2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 25-29, 2020, Las Vegas, NV, USA (Virtual)

Verification of system-wide safety properties of ROS applications

Renato Carvalho, Alcino Cunha, Nuno Macedo, André Santos

***Abstract*— Robots are currently deployed in safety-critical domains but proper techniques to assess the functional safety of their software are yet to be adopted. This is particularly critical in ROS, where highly configurable robots are built by composing third-party modules. To promote adoption, we advocate the use of lightweight formal methods, automatic techniques with minimal user input and intuitive feedback.**

This paper proposes a technique to automatically verify system-wide safety properties of ROS-based applications at static time. It is based in 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 deployed as a plug-in for HAROS, a framework for the assessment of ROS software quality aimed at the ROS community. 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. ROS (ros.org) is one of the most popular robotics middlewares but despite recent attempts to integrate quality assurance mechanisms in its development cycle[1], the extensive and dynamic community, allied to the complexity of modern robotics software systems built modularly from third-party packages, renders classic formal approaches infeasible. Although we believe in promoting reliable software engineering methods transversal to the ROS development cycle, in the short term already developed ROS robots need be addressed. Thus, we advocate the use of *lightweight* formal approaches that can *i*) automatically analyse ROS applications from source code as they are developed, *ii*) be deployed by typical ROS developers, and *iii*) quickly report feedback understandable by all stakeholders.

When verifying modular systems such as ROS applications – comprised by 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 components. The latter 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 [1]. These are particularly useful in middlewares that promote highly-configurable applications, such as ROS where launch configurations determine which nodes run and under which parameters, and may remap communication links.

The authors are with Universidade do Minho and INESC TEC, Portugal. The first author was supported by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 and by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-016826. The remaining were financed by the ERDF through COMPETE 2020 and by National Funds through the FCT within project POCI-01-0145-FEDER-029583. They would like to thank Filipe Santos and Luís Santos for supporting the analysis of AgRob V16.

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

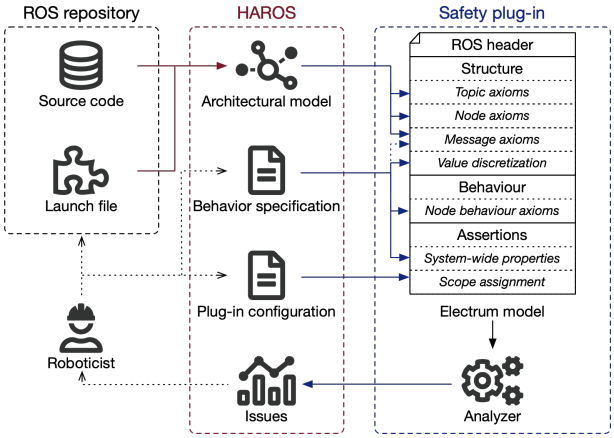

Fig. 1: Architecture of the HAROS safety plug-in for ROS.

This paper presents the first technique to automatically verify at static time system-wide safety properties based on message-passing for ROS applications, given a loose specification of the expected behaviour of the individual nodes. The backend relies on Electrum [2], a formal specification language based on first-order linear temporal logic that extends Alloy [3] and is accompanied by an automatic Analyzer. As depicted in Fig. 1, to integrate the ROS development process we rely on HAROS, a framework for the continuous quality assessment of ROS software developed by our team [4]. HAROS automatically extracts computation graphs from ROS source code, and provides a user-friendly behavioural specification language and a unified reporting interface. The main contributions of this work are thus: *i*) the formalization of the architecture and behaviour of ROS applications in Electrum, and *ii*) a HAROS plug-in that translates HAROS artefacts into Electrum and reports model checking counter-examples back as ROS-flavoured issues.

Section II presents the developed HAROS plug-in. The underlying Electrum architectural and behavioural formalization is presented in Section III. Section IV reports on its application to a real ROS-based robot, AgRob V16. Section V discusses related work on ROS analysis, and lastly Section VI presents conclusions and future research lines.

II. HAROS safety plug-in

HAROS (github.com/git-afsantos/haros) is a plug-in-based framework for the quality assessment of ROS software,

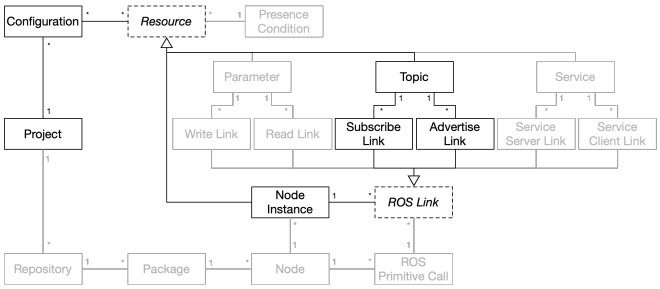

Fig. 2: HAROS meta-model (unsupported features in grey).

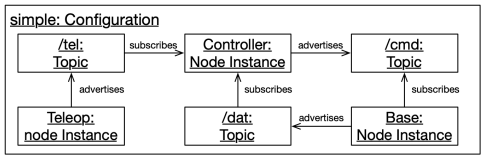

Fig. 3: HAROS architectural model for Controller.

```
property   ::= scope : pattern
scope      ::= globally | after activator [until terminator]
pattern    ::= some events | events causes events
             | no events | events requires events
events     ::= event (|| event)*
event      ::= name [predicate] [as ident]
predicate  ::= { condition (, condition)* }
condition  ::= param binop value-expr
value-expr ::= [ value (, value)* ] | int to int | value
value      ::= int | string | [$ident .] param
binop      ::= = | != | in | not in
param      ::= field [. field]
field      ::= ident [[ n-zero ]]
```

Fig. 4: Supported subset of the HPL language.

automating code retrieval, analysis and reporting for ROS repositories in continuous integration.

As a running example, consider a simplistic ROS Controller, where a `Controller` node publishes commands in a topic `cmd` at a certain rate, according to data provided by a `Teleop` node through topic `tel`. A `Base` node 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 `cmd` message with value 0 and a "stop" warning.

HAROS provides two plug-in entry points for analysing ROS repositories: one for packages and source files, and another for architectural models per launch configuration, automatically extracted from the source code [5]. 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 technique, essentially the ROS computation graph. Support for services and parameters is left as future work. Fig. 3 presents the architectural model for a `simple` configuration of Controller under this meta-model.

HAROS supports behavioural specifications for maintained repositories to be used by the plug-ins. These must be defined by domain experts for each configuration and node, but the latter can be re-used between applications. Figure 4 presents the supported subset of this domain-specific language, HPL, which acts at the message-passing level, treating nodes

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

Fig. 5: Behaviour of Controller nodes in HPL.

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

Fig. 6: Controller system-wide properties in HPL.

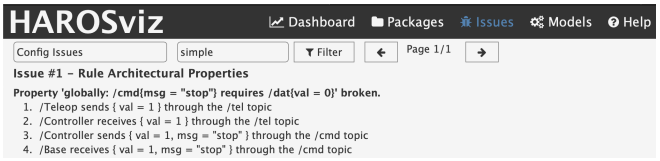

Fig. 7: Safety issue for Controller in HAROS.

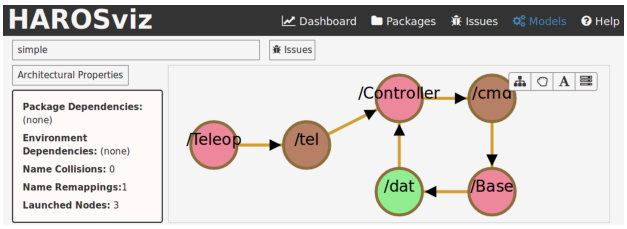

Fig. 8: Safety issue for Controller in HAROS architecture visualizer (involved elements in red).

as black-boxes. Its semantics is formalized in first-order linear temporal logic over ROS event traces based on well-known property patterns [6], supporting both safety – *absence* and *precedence* – and liveness – *existence* and *response* – properties. Events amount to messages occurring in topics, which may be tested according to certain conditions over their fields. Scopes may also be assigned to properties through bounding events. Due to the nature of Electrum the real-time features of HPL are not supported, so the plug-in translates HPL to first-order linear temporal logic.

Part of the behaviour of Controller nodes is specified in Fig. 5. `Teleop` and `Base` must publish messages within the expected value ranges. For `Controller`, three safety properties are enforced: while danger messages are received from `dat` only 0s may be published at `cmd`; 0s published at `cmd` are propagated from `tel` or originate from danger `dat` messages; and $\neq 0$ `cmd` messages are propagated from `tel`. Note that these do not force the actual publication of messages, but specify safe behaviour. A liveness property forces danger `dat` messages to produce a "stop". Two system-wide safety properties, expected to arise from the node behaviour under the `simple` configuration, are shown in Fig. 6: no value other than those published in `tel` ever reaches `cmd` and that "stop" `cmd` messages arise from 0s at `dat`.

Any issue detected by a plug-in is reported in the unified HAROS interface, which for architectural plug-ins are associated with a specific launch configuration. Plug-ins may be provided additional parameters, which in this case are the model checking scopes (explained in Section III). The plug-in keeps a mapping between ROS and Electrum artefacts, so that the abstract execution traces found by the Analyzer can be converted back into the ROS domain. One such issue is depicted in Fig. 7 for Controller: the second specified property does not actually hold, since we are forcing `Controller` to publish "stop" messages after receiving 0 values, but not forbidding their occurrences in other circumstances. Alternatively, the counter-example may be inspected in the architecture visualizer, as shown in Fig. 11.

III. Formalization of ROS configurations

Safety verification is performed by an Electrum backend[2], launched by the plug-in for ROS applications maintained by HAROS, as depicted in Fig. 1 and described in this section.

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.

Formalized ROS configurations share a structural header, depicted in the first part of Fig. 9. Signatures `Node` and `Topic` represent the elements of the configuration; they will be extended exactly by the specific elements of each configuration. For each node, static fields `subs` and `advs` represent the **`set`** of subscribed and advertised topics, respectively, while variable fields `inbox` and `outbox` represent at each instant the **`set`** of messages waiting to be processed and propagated, respectively. Signature `Field` represents message fields. Lastly, signature `Message` denotes the set of available messages in each trace and `Value` abstracts possible message field values, with 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, message types are not explicitly modelled, but are inferred from the assigned topics to enforce the mandatory fields for each message, as will be shown briefly. Unlike the fixed configuration signatures, an arbitrary number of messages and values will be considered in the analysis, within predefined bounds.

[2]The tool, documentation and guides are available at `http://haslab.github.io/Electrum/`.

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

Fig. 9: Encoding of the Controller architecture in Electrum.

The second part of Fig. 9 instantiates the `simple` configuration of Controller. The translation of nodes and topics is straightforward, and results in the declaration of singleton (**`one`**) signatures for topics (e.g., `dat`) and nodes (e.g., `Base`) and the enforcement of subscriptions and advertisements through **`fact`** `Links`, which in Electrum impose model axioms. Here, $\rightarrow$ is the Cartesian product and `+`/`&` set union/intersection. Fields are then declared (e.g., `dat_val`). To avoid encumbering the analysis, fields not mentioned in the specifications are ignored. Fact `Fields` guarantees that *i*) each field value is correctly typed (e.g., l. 19), and *ii*) each message has exactly one value assigned per relevant field (e.g., l. 20). **`all`**, **`some`** and **`no`** are first-order quantifications, and `.` relational composition, thus `topic.tel`, e.g., denotes all messages with topic `tel` assigned.

Being SAT-based, Electrum is not well-suited to deal with numerical values. Thus, message values are discretized by imposing a hierarchy on uninterpreted `Value` atoms so that message content can still be considered without explicitly representing numerical values. This process must consider the behavioural specifications, since different values are relevant for different message conditions (which may test the equality with constant values or message fields, or the membership in sets or numeric ranges), and works as follows:

- for each constant value in a condition test, 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

```
1  fact Messages {
2    no outbox+inbox
3    always {
4      all n: Node |
5        n.inbox.topic in n.subs and n.outbox.topic in n.advs
6      all m: Node.outbox {
7        all n: subs.(m.topic) | eventually m in n.inbox
8        eventually m not in Node.outbox }
9      all m: Node.inbox |
10       before once m in advs.(m.topic).outbox } }
11 -- Application-specific behaviour
12 fact NodeBehavior { always {
13   no m: Teleop.outbox&topic.tel |
14     tel_val.(m.val) not in Int_0_100
15   ...
16   all m: Controller.outbox&topic.cmd |
17     cmd_val.(m.val) != Int_0 implies before once
18       some m0: Controller.inbox&topic.tel |
19         tel_val.(m0.val) = cmd_val.(m.val)
20   all m: Controller.inbox&topic.dat |
21     dat_val.(m.val) = Int_0 implies after eventually
22       some m0: Controller.outbox&topic.cmd {
23         cmd_val.(m0.val) = Int_0
24         cmd_msg.(m0.val) = Str_stop } } }
25
26 assert simple0 { always {
27   no m: Node.outbox&topic.cmd |
28     cmd_val.(m.val) not in Int_0_100 } }
29 assert simple1 { always {
30   all m: Node.outbox&topic.cmd |
31     cmd_msg.(m.val) = Str_stop implies before once
32       some m0: Node.outbox&topic.dat |
33         dat_val.(m0.val) = Int_0 } }
```

Fig. 10: Encoding of the Controller behaviour in Electrum.

the respective signatures (e.g., l. 30);

- identify which numeric values/ranges are completely disjoint with one another and force the disjointedness of the respective signatures (e.g., l. 31).

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 linear temporal logic, including future operators **after**, **eventually** and **always**, and their past counter-parts **before**, **once** and **historically**. Such properties can be used as an axiom in a **fact** – restricting the valid traces that will be considered during model checking – and as an assertion in an **assert** – that will be checked over all valid traces. Thus the same idiom can be used to impose the behaviour of individual nodes and to check system-wide properties, which are both specified in HPL in HAROS.

Since nodes are loosely specified at the interface level and without real-time considerations, our behaviour encoding is under-specified to allow for alternative behaviours and event interleavings. Fact `Messages` in Fig. 10 encodes the message-passing process shared by all formalized configurations. It forces inboxes and outboxes to start empty (without temporal operators, formulas refer to first state, l. 2), and then restricts message-passing in all states (**always**) by enforcing that:

- messages in inboxes and outboxes are correctly typed (l. 5). Note that, e.g., `n.inbox.topic` retrieves the topics of all messages in the inbox of `n`;
- messages in an outbox eventually reach the inboxes of all subscribing nodes (l. 7) and leave the outbox (l. 8). Notice that no timing constraints are imposed and that nodes may receive the message at different states;
- messages in an inbox must have previously been in the outbox of an advertising node (l. 10), so that no messages spontaneously appear in inboxes.

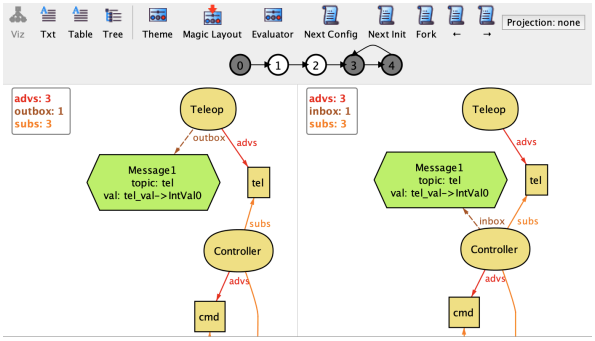

Fig. 11: Counter-example for Controller in Electrum.

The second part of Fig. 10 instantiates the specifications of the Controller nodes and configuration, adapting the LTL formalization of specification patterns [6] to LTL with past operators. The former is encoded in fact `NodeBehaviour`, which restricts how nodes process messages from the inbox to the outbox. For instance, the absence pattern of `Teleop` (ll. 13–14) states that outbox messages must have values within 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 outbox messages require the previous existence of messages in the inbox, such as the one in ll. 16–19 for `Controller`, and dually for response patterns, as in ll. 20–24.

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 `simple1` whether "stop" `cmd` messages are always preceded by a 0 `dat` message. The difference between the encoding of node and configuration specifications is in the scope of the message events: the former focuses on the inbox and outbox of the node being specified; the latter on any message passing in a 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`).

These assertions can be automatically verified by the Electrum Analyzer once a scope is provided for the non-exact signatures (`Value` and `Message`) and the maximum trace length, since by default Electrum performs *bounded* model checking. Scopes are defined in the plug-in option 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, `simple0` holds but not `simple1` (with up to 5 messages/values and maximum trace length of 10). The Analyzer can be used as a stand-alone, in which case the counter-example could be explored in its visualizer as in Fig. 11, currently focusing on the transmission of a message from the outbox of `Teleop` to the inbox of `Controller`. The plug-in translates such traces back into the ROS-domain,

Fig. 12: The AgRob V16 robot monitoring a slope vineyard.

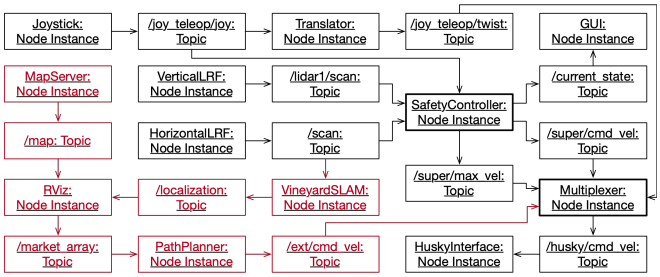

Fig. 13: Simplified HAROS architectural model for AgRob V16 (elements of the `map` configuration in red).

resulting in the issue already presented in Fig. 7.

IV. Evaluation

Our evaluation of the approach 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 it to a real robot under development, AgRob V16.

A. Case study

AgRob V16 (agrob.inesctec.pt, Fig.12) is a modular robot for precision farming in slope vineyards developed in C++ ROS. One of its main challenges is the operation in an unstructured environment shared with human operators, which also renders it safety-critical. Furthermore, it has been developed with a focus on modularity and extensibility, relying on third-party ROS packages, making it prone to launch configuration errors that may affect safety properties. For these reasons, safety feedback in continuous integration would prove valuable to the development team.

The robot follows a typical architecture of sensors, controllers, planners and actuators. Some features are optional, 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 this evaluation we focus on two particular configurations: `startup` for 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.

```
globally: /current_state{data[0]=6} requires
                              /joy_teleop/joy{button[0]=1}
globally: no /super/cmd_vel{linear.x not in 0 to 10}
globally: /super/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}
```

Fig. 14: Part of `SafetyController` behaviour in HPL.

```
globally: /agrobv16/current_state{data[0]=3} requires
            /joy_teleop/joy{button[0]=0, button[1]=1}
globally:
  /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}
```

Fig. 15: Some AgRob V16 system-wide properties in HPL.

A simplified version of the AgRob V16 architecture, as extracted by HAROS, is depicted in Fig. 13, highlighting the differences between the two configurations `SafetyController` and `Multiplexer` are the core nodes and are responsible for ensuring safe behaviour. The controller monitors data from sensors – lasers `HorizontalLRF` and `VerticalLRF` – and the selected operating mode from the `Joystick` to issue safe velocity commands and the current state to be displayed by the `GUI` (through `data[0]`). The multiplexer collects commands from the controller, the `Joystick`, and, under the `map` configuration, the `PathPlanner`. Following a set of priorities, commands are passed to the `HuskyInterface` which communicates with the hardware actuators. Joystick mode is switched on through `joy` messages with `button[0]` $= 1$, and the "follow path" mode through `button[0]` $= 0$ and `button[1]` $= 1$, having lower priority; in joystick mode, `buttons[4]` and `button[5]` issue velocity commands.

Through discussions with the AgRob V16 developers, behaviour of nodes and the desirable safety properties were encoded in HPL. This resulted in 30 specifications over 9 nodes and 4 system-wide properties. Figure 14 presents part of the `SafetyController` specification, the node with richer behaviour, such as how the information published to `current_state` for the `GUI` is determined from the teleoperation buttons; the ranges of linear and angular velocities published at `cmd_vel`; and how certain velocity values are restricted to modes or other teloperation commands.

Figure 15 depicts some configuration properties, namely that `current_state` only contains the "follow path" mode (`data[0]` $= 3$) if the corresponding buttons have once been pressed; and `cmd_vel` messages to rotate in-place (`linear.x` $= 0$ and `angular.x` $\neq 0$) are caused by a `scan` message with a range smaller than 40cm ($0 \leq$ `ranges[0]` ≤ 4), or the system is in teleoperation mode.

The 4 properties were checked for the two configurations with increasing scopes for signatures `Value` and `Message` for traces with up to 10 distinct states, in a 2.4 GHZ Intel Core i5 with 8GB memory running Electrum in bounded mode with SAT4J, as summarized in Table I. One property was actually shown not to hold for the `map` configuration: commands to rotate in-place in "follow path" mode may be caused by the

TABLE I: Evaluation results, in seconds.

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

`PathPlanner` without the lasers detecting obstacles due to accumulated localization errors. A HAROS issue would show a message being passed from the `PathPlanner` to the `SafetyController`, which would identify a dangerous situation and instruct the `HuskyInterface` to rotate, without the lasers publishing any message. This counter-example requires 4 values/messages, but even with scope 10 it is still found under 1min. In fact, all properties were checked with scope 14 at around 2min., feasible if it is to be run in continuous integration, although the trace length may need to be increased for additional confidence.

B. Threats to validity

The assignment of the model checking scope for messages/-values to a ROS repository must be handled with care, since small universes may hide potential safety issues. We believe that with application-specific knowledge of the development team it is possible to infer sensible scopes, but whether this applies to 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

Static analysis techniques to address specific ROS issues have been proposed. The initial release of HAROS focused on internal quality metrics and conformity with coding standards [4]. Ore et al. propose a technique [7] to detect inconsistencies between physical units in C++ ROS code, relying on the *a priori* annotation of standard libraries.

Some approaches extract intermediary models from ROS code but not for general-purposes analyses. Purundare et al. [8] propose a technique that extracts from C++ code a model of the message flow between components to identify code changes that may impact message-passing. Sharma et al. [9] also address the impact of code changes by extracting a data flow model from C++ code, but focus on the impact to the rate of message publication. Muscedere et al. [10] focus on the detection of feature interaction symptoms by extracting a "factbase" from C++ code, over which user-defined code queries are executed. These can be used to identify code patterns but not dynamic behaviour. Our approach builds on work by our team [5] on formalized ROS configurations and proposes a technique for their extraction from C++ code for subsequent analyses. A query language is provided to detect simple architectural patterns, but not dynamic behaviour.

Some work has been done on the verification of safety for ROS applications through off-the-shelf model checkers, namely SPIN by Webster et al. [11] and UPPAAL by Halder et al. [12]. However, these are mostly exploratory works based on the *ad hoc* formalization the of robotic software.

VI. Conclusions

This paper presented a model checking technique to verify message-passing system-wide safety properties based on a formalization of ROS launch configurations and loosely specified behaviour of individual nodes. It is wrapped in a HAROS plug-in that automatically creates such Electrum models – from configurations extracted in continuous integration and specifications provided by the domain experts – and translates abstract counter-examples back into ROS-flavoured HAROS issues. Its application to a real robot showed it to be sufficiently expressive to check certain classes of safety properties, and to have a reasonable performance. The formalization could be applied to other message-passing middlewares, but the focus on ROS enabled the automatic extraction of architectures by HAROS.

In the future we plan to explore techniques to support the user when specifying scopes and techniques to help discard false positives, possibly relying on run-time analyses to check whether counter-examples are possible execution traces. We are also exploring strategies to improve the scalability of the approach and to support certain classes of timed properties.

References

[1] 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.

[2] 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.

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

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

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

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

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.