Can robot navigation bugs be found in simulation? An exploratory study

Thierry Sotiropoulos, Hélène Waeselynck, Jérémie Guiochet, Félix Ingrand

Can robot navigation bugs be found in simulation? An exploratory study

Thierry Sotiropoulos, Hélène Waeselynck, Jérémie Guiochet and Félix Ingrand
LAAS-CNRS, Université de Toulouse, CNRS, UPS, Toulouse, France
Email: firstname.lastname@laas.fr

***Abstract*—The ability to navigate in diverse and previously unknown environments is a critical service of autonomous robots. The validation of the navigation software typically involves test campaigns in the field, which are costly and potentially risky for the robot itself or its environment. An alternative approach is to perform simulation-based testing, by immersing the software in virtual worlds. A question is then whether the bugs revealed in real worlds can also be found in simulation. The paper reports on an exploratory study of bugs in an academic software for outdoor robots navigation. The detailed analysis of the triggers and effects of these bugs shows that most of them can be revealed in low-fidelity simulation. It also provides insights into interesting navigation scenarios to test as well as into how to address the test oracle problem.**

***Index Terms*—simulation-based testing; autonomous systems; safety; domain specific defects; exploratory study.**

I. Introduction

The ability to navigate in diverse and previously unknown environments is a critical service of autonomous robots. The underlying navigation software may be complex and must be thoroughly validated. To do so, the usual practice is field testing, i.e., experimentations with a physical robot in real world. But this approach is quite expensive and potentially risky in case of misbehavior (for the hardware, the environment or the humans). In contrast, simulation-based testing is cheaper and may explore a large number of navigation scenarios in virtual worlds without incurring any physical risk. Existing robotic simulators seem to have reached a sufficient level of maturity to be used for testing. For example, a recent survey [1] points to two open-source general purpose simulators, Gazebo [2] and MORSE [3], that provide adequate physical fidelity with respect to navigation and mission planning functions. However, in practice, such tools are mainly used for prototyping purposes. There is currently no principled test method that would exploit their facilities for intensive validation purposes.

The long-term objective of our research is to develop such a method. This is challenging, as the test selection and test oracle problems are specifically difficult in the case of autonomous navigation missions in unpredictable environments:

- Test selection — The input domain is a hypothetical space of worlds in which the robot is intended to navigate. No need to say, the domain is infinite, incompletely specified (what are the key characteristics to consider in world models?), and it is hard to determine which test selection criteria should be used.
- Test oracle — Given an arbitrary world and mission in this world, there is no ground truth about the decisions that the robot should take. For example, failure to reach a destination point may be due to some misbehavior or to the infeasibility of the mission, but the automated oracle has no means to decide between the two.

The relevance of simulation-based testing is questionable: could it be the case that most bugs only surface in real-world conditions? Thus, there is a need to study the reproducibility of bugs in simulation, their trigger conditions, as well as their observable effect. This paper reports on such an exploratory study applied to an academic software for outdoor robots navigation. The sample of bugs considered is reasonably sized (33 bugs) for an in-depth analysis of each of them. We made a detailed analysis of their triggers and effects, to the point of trying to experimentally reproduce some of them in MORSE simulations. The analysis shows that low-fidelity simulation could reveal most of the navigation bugs found for this software. It also provides feedback about input scenarios and observation data we missed in previous test experiments with this software [4]. More generally, the study confirms that the diversity of misbehavior patterns makes the oracle problem very challenging, and provides useful insights into how to approach it.

The structure of the paper is as follows. After a discussion of related work (Section II), we present the navigation software under study in Section III. The case study design is detailed in Section IV. Section V discusses results, derived insights and threats to validity. Section VI concludes.

II. Related Work

The testing of autonomous systems has started to attract interest from the testing community. Test selection strategies have been investigated, based on an abstract model of test situations, that describes the involved entities, their relationships and some interaction patterns. The authors of [5] use UML (Unified Modeling Language) to specify a metamodel of entities and a set of interaction scenarios. The approach is applied to a vacuum cleaner robot, using metaheuristic search techniques to generate abstract test data from the models. In [6], the structural model of entities is in UML and the dynamic part consists of Petri nets. In [7], the behavior of the load handling device of an industrial transport robot is modelled and tested using label transition system. The work of [8] defines several types of mid-air collision situations, and uses them to guide the evolutionary testing of a drone collision avoidance algorithm. The same authors extend their work to find challenging situations with a Genetic-Algorithm-based approach for a UAV collision avoidance system [9].

All these approaches have a very simplified view of the simulated environment and do not call for sophisticated simulation means. To the best of our knowledge, the work of [10] is the only one to consider complete virtual world environments, in the framework of 2D simulations. An interesting contribution of this work is to establish a connection with world content generation techniques used in the domain of video games [11], which we believe is a promising direction of research. Work along these lines has recently started at LAAS. We are developing a test framework based on MORSE and using the generation of virtual 3D worlds to challenge the navigation of robots. Preliminary results consider the ability to control the difficulty of a navigation mission (feasibility, time and detours to reach the destination point) by tuning some 3D world generation parameters [4].

None of these works is based on fault studies in autonomous systems, and indeed there are not many such studies. One of them is [12] where the authors designed the RoboX tour guide robot and studied the deployment of 11 of them during 5 months (13,313 hours of operational time) at a robotics exhibition. A significant amount of robot failures - 96% of the critical ones, leading to the robot stopping and requiring human intervention - were caused by software. The navigation system represented 17% of those critical software failures, vs. 83% for the payload software, which was still in a test phase at the beginning of the exhibition.

In [13], Steinbauer presents the results of a study conducted in the context of the roboCup competition [14], to acquire knowledge on the nature of faults that affects autonomous robots. This study involved the participants of the competition, who were asked to answer a questionnaire investigating which part of robot systems are affected by faults, what the root causes are, what the failures caused are and what their impact and frequency are. The study incorporated the responses for 17 robots. It showed that software faults are more frequent than hardware faults.

In contrast to the above two studies, this paper focuses on the navigation software only. It performs an in-depth analysis of a (necessarily much smaller) dataset. Examples of in-depth analysis of faults can be found for other types of software. In [15], the authors perform an exploratory study of 109 performance bugs collected from Apache, Chrome, GCC, Mozilla and MySQL. They report having spent more than one year to analyze these bugs. The focus of [16] is on variability bugs, in the sense of variability in highly configurable systems. The study involves a qualitative analysis and documentation of 42 bugs from the Linux Kernel repository, and required an effort of several person-months. A third example is [17] on environment-dependent bugs in MySQL Server software. The authors extracted 7 hard-to-reproduce bugs from the manual analysis of 568 bug reports, and performed experimental test campaigns to study the influence of memory occupation, disk usage and level of concurrency on the failure reproducibility.

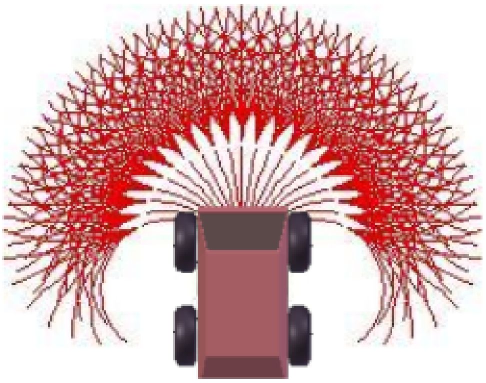


Fig. 1. P3D Arcs in front of robot (for `depth`=2 and `nbArcs`=20.

Our study is comparable in terms of effort and number of analyzed bugs. It involves a mix of manual analysis and test experiments, for a total effort of 6 person.months. Thirty-three bugs were extracted from 356 Git commits, analyzed in-depth and documented, and we additionally carried out test runs for a subset of them. For the bug documentation, we adapted the recommendations provided by [18], who propose a method to identify domain-specific defect classes based on reading change history. The authors provide an exemplary template form, which the analyst can fill in to document the findings. We modified the form to better integrate our concern for reproducibility in simulation (see Sect. IV-B).

III. Targeted Navigation Software

The targeted navigation software is part of the OpenRobots software repository[1], which includes software mostly developed at LAAS for the study and design of various kinds of robotic platforms: outdoor, indoor, UAVs. We use the Robotpkg infrastructure[2] to install and compile OpenRobots components and dependencies for the Mana platform dedicated to outdoor navigation. The Mana meta-package in OpenRobots[3] contains all the necessary software to handle localization, mapping, image acquisition and processing, hardware, path-planning, mathematical operations, motion and communication. We focus here on the navigation service encompassing localisation (pom-genom module), local path-planning (p3d-genom module) and 3D mapping (dtm-genom module).

The P3D local path-planning is an academic implementation of NASA's GESTALT algorithm [19] for Mars exploration rovers. Its principle is to choose the path that minimizes both a traversability-stability cost and the distance to the goal [20]. The algorithm considers a fixed number of arc-shaped paths

[1] https://www.openrobots.org/wiki

[2] http://robotpkg.openrobots.org/

[3] http://robotpkg.openrobots.org/robotpkg/meta-pkgs/mana/index.html

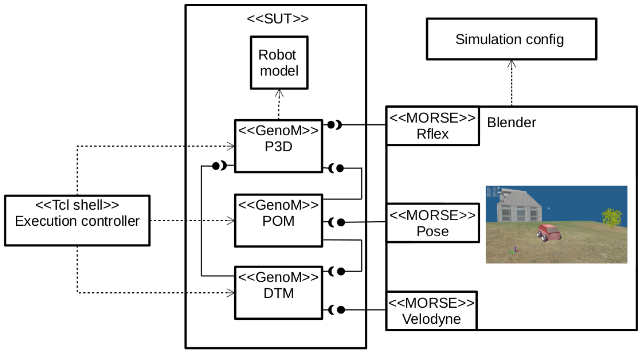


Fig. 2. Simplified diagram of the test architecture

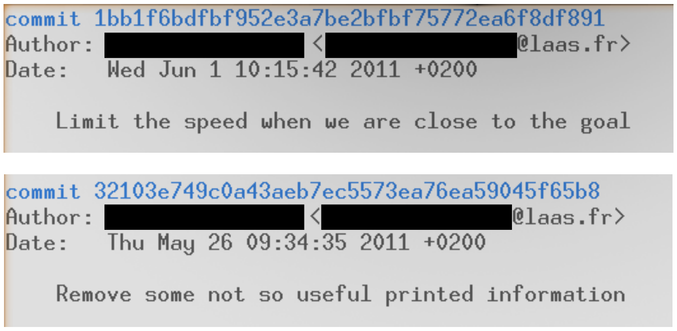
Module name	.c	.h	.gen	Total
pom-genom	1929	340	435	2704
p3d-genom	1984	677	592	3253
LibP3D	7526	1115	0	8641
dtm-genom	2256	232	434	2922
Total	13695	2364	1461	17520

Table I
Lines of code numbers for all modules of the targeted navigation software by type of considered files.

in front of the robot, and different points (called nodes) along each arc as one can see in Figure 1. The cost of putting the robot at a particular point increases with the slope at this location. Cost is infinite if the terrain is unknown (no perception). P3D computes the cost and reward associated with each arc and selects the best one.

The study covers the successive versions of this software between 2005 and 2015. In 2005, the software had already reached a relatively mature level (most of the initial developments were made before 2005, but the commit logs were not archived). During the 2005-2015 period, this navigation service was used for all outdoor field experiments conducted by LAAS researchers in various collaborative projects. Although it is an academic software, the functionalities and the complexity are representative of a real navigation service.

Figure 2 gives a schematic view of the existing test platform for this software in [4]. The navigation software is immersed into virtual 3D worlds by using simulated versions of the sensors and actuators: the simulated modules Rflex (wheel control and odometry), Velodyne (3D laser scanner sensor) and Pose (position sensor) are all handled by MORSE. The tested modules DTM[4] (3D mapping manager), POM[5] (position management) and P3D[6] (motion planning) are the exact replicas of the modules running on the real robot. They are developed with GenoM (Generator of Module) [21]. GenoM is a tool to specify, deploy and encapsulate the needed algorithms into standardized server components. In our experiments, GenoM modules communicate via the pocolibs middleware, using shared memory primitives called *posters*. GenoM uses a description file (.gen) and a set of algorithms written in C language (also called *codels*) to automatically generate a robotic module. The System Under Test (SUT) includes the .gen and .c files, plus the library for the execution support of modules, plus functional libraries dedicated to a specific module (e.g., LibP3D) or to a set of modules (e.g., the generic LibT3D library for 3D geometry), for a total of 35K lines of code. The various modules and libraries are developed and archived in separate git repositories. To keep the manual effort tractable, the collection and documentation of bugs focuses on a subset of these repositories, the ones that seemed to us the most relevant to navigation issues. This subset contains the three core modules and the functional library of P3D, as shown as Table I. It represents a bit more than 17K lines of code.



```
commit 1bb1f6bdfbf952e3a7be2bfbf75772ea6f8df891
Author: [redacted] <[redacted]@laas.fr>
Date:   Wed Jun 1 10:15:42 2011 +0200

    Limit the speed when we are close to the goal
```

```
commit 32103e749c0a43aeb7ec5573ea76ea59045f65b8
Author: [redacted] <[redacted]@laas.fr>
Date:   Thu May 26 09:34:35 2011 +0200

    Remove some not so useful printed information
```

Fig. 3. Comments example. Top: comment of a commit correcting a bug, bottom: comment of a commit not related to a bug.

The authors of the code were PhD students and postdocs who have now left the lab. No bug tracking tool was used. The only information available to us is the diff of the successive commits and the comment entered by the author of the commit. Figure 3 shows two examples of comments. As can be seen, they are typically quite succinct. In addition, code changes may have other motivations than bug fixes such as code cleaning or re-factoring, upgrades driven by new versions of libraries, etc. Manual analysis has to determine which commits correspond to a bug, and what the bug consists of.

IV. Design of the Exploratory Study

The aim of this exploratory study is to gain new insights into navigation bugs. It involves an in-depth qualitative analysis of a set of bugs extracted from the commit history of the above software.

A. Research Questions

The main objective of this study is to have a feedback on the relevance of testing in simulation, compared to testing in real worlds only. Our hypothesis is that many bugs do not require the reproduction of complex physical phenomena to be revealed. For example, let us consider the test platform described in Figure 2 as a baseline. This platform is based on an existing simulation configuration used in the prototyping

[4] http://trac.laas.fr/git/robots/dtm-genom.git
[5] http://trac.laas.fr/git/robots/pom-genom.git
[6] git://trac.laas.fr/git/robots/p3d-genom

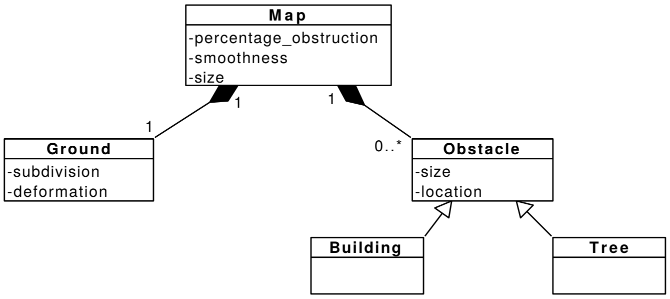


Fig. 4. Old input model.

of the Mana robot, with additional aspects related to the generation of virtual worlds and the observation of the robot behavior [4]. It corresponds to a low-fidelity simulation of the dynamics of the robot (for example the inertia is not simulated, neither are the reactions between wheels and ground during collision with rocks, or in slippery areas). MORSE may offer more realistic simulation of the physics, but at the price of longer computing times and greater effort to develop realistic actuator modules. So we want to evaluate whether a low-fidelity configuration like this is sufficient for efficient simulation-based testing, possibly with some improvements. It leads us to our first research question:

RQ1 : Can the triggers and effects of robot navigation bugs be reproduced in low-fidelity simulation?

Furthermore, we want, from the detailed analysis of triggers and effects, to draw lessons on the design and implementation of simulation-based testing. We are seeking new insights into the input domain definition, the raw output data to collect, and the oracle procedures to automate their analysis. Here again, we can take our previous work as a baseline.

The inputs to a robot navigation run were a randomly generated world instance and a navigation mission in this world. We used the simple world model shown in Figure 4. It was designed based on the 3D image of an area where the real robot was deployed for experimentation. A world is composed of the ground and obstructing objects like trees and buildings. The ground is rather smooth but with many local irregularities. World generation parameters allowed us to control the granularity ($subdivision$) and amplitude ($deformation$) of these irregularities, as well as the percentage of obstruction. A navigation mission was defined by a starting position and a target arrival position. Besides this simple model, we seek feedback on the important characteristics of worlds and missions to consider for test input selection.

RQ2 : From the analysis of triggers, which elements are to be considered in the input model of worlds and missions used for testing?

Some additional input configuration data are required to define test experiments. They concern the physical robot configuration (e.g., size, number and placement of sensors) and the parameters of the navigation algorithms (e.g., the number of arcs explored by P3D). Unfortunately, the developers did not archive the configuration files used for experiments in the field. Our previous work used an exemplary configuration that was given to us.

RQ3 : From the analysis of triggers, which elements are to be considered in the input data configuration (for the robot, for the navigation algorithms)?

Finally, how to approach the oracle problem is an open issue, for which we crucially need new insights. Our previous experiments did not put emphasis on revealing faults — rather, it focused on world generation issues and on tuning the difficulty level of navigation missions by means of a few parameters. We implemented a set of observation mechanisms to monitor the trajectory of the robot, record collision events, timeouts, error messages issued by the software, and also collect data relevant to the robot's subjective view (its perceived position, the last map it sees at the end of the run). However, the processing of the collected raw data was oriented toward assessing the difficulty of missions, not toward detecting incorrect behavior. Failure of a mission was not interpreted as failure of the navigation software, except for failure due to a collision. The analysis of the effect of real bugs will provide knowledge about misbehavior patterns, and may generate ideas on how to automatically detect them.

RQ4 : From the analysis of effects, which observation data and oracle procedures should be considered?

B. Approach to address the Research Questions

We consider the commits of P3D, LibP3D, DTM and POM from their respective Git repositories. Their total number is not too large (less than 400), making it possible to examine all of them. Otherwise, we might have used strategies to choose the commits that are more likely to exhibit a fault correction — e.g., the commits with large changes [18].

For each commit, a first analysis of the author's comment and of the code diff is performed. Its aim is to determine whether or not the change may correspond to a bug fix. If it may, the commit is retained for more in-depth analysis. The findings of the in-depth analysis of bugs are recorded in the form shown in Figure 5 (more details to be provided below). The form is inspired by the template proposed by [18] to derive domain-specific knowledge about bugs. We add a section on reproducibility in simulation in order to adapt the template to our research questions. If the overall analysis is in favor of reproducibility, we try to validate our understanding of the bug trigger and effect by dedicated test runs using the existing test platform. Once in-depth analysis of all identified bugs has been completed, the low-level findings of each bug are gathered to derive answers to the research questions. The answers are intended to provide insights into how to improve the baseline test platform, as well as more general insights into testing robot navigation software.

The form to be filled for each bug has 6 sections. **Location** contains factual data that identifies the bug. The Line(s) subsection reports on the bug fix according to the diff result. **Fault** contains free text to describe the bug. As noted by [18], the success of bottom-up knowledge building depends on how

- **Location**: *location of the patch*
  - Version:
  - Commit id:
  - Fault id:
  - File(s):
  - Line(s):
  - Function(s):
- **Fault**: *what was wrong in the code*
- **Failure**: *how the fault manifested itself*
- **Time to fix**: *when the fault was inserted and when the patch was inserted*
  - Fault inserted:
  - Fault fixed:
- **Reproducibility**: *how the fault can be triggered and the failure observed in simulation*
  - Overall Judgment : *no/yes/YES*
  - Constraint(s) on the simulation fidelity:
  - Constraint(s) on the world/mission:
  - Constraint(s) on the configuration data:
  - Raw data to observe:
  - Post-Processing to detect misbehavior:
- **Description**: *other findings and context*

Fig. 5. Form to be filled for each bug.

much information the low-level bug reports contain. Hence, the bug description should be sufficiently detailed to clearly state what was wrong in the code. For example, *"Variable* `uTurning` *is not re-initialized when a new destination point is passed as a parameter"* is preferable to the mere indication of an initialization fault. Similarly, **Failure** contains free text to describe the induced failure with a sufficient degree of details. **Time to fix** reports the dates for bug insertion and bug fix. While the fix is at the currently analyzed commit, the insertion date must be determined by a manual analysis, backwards in the history of commits. Note that the duration of the bug is actually not meaningful for our target academic software, because it underwent a sporadic development process. For example, we identified a crude bug preventing P3D to start, which remained nine months unnoticed. It could mean that there was no outdoor experiment planned during those nine months, or that the experiments used a P3D version not yet archived into the repository. While duration is irrelevant, we kept this section in case we need to analyze the sequential ordering of related bugs. **Reproducibility** is the core section of the form. It synthesizes the results of the analysis of the bug with respect to the research questions. Based on the understanding of the bug, we identify triggering conditions that must be fulfilled by the input cases (world/mission, input configuration data), and determine observation means to detect the failure. We also determine whether physical phenomena need to be reproduced. Finally, an overall judgment about reproducibility is issued based on the manual analysis and possibly on additional test experiments: no if we judge that the bug exposure would require a high degree of fidelity with respect to real-world navigation, yes if low-fidelity simulation appears sufficient, but the analysis results could not be validated by test runs, YES if we successfully reproduced the bug by runs in simulation.

For the test experiments intended to turn a yes bug into a YES, we considered two options:

1) Re-create the software version before the commit,
2) Inject the identified bug into the current version of the software.

Option 1 was not retained. A first problem is that the developers did not systematically archive all versions of the various modules and libraries, and did not archive the configuration files at all. Given a commit date for one module, if we take the versions of other modules corresponding to this date, we are not sure to recreate the software that was executed when the bug was revealed. Moreover, a second problem is that the current test scripts no longer work for the old versions of navigation. We considered that it would require a significant amount of effort to downgrade the scripts for the various versions to test.

Option 2 was then selected, so that the current test platform can be used. It also allows us to study the bugs one by one, by injecting a single bug at a time. However, injecting the bug may prove technically difficult. In some cases, the bug affects a function that no longer exists. Or the code changed so much that the injection is not merely a matter of re-introducing the lines changed by the fix: one has to undo changes in other parts of the software. It was not always possible to identify the changes required to inject the bug. In these cases, we discussed our understanding of the bug with a research engineer having contributed to the LAAS robot software architecture, and more specifically to GenoM modules. It allowed a form of expert validation of our findings for the bugs.

V. Empirical Results

We analyzed 356 commits:

- P3D: 69 commits
- LibP3D: 154 commits
- DTM: 50 commits
- POM: 83 commits

A. Overview of Bugs and their Reproducibility (RQ1)

We identified 33 bugs from the commits. Table II gives the breakdown of fault types across components. We use the ODC classification [22], but add a separate memory class to emphasize the high number of such bugs in the studied software. Programmers forgot to free the dynamically allocated memory, yielding memory leaks. The other important classes of bugs concern data assignments, incorrect implementation of algorithms and interface problems due to the handling of parameters in different measurement units or in different storage formats.

	P3D	LibP3D	DTM	POM	Total
Assignment	2	2	2	1	7
Checking	2	0	0	0	4
Algorithm	4	3	0	0	7
Timing	1	0	0	0	1
Memory management	0	9	1	0	10
Function	1	0	0	0	1
Interface	2	0	0	3	3

Table II
Orthogonal Defect Classification.

Name	no	yes	YES
P3D	1	7	4
LibP3D	0	9	5
DTM	0	1	2
POM	0	4	0
Total	1	21	11

Table III
Judgment about the reproducibility of bugs. No: not reproducible; yes: reproducible in theory; YES: reproduced.

Table III shows the overall judgment about the reproducibility of bugs in simulation. Only one bug was deemed impossible to reproduce in low-fidelity simulation (no column). It corresponds to the P3D fault put in the $function$ class of Table II. The function is the spot turn one. The Mana robot is a four wheel drive platform and can rotate on the spot by having the left and right wheels speed opposite. However, this rotation induces mechanical vibrations of the physical structure, affecting the sensors in such a way that the robot can loose its localization and have its 3D map of the environment corrupted. This functionality has been removed from P3D, and the spot turn is now managed differently and suspends the map building while rotating. Revealing this problem in simulation would require an accurate reproduction of the interactions between the wheels and the ground, and of the resulting vibrations.

None of the other bugs depends on the reproduction of complex physical interactions. The only physics we felt useful to add to the baseline test platform was inertia. One of the reproduced P3D bug from the YES column leads the robot to arrive too fast at the destination point, forcing it to brake abruptly. The baseline platform is sufficient to observe the abrupt braking, but has an unrealistic immediate stop of the robot. Inertia is introduced by modifying the avatar of the robot so that MORSE knows that robot movement is due to the rotation of the wheels. Then the Bullet Engine (the physics engine used by MORSE) is able to take care of inertia and friction during simulation, and the effect of braking is no longer instantaneous.

As can be seen on Table III, we confirmed reproducibility by test runs for a subset of 11 bugs. The remaining ones (Column yes) correspond to the following cases:

- 10 LibP3D and DTM bugs inducing memory leaks. These bugs were considered as out of the scope of the study, since they are not specific to navigation issues. They may be addressed by dedicated tests with the help of dynamic analysis tools such as Valgrind [23]. But in theory, the bugs could also be reproduced by regular navigation tests, in simulation or in the field, provided that the runs are sufficiently long to exhaust memory.
- 4 bugs in the P3D spot turn function (independent of the vibration problem we already mentioned). As the function no longer exists, we cannot inject the bugs in the current version of the software. However, the bugs are not difficult to understand, and we are quite confident that we correctly identified the triggers and effects.
- 3 P3D bugs affecting the logic of a `P3D_Blocked` error report. This error is issued when the robot ends up in a dead-end with obstacles around it, there is an arc selected by P3D and the first obstacle along the arc is too close to the robot. We spent a lot of time trying to reproduce the bug in simulation. But each time, P3D detected that it could not select any arc and activated the logic of another error report. We finally came to the conclusion that the prevalence of the other error report over the desired one could be due to the used P3D input configuration. We changed the value of one of the parameters, and could reproduce the expected triggers and effects. However, since the tested configuration is not representative of a realistic value for the parameter, we keep these 3 bugs in the yes category rather than the YES one.
- 3 POM bugs affecting the processing of sensor data. The processing can accommodate data in different formats and the bugs affect the conversion logic of one of them. The baseline platform with the Pose sensor does not allow the faulty code to be executed because it does not have the adequate output format. Unfortunately, no MORSE simulation component is available for sensors with this format, so the bug could not be reproduced. However, it is clear that the trigger is related to the robot configuration (the sensors placed on it), and not to complex physic interactions. The effect is less obvious. Most probably would the ill-converted data make the robot unable to determine its position, but an experimental confirmation would have been useful.
- 1 POM bug corresponding to an out-of-range indexing of array (a common fault that that could be caught by analysis tools). It turns out that the array is embedded into a union data structure with extra memory space: the illegal accesses end up there and nothing bad happens. Our judgment is that this fault cannot yield any failure, be it in real world or in simulation (we confirmed this in simulation). But should the embedding union structure change, an out-of-range error would systematically occur. Hence, this bug is as reproducible in simulation as in real world.

In conclusion, we are quite confident on the validity of our analysis about the reproducibility of bugs, including for the 21 yes bugs. The results support the relevance of simulation-based testing in virtual worlds. While some faults obviously require testing in real conditions (like the vibration problem we mentioned), many bugs can be found in low-fidelity simulation, with possibly some manageable improvements like the account for inertia.

B. Insights Into the Input Cases: Worlds, Missions and Configuration Data (RQ2, RQ3)

	Trigger conditions
Mission	Destination point behind the robot at start point
	Start point and destination points do not have the same Y value
	Long distance between start and destination point
	Several way-points per mission, or several missions in sequence
	Mission abortion and replacement
World	Dead end
	Hole
	Large map
Config.	P3D depth > 1
	Goal tolerance is small (tested with 0.1)
	Specific sensors
	Incorrect P3D parameters

Table IV
inputs and configurations used to trigger the faults

Seven bugs do not need specific trigger conditions as regards the tested world/mission and input configuration data. A basic test case suffices, e.g., asking for a straight line navigation mission in a flat world without obstruction. Table IV shows the list of conditions we identified for the other bugs.

The conditions on missions may concern the location of the destination point, the request to reach a sequence of destination points, or the abortion of a mission to replace it by another one. Our previous experiments in [4] completely missed these aspects. We simply had a start point at the bottom left of the map, a unique destination point at the top right, with work focusing on the random generation of terrain and obstacles between the two. For some of the bugs, the mission conditions have to be combined. For example, a P3D bug affects the management of a new destination point while turning on the spot. The revealing case deduced from our manual analysis sends a first destination point located behind the robot so that it starts to turn on the spot, and then replacement by a new destination point occurs while the robot is still turning. Another interesting example is a DTM bug that affects the calculation of the Y coordinate value of the robot in its perceived map. The trigger combines two conditions on the location of the destination point: it must not have the same absolute Y-value as the start point and it must be sufficiently far away to ensure the gradual degradation of the robot's map until the navigation's ability is affected. This is illustrated by the robot trajectory in Figure 6 that starts correctly but becomes absurd after a certain distance has been covered. Note that long distance missions require large world dimensions (see the large map condition in Table IV).

Other triggers require the robot to be exposed to specific world elements while accomplishing its mission: it must be trapped into a dead-end, or its trajectory must cross a hole. We already mentioned dead-ends in the discussion of the P3D bugs affecting the `P3D_Blocked` error report. For these bugs, the dead-end must be an area sufficiently narrow to force the robot to stop and issue the error report. Otherwise, the robot will keep moving inside the area in order to find the exit. Another P3D bug requires the dead-end area to surround the destination point: when the robot reaches the point, it reports an error instead of reporting success of the mission. Note that our previous work did not explicitly include dead ends in the world model. But the tuning of the percentage of obstruction produced virtual worlds with dead ends of various sizes, and we did trigger trap situations. The specific case with the dead end at the destination point could however not occur, because we kept a free area around the start and destination points, in order to avoid trivially infeasible missions.

Holes were missing in our world model. We only considered local depressions of the terrain. Still, deeper holes are interesting to add because they correspond to areas not perceived by the robot: for one of the LibP3D bugs, the robot could choose an arc ending at an unperceived node.

Conditions on the input configuration data were the most difficult to determine. We would really recommend that all configuration files be archived with the source code. In some cases, it was clear that the tested configuration had been incorrect: the fix introduced a check of the P3D parameters, or a comment about the parameter value was inserted. In other cases, we could determine reasonable variants of the baseline P3D configuration that were necessary to trigger the bug, like increasing the depth of the arc exploration or diminishing the tolerance value of the distance to the goal. Such triggers are reported for the sake of completeness, but we do not intend to elaborate on them to generically test the robustness of the software with respect to a set of configurations, correct or incorrect. Likewise, the hardware configuration (e.g., sensors) can be the trigger of bugs, but we do not intend to systematically explore configuration variants. However, these bugs show that configuration files are as important as the source code version. We would recommend that a change in configuration yields a regression test, in the same way as a change in the code would do.

C. Insights Into Observation Data and Oracle Procedures (RQ4)

The bugs induce diverse failures, as shown in Table V. The effects range from obvious failures (crash, P3D does not start), to performance issues (suboptimal trajectory due to initial bad alignment to the destination point, mission failure while the mission is feasible) and even failures that would be dangerous if the test occurred in real world (the robot falls into a hole, the speed commands are not refreshed and retain their value forever).

► Infinite spot turn

► Failure to align to the target destination point

► Jerks in angular speed commands

► Robot does not immediately stop after detecting an error

► The robot arrives successfully at destination

but considers itself as blocked

► The robot brakes too late when arriving at destination

► The speed commands are not refreshed and retain

their value forever

► P3D does not start

► Execution crash

► Unexpected mission failure

► The robot goes round and round in circles until time-out

► The robot falls into a hole

► The robot has an absurd trajectory

Table V
List of encountered failures.

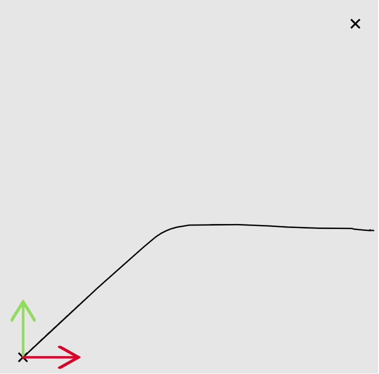


Fig. 6. Absurd trajectory. Starting point and goal point are represented by a cross. X and Y axis are represented by a red and a green arrow.

If we look at the logs collected by our baseline test platform, we missed few data to be able to detect these failure. The only missing observations are the commands sent to the wheels. But the most difficult problem is not the logging of raw data: it is their interpretation to automatically detect the misbehavior. Indeed, the diversity of failure patterns is very challenging for the test oracle. Also, performance-related issues are hard to distinguish from legitimate behavior: the misbehavior is only with respect to some (so far, implicit) reference. For example, mission failure is a misbehavior only if the mission is expected to succeed. The trajectory of Figure 6 is absurd only because we know that the robot should move straight away. Even going round and round in circles may not be abnormal or, say, is a known and accepted behavior of the used path planning algorithm. This is due to the fact that the algorithm is based on local search. As a result, when the robot is trapped into a dead end, it is unable to escape if the only exit would increase its distance to the destination point.

To address the diversity of misbehavior patterns, we advise trying to devise as many error detectors as possible, each focusing on a simple property. From our analysis of bugs, the identification of such simple properties could consider the following elements:

- *Requirements attached to mission phases*. For example, we might not be able to detect a suboptimal trajectory, but we might be able to detect initial bad alignment to the destination, should alignment have been identified as a requirement for the initial phase of the mission. Similarly, the arrival at the destination point may call for specific requirements, like sending a success report, not going beyond the point, etc.
- *Thresholds related to robot movement*, like the maximal variation of speed commands (would detect jerks), or the maximal angle covered by spot turns (should not exceed 180°, to detect suboptimal turn direction or infinite turns).
- *Catastrophic events*, like falling into a hole or bumping into an obstacle.
- *Requirements attached to error reports*, like the requirement that the robot should stop immediately when reporting any error.
- *Perception requirements*, like always having a correctly perceived position with some tolerance. Detecting bad map perception is more difficult. For the bug yielding the absurd trajectory of Figure 6, a detector could have been the high percentage of unknown areas in the perceived map.

The set of detectors might be enriched as more experience is gained on the target system or on similar ones.

The detection of performance-related issues, that would not be caught by any of the error detectors, is an open problem. A partial solution is to start with test cases for which a reference behavior can be pre-determined. For example, the robot is assigned a trivial mission that it must succeed, and its trajectory should not depart too much from a pre-calculated one. But this will not be sufficient for bugs with complex triggers. Our future work will explore whether our previous experiments on difficulty levels can offer a partial solution, in the framework of regression testing. For example, the introduction of a performance bug can be suspected if navigation missions that were easy for the previous software version now become challenging or very difficult for the new version (the percentage of mission failure increases, the missions take longer time and involve more detours).

D. Threats to Validity

Our results are based on the analysis of a specific academic navigation software, which can be a threat to external validity. However, we argue that this software is representative of many navigation services, in terms of both the underlying algorithms and code complexity. During the period covered by the study, it was actually used in several real applications, some of them deployed out of the lab.

The development process may not be representative of an industrial one. An academic development is often not continuous, and furthermore we cannot assume a systematic archiving of versions. To account for this, we avoided to interpret data that might be not significant. For instance, the "time to fix the bug" information does not really represent a continuous use of the targeted software. It may reflect a period of non-activity or simply a gap in the archiving process. Hence, we excluded the analysis of this data.

Like in all similar studies based on bug fixes, our analysis considers bugs that have been found. It may induce an over-representation of the easy-to-reveal bugs compared to the more difficult and still unknown ones. Moreover, our dataset is composed of bugs that have been found with non-formalized test procedures, with field tests that are usually constrained in terms of testing inputs (few real scenarios are tested). We consider that the over-representation of easy bugs would be more a problem for deriving quantitative findings than for the kind of qualitative insights we were looking for. Besides bugs that can be revealed by "easy" cases (i.e. flat world, no obstacle, straight movement), we also found interesting examples of triggers and could identify diverse failure patterns. As regards the issue of reproducibility, the results are sufficient to convince that simulation-based testing is an approach worth considering. Even if our data set missed some hard-to-reveal bugs that would not be reproducible in simulation, it would still be the case that many bugs currently found in field experiments could also be caught by simulation-based testing.

Regarding the internal validity of our experimental study, we may have missed some bugs due to our manual analysis, and thus missed some lessons learnt. It is actually an issue in every bug study based on manual analysis, but this does not reduce the relevance of the lessons learnt from the bugs we identified. Another debatable issue is the way we reproduced the bugs: we opted for fault injection in the current software version rather than replay of a complete deprecated version, which could induce differences in the triggers and effects. Nevertheless, we performed an in-depth manual analysis of each bug in the context of its version. For the bugs that we successfully reproduced with the current version, the triggers and effects were the ones expected from analysis. For the bugs we unsuccessfully tried reproduce, we validated our manual findings with a software robotic engineer involved in the development of the considered software.

VI. Conclusion

In this paper, we explored the reproducibility in simulation of bugs that affect the navigation software of an outdoor robot. The bugs were collected using manual analysis of the commit history of the software.

The in-depth analysis of bugs provided useful insights into domain-specific triggers and effects, as intended, but also suggests a few common sense recommendations about the validation process in general. One of them is to consider the configuration files as an integral part of the software. Indeed, several bug triggers were related to the hardware (e.g., sensors put on the robot) or software (e.g., parameters used to tune the path-planning algorithm) configuration. It is then very important to archive the configuration, and to validate configuration changes in the same way as one must validate code changes. This was not done in the case of the academic software we studied. Another general recommendation is to have separate consideration for common and not domain-specific bugs like memory leaks and out-of-range array indexing. They represented one third of the bugs we analyzed. Dedicated analysis with the help of tools like Valgrind can catch those bug before one concentrates on the validation of the core navigation functionalities, which is the primary objective of our work.

Overall, the study supports our hypothesis that many navigation bugs do not require the reproduction of complex physical phenomena to be revealed. A single bug is related to tough aspects of the physics. It concerns an incompatibility of the spot turn and map building functions, which surfaces via mechanical vibrations of the platform. The triggers and effects of the other analyzed bugs are well-amenable to reproduction in low-fidelity simulation. As far as physics is concerned, the only suggested improvement to our platform was to account for inertia, allowing us to observe that the robot may brake too late when arriving at destination. Interestingly, a few bugs induced a dangerous behavior (the robot falls in a hole, does no longer refresh the speed commands), which further supports the relevance of simulation-based testing for safety concerns.

The triggers we identified suggested improvements to the input model we considered in previous experiments. We realized we did not pay sufficient attention to the mission definition in terms of the relative position of start and destination points, the number of waypoints to reach, or to mission management aspects like aborting and replacing missions. We also missed elements on the 3D terrain like holes that the robot must consider as unknown and potentially dangerous area to avoid. Some triggers were found to combine several conditions on missions, world elements, or the current robot status (e.g., mission replacement while the robot is turning on the spot to align to a point behind it, narrow dead-end blocking the robot at the destination point). Such tricky cases support the recommendation made by others to consider *situation*-based testing [24]. Our future work will have a closer look at how to guide and observe situation coverage, thinking in terms of interactions of the robots with (a combination of) world/mission model elements.

The observed effects of bugs were diverse and not easy to distinguish from normal behavior in the absence of a precise reference. Such a precise reference (e.g., including a reference trajectory) can be pre-determined only for the simplest cases. To alleviate the problem, we propose to insert numerous fine-grained error detectors, to be used generically for all tested navigation missions. From the bug we analyzed, the checked properties should at least cover the following dimensions: requirements attached to mission phases, threshold-based invariants related to robot movement, absence of catastrophic events, requirements attached to error reports, and good perception requirements. Our future work will also consider the detection of performance bugs by means of regression tests, using a sample of worlds and missions of various difficulty levels [4].

Work along the above-mentioned lines — input modeling, situation coverage, fine-grained detectors and performance-related regression tests — will benefit from access to an industrial case study: an agricultural robot to weed vegetable crops. The navigation software of this robot will provide a case study to develop and transfer the insights gained from the analysis of bugs presented in this paper.

Acknowledgement

This work was supported in part by the EU CPSE Labs project funded by the H2020 program under grant agreement No 644400. We wish to acknowledge the contribution of Anthony Mallet and Simon Lacroix from LAAS-CNRS, whose participation helped us get a better understanding of the code and faults.

References

[1] D. Cook, A. Vardy, and R. Lewis, "A survey of auv and robot simulators for multi-vehicle operations," in *Proceedings of the IEEE/OES Conference on Autonomous Underwater Vehicles AUV*, pp. 1–8, 2014.

[2] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *International Conference on Intelligent Robots and Systems (IROS)*, vol. 3, pp. 2149–2154, 2004.

[3] G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan, "Modular open robots simulation engine: Morse," in *IEEE International Conference on Robotics and Automation ICRA*, pp. 46–51, 2011.

[4] T. Sotiropoulos, J. Guiochet, F. Ingrand, and H. Weaselynck, "Virtual worlds for testing robot navigation: a study on the difficulty level," in *IEEE 12th European on Dependable Computing Conference EDCC,*, pp. 153–160, 2016.

[5] Z. Micskei, Z. Szatmári, J. Oláh, and I. Majzik, "A concept for testing robustness and safety of the context-aware behaviour of autonomous systems," in *Agent and Multi-Agent Systems. Technologies and Applications*, pp. 504–513, Springer, 2012.

[6] A. Andrews, M. Abdelgawad, and A. Gario, "World model for testing autonomous systems using petri nets," in *IEEE 17th International Symposium on High Assurance Systems Engineering HASE*, pp. 65–69, 2016.

[7] C. Mühlbacher, S. Gspandl, M. Reip, and G. Steinbauer, "Improving dependability of industrial transport robots using model-based techniques," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3133–3140, 2016.

[8] X. Zou, R. Alexander, and J. McDermid, "Safety validation of sense and avoid algorithms using simulation and evolutionary search," in *Computer Safety, Reliability, and Security*, pp. 33–48, Springer, 2014.

[9] X. Zou, R. Alexander, and J. McDermid, "On the validation of a uav collision avoidance system developed by model-based optimization: Challenges and a tentative partial solution," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshop*, pp. 192–199, 2016.

[10] J. Arnold and R. Alexander, "Testing autonomous robot control software using procedural content generation," in *Computer Safety, Reliability, and Security*, pp. 33–44, Springer, 2013.

[11] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[12] N. Tomatis, G. Terrien, R. Piguet, D. Burnier, S. Bouabdallah, K. O. Arras, and R. Siegwart, "Designing a secure and robust mobile interacting robot for the long term," in *Proceedings of IEEE International Conference on Robotics and Automation ICRA*, vol. 3, pp. 4246–4251, 2003.

[13] G. Steinbauer, "A survey about faults of robots used in robocup," in *RoboCup Robot Soccer World Cup XVI*, pp. 344–355, Springer, 2013.

[14] U. Visser and H.-D. Burkhard, "Robocup: 10 years of achievements and future challenges," *AI magazine*, vol. 28, no. 2, p. 115, 2007.

[15] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 77–88, 2012.

[16] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: a qualitative analysis," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 421–432, 2014.

[17] D. G. Cavezza, R. Pietrantuono, J. Alonso, S. Russo, and K. S. Trivedi, "Reproducibility of environment-dependent software failures: An experience report," in *IEEE 25th International Symposium on Software Reliability Engineering ISSRE*, pp. 267–276, 2014.

[18] T. Nakamura, L. Hochstein, and V. R. Basili, "Identifying domain-specific defect classes using inspections and change history," in *Proceedings of the ACM/IEEE international symposium on Empirical software engineering*, pp. 346–355, 2006.

[19] J. J. Biesiadecki and M. W. Maimone, "The mars exploration rover surface mobility flight software driving ambition," in *IEEE Aerospace Conference*, pp. 15–pp, 2006.

[20] D. Bonnafous, S. Lacroix, and T. Siméon, "Motion generation for a rover on rough terrains," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 2, pp. 784–789, 2001.

[21] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand, "GenoM3: Building middleware-independent robotic components," in *IEEE International Conference on Robotics and Automation*, pp. 4627–4632, 2010.

[22] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on software Engineering*, vol. 18, no. 11, pp. 943–956, 1992.

[23] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley, "Tracking bad apples: reporting the origin of null and undefined value errors," in *ACM SIGPLAN Notices*, pp. 405–422, 2007.

[24] R. Alexander, H. Hawkins, and D. Rae, "Situation coverage–a coverage criterion for testing autonomous robots," tech. rep., University of York, Department of computer science, 2015.