Property-Based Testing for the Robot Operating System

André Santos
INESC TEC & University of Minho
Braga, Portugal
andre.f.santos@inesctec.pt

Alcino Cunha
INESC TEC & University of Minho
Braga, Portugal
alcino.cunha@inesctec.pt

Nuno Macedo
INESC TEC & University of Minho
Braga, Portugal
nuno.m.macedo@inesctec.pt

ABSTRACT
The Robot Operating System (ROS) is an open source framework for the development of robotic software, in which a typical system consists of multiple processes communicating under a publisher-subscriber architecture. A great deal of development time goes into orchestration and making sure that the communication interfaces comply with the expected contracts (e.g., receiving a message leads to the publication of another message). Orchestration mistakes are only detected during runtime, stressing the importance of component and integration testing in the verification process. Property-based Testing is fitting in this context, since it is based on the specification of contracts and treats tested components as black boxes, but there is no support for it in ROS. In this paper, we present a first approach towards automatic generation of test scripts for property-based testing of various configurations of a ROS system.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging; Publish-subscribe / event-based architectures; • Computer systems organization → Robotics;

KEYWORDS
Software Testing, Test Automation, Property-based Testing, Robot Operating System

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

A-TEST ’18, November 5, 2018, Lake Buena Vista, FL, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6053-1/18/11...$15.00
https://doi.org/10.1145/3278186.3278195

1 INTRODUCTION
Software testing has become a staple in most quality assurance processes, providing valuable feedback regarding the correctness of the software under test. In particular, testing is often a mandatory step in the verification of safety-critical systems [10].

With the recent developments in robotics, robots are taking their place in various safety-critical application domains, such as health, industry and agriculture. As their capabilities increase, so does the complexity of the software behind these systems. Considering the possible dangers of human-robot interaction, and how expensive the systems themselves are, ensuring that they are correct and well tested is a necessity. However, developing a fairly complete test suite for a robot, covering a wide variety of scenarios, can be challenging, especially at the level of integration and system testing, due to the heavy coupling of these systems on input and output (sensors and actuators).

The Robot Operating System (ROS)\(^1\) [11] is one of the most popular open source frameworks for the development of robotics systems, with thousands of users worldwide. It provides middleware, libraries and tools that aim to shorten development time and encourage re-use of existing components. A typical ROS system is a set of independent processes (called nodes) communicating with each other through message-passing. Most of the time, nodes communicate asynchronously in a publisher-subscriber fashion (called topics), although a client-server model is available (called services).

Due to the distributed and re-usable nature of ROS components, part of a ROS developer’s job is to integrate nodes, using configuration files and name aliases, to ensure that topics and services match. Component integration is a challenging task [4] that is very prone to human error, and a common source of bugs. Such integration bugs tend to be detected early on, as developers notice when a node is not receiving expected messages. Nonetheless, these issues are only detected manually with the assistance of runtime inspection tools. In this regard, automated testing would decrease development time, besides providing a systematic way to make sure that the whole configuration adheres to an intended architecture.

There is some support in ROS for automated testing\(^2\) using popular testing libraries, such as Google’s gtest for C++ code and unittest for Python code, but this is mostly appropriate for library unit tests and node unit tests (testing the ROS interface of a single node). ROS also provides simulation environments and message replay tools, which alleviate hardware dependencies when testing.

Many desirable safety properties in a ROS system apply at the interface and integration levels, and, thus, unit testing does not suffice. For instance, consider a simple system in which a node handles safety behaviours, while another node publishes sensor readings and subscribes to actuator commands. This is the case, for instance, with Kobuki\(^3\), a popular ROS robot used in research and education. A desirable property for this system would be to ensure that the safety controller node publishes a command (e.g. stopping) after receiving a sensor message signalling a bump into an obstacle, and that this message is received by the base node, to relay it to the actuators. These properties can be further refined by adding

---

\(^1\)http://www.ros.org/
\(^2\)http://wiki.ros.org/Quality/Tutorials/UnitTesting
\(^3\)http://kobuki.yujinrobot.com/
Timing constraints (e.g. the stop command being published within, at most, one second after the bump).

Such properties can be specified and tested with manually crafted test scripts, but this can prove to be cumbersome with the default testing framework. A more efficient strategy is to exploit Property-based Testing (PBT), a property-oriented testing method in which common approaches use seemingly random input generators in a systematic way, attempting to falsify properties specified by the developers. Using random inputs may lead to redundant tests, but, on the other hand, they have the potential to uncover defects that a test developer would not come up with.

Our approach adapts this idea to the context of a ROS system. Simply put, we take a set of nodes, ranging from a single node to an entire application, and we consider it as a black box. The inputs for this black box are open subscribed topics (topics where there are no active publishers) and the outputs are open published topics (topics where there are no active subscribers). Then we use Hypothesis, a PBT library, to find a sequence of ROS messages that either crashes the configuration under test, or falsifies a specified property. Due to the random nature of PBT, this approach is most suitable for pure software nodes or hardware controllers using simulation.

Still, writing such tests for various ROS configurations would be as error prone as orchestrating the systems themselves, since node and topic names must be specified. Ideally, this task should be automated, so that, for instance, typing mistakes are completely avoided. As such, inspired by Model-based Testing approaches, we propose an automatic test generation method from configuration models extracted using HAROS, a static analysis framework for ROS applications. Our contribution, thus, is a prototype that is capable of taking models of ROS configurations and generating customisable, property-based test scripts for said configurations.

Throughout the paper, we will use a fictitious mobile ROS robot, called FictiBot, as our running example. This robot is essentially composed of two ROS nodes: fictibase, a low-level driver node that directly controls the robot’s sensors and actuators; and ficticontrol, a higher-level controller node that generates a random robot command based on sensor readings. When both nodes are properly integrated, the driver feeds the controller by publishing sensor information, while the controller feeds the driver back by publishing robot commands.

In the remainder of this paper, we start, in Section 2, by providing necessary background for understanding the problem and our contribution. Then we present our prototype test generator, in Section 3, going into detail on the test generation process, from HAROS models to a concrete test script, but also describing the inner workings of the test script itself. We compare our approach to other relevant work in Section 4. Finally, we wrap up with a few conclusions and our directions of future work, in Section 5.

2 BACKGROUND

2.1 Property-based Testing and Hypothesis

Property-based Testing is a testing technique in which the testing process is driven by the specification of generic properties that a system should satisfy [7]. Properties are often assertions on how outputs relate to inputs, and often fall into common patterns, such as how two procedures achieve an equivalent result. A typical example is that applying reverse on the reverse of a list returns the original list, regardless of the list. This immediately contrasts with typical example-based testing methods, where the goal is to verify that the system behaves well for a number of crafted scenarios. An advantage of PBT is that it requires a more formal reasoning about the tested systems than traditional testing methods do. The formulation of specifications, in turn, helps in documenting and building hierarchical models of the system behaviour [2].

In practice, PBT libraries take the property-based test cases and execute them repeatedly, in an attempt to find a counterexample for the specification. Inputs are automatically selected from a large corpus, possibly dynamically generated, and possibly selected at random. Going back to the reverse example, that property would be tested with lists of varying length and contents, ranging from an empty list up to lists of a certain length limit.

PBT is most often applied in unit testing, although it has been applied in testing of web services, for instance. It became popular as a testing method with QuickCheck [5] for the Haskell programming language. It fits very naturally with the pure functional programming style of Haskell, although modern PBT libraries [2, 5, 8] (QuickCheck included) are able to handle stateful systems and software with side effects just as well. Hypothesis is an example of such a PBT library, mostly for the Python programming language.

Stateful systems can be tested with Hypothesis using a construct called a RuleBasedStateMachine. Test developers extend the RuleBasedStateMachine class to add their own internal state, to define set up and tearing down of class instances, and to define allowed operations (state transitions, called rules in Hypothesis). Additionally, rules can be decorated with preconditions, and invariants can also be specified. Properties in this case are regular Python assertions, written within rules or invariant definitions. The test execution consists of Hypothesis repeatedly creating instances of the state machine, and executing a sequence of rules with generated inputs.

An example of stateful testing in Hypothesis is provided in Figure 1. This is a naive example, asserting that trying to remove a random integer from a list of integers will result in a list of the same length or shorter. This property would be correct, in theory, but calling remove in Python with an element that is not present in the list results in an error. To fix this example, either the call to remove must be guarded with a conditional, or the error handled.

2.2 The Robot Operating System

ROS is a multi-language framework, where nodes written in different languages – the two most common ones being C++ and Python – can interact seamlessly. A typical ROS system consists of a peer-to-peer network of nodes, exchanging messages using topics for a publisher-subscriber model, or services for a client-server model. All nodes using a topic or service should exchange messages of the same type. ROS provides a few basic message types, but users can define their own using a message definition language. During the
Property-Based Testing for the Robot Operating System

```python
class ListMachine(RuleBasedStateMachine):
    def _init_(self):
        super(ListMachine, self)._init_()
    @rule(value=integers())
    def append(self, value):
        self.state.append(value)
    @rule(value=integers())
    def remove(self, value):
        self.state.remove(value)
        previous_length = len(self.state)
        assert len(self.state) == previous_length
```

Figure 1: Minimal Hypothesis example for stateful testing.

![Minimal Hypothesis example for stateful testing.](image)

Figure 2: ROS Computation Graph of FictiBot. Circles represent nodes, rectangles represent topics. Outgoing arrows represent publications, incoming arrows represent subscriptions.

project’s build phase, ROS tools generate source code for message types in all target languages.

There is a special node in all ROS systems, called the ROS Master, which is always started by default. Its purpose is to help set up the network, by providing nodes with peer discovery, and to hold a shared key-value store where nodes can read and write key-value pairs (called parameters) at runtime. Collectively, nodes, topics, services and parameters are called resources, and the network of ROS resources is called the ROS Computation Graph\(^\text{8}\). Figure 2 shows a diagram of the Computation Graph for our FictiBot running example. Note that the ROS Master is omitted from the diagram.

Computation Graphs are dynamic, meaning that resources can join and leave at any time. Developers use XML-like files, called launch files, to deploy sets of nodes and parameters into a Computation Graph (or to create a new one), using a tool called roslaunch. Launch files also provide mechanisms to access environment information, group resources under a namespace and deploy resources conditionally. Another core feature of launch files is a name aliasing mechanism, called remappings or remaps, which is fundamental to orchestrate and re-use nodes. For instance, if a node is configured with a remapping A → B, it would be transparently forwarded to a resource named B upon requiring a resource named A. Verification of launch files is mostly limited to syntax and lookup of node executables, which makes component integration all the more challenging. Figure 3 shows an example launch file, used to deploy the driver and controller nodes of our running example.

\(^8\)http://wiki.ros.org/ROS/Concepts

![Diagram of the Computation Graph for FictiBot.](image)

Figure 3: A minimal ROS launch file that deploys a driver node and a higher-level controller node.

![Diagram of the Computation Graph for FictiBot as extracted by HAROS.](image)

Figure 4: Computation Graph of FictiBot as extracted by HAROS. Nodes are shown in white and topics in green.

2.3 The HAROS Framework

HAROS is a framework for static analysis of ROS applications. As of version 3.0, it is capable of extracting Computation Graph models from the source code, although this feature is currently only available for C++ code. This works by the user defining Configurations in YAML project files, essentially named lists of launch files which are parsed in order. For each launch file, HAROS extracts the set of participating nodes. Afterwards, the source code for each node is parsed and the model extended with the usage of ROS primitives (e.g. subscribing to a topic). As topics are detected, the associated message type is determined and annotated too.

HAROS tries to resolve as many conditions and variables as possible in static time, but, due to the dynamic nature of ROS, this analysis is limited. Nodes may not be launched from a launch file if a condition is not satisfied, and topics are created on demand, as nodes call the ROS API with arguments that may be themselves the result of some computation, or collected from external sources (e.g. parameters). The resulting model marks conditional resources as such (including the respective conditions) when HAROS is unable to fully resolve them. To alleviate this issue, users can provide hints when defining a configuration (e.g. topics that a node may subscribe to), which are used by HAROS in the extraction process. Fortunately, conditional topics are not very common in practice [12], and thereby HAROS is able to cover a significant ROS corpus. Figure 4 shows the diagram of our running example, as extracted and depicted by HAROS.

3 PROPERTY-BASED INTEGRATION TESTING FOR ROS

Property-based Testing, as presented in Section 2.1, fits very naturally in unit testing and, with the addition of stateful testing, can even be used to test full components. To leverage PBT at the integration testing level, our approach is to consider the set of integrated

\[\text{assert \text{len}(\text{self.state}) <= \text{previous_length}}\]

\[\text{self.state.remove(value)}\]

\[\text{previous_length = \text{len}(\text{self.state})}\]

\[\text{assert \text{len}(\text{self.state}) == previous_length}\]
whereas PBT is often used to test single functions or synchronous

A-TEST '18, November 5, 2018, Lake Buena Vista, FL, USA
André Santos, Alcino Cunha, and Nuno Macedo

configuration as the product of such lists. Providing a list of launch
candidates can join and leave at any time, there is no clear definition
of what is a ROS application. Lists of launch files are possibly the
presented in Figure 3 to construct the model depicted in Figure 4.

Before the test script generation process begins, configuration mod-
els must be extracted. Thus, the first step, is to specify the intended
ROS configurations in the HAROS project files. As mentioned in
Section 2.3, this boils down to a list of launch files and optional
extraction hints. We assume that the user provides enough spec-
fication for the extracted model to be correct. Test settings that
play a role in the test generation process are also specified at this
stage. Figure 6 shows a HAROS project file using the launch file
presented in Figure 3 to construct the model depicted in Figure 4.

Since ROS promotes open and dynamic systems, where partic-
cipants can join and leave at any time, there is no clear definition
of what is a ROS application. Lists of launch files are possibly the
best candidate definition, which is why HAROS considers a Con-
figuration as the product of such lists. Providing a list of launch

Hypothesis already handles basic types and lists, so we just had
files, instead of a simple list of nodes, is also advantageous for the
purposes of test generation, since the generated test script is able to
 reproduce the configuration as it is in a normal setup. This is an
important detail, because the test routine requires launching
the same configuration multiple times, as we explain in Section 3.2,
and it must be accurate in doing so, including all ROS parameters
and evaluated variables.

After the model extraction takes place, the test generation pro-
cess begins. Identifying the list of participating nodes is straightfor-
ward, so the next step is to identify the input and output topics. We
 treat the configuration under test as a black box. As such, our cur-
rent approach only considers open topics (topics with subscribers
but no publishers, or vice-versa) as candidates for testing. Further-
more, we also discard nodes and topics that the analysis could not
fully resolve.

For instance, when testing the whole minimal configuration from
our example, the only open topic is teleop_cmd, with a single
subscriber. The corresponding test script would generate a publisher
for this topic, and the testing would boil down to trying to make
the configuration crash with random messages on this topic. If we
tested each node in isolation, however, the generator would create
all the corresponding publishers and subscribers for each topic. We
considered generating publishers and subscribers for connected
topics as well, but this would interfere with the integration we are
supposedly testing for correctness.

The final step in the test generation process is to construct ROS
message generators for Hypothesis, called strategies in Hypothe-
sis terminology. PBT libraries, Hypothesis included, often provide
generators for common and basic types of data out of the box, but
custom or complex data types must be specified manually. ROS
messages follow a well-defined format\(^\text{10}\), in which message fields
can be of a basic type (numbers and strings), another message type
(composition) or lists of one of the previous. As such, the genera-
tion of message strategies can be automated by traversing message
fields and producing the respective sub-strategy.

Hypothesis already handles basic types and lists, so we just had
to provide the corresponding value limits (e.g. a strategy for un-
signed 8-bit integer fields that generates values between 0 and 256).
To handle composition, we extended the generation algorithm with
recursion and caching. Although this is enough to achieve a work-
ing implementation, in some cases developers do not expect the full
range of values a type may provide, but intend for certain fields to
behave like an enumeration (e.g. LED flashing red, green, or turned
off). This is a known limitation of ROS, as it supports constants, but

\(^9\)https://github.com/git-afsantos/haros/blob/dev-make-tests/haros/gen_tests.py

\(^{10}\)http://wiki.ros.org/msg

Figure 5: Workflow of the test generation process.
not proper enumerations. Thus, we have implemented support for users to define simple enumerations, using either literal values or constants defined within a message type. This is a way to eliminate many random tests that would provide little value in the end. On the other hand, use of this feature must be pondered, as unexpected inputs can help uncover more faults. Figure 7 shows the automatically generated Hypothesis strategy for a custom message type, illustrating enumerations (the state field), composition (the pose field) and lists (the bumpers field).

3.2 The Test Script

The entry point of a test script performs only a few operations before the test routine starts. First off, in order to use ROS interfaces, the test script has to register itself as a ROS node. Then it creates a TestSettings object and an InternalState object, which it passed down to the test routine. TestSettings objects contain configuration-specific data, such as the required launch files, or the published topics. This is required, since the test routine is generic. In general, users shall not find the need to edit these, unless they want to ignore a specific topic, for instance.

We cannot determine in advance which properties users might want to verify, so the only way to specify custom properties, besides not crashing, is to edit the test script and add assertions manually. The InternalState class, which is just a template, is meant for this purpose. Users can extend it, and define state variables and callback functions for messages that the test node publishes or receives. The idea behind this is to allow relatively complex properties to be expressed, such as temporal properties (e.g., a message has arrived within X seconds of a previous one), or properties that depend on the history of exchanged messages (e.g., a message field contains monotonic increasing values over time). Expressing properties this way is also in line with the style advocated in Hypothesis stateful testing. Figure 8 shows the generated InternalState template when testing the ficticontrol node of our example in isolation.

We use Hypothesis' RuleBasedStateMachine, as presented in Section 2.1, to perform stateful testing. Since we are focusing on the publisher-subscriber aspect of ROS, we define two simple operations, publishT and spin, where publishT publishes a randomly generated message on topic T, and spin (using a common terminology in ROS), processes incoming messages and sleeps for a given time, to achieve a certain loop rate. For each input topic, as determined in the test generation process, a publish operation is dynamically defined.

Hypothesis will repeatedly try various sequences of publishT and spin operations, until a property is falsified or a limit is reached and all properties are deemed true. This implies that, if the first sequence of operations does not run into an error, there must be a way to reset the complete internal state, so the second iteration starts from a deterministic state. This is also one of the big design challenges we faced. It is a simple matter in regular unit tests, where it suffices to define custom set up and tear down functions that manipulate the state as necessary. When testing a generic set of ROS nodes – which are completely independent processes from the testing script – there is no standard way to reset the internal state of such nodes as required. Thus, we settled on a sensible, although not very performant option, which is to shutdown the whole configuration under test after a successful iteration, so that the configuration can be restarted and likely be in a clean state. It is mostly for this reason that we keep references to the original launch files.

By default, the test node already checks two simple properties. The first property states that all nodes under test must be alive. We make use of internal libraries of ROS to ping other nodes with a timeout. The test fails if any node terminates (due to error or otherwise) or becomes unresponsive. The second property we test for is that the tested ROS interface is stable, i.e. the set of published and subscribed topics should not change during the test run. It is uncommon, in general, to come up with a use case in which closing a topic is a requirement, and so we treat it as an error.

3.3 Preliminary Experiments

To evaluate the usefulness of our framework in a more realistic scenario, we applied it to Kobuki, an education oriented mobile ROS robot that comes out of the box with various different configurations. At the most basic level, it simply runs the base node, which is responsible for publishing sensor data and subscribing to velocity commands, similar to the fictibase node from our example. Other configurations, enabled in separate launch files, add various features such as teleoperation, a safety controller, or a random walk controller. We chose the safety controller as our initial test target.

The safety controller subscribes to sensor topics (bumper, wheel drops and cliff detection) and publishes velocity commands. Upon receiving sensor messages, the callback functions simply update the controller's internal state with the new information. Its publishing
class InternalState(object):
    def __init__(self):
        self.on_setup()
    def on_setup(self):
        self.bumper_left = False
        self.bumper_right = False
        self.bumper_center = False
        # other boolean variables...
        if event.msg.linear.x <= 0:
            assert (self.bumper_left or self.bumper_right
                    or self.bumper_center or self.cliff_left
                    or self.cliff_right or self.cliff_center)
    def on_events_bumper(self, event):
        if event.msg.bumper == BumperEvent.LEFT:
            self.bumper_left =
            # repeat for bumper_center and bumper_right
            def on_events_wheel_drop(self, event):
                # update internal state
                # update internal state

Figure 9: Model of Kobuki’s safety controller.

Figure 10: Specification of a safety property with internal state.

loop checks the internal state and publishes a zero velocity message
if a wheel is dropped, and a negative velocity message (backward
movement) when one of the other sensors is triggered. Figure 9
illustrates the node’s behaviour.

An immediate property to take out of this setup is that receiving a
message of negative velocity implies that the last published message
of bumper or cliff sensors should contain an active state. For the
sake of an illustrative example, we specified that a velocity of zero
or less implies a bump or cliff, as seen in Figure 10. Hypothesis
has been able to consistently find a minimal counterexample to
this property, which is sending a single wheel drop message and
waiting for the corresponding zero velocity message. Figure 11
shows a sample of the produced output.

We ran this experiment within a 32-bit virtual machine using a
single 2.0 GHz processor and 2 GB of memory. On average, the test
runs take about 62 seconds, of which more than 90% of the time
is spent setting up and tearing down the ROS configurations (spawning,
killing and waiting for processes and network connections),
and only a minimal fraction is spent on the actual testing, publish-
ning and waiting for messages. These results clearly show how
detrimental to performance our approach to resetting in-between
test iterations is. On the other hand, the fact that Hypothesis is able
to find a counterexample shows that our approach is feasible, even
if it needs optimisation.

4 RELATED WORK

Automated software testing is very appealing for its promise of
finding defects without much effort from the user, but, in [14],
Vincenzi et al. show that manual tests tend to be more effective.
Even though their work focuses on unit tests for Java programs,
we expect the picture to be the same in our context. They state
that automated and manual tests have a complementary aspect –
whereas manual tests are better to test specific scenarios, automated
tests can uncover unexpected faults. We support this view, since,
in our case, it would be very hard to test specific scenarios with
randomly generated messages.

A proposal for unit testing of publisher-subscriber architectures
is presented in [9]. Their proposal is based on Java programs, anno-
tated with preconditions, postconditions and invariants, specified in
Linear Temporal Logic. The testing framework mocks the publisher-
subscriber infrastructure, so that components can be tested in iso-
lation. Our approach differs in that we are focusing on the ROS
infrastructure, and thus we can make use of domain knowledge.
Besides, our approach does not require as much (neither as formal)
specification in order to function.

When testing stateful systems, most PBT tools generate random
transitions based solely on the current state of the model. With the
integration of external test case generators, more information can
be taken into account, and more meaningful test sequences can
be generated. This testing method is presented in [1], and it is an
approach that we intend to explore further, since, in many cases,
purely random test cases might not make much sense in the context
of a ROS application. Our work differs mostly in that we are testing
asynchronous distributed systems, our test scripts are automatically
generated, and we check architecture-related properties as well.

Despite the importance of testing in robotics, research on the
topic is relatively scarce. In [3] the authors present a methodology
that bridges the gap between modern software testing techniques
and the basic unit testing seen in robotics environments. They apply
it to ROS, incorporating simulation environments as substitutes for
real hardware. In [6], this idea is extended to implement Model-based Testing for ROS systems. As is our case, their work is aimed at the integration testing level, but they focus on the navigation and localisation capabilities of mobile robots only. Topological maps, describing the places a robot should be able to move to, are used to generate Timed Automata that simulate the robot’s behaviour. Generated tests verify that the robot can move according to the given map, and specific test scenarios can be manually specified, but finding the causes of a test failure is left for the user. While our approach is more user-friendly, in that it produces counterexamples for violated properties, testing of navigation and localisation is possibly one of the hardest features to implement correctly with our approach. As such, our approach might be considered as a complementary test method.

5 CONCLUSIONS AND FUTURE WORK

In this paper we presented our initial approach towards providing a PBT tool for robotic software using ROS. Much of what happens in a ROS system is dependent on messages exchanged between components. This makes it so that specifying component contracts is a given, even if implicitly. PBT brings forth an opportunity to make such contracts explicit, besides providing a mechanism to test them systematically.

Our preliminary results lead us to believe that PBT can be effectively applied to ROS systems, despite the challenges of a dynamic, asynchronous architecture. Testing in the ROS community consists mostly of manual unit tests or test scripts that reproduce a specific scenario in a simulated environment. Introducing automated random testing helps uncover unforeseen faults and provides useful counterexamples for user-specified properties. An interesting point of future work would be to generate specific test cases for each counterexample found.

Another benefit of our approach is to have an automated method of verifying that a ROS configuration produces an expected Computation Graph, where nodes and topics are well integrated. This is a common source of bugs during development, as components are re-used and reconfigured multiple times for different applications. On the other hand, we acknowledge that our proposed tool is no substitute for manual testing, but rather a complementary test method.

Regarding future work directions, there are a few paths in our approach that can be further explored. First off, the languages for user-specified properties and constraints should be extended, for instance to allow other common message constraints, such as ranges for numeric values. There is also the possibility of integrating proper Temporal Logic specifications (e.g. imposing an order on published topics, even though the messages are random). Alternatively, we could integrate external test case generators, using a similar approach to the one presented in [1].

Our current prototype can only generate test scripts for full configurations. If a user wants to test just a subset of the nodes, a new configuration has to be created for that purpose. We intend to work on this matter, either by allowing constraints to be specified, or by generating test scripts for all subsets of a given configuration with either single nodes or nodes connected by at least one topic.

Finally, our current method of resetting configurations between test iterations is lackluster in terms of performance, where a great deal of time is spent in setting up and tearing down processes. This is an immediate target for optimisation, although more performant solutions might not be as generic.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. 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