Fuzz Testing in Behavior-Based Robotics

Delgado, Rodrigo; Campusano, Miguel; Bergel, Alexandre

Published in:
2021 IEEE International Conference on Robotics and Automation, ICRA 2021

DOI:
10.1109/ICRA48506.2021.9561259

Publication date:
2021

Document version:
Accepted manuscript

Citation for published version (APA):

Go to publication entry in University of Southern Denmark's Research Portal
Fuzz Testing in Behavior-Based Robotics

Rodrigo Delgado1, Miguel Campusano2, Alexandre Bergel1

1ISCLab, Department of Computer Science (DCC), University of Chile
2SDU UAS, MMMI, University of Southern Denmark

Abstract—The behavior of a robot is typically expressed as a set of source code files written using a programming language. As for any software engineering activity, programming robotic behaviors is a complex and error-prone task. This paper proposes a methodology that aims to reduce the cost of producing a reliable software describing a robotic behavior by automatically testing it.

We employ a fuzz testing technique to stress software components with randomly generated data. By applying fuzz testing to a complex robotic software, we identified errors related to the coding, the way data is handled, the logic of the robotic behavior, and the initialization of architectural components. Furthermore, a panel of experts acquainted with the analyzed behavior have highlighted the relevance and the significance of our findings. Our fuzzer operates on the SMACH and ROS frameworks and it is available under the MIT public open source license.

I. INTRODUCTION

The behavior of a robot is usually modeled and expressed in software. A software for a robot, as any software artifact, is prone to errors and bugs. The software engineering community has produced many techniques to test and discover potential problems in an early software development phase. One popular technique is fuzz testing, which consists in loading a program with random data with the hope of identifying a software failure [1]. Fuzzing complements classical testing techniques by allowing one to discover bugs and errors that would be hard to find using ad-hoc generated input, as frequently happens in a laboratory setting. As far as we are aware, fuzzing for software testing has been superficially considered by the robotic community despite its property to identify software bugs and security issues.

This work describes and empirically evaluates a fuzz testing technique designed to identify bugs in a robotic behavior. Our experiment resulted in identifying critical bugs in a robot competing at the RoboCup. The severity of the bugs and the usefulness of our approach is empirically validated by a panel of experts.

Contributions. This paper proposes a methodology to test a state machine-based software describing the behavior of a robot, testing each state in isolation. Our fuzzer operates with the SMACH framework and is available to the community to replicate our results and apply our technique in this location https://github.com/rdelgadov/fuzz_testing. Note that our fuzzer is distributed under the MIT License, making it business-friendly and available for academic purposes.

Hypothesis and research questions. The hypothesis this paper is based upon is that fuzzing can automatically identify non-trivial bugs in robotic software behaviors. We refer to “non-trivial bugs” as bugs that occur in situations that are not contemplated by a laboratory and controlled setting. To verify this hypothesis, we formulate the following research questions:

RQ1: What are the characteristics of the errors identified by fuzz testing? This question seeks to identify what type of errors and categories of bugs fuzzing in a software-based robotic behavior can be identified.
RQ2: Can fuzz testing detect representative and realistic problems in robotic behaviors? This question relates to the potential of fuzzing to identify bugs in the field vs in a laboratory setting.
RQ3: Can fuzz testing find hard-to-spot errors? This question relates to the complexity of identifying bugs and errors. We rely on the perception of experts to qualify on the complexity of the identification.

II. RELATED WORK

The interdisciplinary nature of our paper’s approach shares various complimentary topics with a number of related works, such as mixing robotic behavior, software development and fuzz testing.

Fuzz testing. Fuzz testing is a technique originally designed to detect vulnerabilities in software systems. The techniques associated with fuzz testing are numerous. For example, the fuzzing can be guided by a grammar [2], [3], [4] or mutation [5], [6] to generate complex structured data.

Fuzz testing mixes random with automated data generation to produce a wide range of plausible values to test the software source code with. It has been shown that different fuzz techniques can find different types of bugs in the same software system [7]. There are some fuzzers that use finite state machines to model the software and apply fuzz tests to the model [8], [9]. This suggests that fuzz testing can be beneficial to a particular context, such as using state machines to define robotic behavior.

Testing robot behavior. Robot unit testing [10] is a close approximation to the classical unit testing in software engineering applied to robotics. Robot unit testing considers simulators as a valid and sufficiently accurate tools to test a robot software. Unit testing automatizes the monitoring of some well defined scenarios. Our approach is therefore complementary to robot unit testing since fuzz testing explores a space of possible scenarios instead of restricting the testing phase to a set of fixed and well determined scenarios.

Laval et al. propose a multi-layered testing methodology [11], in which the safety of the human operators plays
a central role. The methodology they propose relies on the
definition of repeatable, reusable, and semi-automated tests.
A robot is then tested on a wide range of different aspects,
ranging from hardware actuators to software and the robot
behavior. Similarly to robot unit testing, Laval et al. heavily
rely on well defined scenarios the robot has to execute. Our
approach takes a different, yet complementary stance by not
being tied to any fixed and inevitably biased scenarios.

**Fuzzing and robotic.** Our effort is not the first attempt at using
fuzz testing for robotic development. The ros1_fuzzer\[1\]
operates on ROS topics to simulate the data of sensors. Our
fuzzer operates on userdata, and as such, can test an individual
state. Furthermore, our fuzzer uses a grammar, which signifi-
cantly increases the capability of entering into conditional
branches, and therefore identifying more vulnerabilities. Also,
RvFuzzer [12] uses a model-based fuzzer to find valid inputs
that generate errors in command-driven robots. However,
RvFuzzer works at the outermost layer of the software by
testing commands that execute a complete behavior and does
not perform tests between components of the behavior.

### III. STATE MACHINES AND SMACH

SMACH is a task-level programming architecture to
develop robotic behaviors\[2\] [13]. SMACH is written in
Python and is designed to operate with ROS [14]. SMACH
implements hierarchical state machines in which a state can
be a state machine.

**State.** The central component in SMACH is the state defi-
nition. A state is modeled as an object with 3 parameters:
outcomes, input keys, and output keys. The outcomes are
string characters that represent a possible outcome transition
from the state used to make connection between states. The
input and output keys are the keys available to read and write
data passed through the states. The input and output keys are
passed through an object called userdata.

A state execution is expressed by evaluating its associated
function, which must return a string representing the outcome
taking place in the state. This outcome must be linked to another state when the state machine is built,
this represents a transition. Returning a non-defined outcome
leads to an error.

**State machine.** Each state represents an atomic behavior.
A group of states can be contained in a structure called
state machine. As such, a combination of states represents
a composed and therefore complex behavior. In SMACH, state
machines are hierarchical, which means that a state machine
may include other state machines in addition to individual
states.

**Userdata.** To pass data through a state transition, SMACH
uses the notion of userdata, which is essentially a thread-safe
dictionary, mapping keys to values. Userdata instances are
passed through the states when a transition is triggered.

**Internal and external inputs.** We call internal inputs to
data passed through the state as userdata. We call external
inputs data obtained externally from the source code, e.g.,
sensors. As an example, a state that searches for a person in a
room first identifies the person and then delivers the position
information to the following states. The position given to the
following states is an internal input. The image to calculate
where the person is and the actual position of a robot are
external inputs.

### IV. FUZZ TESTING IN ROBOTICS

Fuzz testing is a software engineering technique designed to
test a software application using automatic data generation
provided as inputs. This section illustrates fuzz testing for a
robotic behavior and details some properties of it.

**A. Illustrating example**

Consider the following simplified example inspired by our
experiment. Assume the following execute function of a
SMACH state that simply checks for a provided confirmation:

```python
def execute(userdata):
    text = userdata.confirmation_text
    if 'yes' in text:
        return 'yes'
    elif 'no' in text:
        return 'no'
    else:
        return 'aborted'
```

The function takes as argument an userdata ob-
ject, provided by another calling state (e.g., a voice-to-
text state). The object userdata contains a variable
confirmation_text, which contains a string character
representing the confirmation text. The state, and therefore
the execute function, has three possible outcomes: "yes",
"no" or "aborted".

**Presence of an error.** The execute function given above
has an error. The equality of string characters must be
performed using == and not with the operator in. We have
the expression 'yes' in 'yes' that evaluates to True,
which therefore complies with the intention of the execute
function. However, the expression 'yes' in 'eyes' also
evaluates to True since the word 'eyes' contains all the
letters of 'yes'. As such, saying "eyes" would be interpreted
as a positive confirmation message, which is obviously wrong
and constitutes an error.

**Fuzz testing to the rescue.** The error, which consists in using
== instead of in can be easily identified with a simple value
generation. Consider the function:

```python
def fuzz(init=2,fin=3):
    size = random.randrange(init,fin)
    alphabet = 'yesno'
    return ''.join(random.choice(alphabet) for i in range(size))
```

The function fuzz generates a word of 2 or 3 letters long
picked from the basket 'yesno'. Providing such generated
words as input of the state and monitoring the outcome of
the state can easily uncover the code error. For example, the
word 'noe' or 'eno' would be interpreted as a 'no'.
This example illustrates the principle of fuzz testing, which
is elaborated in the next subsection.

---

1. https://github.com/aliasrobotics/ros1_fuzzer
B. Our Fuzzer

We built a fuzzer that operates on SMACH state machines. Our fuzzer is based on a grammar [2] and generates random values as input keys of state machines. Thanks to the hierarchical nested state machines, our fuzzer is able to operate at the level of an individual state (fine-grained and white-box testing) or a whole state machine (coarse-grained and black-box testing). Relying on a grammar to generate random input is essential to ensure a coherent structure in the provided inputs.

White-box testing. Our fuzzer randomly generates userdata and provides it to a particular state, possibly part of a larger state machine. In this operation mode, the fuzzer is stressing the behavior of one single state, thus we qualify this mode as white box testing by searching for issues in the internal logic (as opposed to the machine interface as we will see later).

To be able to generate appropriate userdata, it is crucial to restrict the space of the values accepted by a state. For example, assume a state expects an integer value as input. If our fuzzer provides a string character to that state, a type error will be inevitably produced. Since Python is a dynamically-typed language (as JavaScript and contrary to Java and C++), it is not possible to determine whether a variable accepts a string or a number by solely looking at the source code definition. However, our fuzzer needs this crucial piece of information to generate random values of the appropriate type. Without such information, our fuzzer will identify trivial and non-relevant type errors instead of valuable bugs and errors in the robotic behavior logic.

To address this obstacle, we have designed a state-monitoring technique to determine the type of values expected by a state. The technique consists in monitoring values provided by a calling state to the called state. When values are provided to a state during a seed execution, the userdata structure is simply logged for offline analysis. We qualify as a seed the executions exercised on a robot that involves the state in which one wishes to obtain type information of the userdata. Such a seed execution is meant to represent a representative execution, which typically happens within a laboratory setting.

After one or more seed executions, types are inferred from the logs. Luckily, values commonly used in SMACH to pass through userdata are most of the times numbers or string characters. Complex data structure such as object instances are rarely employed, which greatly simplifies both the type inference and the value generation.

White-box testing is able to identify bugs and faults that are contained within one single state. As such, it is convenient to use white-box testing in critical and hub states in a possible large web of inter-connected states.

Black-box testing. Typically, a robot interacts with an environment in which noise in sensor reading and unexpected perturbations are unfortunately way too frequent. In the black-box testing mode, our fuzzer stresses the behavior of a whole state machine, instead of focusing on internal and individual states. As such, black-box testing operates at a coarse grain (by considering a whole state machine) while white-box testing operates at a fine grain (by considering individual states).

Concretely, black-box testing generates values representing external values that are provided by the robot. Random values are generated and provided to a SMACH state machine, simulating a tunneling of external values. Black-box testing is adapted to simulate changes on robot data with values that may be interpreted as inadequate or wrong.

V. Experiments

Our hypothesis is that fuzz testing can identify erroneous and complex robotic behaviors. To verify this, three questions are stated in Section I, covering the nature of the identified problems (RQ1), the difficulty of finding these issues (RQ2), and the severity of the bug of robot behavior (RQ3). This section describes the experimental design and the methodology we used to answer the research questions and verify our hypothesis.

A. Experimental context

The UChile Peppers team, from the University of Chile, uses a Pepper-based robot from SoftBank Robotics to support human-robot interaction in domestic environments.

As for most robots developed in a University, large parts of the software source code is written by under- and postgraduate students during the development of their thesis. As such, it is crucial to rigorously and extensively test the robotic behavior to remain competitive. Note that the authors of this paper are not involved in the development of the robot nor part of the UChile Peppers team.

We wish to maximize the feedback from the participants in our experiment. We restrict our evaluation to white-box testing only to have a better description of the encountered errors.

B. Methodology

We have designed a methodology to answer our three research questions, as presented in Figure 1. This subsection details each step of it.

![Fig. 1: Steps of methodology to test a behavior](https://uchile-robotics.github.io/bender-index.html)

S1: behavior selection. A robot, such as the one implemented by the UChile Peppers team is complex and offers a wide range of different specific behaviors. The first step of our methodology involves the identification of the set of specific behaviors to be tested. It is important to identify such specific

---

3 We do not consider the optional type information supported by Python as a reliable source of information due to its scarce use in robotic engineering.

4 https://uchile-robotics.github.io/bender-index.html
behaviors and clearly determine the expected outputs. As such, it is important that the specific behaviors are deterministic and may be re-executed at will.

S2: input description. Accepted input values must be adequately described and characterized in order to apply fuzz testing to a state or a state machine. This involves identifying (i) the names of the inputs, (ii) the type of the data used by the state code (using the state-monitoring technique), and (iii) whether it is external or internal to adequately tune the value generation by our fuzzer. The output of that step is a clear and unambiguous description of the input. The fuzzer is also correctly tuned to produce inputs properly structured by designing a grammar.

In case the fuzzer produces some values of a wrong type or inadequately defined (e.g., providing positive numbers while only negative numbers are expected), additional seed executions may be run or the grammar may be refined.

S3: run the fuzzer. The third step consists in executing the fuzzer on each state of the state machine involved in the selected robot behavior to be tested. To execute a state, input values must be generated using the grammar and the type information described above.

The more a state is tested (i.e., executed with generated input values), the probability to identify faults and anomalies increases. A state execution represents an incremental unit of exploring the space of plausible input values, thus increasing the likelihood to trigger an anomaly.

The number of executions of each state is specified by one of the hyperparameters associated to the fuzzing process. In our actual setting, we execute each state 100 times. As we will discuss later on, this arbitrary value produced satisfactory results, but this number can be increased in presence of solid computational resources (e.g., a cluster of CPU or multiple and identical instances of the robot).

The output of this step S3 is a set of logs of each state execution that contains (i) the input values, (ii) state outcomes, (iii) errors if any, (iv) state execution time, and (v) stack trace if an error occurred.

S4: result analysis. After executing the states, logs are carefully and semi-automatically reviewed to identify anomalies in the behavior execution. From our experience, anomalies are best identified by using two different approaches: (i) looking for errors and stack traces, and (ii) comparing the fuzzer outputs with executions that are known to be correct (as obtained in a laboratory and controlled setting).

S5: error selection. Many possible software anomalies may have been identified from the previous step. As such, the practitioner needs to filter false positives from relevant software errors. Furthermore, duplication of errors must be filtered out: different input values may lead to the same error. The associated stack traces are likely to be sufficient at identifying whether the error is the same or not.

S6: classification. When a situation is encountered, be it an error or an expected behavior, it is crucial to determine the cause of, as it is most likely caused by either incorrect inputs or an error in the state behavior.

S7: specific value. If an error is caused by incorrect inputs, the grammar used by the fuzzer has to be refined, thus making the process jump to Step S2. Consider the case where a state expects a string as input value, but only two different strings are accepted, 'yes' and 'no' as in our previous example. Providing any other strings may lead to an abnormal behavior. In this particular case, the grammar needs to restrict the string generation under these constraints.

As software defining a robotic behavior grows over time, it frequently happens that a developer has little knowledge about what she needs to operate with. This step allows you to test a state without knowing its inner workings. A practitioner can iterate by refining the grammar until relevant software anomalies are spotted. Note that in case of providing different types of input values that allow a seemingly successful execution, the behavior of the state may have a vulnerability or simply not use the provided input.

S8: report. At this stage, we, as external agents of the development team, are confident that the filtered errors designate situations that must be reported to the robotic development team.

For each identified error, we asked the following questions to a panel of experts:

- Q1 – Do you think this situation is a software problem?
- Q2 – Do you have a connection with the situation?
- Q3 – Do you think you could have found the situation on your own?
- Q4 – Have you seen this situation before?
- Q5 – How difficult do you think it is to find the situation? Answers ranges from 1 (easy) to 5 (hard)
- Q6 – Is it important for RoboCup competition? Answers range from 1 (not important) to 5 (very important)

This questionnaire collects opinions about our findings in an unbiased fashion. Each panel member was individually questioned and not in a group to not influence other members. Our questionnaire does not mention the word “error” to not bias the participant in taking our findings as an actual error. Instead, we use the generic term “situation”, which is more neutral on the kind of problems reflected by our findings.

Each participant received and evaluated three situations per errors. We indicated to the participants the exact location of an error in the source code, the input values our fuzzer generated, and the outcomes from the state execution. We also encourage the participant to reproduce the error.

We voluntarily kept a low number of evaluations to avoid fatigue from our participants, which would inevitably affect the quality of their opinion. The following section presents our results.

VI. RESULTS AND DISCUSSION

We applied our methodology on a robotic behavior provided by the UChile Peppers team. The robotic software we tested is composed of 124 different states, defining 6 different specific robotic behaviors. In total, our fuzzer operated on 24 different internal inputs using a dedicated grammar and 9 external inputs with different options.
The code contains a syntax error because a comma is missing between the string "I am going left." and "right.". Our fuzzer identified one syntax error, which was not detected during the development and the test made in laboratory because it was fixed on the robot itself.

Data handling error. By being dynamically typed, Python does not offer a safety net to prevent elementary incorrect data. Furthermore, it frequently happens that collections of heterogeneous elements are provided, e.g., if the tuple ("move_left", 15) is provided to a state, then that state needs to assume that the first element of that tuple is a string and the second is an integer. If not done, the code will likely suffer from a data handling error.

In a previous version of the code, the variable userdata.list_objects was containing the two coordinates of a position in a two-dimensional space. After a new version of the code was produced, the variable now provides a label indicating a physical position (e.g., ‘Door’) instead of coordinates. It will return a tuple of chars (e.g., ("D", "O")) and will produce an error on the following states.

We also qualify as a data handling error situations where a declared parameter is not accessible (e.g., by using the remapping ability of SMACH), or if some non-existent data are being accessed (e.g., a_list = userdata.non_existing_field). We found 3 errors where data was not well defined or the state tries to access non-existing data.

Data handling errors may be complex to find and debug because they usually require a solid knowledge about the state producing the data, the data itself, and the state consuming the data. Furthermore, the producer and the consumer of data can be distant. This may happen in the case that a state simply forwards the received data. In such a case, our fuzz is therefore valuable at identifying such an error.

Logic error. An incorrect conditional statement or loop control may lead to a logic error. During the execution, a logic error may be expressed by executing a wrong branch in a condition (e.g., the else branch is executed instead of the then branch). Consider the following code snippet:

```python
# text_confirmation='Yes'
if userdata.text_confirmation=='yes':
    return 'yes'
elif userdata.text_confirmation=='no':
    return 'no'
return 'aborted'
```

This code contains a logic error because the variable text_confirmation points to the capitalized string 'Yes', while the first condition is expressed with lowercase 'yes'. As a result, the branch return 'aborted' will be considered while obviously the first one should be considered.

From our experience with the software provided by the UChile Peppers team, logic errors are usually difficult to catch and identify. One reason for this difficulty is the fact that a logical error is usually not expressed by an application crash. Instead, in our experiment, a logical error is expressed by an unexpected behavior of the robot. In the example given above, orally saying "yes" to the robot was transcribed as 'Yes' instead of 'yes' by the voice-to-text module. Thus...
In our experiment, we found four kinds of errors: syntax errors, data handling errors, logical errors, and architecture configuration error. To perform a sophisticated task, a robot will typically involve various independently designed components. When initiating or resetting a robot behavior, a long sequence of component initializations must be performed. We experienced an unexpected behavior in the way the main state machine of the robot is initialized. By providing a particular list of robot features to activate, an extraordinary large amount of logs were produced by both SMACH and ROS. After the initialization, the robot did not seem to have any anomalies. After discussing with a core developer of the UChile Peppers team, it seems that some components associated to a particular robot features are initialized more than once in some non-obvious circumstance. As a result, significantly more logs were generated, however, no apparent dysfunctions were experienced. We experienced only one instance of this architecture configuration error.

**Answering RQ1.** In our experiment, we found four kinds of errors: syntax errors, data handling error, logical errors, and architecture configuration error. It is likely that other kinds of errors may be found, for example involving an incorrect use of an API, or an incorrect interaction with a sensor driver. However, in our experimental setting, we have not seen such a case.

### B. Realistic error

We aim to identify realistic errors and anomalies in software modeling robotic behaviors. We surveyed a panel of experts to assess the practical values of the software errors we identified. Collecting opinions and feedback about our findings is essential to assess how close our technique meets practitioners expectation. We formulated our second research questions as:

**RQ2:** Can fuzz testing detect representative and realistic problems in robotic behaviors?

Answers of the questions defined in Section V-B are listed in Table II. The situations we presented to the participants are largely identified as a software problem (Q1). The errors we identified were known to the participants, but they were not fixed (Q4). The reported errors are perceived as important or very important (Q6).

<table>
<thead>
<tr>
<th>Error</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>3/3</td>
<td>0/3</td>
<td>2/3</td>
<td>3/3</td>
<td>2.7</td>
<td>5</td>
</tr>
<tr>
<td>E2</td>
<td>5/5</td>
<td>2/5</td>
<td>5/5</td>
<td>3/5</td>
<td>3</td>
<td>4.8</td>
</tr>
<tr>
<td>E4</td>
<td>2/3</td>
<td>1/3</td>
<td>3/3</td>
<td>3/3</td>
<td>1.7</td>
<td>4</td>
</tr>
<tr>
<td>E5</td>
<td>3/3</td>
<td>3/3</td>
<td>3/3</td>
<td>3/3</td>
<td>2.3</td>
<td>4.7</td>
</tr>
<tr>
<td>E6</td>
<td>3/3</td>
<td>1/3</td>
<td>1/3</td>
<td>2/3</td>
<td>2.3</td>
<td>4.5</td>
</tr>
</tbody>
</table>

**TABLE II: Result of our experiment. Questions are listed in Section V-B. In X/Y, X = number of “yes” and Y = total number of answers. Average score is given for Q5 and Q6.**

leading to an unexpected behavior. Identifying logical errors involves a close comparison of the fuzzer output with a correct execution performed in a controlled setting. During our experiment, we found one case of a logical error.

**Architecture configuration error.** To perform a sophisticated behavior, a robot will typically involve various independently designed components. When initiating or resetting a robot behavior, a long sequence of component initializations must be performed. We experienced an unexpected behavior in the way the main state machine of the robot is initialized. By providing a particular list of robot features to activate, an extraordinary large amount of logs were produced by both SMACH and ROS. After the initialization, the robot did not seem to have any anomalies. After discussing with a core developer of the UChile Peppers team, it seems that some components associated to a particular robot features are initialized more than once in some non-obvious circumstance. As a result, significantly more logs were generated, however, no apparent dysfunctions were experienced. We experienced only one instance of this architecture configuration error.

### Answering RQ2.

We therefore answer to RQ2 by stating that our fuzz technique was able to identify representative and realistic problems for the robotic behavior defined by the UChile Peppers team.

**C. Hard-to-spot errors**

Intuitively, an error that is easy to spot is likely easy to fix. Oppositely, an error that is well hidden in the internal software is likely to be damaging in the long run since people may build on top of it. Assessing the difficulty for practitioners to identify the errors is therefore important. We formulated our third research question as follows:

**RQ3:** Can fuzz testing find hard-to-spot errors?

We found that although participants do not have a connection to the presented errors (Q2), however, they think they could have found the errors (Q3) without too much effort (Q5). Interestingly, these errors were not investigated or even reported before our experiment.

We found out that the expertise of a practitioner directly contributes to the ability of identifying errors. A practitioner with high expertise in Python and the employed frameworks will likely perceive syntax errors as easy to find, and our experiment confirmed it. An experienced practitioner may also be familiar with debugging and memory inspecting tools, which is apparently key to identifying and addressing data handling errors. However, the logic errors seem to be the most complicated kinds of errors to identify and address. Although we do not pretend to generalize our case, some of the participants indicated that the logic error we identified was hard to spot.

**Answering RQ3.** Overall, we answer to RQ3 by stating that our fuzzer identified errors that are perceived as mildly difficult to be found, therefore not considered as hard-to-spot despite being latent for possibly a long time and not being reported by the team.

**VII. CONCLUSION AND FUTURE WORK**

Our overall effort indicates that software artifacts developed by the robotic community could benefit from state-of-the-art techniques produced by the software engineering community. We have produced a fuzzer, available to the SMACH and ROS community. Our experiment indicates clear benefits for the robotic behavior we have analyzed. Currently, one significant threat to the validity of our work is that we only have conclusive results for one single robot. As future work, we plan to replicate our results with different robots and robotic behaviors.

**Acknowledgments**

The work presented in this paper was partially funded by Fondecyt Regular 1200067, Lam Research, and by the Innovation Fund Denmark project HealthDrone.
REFERENCES


