, Clare Dixon, Michael Fisher, Maha Salem, Joe Saunders, Kheng Lee Koay, Kerstin Dautenhahn, and Joan Saez-Pons. Toward reliable autonomous robotic assistants through formal verification: A case study. *IEEE Trans. Human-Machine Systems*, 46(2):186–196, 2016.

[26] Kai Weng Wong and Hadas Kress-Gazit. Robot Operating System (ROS) introspective implementation of high-level task controllers. *Journal of Software Engineering for Robotics*, 8(1):65–77, 2017.