Verification of system-wide safety properties of ROS applications

Renato Carvalho, Alcino Cunha, Nuno Macedo, André Santos
Universidade do Minho & INESC TEC
Portugal

***Abstract*—Despite being currently deployed in safety-critical scenarios, proper techniques to assess the functional safety of robotics software are yet to be adopted by the community. This is particularly critical in ROS, where highly configurable robots are built by composing third-party modules of varying quality. To promote adoption, we advocate the development of lightweight formal methods, automatic techniques with minimal input from the user and intuitive feedback.**

This paper proposes a technique to automatically verify system-wide safety properties of ROS-based applications. At its base is the formalization of ROS architectural models and node behaviour in Electrum, over which system-wide specifications are subsequently model checked. To automate the analysis, it is then deployed as a plug-in for HAROS, a framework for the assessment of ROS software quality aimed at the ROS software developers. The technique is evaluated in a real robot, AgRob V16, with positive results.

I. Introduction

Safety certification for robotics software is increasingly vital with the ever-closer robot-human interaction, but suitable safety verification techniques are scarce. This particularly affects robotics software developed in ROS [1], one of the most popular robotics middlewares, which heavily relies on open-source and third-party components. Despite recent attempts to integrate quality assurance mechanisms in the ROS development cycle[1], the extensive and dynamic community, allied with the complexity of modern robotics software systems, renders classic formal approaches to program verification infeasible.

Although we believe in developing and promoting proper software engineering methodologies and techniques transversal to the complete ROS development cycle, it is undeniable that in the short term the quality of the existing ROS code corpus – now numbering over 3000 packages in the official index[2] – needs to be assessed. Thus, we advocate the use of *lightweight* formal approaches to analyse ROS-based software that can *i*) automatically analyse components as they are developed, *ii*) be deployed by typical ROS software developers, and *iii*) report feedback understandable by all stakeholders.

When verifying modular systems such as ROS applications – comprised by multiple nodes developed in general-purpose programming languages, and organized in heterogeneous architectures, dubbed the ROS *computation graph* – it is common to split the analysis in two stages: one analysing the behaviour of the individual components, and the other the end-to-end system behaviour, assuming the correctness of the individual components. Such system-wide analysis sees components (here, ROS nodes) as black-boxes, acting only on the interface level (here, message-passing), and has been shown to be prone to lightweight formal analyses [2]. These are particularly useful in middlewares that promote highly-configurable applications composed of third-party components of varying quality, such as ROS, where launch configurations can change which nodes are running, possibly selected from a community-driven index, pass them additional parameters, and remap the communication links.

This paper presents the first technique to automatically verify system-wide safety properties based on message-passing for ROS applications, based on a loose specification of the expected behaviour of the individual nodes. The backend relies on Electrum [3], a formal specification language based on first-order linear temporal logic that extends Alloy [4], which is accompanied by an automatic Analyzer [5]. To integrate the ROS development process we rely on HAROS, a plug-in-based framework for the continuous quality assessment of ROS software, previously developed by our team [6]. HAROS automatically extracts architectural models from ROS applications, provides a user-friendly behavioural specification language, and provides a unified reporting interface. The model checking technique is thus wrapped in a plug-in that automatically translates HAROS artefacts into Electrum, and found counter-examples back into ROS-flavoured issues.

Section II presents an overview of the proposed verification technique and how it integrates the HAROS framework. Technical details regarding the formalization in Electrum of architectural models and behavioural specifications are then presented in Section III. The Electrum language is presented as needed throughout these sections. Section IV reports on the application of the technique to a real ROS-based robot, AgRob V16. Section V discusses related work on the verification of ROS applications, and lastly Section VI presents conclusions and points to future work.

II. Approach overview

The proposed technique is based on a formalization of launch configurations of ROS applications in Electrum and its wrapping as a HAROS plug-in, as depicted in Fig. 1. This section provides an overview of this approach. As a running example, consider a simplistic ROS Controller, where a `Controller` node publishes commands in a topic `cmd` at

[1] https://discourse.ros.org/c/quality
[2] https://index.ros.org

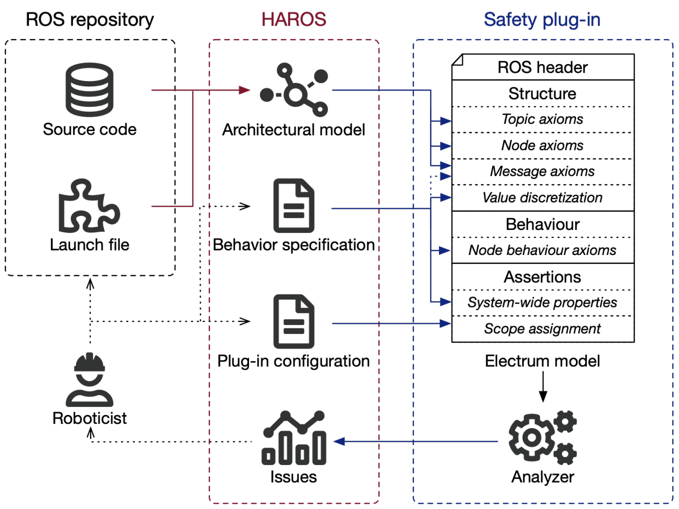


Fig. 1: Architecture of the HAROS safety plug-in for ROS.

a certain rate, according to some data provided by a `Teleop` node through `tel`. A `Base` node then subscribes to `cmd` and publishes back data from sensors through `dat`, that is either value 1 (everything ok) or 0 (dangerous situation). When `Controller` receives a danger message, it publishes a message at `cmd` with value 0 and a "stop" warning message.

A. *Model checking backend*

At the core of the proposed technique is the encoding of the architecture (i.e., the computation graph) and behaviour of a launch configuration as an Electrum model, whose structure is depicted on the right-hand side of Fig. 1. Besides a shared header declaring common elements, this model starts with the codification of the architectural model (including nodes, topics, messages and their relationships), as will be presented in Section III-A. Being SAT-based, Electrum is not well-suited to deal with numerical values, so a discretization of message values is performed to keep the problems manageable.

This is followed by the axiomatization of the behaviour of the individual nodes and the declaration of assertions to check the system-wide properties over the launch configuration under analysis, as will be presented in Section III-B. Our goal is to verify architectural safety properties, so node behaviour is specified at the ROS interface level, that is, at the level of message processing and publication. Thus, node behaviour is loosely specified to allow the analysis of alternative behaviours and event interleavings. This may lead to false positives, especially when verifying liveness properties.

The resulting model is then passed to the Analyzer, which will automatically check whether the end-to-end properties hold, returning an execution trace if a counter-example is found. Electrum acts in a bounded universe, so scopes for certain elements must be provided prior to the analysis.

B. *HAROS plug-in*

To automate the creation of the Electrum models and ease the deployment and integration of the technique in the development process, the model checking backend is integrated into

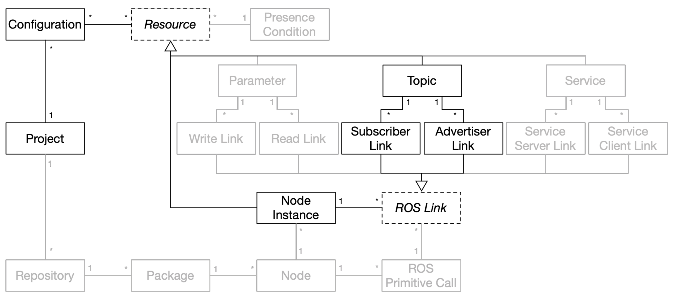


Fig. 2: HAROS architectural meta-model (unsupported features greyed out).

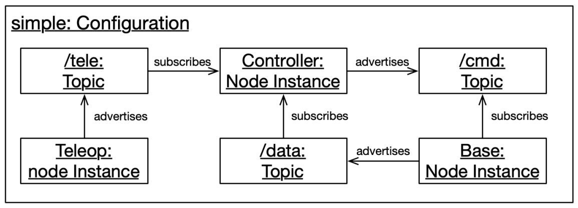


Fig. 3: HAROS architectural model for Controller.

HAROS, as depicted in the left-hand side of Fig. 1. HAROS[3] is a plug-in-based framework for assessing the quality of ROS software in continuous integration.

HAROS provides two plug-in entry points for a ROS repository under analysis: one for packages and source files, and another for architectural models per launch configuration, automatically extracted from the source code [7]. The proposed technique is deployed as a plug-in for the latter. The HAROS architectural meta-model is depicted in Fig. 2, highlighting the elements relevant for this approach, which amounts essentially to the ROS computation graph, containing information regarding launched nodes and which topics are subscribed and advertised. Support for other computation graph elements, such as services and parameters, is left as future work. The architectural model for a `simple` configuration of Controller under this meta-model is presented in Fig. 3.

Over each maintained ROS repository, HAROS allows the definition of behavioural specifications to be used by the plug-ins [8]. Fitting for the goal of this work, this domain-specific language, depicted in Fig. 4, acts at the message-passing level, treating nodes as black-boxes. Its semantics is formalized in metric first-order linear temporal logic over ROS execution traces based on well-known temporal property patterns [9], supporting both safety – *absence* and *precedence* patterns – and liveness – *existence* and *response* patterns – properties over events. Events essentially amount to messages occurring in topics, which may be filtered according to certain conditions over the message fields. As stated, our approach assumes the correct behaviour of individual nodes in order to verify end-to-end properties, and HAROS distinguishes between node and configuration specifications. The plug-in translates these into the first-order linear temporal logic of Electrum.

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

```
property     ::= scope : pattern
scope        ::= globally
               | after activator [until terminator]
               | time-bound after activator
pattern      ::= some events | no events
               | events causes [time-bound] events
               | events requires [time-bound] events
events       ::= multi-event (|| multi-event)*
multi-event  ::= event | event-set | event-chain
event        ::= name [predicate] [as ident]
predicate    ::= { condition (, condition)* }
condition    ::= param-expr binop value-expr
param-expr   ::= param | builtin ( param )
value-expr   ::= set | range | value
set          ::= [ value (, value)* ]
range        ::= number to number
value        ::= number | string | ref-expr
ref-expr     ::= [$ident .] param | builtin ( [$ident .] param )
binop        ::= = | != | in | not in | < | <= | > | >=
param        ::= field [. field]
field        ::= ident [[ (n-zero | slice) ]]
```

Fig. 4: HAROS behavioural language (unsupported features greyed out).

A specification of node behaviour for Controller is provided in Fig. 5a. For `Teleop` and `Base`, it is simply stated that they publish messages within the expected value constraints. For `Controller`, 2 safety properties are enforced: that it only publishes values $\neq 0$ at `cmd` if they have been received from `tel`; and that 0 values may be propagated from `tel` or originate from danger messages at `dat`. Note that these do not force the actual publication of messages, but rather restrict what are the valid publications. Figure 5b then shows some system-wide expected properties, that should emerge from the node behaviour and the architecture under consideration: that no value other than those published in `tel` ever reaches `cmd`, that values of 0 in `cmd` arise either from 0s at `tel` or `dat`, that "stop" messages at `cmd` arise from 0s at `dat`, and that 0s at `dat` are eventually propagated to `cmd`. Currently only global pattern scopes are supported (support for richer scopes is under way, and will allow the definition of rules such as "*after* a danger message, only 0 values may be published"). Real-time specification features are out of the scope of this technique due to the nature of the underlying model checker.

Each repository maintained in HAROS may also be provided additional configurations, which includes which plug-ins will be run and under which options. Required analysis scopes for our technique are specified in this file.

Lastly, HAROS provides a unified interface for reporting issues, which for architectural plug-ins are associated a concrete launch configuration. If the Analyzer finds a counter-example that violates a desirable property, it is reported back as a HAROS issue. During the translation a mapping between ROS and Electrum artefacts is kept, so that the abstract execution traces can be converted back into the ROS domain. One such issue is depicted in Fig. 6 for Controller: the third specified property does not actually hold, since from Fig. 5a we are forcing `Controller` to publish "stop" messages after receiving 0 values, but not forbidding their occurrences in other circumstances. The depicted counter-example shows `Teleop` publishing a value 1, which is processed by `Controller` that publishes the same value with a "stop" message.

III. Formalization of ROS configurations

This section presents the formalization of launch configuration of a ROS application in Electrum. The language is presented as needed, for a more thorough presentation see [3].

A. Architectural model

In Electrum, much like in Alloy, structure is defined through the declaration of *signatures* (**`sig`**) – which introduce sets of uninterpreted atoms – and *fields* – which introduce relations between atoms. A hierarchy may be imposed over signatures, either through *extension* (**`extends`**) – guaranteeing the disjointedness of the sub-signatures – or through *inclusion* (**`in`**) – allowing for overlapping sub-signatures. A signature may be declared as **`abstract`** – so that its atoms must belong to a sub-signature – and assigned a simple *multiplicity* (**`some`**, **`no`**, **`one`**, **`lone`**) – restricting the number of atoms assigned to it. Fields are associated to a parent signature, may be of arbitrary arity, and can also be restricted by multiplicities. In Electrum, unlike in Alloy, signatures and fields may be declared as mutable (**`var`**), allowing them to change in time.

The structural ROS header is presented in the first part of Fig. 7. Signatures `Node` and `Topic` represent the nodes and topics of a ROS configuration; they are declared as abstract since they will be extended by the particular objects of each configuration. Node static fields `subs` and `advs` will contain the **`set`** of subscribed and advertised topics, respectively. Variable fields `inbox` and `outbox` will contain a each instant the **`set`** of messages waiting to be processed and to be propagated, respectively, by each node. Signature `Field` will contain all message fields relevant for verifying the configuration specification. Lastly, signature `Message` will denote the set of available messages in each trace and `Value` abstracts possible message field values, with two sub-signatures representing the supported primitive types, `IntVal` and `StrVal`. Messages are assigned exactly **`one`** `topic`, and may have at most one value assigned to each message field (`val`, multiplicity **`lone`**). For simplicity purposes, we do not explicitly model message types, which can be inferred from the assigned topics and are used to enforce the mandatory fields for each message, as will be shown briefly. Unlike the other signatures whose values are fixed exactly for a particular configuration, during the analysis an arbitrary number of messages and values will be considered, within predefined bounds.

The second part of Fig. 7 presents the encoding of the Controller architecture from Fig. 3. The translation of nodes and topics is straightforward, and results in the declaration of singleton (**`one`**) signatures for each topic (e.g., `dat`) and node (e.g., `Base`), as well as the restriction of the subscriptions and advertisements of each node through **`fact`** `Links`, which in Electrum can be used to impose additional constraints on the model. Here, $\rightarrow$ denotes the Cartesian product and `+`/`&` set union/intersection. This is followed by the declaration of relevant fields (e.g., `dat_val`). To avoid encumbering

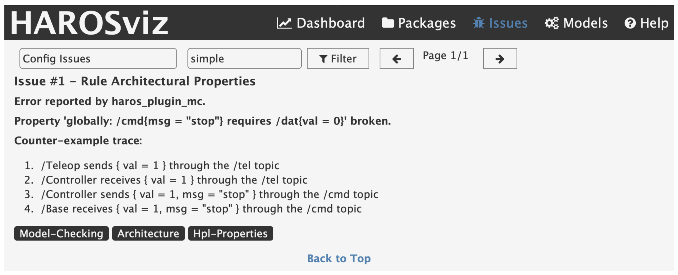
```
Teleop:
  globally:
    no /tel{val not in 0 to 100}
Base:
  globally:
    no /dat{val not in {0,1}}
Controller:
  globally:
    /cmd{val != 0} as m requires /tel{val = $(m.val)}
  globally:
    /cmd{val = 0} requires /tel{val = 0} || /dat{val = 0}
  globally:
    /dat{val = 0} causes /cmd{val = 0, msg = "stop"}
```

(a) Node behaviour.

```
simple:
  globally:
    no /cmd{val not in 0 to 100}
  globally:
    /cmd{val = 0} requires /dat{val = 0} || /tel{val = 0}
  globally:
    /cmd{msg = "stop"} requires /dat{val = 0}
  globally:
    /dat{val = 0} causes /cmd{val = 0, msg = "stop"}
```

(b) System-wide properties.

Fig. 5: HAROS specifications for Controller.



Issue #1 – Rule Architectural Properties

Error reported by haros_plugin_mc.

Property 'globally: /cmd{msg = "stop"} requires /dat{val = 0}' broken.

Counter-example trace:

1. /Teleop sends { val = 1 } through the /tel topic
2. /Controller receives { val = 1 } through the /tel topic
3. /Controller sends { val = 1, msg = "stop" } through the /cmd topic
4. /Base receives { val = 1, msg = "stop" } through the /cmd topic

Fig. 6: HAROS safety issue for Controller.

the analysis, this process also takes into consideration the behavioural specifications, so that irrelevant message fields are not created. Fact `Fields` then guarantees that each message is correctly typed, by enforcing that *i*) each message field correctly typed (e.g., l. 32), and *ii*) each message has exactly one value assigned per relevant message field (e.g., l. 33). In Electrum **`all`**, **`some`** or **`no`** denote first-order quantifications, and **`.`** relational composition, so that `topic.tel`, e.g., denotes all messages with topic `tel` assigned.

Lastly we deal with the discretization of message field values. Again, this process takes into consideration the behavioural specifications, since different values may be relevant for different message conditions. Message conditions may test equalities with constant values or message fields, set membership and numeric range membership. The goal is to impose a hierarchy on uninterpreted `Value` atoms so that messages can still be tested for those conditions without explicitly representing every numerical value. Thus, this process works as follows:

- for each constant value in an equality test or inside a set in a condition, create a **`lone`** signature (e.g., `Int_0` or `Str_stop`);
- for each range test in a condition, create a signature without any multiplicity imposed (e.g., `Int_0_10`);
- identify which numeric values/ranges are completely contained in one another and force the containment of the respective signatures (e.g., ll. 40–41);
- identify which numeric values/ranges are completely disjoint with one another and force the disjointedness of the respective signatures (e.g., l. 42).

```
1   -- ROS header
2   abstract sig Topic, Field, Value {}
3   sig IntVal, StrVal extends Value {}
4
5   abstract sig Node {
6     subs, advs       : set Topic,
7     var inbox, outbox: set Message }
8
9   sig Message {
10    topic: one Topic,
11    val  : Field→lone Value }
12
13  -- Application-specific architecture
14  one sig tel, dat, cmd extends Topic {}
15  one sig Teleop, Base, Controller extends Node {}
16
17  fact Links {
18    advs = Teleop→tel + Base→dat + Controller→cmd
19    subs = Controller→(tel+dat) + Base→cmd }
20
21  one sig tel_val, dat_val,
22    cmd_val, cmd_msg extends Field {}
23
24  fact Fields {
25    all m: topic.tel {
26      (m.val) in tel_val→IntVal
27      (m.val) in tel_val→one IntVal }
28    all m: topic.dat {
29      (m.val) in dat_val→IntVal
30      (m.val) in dat_val→one IntVal }
31    all m: topic.cmd {
32      (m.val) in cmd_val→IntVal + cmd_msg→StrVal
33      (m.val) in (cmd_val+cmd_msg)→one (IntVal+StrVal) } }
34
35  lone sig Int_0, Int_1 in IntVal {}
36  sig Int_0_10, Int_0_100 in IntVal {}
37  lone sig Str_stop in StrVal {}
38
39  fact Values {
40    Int_0+Int_1 in Int_0_10
41    Int_0_10 in Int_0_100
42    no Int_0&Int_1 }
```

Fig. 7: Encoding of the Controller architecture in Electrum.

Acting on a finite universe, the set of values considered during model checking will necessarily be finite, but the procedure will check every possible valuation within that scope.

B. Behavioural specifications

Electrum supports properties in first-order linear temporal logic, including future operators **`after`**, **`eventually`** and **`always`**, and their past counter-parts **`before`**, **`once`** and **`historically`**. Primed expressions can also be used to refer to an expression value in the succeeding state. Such declarative

```
1  -- ROS header
2  fact Messages {
3    no outbox+inbox
4    always {
5      all n: Node {
6        n.inbox.topic in n.subs
7        n.outbox.topic in n.advs }
8      all m: Node.outbox {
9        all n: subs.(m.topic) | eventually m in n.inbox
10       eventually m not in Node.outbox }
11     all m: Node.inbox |
12       before once m in advs.(m.topic).outbox } }
13
14 -- Application-specific behaviour
15 fact NodeBehavior { always {
16   no m: Teleop.outbox&topic.tel |
17     tel_val.(m.val) not in Int_0_100
18   no m: Base.outbox&topic.dat |
19     dat_val.(m.val) not in Int_0+Int_1
20   all m: Controller.outbox&topic.cmd |
21     cmd_val.(m.val) != Int_0 implies before once
22       some m0: Controller.inbox&topic.tel |
23         tel_val.(m0.val) = cmd_val.(m.val)
24   all m: Controller.outbox&topic.cmd |
25     cmd_val.(m.val) = Int_0 implies before once (
26       (some m0: Controller.inbox&topic.dat |
27         dat_val.(m0.val) = Int_0) or
28       (some m0: Controller.inbox&topic.tel |
29         tel_val.(m0.val) = Int_0))
30   all m: Controller.inbox&topic.dat |
31     dat_val.(m.val) = Int_0 implies after eventually
32       some m0: Controller.outbox&topic.cmd {
33         cmd_val.(m0.val) = Int_0
34         cmd_msg.(m0.val) = Str_stop } } }
35
36 assert simple0 { always {
37   no m: Node.outbox&topic.cmd |
38     cmd_val.(m.val) not in Int_0_100 } }
39 assert simple1 { always {
40   all m: Node.outbox&topic.cmd |
41     cmd_val.(m.val) = Int_0 implies before once (
42       (some m0: Node.outbox&topic.dat |
43         dat_val.(m0.val) = Int_0) or
44       (some m0: Node.outbox&topic.tel |
45         tel_val.(m0.val) = Int_0)) } }
46 assert simple2 { always {
47   all m: Node.outbox&topic.cmd |
48     cmd_msg.(m.val) = Str_stop implies before once
49       some m0: Node.outbox&topic.dat |
50         dat_val.(m0.val) = Int_0 } }
51 assert simple3 { always {
52   all m: Node.outbox&topic.dat |
53     dat_val.(m.val) = Int_0 implies after eventually
54       some m0: Node.outbox&topic.cmd |
55         cmd_msg.(m0.val) = Str_stop } }
```

Fig. 8: Encoding of the Controller behaviour in Electrum.

property can be both used as a model axiom in a **fact** – restricting the valid traces that will be considered during model checking – and as a model assertion in an **assert** – that will be checked over valid execution traces. Thus the same temporal idiom can be used to impose the behaviour of individual nodes and to check system-wide properties, which in HAROS are formalized in the same language.

Acting upon loose node specifications and without any real-time considerations, our encoding of ROS behaviour is under-specified to allow for alternative behaviours and event interleavings. The first part of Fig. 8, fact `Messages`, encodes the message-passing process that is agnostic of the ROS configuration under analysis. It starts by stating that the inboxes and outboxes start empty (l. 3), and then restricts message-passing in all states (**always**) by enforcing:

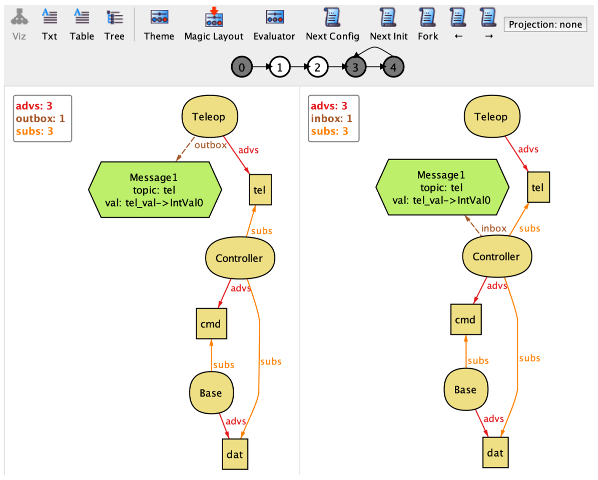


Fig. 9: Counter-example for Controller in Electrum.

- the correct typing of messages in inboxes and outboxes (ll. 6–7). Notice that, e.g., `n.inbox.topic` retrieves the topics of all messages in the inbox of `n`;
- that every message in an outbox eventually reaches the inboxes of all subscribing nodes (l. 9) and leaves the outbox (l. 10). Notice that no timing constraints are imposed on this process, and in fact subscribing nodes may receive the message at different states;
- all messages that exist in an inbox must have previously been in the outbox of an advertising node (l. 12), so that no messages spontaneously appear in inboxes.

The second part of Fig. 8 depicts the translation of the Controller node and configuration specification (Fig. 5), adapting the LTL formalization of the specification patterns [9] to LTL with past operators. The former is encoded in fact `NodeBehaviour`, which restricts how each node processes messages from the inbox to the outbox. For instance, the absence pattern of `Teleop` is straightforward (ll. 16–17): no message may be in the outbox with values outside the valid integer range (note that, e.g., `Teleop.outbox&topic.tel` denotes all messages in the outbox of `Teleop` for topic `tel`). In precedence patterns messages in the outbox require the previous existence of another message in the inbox, such as the one in ll. 20–23 for `Controller`, while in the dual response pattern messages in the inbox require the eventual existence of another message in the outbox, such as the one in ll. 30–34 for `Controller`.

Lastly, an assertion is created for each configuration specification. For instance, `simple0` checks whether messages published in `cmd` may be outside the expected integer range, and `simple2` whether "stop" messages at `cmd` are always preceded by a value 0 at `dat`. The main difference between the translation of node and configuration specifications is in the scope of the message-passing events: the former focuses on the inbox and outbox of the node being specified; the latter on any message passing in a given topic, abstracted by being in the inbox of any subscribing node (e.g., `Node.inbox&topic.cmd` denotes every message published by any node in topic `cmd`).

Fig. 10: The AgRob V16 robot monitoring a slope vineyard.

These assertions can be automatically verified by the Electrum Analyzer once a scope is provided for the non-exact signatures (`Value` and `Message`). Moreover, by default Electrum performs *bounded* model checking [10], meaning that traces are considered only up to a maximum length. The scope for `Value` and `Message`, as well as a maximum trace length, are defined in the plug-in configuration file of each ROS repository under analysis, and should depend on the complexity of the application and on the needed level of confidence. As expected, all assertions except `simple2` are shown to hold (with up to 5 messages/values and maximum trace length of 10). The Analyzer could be used as a standalone, in which case the counter-example for `simple2` could be explored in its visualizer as depicted in Fig. 9, currently focusing on the transmission of a message from the outbox of `Teleop` to the inbox of `Controller`. The HAROS wrapper translates such traces back into the ROS-domain, resulting in the issue already presented in Fig. 6.

IV. Evaluation

Our evaluation of the proposed technique had two main goals, namely to assess *i*) whether its expressibility is sufficient to address relevant systems and properties, and *ii*) whether it scales to real world robotics software. To that purpose, we have applied to a real robot under development, AgRob V16.

A. Case study

AgRob V16 (Fig.10) is a modular robot for precision framing in slope vineyards developed in C++ ROS[4]. One of its main robotic challenges is the operation in an unstructured environment. This also renders it a safety-critical system, as the environment is shared with human operators. Furthermore, it has been developed with a focus on modularity and extensibility, relying on several third-party ROS packages, making it prone to launch configuration errors that may undermine safety properties. For these reasons, the feedback provided by the proposed technique in continuous integration could prove valuable to the ROS development team.

The robot follows a typical architecture of sensors, controllers, planners and actuators. Some features are optional,

[4]http://agrob.inesctec.pt/

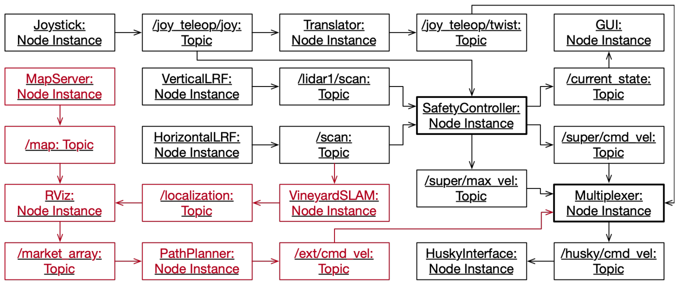


Fig. 11: Simplified HAROS architectural model for AgRob V16 (red denotes elements of the `map` configuration).

and multiple launch files are provided with different presets. It has two main operating modes: one where the robot follows a path, avoiding possible obstacles, and another where the robot is controlled by a teleoperation joystick; the operating mode is also switched through teleoperation. For the purpose of this analysis we focus on two particular configurations: `startup` which launches a minimal configuration, and `map` that additionally launches localization and navigation features. In the former, obstacles are only detected by laser sensors, while in the latter localization information is also considered.

A simplified version of the AgRob V16 architectural model, as extracted by HAROS, is depicted in Fig. 11, highlighting the differences between the two configurations. Roughly, the core nodes are the `SafetyController` and the `Multiplexer`, which are responsible for ensuring safe behaviour. The controller monitors data coming from sensors – lasers `HorizontalLRF` and `VerticalLRF` – and the selected operating mode provided by the teleoperation `Joystick` to issue safety velocity commands and the current state to be displayed by the `GUI`. The multiplexer collects commands from the safety controller, from the `Joystick` teleoperation and, under the `map` configuration, also from the `PathPlanner`. Following a set of priorities, the selected commands are then passed to the `HuskyInterface` which communicates with the hardware actuators.

After discussions with the ROS development team, the message-passing behaviour of the individual nodes was established, as well as the desirable system-wide properties, which were encoded in the HAROS specification language. This resulted in 30 specifications over 9 nodes, and 4 configuration safety properties, some of which depicted in Fig. 12. Figure 12a focuses on the specification for `SafetyController`, the node with richer behaviour. Note that the joystick operating mode is switched on through `joy` messages with `button[0]` $= 1$, and the "follow path" mode through messages with `button[0]` $= 0$ and `button[1]` $=$ 1, having lower priority; field `data[0]` at `current_state` messages reports the current operating mode; and when in joystick mode, `buttons[4]` and `button[5]` issue velocity commands. In that context, the specification includes:

- how the information published to `current_state` for the `GUI` is determined from the teleoperation buttons;
- restricting the ranges of linear and angular velocities

```
globally:
  /current_state{data[0] = 6}
    requires /joy_teleop/joy{button[0] = 1}
globally:
  /current_state{data[0] = 3}
    requires /joy_teleop/joy{button[0] = 0, button[1] = 1}
globally:
  no /super/cmd_vel{linear.x not in 0 to 10}
globally:
  no /super/cmd_vel{angular.x not in -100 to 100}
globally:
  /supervisor/cmd_vel{linear.x in 3.8 to 4.2}
    requires /joy_teleop/joy{button[0] = 0, button[1] = 1} ||
             /joy_teleop/joy{button[4] = 1, button[5] = 0}
globally:
  /supervisor/cmd_vel{linear.x in 5.8 to 6.2}
    requires /joy_teleop/joy{button[0] = 0, button[1] = 1} ||
             /joy_teleop/joy{button[4] = 0, button[5] = 1}
...
```

(a) `SafetyController` behaviour.

```
globally: // Prop1
 /agrobv16/current_state{data[0] = 3}
   requires /joy_teleop/joy{button[0] = 0, button[1] = 1}
...
globally: // Prop4
  /husky/cmd_vel{linear.x = 0, angular.x in -100 to 100}
    requires /scan{ranges[0] in 0 to 4} ||
             /joy_teleop/joy{button[0] = 1}
```

(b) System-wide properties.

Fig. 12: Snippet of HAROS specifications for AgRob V16.

published at `cmd_vel`;
- how certain velocity values are restricted to modes (determined by `joy` buttons 0 and 1) or other teleoperation commands (by `joy` buttons 4 and 5)

Figure 12b depicts some configuration properties, namely:
- that `current_state` only contains the "follow path" mode ($\texttt{data[0]} = 3$) if the corresponding buttons have once been pressed in `joy`;
- `cmd_vel` messages to the base to rotate in-place ($\texttt{linear.x} = 0$ and $\texttt{angular.x} \neq 0$) are caused by a `scan` message with a range smaller than 40cm ($0 \leq \texttt{ranges[0]} \leq 4$), or the system is in teleoperation mode ($\texttt{button[0]} = 1$).

All the 4 system-wide properties were checked for the 2 configurations with increasing scopes for values and messages for traces with up to 10 states, in a 2.4 GHZ Intel Core i5 with 8GB memory running the bounded model checking engine of Electrum with SAT4J, as summarized in Table I. One of the properties was actually shown not to hold for the `map` configuration due to introduction of additional nodes: commands to rotate in-place may be published in "follow path" mode by the `PathPlanner` without obstacles being detected by the lasers due to accumulated localization errors, which may wrongly identify dangerous situations. The counter-example returned as a HAROS issue would roughly describe a execution trace where a velocity command from the `PathPlanner` would be identified by the `SafetyController` as a dangerous situation, not flagged by the lasers, and thus instruct the `HuskyInterface` to rotate. This counter-example requires scopes of at least 4 values/messages, but can still be found under 1m for 10 values/messages. All properties were checked with up to 14 values/messages at around 2m, which is feasible if the technique is to be executed in continuous integration.

B. Threats to validity

As already discussed, the assignment to each ROS repository of a model checking scope for messages/values must be handled with care, since small universes may hide possible safety issues. We believe that with application-specific knowledge of the development team it is possible to infer sensible scopes for most relevant traces, but whether that will scale for more complex applications needs further evaluation.

The loosely specified behaviour of our model may lead to false positive counter-examples, since no particular scheduling is imposed on the message-passing process. We did not detect such cases in our case study, but we expect them to arise mostly when dealing with desirable liveness properties, which have not been identified in AgRob V16.

V. Related Work

As far as we aware, ours is the first approach for automatically verifying safety properties of ROS applications.

Dedicated static analysis techniques to address specific issues have been proposed. The initial release of HAROS by our team focused on calculating internal quality metrics and assessing conformity with coding standards [6]. Ore et al. propose a technique [11], backed by the Phriky-Units tool [12], to detect inconsistencies between physical units in C++ ROS code, relying on the a priori annotation of standard libraries. This work has also been extended to support the probabilistic inference of units from non-annotated libraries [13].

Some approaches extract an intermediary model from the code, but not with the goal of performing general-purposes safety analyses. Purundare et al. [14] propose a technique that statically extracts, from C++ ROS code, a model of the message flow between components, which is then used to highlight code changes that may impact message-passing. Sharma et al. [15] also address the impact of code changes by statically extracting a data flow model from C++ ROS code, but focus on changes to the rate of message publication and identifying nodes that are sensible to the rate of incoming data. Muscedere et al. [16] instead focus on the detection of feature interaction symptoms by extracting a "factbase" from C++ ROS code, over which user-defined code queried are executed. Such queries can be used to identify code patterns but not

Configuration	**Spec**	n `Value`, n `Message`, 10 **`Time`**						
		2	4	6	8	10	12	14
`startup`	Prop1	✓ (3.0s)	✓ (7.5s)	✓ (19.0s)	✓ (29.3s)	✓ (45.5s)	✓ (70.1s)	✓ (102.7s)
	Prop1	✓ (2.9s)	✓ (8.8s)	✓ (17.4s)	✓ (26.9s)	✓ (49.3s)	✓ (75.8s)	✓ (104.8s)
	Prop2	✓ (3.0s)	✓ (7.5s)	✓ (15.6s)	✓ (32.3s)	✓ (44.7s)	✓ (63.4s)	✓ (90.6s)
	Prop4	✓ (2.6s)	✓ (11.0s)	✓ (26.8s)	✓ (45.0s)	✓ (67.2s)	✓ (108.7s)	✓ (142.3s)
`map`	Prop1	✓ (3.8s)	✓ (10.7s)	✓ (23.0s)	✓ (37.6s)	✓ (59.8s)	✓ (93.8s)	✓ (121.0s)
	Prop1	✓ (3.6s)	✓ (9.1s)	✓ (23.9s)	✓ (39.5s)	✓ (61.1s)	✓ (103.0s)	✓ (138.5s)
	Prop2	✓ (3.6s)	✓ (10.6s)	✓ (24.4s)	✓ (38.9s)	✓ (61.9s)	✓ (103.1s)	✓ (126.3s)
	Prop4	✓ (3.1s)	✗ (8.6s)	✗ (20.0s)	✗ (36.5s)	✗ (54.6s)	✗ (85.1s)	✗ (128.8s)

TABLE I: Results for the 4 desirable properties for the 2 configurations of AgRob V16, including execution times.

dynamic behaviour. The proposed approach builds on work by our team that formalizes the meta-model for the ROS computation graph and proposes a general approach for its extraction from ROS C++ applications [7]. A code query language is also provided to detect simple architectural patterns, but not dynamic behaviour. Such models have recently been used as the basis of a property-based testing technique [8].

Some work has been done on attempting to verify safety properties of ROS applications through state-of-the-art model checkers, namely SPIN by Webster et al. [17] and UPPAAL by Halder et al. [18]. However, these are mostly exploratory approaches based on the ad hoc codification of robotic software into the target model checking language.

VI. Conclusions

This paper presented a technique based on model checking to verify message-passing system-wide safety properties based on a formalization of ROS launch configurations and loosely specified behaviour of individual nodes. It has been wrapped in a plug-in for HAROS that automatically creates such Electrum model – from the architectural model extracted in continuous integration and the specifications in the provided domain-specific language – and translates back abstract counter-examples into the ROS domain – reported in a unified interface for issues. Through the application to a real robot, it has proven to be sufficiently expressive to check certain classes of safety properties, and to have a reasonable performance.

We are currently extending the support for richer property patterns, in particular for scopes other than the global one, which would allow the specification of desirable properties depending on the current operating mode. Future work is planned to address the two main limitations of this approach. First, on a more fundamental line, techniques to support arbitrary scopes for record-like signatures, such as `Message`, in the underlying model check Electrum. Second, on a more domain-specific line, techniques to help discard false positives, possibly relying on run-time analyses to check whether the generated counter-examples are actually valid execution traces.

Acknowledgment

The authors would like to thank Filipe Neves dos Santos and Luís Santos for supporting the analysis of AgRob V16. 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 PTDC/CCI-INF/29583/2017 (POCI-01-0145-FEDER-029583).

References

[1] 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.

[2] J. P. Near, A. Milicevic, E. Kang, and D. Jackson, "A lightweight code analysis and its role in evaluation of a dependability case," in *ICSE*. ACM, 2011, pp. 31–40.

[3] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg, "Lightweight specification and analysis of dynamic systems with rich configurations," in *SIGSOFT FSE*. ACM, 2016, pp. 373–383.

[4] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, revised ed. MIT Press, 2012.

[5] J. Brunel, D. Chemouil, A. Cunha, and N. Macedo, "The Electrum Analyzer: Model checking relational first-order temporal specifications," in *ASE*. ACM, 2018, pp. 884–887.

[6] A. Santos, A. Cunha, N. Macedo, and C. Lourenço, "A framework for quality assessment of ROS repositories," in *IROS*. IEEE, 2016, pp. 4491–4496.

[7] A. Santos, A. Cunha, and N. Macedo, "Static-time extraction and analysis of the ROS computation graph," in *IRC*. IEEE, 2019, pp. 62–69.

[8] ——, "SeaBASS: System and behaviour abstraction with short specifications," 2019, submitted.

[9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *ICSE*. ACM, 1999, pp. 411–420.

[10] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[11] J. Ore, C. Detweiler, and S. G. Elbaum, "Lightweight detection of physical unit inconsistencies without program annotations," in *ISSTA*. ACM, 2017, pp. 341–351.

[12] ——, "Phriky-Units: A lightweight, annotation-free physical unit inconsistency detection tool," in *ISSTA*. ACM, 2017, pp. 352–355.

[13] S. Kate, J. Ore, X. Zhang, S. G. Elbaum, and Z. Xu, "Phys: Probabilistic physical unit assignment and inconsistency detection," in *ESEC/SIGSOFT FSE*. ACM, 2018, pp. 563–573.

[14] R. Purandare, J. Darsie, S. G. Elbaum, and M. B. Dwyer, "Extracting conditional component dependence for distributed robotic systems," in *IROS*. IEEE, 2012, pp. 1533–1540.

[15] N. Sharma, S. G. Elbaum, and C. Detweiler, "Rate impact analysis in robotic systems," in *ICRA*. IEEE, 2017, pp. 2089–2096.

[16] B. J. Muscedere, R. Hackman, D. Anbarnam, J. M. Atlee, I. J. Davis, and M. W. Godfrey, "Detecting feature-interaction symptoms in automotive software using lightweight analysis," in *SANER*. IEEE, 2019, pp. 175–185.

[17] M. Webster, C. Dixon, M. Fisher, M. Salem, J. Saunders, K. L. Koay, K. Dautenhahn, and J. Saez-Pons, "Toward reliable autonomous robotic assistants through formal verification: A case study," *IEEE Trans. Human-Machine Systems*, vol. 46, no. 2, pp. 186–196, 2016.

[18] R. Halder, J. Proença, N. Macedo, and A. Santos, "Formal verification of ROS-based robotic applications using timed-automata," in *FormaliSE@ICSE*. IEEE, 2017, pp. 44–50.