Testing a non-deterministic robot in simulation - How many repeated runs ?

Clément Robert, Jérémie Guiochet, Hélène Waeselynck

Testing a non-deterministic robot in simulation - How many repeated runs ?

Clément Robert, Jérémie Guiochet and Hélène Waeselynck
LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France
Email: firstname.lastname@laas.fr

***Abstract*—Testing autonomous robots requires test campaigns in the field that could be expensive, risky for the system and its environment, or even impossible to set up. To cope with these limits, an expanding approach is to perform tests in virtual environments using simulators. Due to the non-determinism of the robot control and simulation execution, a test might fail or pass with exactly the same inputs (same world and mission). An important question is thus: how many runs are required to activate a fault? We particularly focus in this paper on the efficiency of repeated runs and world diversity in the context of functional testing. We address this research question with an industrial case study – an agricultural weeding robot developed by Naïo Technologies. We conclude that, in this case study, test input diversity is more efficient than repeated runs in order to activate faults. More importantly, we propose an experimental approach to assess the impact of the non-determinism, which may be reused in other case studies.**

***Index Terms*—testing, autonomous robot, non-determinism, simulation, Software in the Loop (SiL) testing.**

I. Introduction

The validation of an autonomous robot involves mission-level tests, which may be done partly in the real world (field testing) and partly in virtual worlds (simulation-based testing). In both cases, a concern is that the robot may not behave in the same way for repeated runs with similar environmental conditions. It is true for field tests, not only because reproducing identical inputs and environmental conditions is actually impossible, but also because the system itself is non-deterministic. The embedded software typically has concurrent threads and processes executed at the same time, yielding execution non-determinism. This execution non-determinism is still observed in simulation where identical test environments can be fully reproduced. It may have a strong impact on the Pass/Fail outcome of a test case. Indeed, a single pass observation does not mean that the test case would pass for all possible executions. Now, suppose we adopt a repeated execution scheme where each test case is run n times. On the one hand, the repetition allows us to possibly catch non-deterministic fails of the test case, that did not occur at the first time. But on the other hand, we may end up spending the test budget on the repetition of tests that pass n times, instead of exploring new tests that would fail. This raises the question of how many runs of a test case are to be performed to efficiently uncover faults.

We address this question in the context of simulation-based testing, where at least identical environment conditions can be repeated for several runs. We consider an industrial case study, an agricultural robot for autonomous weeding. The system is tested in a Software-in-the loop (SIL) configuration, that is, the real software is immersed in virtual worlds and challenged with missions. The test platform includes several components that we developed around the robot simulator: a world generation facility, a test harness to automatically run many simulations (400 in this case study) and a test oracle (i.e., the mechanism determining the Pass/Fail verdict) that checks a set of expected properties. We use this platform to empirically study the variability of the test verdicts over repeated runs, and to experiment with various repeated execution schemes.

The structure of the paper is as follows. Section II discusses related work. Section III introduces the case study: the Oz robot. The experimental design is detailed in Section IV. Section V presents the results and Section VI concludes.

II. Related work

A deterministic program produces identical output sequences when fed with identical input sequences. This is the case for purely sequential program. In autonomous robotics, the embedded software is generally composed of multiple threads to interact with the environment, one for each subsystem. In such systems, inter-process communication is generally done at best effort, resulting in an asynchronous and non-deterministic behavior. For example, ROS [17], a robotic middleware widely use in the research community, provides basic services for the development of robot software. It relies on a concurrent programming paradigm where nodes encapsulate subsystems. Moreover, the decisional functions of autonomous systems commonly use algorithms involving randomness (e.g. heuristics for mission or trajectory planning), which also induces non-determinism [8].

The problem of non-determinism is particularly relevant in the context of testing cyber physical system in simulation. The use of simulation for testing is increasing, mainly pushed by automotive, with high fidelity simulators (e.g., see [26], [16] or [15]). They provide specific virtual road environments and have proved useful to thoroughly test advanced driver assistance systems in simulation (see e.g., [1], [10]). On the other hand, robots are as various as their applications and each simulator must be built specifically. They are developed based on simulation platforms that are more generic (Gazebo [9], MORSE [6]) or also based on game engines like Unreal [25] and Unity [24]. Such simulators can then be used for testing the robots, like a rough terrain robot tested in [21], a rescue

robot in [2] or a drone in [12]. Generally speaking, a simulator may be a complex system in its own right, introducing extra sources of non-determinism into the simulation: concurrency of the simulator's components, use of remote services, time (few things are more non-deterministic than a call to the system clock), resource leaks, etc.

With the non-determinism effect – from the tested system and possibly also the simulator – come difficulties for testing. Automatic testing relies on a component that can determine if a test has passed or failed: the oracle. Testing autonomous systems is already hard as there is no ground truth about the decisions to take during a mission. For example, as noted by Tain and al. [22], specifying what an autonomous driving system should do would essentially involve recreating the logic of a human driver. Adding non-determinism makes the task even harder as the same test inputs can yield different oracle verdicts: test failures are seemingly random. The non-determinism results in two main issues for testing: the confidence in the oracle verdict to draw conclusions, and the ability to reproduce the fail for diagnosis purposes (the same sequence of events that previously led to a failure may not fail again). In this paper, we focus on the first issue and study the variability of the verdict for different runs of the same test case.

To remove or reduce non-determinism inside a system, an idea is to come back to a temporal deterministic approach, using for instance synchronous programming languages like Esterel [4], Signal [3] or Lustre [7]. Examples of application of synchronous languages to robotic architectures can be found in [5] and [11]. Esterel (version 3) translates concurrent program into equivalent sequential automata. It relies on the synchrony hypothesis that states that every process is instantaneous (the system and the environment states do not change during a process execution). However, some systems cannot be split into atomic processes, thus cannot be fully converted to a deterministic version. Autonomous systems typically involve asynchronous communication between subprocesses on several processors. Such is the case for the weeding robot studied in this paper.

From the simulation side, Reymann and al. [18] propose a framework to manage non-determinism induced by time in distributed simulations. The approach consists in adding a software layer that synchronizes the system by handling the interprocess communication by batch. To ensure that there is no deadlock, the method assumes that there is an upper bound time value for subprocesses to produce a message. This method is well-suited to make multiple simulators run together deterministically, but does not address the non-determinism inside each simulator or system.

Previous solutions focus on removing or reducing the non-determinism. Another approach for testing is to take the non-determinism explicitly into account. A classical way is to repeat the same test several times in order to obtain various execution patterns. The authors of [21] used repeated runs to evaluate the difficulty level of generated navigation missions for Mana, an outdoor robot developed at LAAS-CNRS. They also studied the impact of non-determinism by comparing the trajectories of the robot for a given mission in the same 3D environment. The differences in the trajectories were significant among the runs, in the order of meters or even tens of meters. The authors of [14] used repeated runs for the evolutionary testing of an autonomous cleaner agent. The evolutionary approach requires a fitness function that quantifies the adequacy of the generated test and guides the generation of new ones. The evaluation of the fitness function required multiple executions of the test cases to take the non-determinism into account. A statistical analysis showed that, for that specific case study, 5 executions were enough to represent the overall result and for the fitness function to converge.

Fig. 1. Oz in operation (source: Naïo Technologies)

Repeating the same test case may be necessary but is time consuming. For a given test budget, it forces us to reduce the number of test cases executed and therefore it reduces the exploration of diverse test cases. In this paper we provide a statistical study of repetition versus exploration to uncover faults.

III. Case study: The Oz robot

Oz is an autonomous robot developed by Naïo Technologies [13] and aims at weeding crop fields (Figure 1). The base frame is mounted over 2 by 2 powered wheels for tank like movement. It is considered harmless for humans as it is small ($75\ cm \times 45\ cm \times 55\ cm$) and it can only reach a maximum speed of $0.4\ m.s^{-1}$. On the other hand, it can still be dangerous: a faulty navigation function could send the robot outside of the crop field (e.g., a road). Moreover, it would be highly undesirable that the robot causes damages to the crop plants. To complete its mission, the robot must navigate inside the field while pulling a specific weeding tool. A field is usually composed of multiple rows of vegetables. Thus, Oz has to make U-turns when it exits a row and needs to enter the next one. When the row interspace is considered too large by the robot, it can decide that it needs a second pass to complete the weeding. Figure 2 shows an example of a mission where the robot performs one pass in the first interrow and two in the second one.

A mission can be divided in 4 main phases that loops until the mission is completed (See the letters on Figure 2): The

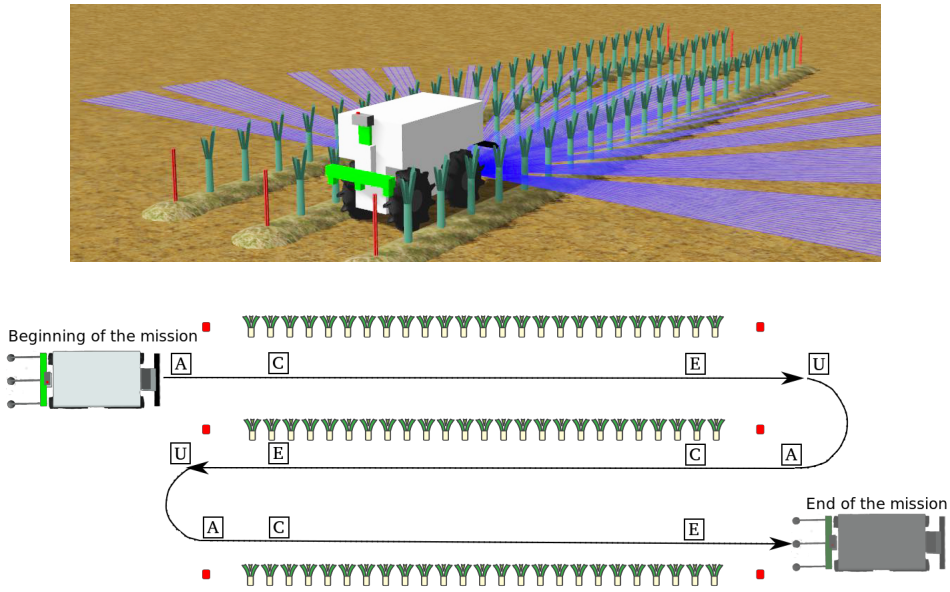


Fig. 2. Oz in simulation. Top: virtual crop field using Gazebo. Bottom: expected mission.

Approach (entering an interrow), the **C**ruise (while between two rows), the **E**xit (exiting an interrow) and the **U**-turn. To help the robot performing its mission, landmarks (red stakes) must be placed at the extremities of each row. Oz is equipped with several sensors that are used according to the mission phase: a laser sensor (LiDAR 2D) and two cameras at the front. It also has contact sensors around it to trigger an emergency stop in the event of collision. The software version presented in this paper is an R&D version that only uses the laser data to perform the line tracking during the cruise phase. Cameras are used to detect the red stakes, evaluate the robot position in the field and consequently trigger a change in the mission phase. The cameras also allow the use of stereo visual odometry techniques to detect possible skidding of the robot during U-turns.

IV. Experimental design

Naïo developed a SiL simulation platform for testing the navigation of Oz. Naïo engineers and us have independently tested an R&D version of the Oz controller (respectively by field tests and by virtual tests) to compare the results and evaluate the potential of simulation-based testing. In another paper [19], we showed that many of the software issues found by field tests are also found in simulation. The virtual tests even found a new issue that was missed in the field. This paper uses the same test platform and the same virtual tests as in [19], but for a different purpose: here, our aim is to study the impact of the non-determinism on the test results. We first present the research questions addressed by our study and then the test platform used by our experiments.

A. Research questions

The first question concerns the variability of test verdicts in repeated runs of the same test case:

RQ1 - Does non-determinism strongly impact the verdict assigned to a given test case?

Note that it could be the case that the detailed behavior is not exactly the same, but the overall Pass/Fail verdict is consistent from one run to the next. To study this question, we consider 5 repeated runs of a sample of 80 test cases, and measure the proportion of test cases that inconsistently fail. Five runs correspond to the number of repetitions considered in the original experiment [19], mainly for pragmatic reasons: each run takes minutes, and it seemed reasonable to keep the total test time (80x5 runs) around 24H. For our study of non-determinism, we chose not to increase the number of runs. We focus on large variation effects that make the inconsistent verdicts likely, rather than on small effects that could surface only after an unrealistically high number of repetitions.

The second question concerns the trade-off between the exploration of new test cases and the repetition of previous ones.

RQ2 - Repetition vs exploration: For this case study, what would be the most efficient repetition strategy in order to reveal the faults?

Overall, the original test campaign with 5 repeated runs proved effective at revealing faults, but it is not clear what the added value of the repetitions was. From the complete set of runs, we extract different subsets to mimic strategies with a lower number of repetitions per test case. Then, we assess their efficiency by considering both the first detection and the relative frequency distribution of the various failure types. The first detection allows us to determine how repetition speeds

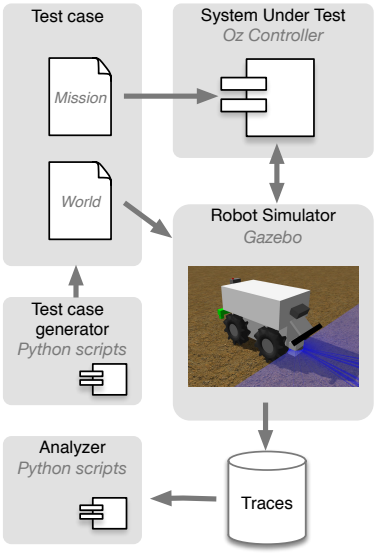


Fig. 3. Simulation architecture.

up or slows down the detection of new failure types. The frequency distribution reflects the relative importance of each failure type, and we study how it is impacted when varying the number of different test cases and the number of runs per test cases.

B. Experimental platform

The test platform used for our experiments is described in Figure 3. It is composed of four components: The System Under Test (SUT), the robot simulator, the test case generator, and the analyzer.

1) **The system under test.** The SUT, i.e., the Oz Controller developed by Naïo Technologies, is written in C and C++ for total of about 151 KLOC. It consists of sensors processing modules, the navigation module and several mission-specific modules. The navigation module needs some information about the crop field and the robot initial position (e.g., the number of rows, the direction of the first U-turn) that are contained in the mission file (a `json` file). The Oz controller receives simulated sensors data and produces speed commands sent to the simulator.
2) **The robot simulator.** The simulator is based on Gazebo, a simulator widely used in robotics research [9]. It has been developed by Naïo.
   To instantiate a virtual world, the simulator requires two main files: a heightmap which is a grayscale `jpg` file that encodes the terrain elevation and a world file (`sdf`) that describes the initial position of every object in the scene. Figure 2 illustrates a simulated environment. Note that the virtual crop field is quite small compared to the real-world ones. The virtual example only contains 3 crop rows of 15 meters whereas a real world field can contain tens of rows that can be as long as a hundred meters. The difference is due to simulation performance and time constraints. Although the simulation seems simple, it is very demanding regarding resources. Our available computing power resources consists of: a PC with 2 Quad core Intel Xeon E5-2623 v3s CPU at 3.5GHz, and 64GB of RAM. To ensure tractability of the simulation, we choose to restrict the size of the generated crop field. The second issue is the testing time: the robot is pretty slow and there is no way to accelerate the simulation. Hence, each individual run can take minutes. As an example, a run on the virtual field in Figure 2 takes about five minutes. Even if it was possible in term of resources, running a test campaign on a real-world-sized crop field would obviously be too time consuming. Despite these constraints, the Gazebo-based simulation proved effective to reveal issues in the Oz Controller [19].
3) **The test case generator.** A test case consists of a mission in a virtual crop field. As already explained, the mission file is read by the SUT, while the crop field description (the virtual world) is fed into the simulator. To automatically produce a sample of diverse test cases, we implemented techniques derived from the Procedural Content Generation of worlds [23]. These techniques aim to create randomly generated environments within a structured framework. The basic idea of PCG is to use a set of high-level parameters to control the production of concrete world contents. In our test input generator, the generation parameters are modelled using Object Oriented Modeling. Then, a Python script randomly generates the mission and world files in the appropriate formats. For that particular case study, 15 high level parameters have been chosen to model a test case. The main generation parameters are the number of rows, the type of vegetable, the space between the rows, the space between the vegetables, the magnitude and the granularity of the ground deformation. In the other paper on the same case study [19], we provide a fully detailed explanation on the generation process. While the random generation gives no guarantee on the input domain coverage, we performed a check (a posteriori) of the sample of 80 generated test cases. It shows that every generation parameter is well covered by considering a subdivision of its range in tenth. That is to say, for a parameter $p \in [0;1]$, there is at least one test case with $p \in [0;0.1]$, one with $p \in [0.1;0.2]$, etc.
4) **The analyzer.** The analyzer is in charge of reading the traces of the run tests, and of checking for unwanted behavior of the robot (test oracle). This component is really important because it produces the failure reports that are being used to study the non-determinism in this paper.
   Table I displays the timestamped data collected by the test platform and made available for the implementation of the checks. The robot logs are collected at the SUT interface. They include error or success reports sent by

the robot at the end of a run, as well as some debug data logged by the robot during cruise phases (perceived position and yaw). The simulator logs provide the actual – rather than perceived – positions and attitudes of the robot during the run.

The Pass or Fail verdict is based on a set of required properties. If at least one property failed, the entire run is considered failed. Initially, there was no specification from which to extract possible failure definitions for Oz. The list of properties had to be specified from scratch in collaboration with Naïo engineers. To do so, we followed guidelines from our previous work [20], and structured the discussion of candidate properties according to five broad categories of requirements:

a) **Requirements attached to mission phases.** The focus is on how to perform a mission. A mission typically consists of a series of phases, with some expectations on what the robot should or should not do in each phase.
b) **Thresholds related to robot movement.** Here, the aim is to detect abnormal values of kinematic or kinetic variables.
c) **Critical events,** like collisions.
d) **Requirements attached to error reports.** A robot has capabilities to monitor its operation and report errors. Requirements can be attached to the handling of these errors.
e) **Perception requirements,** focusing on unacceptable mismatches between the ground truth and its perception by the robot.

It resulted in the elaboration of 8 properties listed in Table II. Further discussion with Naïo engineers led us to consider P2 and P7 as performance-related properties that should not yield a Fail verdict. Moreover, preliminary runs to debug the checks revealed many transient violations of property P4. Looking further into the matter, we could determine that this was an artifact of the simulation. Indeed, the simulation ignores the engine braking force and overestimates velocity on downward slopes arising from terrain irregularities. We decided to de-activate the P4 check to get rid of the spurious violations. The results presented in the next section are thus for test verdicts based on P1, P3, P5, P6 and P8.

V. Experimental results

In this section, we present the results of the tests in simulation and we particularly focus on the impact of non-determinism using oracle properties violations.

Robot logs	Simulator logs
Perceived position x, y	Position x, y, z
Perceived yaw	Quaternion X, Y, Z, W
Mission success report	
Error report	
Wheel commands	

TABLE I
TIMESTAMPED DATA LOGGED DURING THE TESTS

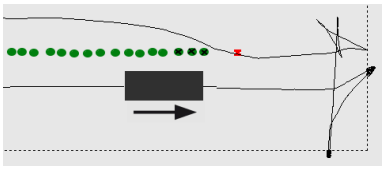


Fig. 4. Visualization of a run with multiple failure types (P6, P5)

A. Variability of the test verdicts (RQ1)

The generated set of tests consists of 80 different virtual crop fields along with their corresponding weeding mission. we perform 5 runs in each case to study the non-determinism of test executions.

Our generated environments are very stressful for the robot as a **Fail verdict occurs in as much as 48% of the runs (192/400)**. All properties but P1 (U-turn in 5-7 maneuvers) are violated by some runs. Table III provide the number of failing runs per property. Since a run may violate several properties, the counts sum up to a number greater than 192. Figure 4 illustrates a run with multiple failures. Its view uses a test data visualization facility we developed. The robot (shown as a black rectangle) starts at the external side of the field, along a row of leeks represented by green circles. While performing the U-turn at the end of the row, the robot crosses the specified limits of the field (indicated by a dotted line) and later collides with the red stake and three leeks. This run counts for both P6 (outside of the crop field) and P5 (collision). Overall, collision is the most frequent failure type, observed in 35.5% of the test runs.

Table IV gives the proportion of test cases yielding from $n = 0$ to 5 Fail verdicts over the five repeated runs (A verdict is "fail" when at least one property Px is violated). A test case has an inconsistent outcome when at least one (but not all) of its repeated runs fails, i.e., $n \in \{1, 2, 3, 4\}$. About 29% of the test cases (23 out of 80) have such an inconsistent verdict from one run to the other. Moreover, consistent Fail verdicts ($n = 5$) do not mean that the detected misbehavior is the same across all runs. Figure 5 illustrates different failures of the same test case. The upper part of the figure visualizes the run also presented in Figure 4. As previously discussed, the first U-turn yields multiple failures: one of the maneuvers exceeds the limits of the field, and there are collisions upon entering the next crop row. The rest of the mission is correctly performed. The bottom part of Figure 5 visualizes another run of exactly the same test case. This time, a new failure adds up to the others: the U-turn misses the next row and goes back to an already weeded area (P3 violation). At the end of its return trip, the robot detects that something is wrong and stops with an error report. Clearly, those two runs in Figure 5 consistently yield a Fail verdict but have different property violation patterns.

We thus conclude that the non-determinism strongly impacts the verdict assigned to a given test case.

Mission Phases	P1	U-turn in 5-7 maneuvers
	P2	Robot maintains reference distance to the vegetables
	P3	Sequence of weeded rows is correct
Movement thresholds	P4	$Velocity < V_{max}$
Catastrophic events	P5	No collision with vegetables or red stakes
	P6	Robot does not go outside of the crop field
Perception	P7	Self-localization with a certain precision
Error reports	P8	$Stopping\_distance < d_{max}$ after reporting an error

TABLE II
ORACLE CHECKS FOR OZ

P1	P3	P5	P6	P8
0	70	142	57	14

TABLE III
FAILURE TYPE COUNTS

#Test cases (total: 80)	#Fails (over 5 runs)
32	0
5	1
4	2
2	3
12	4
25	5

TABLE IV
PROPORTION OF TEST CASES WITH n FAIL VERDICTS OVER THE FIVE REPEATED RUNS.

B. Repetition vs exploration (RQ2) ?

Performing repeated runs is a classical way to account for execution non-determinism. Since each run takes minutes, one may wonder whether less than five repetitions would penalize effectiveness. Starting from the complete set of runs $X_{80,5}$ (80 test cases x 5 repeated runs) that are already performed and analyzed, we randomly extract subsets $X_{t,r}$ to mimic strategies with t test cases repeated r times, $t \in \{1, 80\}$ and $r \in \{1, 5\}$.

We first study the test size to observe at least one failure of each type (P_3, P_5, P_6, P_8). Suppose we consider r runs per test case. We keep adding new test cases, and select a subset of r runs for each of them, until we observe every property violation. The size of the test, i.e., the total number of runs, is then $r * t$. We evaluate the test size 10,000 times for statistical analysis. Figure 6 shows the result of this process in a boxplot. With only one run per test case (the first box), the median test size to violate every property is 20 (r=1; t=20). The size is more than twice as much (46) if we choose 5 runs per test case. It thus seems more efficient to favor test case diversity over execution repetition: strategies with $r > 1$ slow down the detection of new failure types.

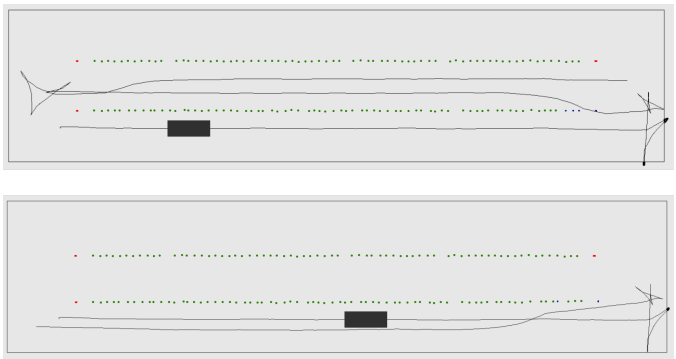


Fig. 5. Two different results with the same test case. On the top: Oz weeds the correct rows but has collisions (P5) and exceeds the field limits (P6). On the bottom: a wrong row failure (P3) adds up to the previous ones.

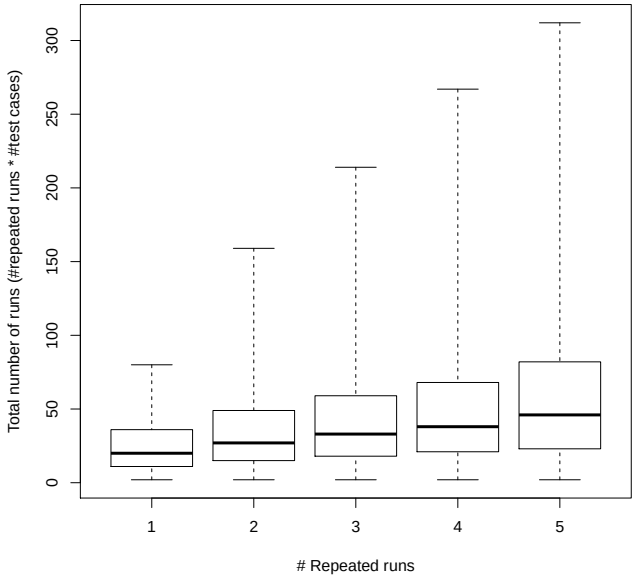
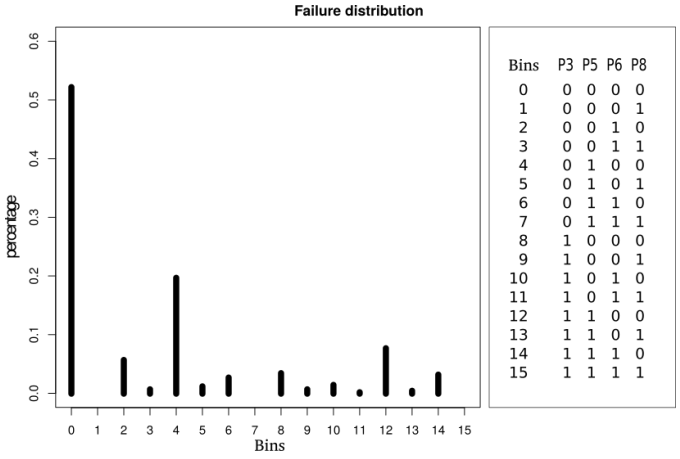


Fig. 6. Total number of runs to observe every failure type ($P_3 P_5 P_6 P_8$) at least once

We further study the relative frequencies of property violations. Figure 7 displays the histogram for the set $X_{80,5}$. The data is split into 16 bins, corresponding to every potential combination of property violation: Bin 0 gives the percentage of runs violating no property, and Bin 15 is for runs violating all four properties. We use this histogram $X_{80,5}$ as a reference to study the impact of varying t and r. Dissimilarity between $X_{80,5}$ and a set $X_{t,r}$ is measured by calculating the Manhattan distance of the bin frequencies:

$D = 0.5 * \sum_{i=0}^{15} |f_i - f_{Refi}|$,



Bins	P3	P5	P6	P8
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

Fig. 7. Relative frequencies of property violation combinations for the 400 runs ($X_{80,5}$)

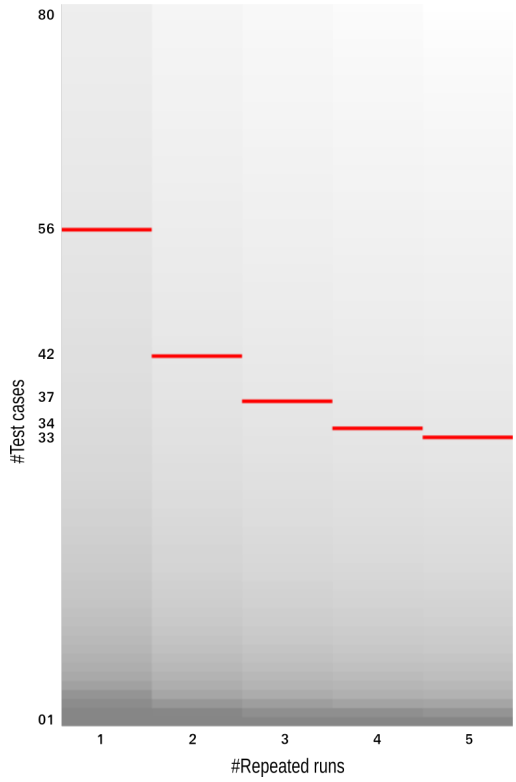


Fig. 8. Median distance to the histogram in Figure 7. In red: isodistance lines for $D = 0.1$

where the 0.5 coefficient rescales the distance to $[0, 1]$, 15 is the maximum index of bins, f_{Refi} is the relative frequency of bin i of $X_{80,5}$ (e.g., $f_{Ref0} = 52\%$ in Figure 7), and f_i is the relative frequency of bin i of the considered set $X_{r,t}$. The objective is to assess for each subset the distance D to the reference $X_{80,5}$ in order to identify which $X_{r,t}$ are sufficient to obtain a close distribution with the reference. For each pair (t, r), we randomly select 10,000 sets $X_{r,t}$ and measure their distance to the reference ($X_{80,5}$). When there are less than 10,000 possibilities, as is the case for small values of t and r, we exhaustively consider all the possible sets. Figure 8 shows the median distances we obtain. Their value is encoded by a tone on a linear grey scale, where white means $D = 0$ and black means $D = 1$. As can be seen for all repeated runs schemes, the distributions of failure violation patterns tend to become more similar as the number of test cases grows (the tone becomes lighter). An interesting point is to look at isodistances. For instance, the red lines in Figure 8 delimitates the (t, r) configurations with a distance $D = 0.1$ to the reference. One can observe that it takes much longer to get within this distance with repeated runs than without: the total test size ranges from 56 runs up to 165 ($5 * 33$). Intuitively, it suggests that the final frequencies observed for $X_{80,5}$ are more determined by the diverse test cases we execute than by the repeated execution of each test case.

We thus come to a surprising conclusion. Although execution non-determinism has a high impact on the test verdict, it seems cost-effective to simply ignore the issue and use the test budget to execute diverse test cases without repetition. Cost-effectiveness is of course relative to the specific case study. More generally, an interesting outcome is to demonstrate that repeated runs are not always the best approach to test non-deterministic applications.

VI. Conclusion

The validation of autonomous robots strongly depends on the resources available to perform test campaigns. Through simulation, more intensive campaign can be carried out but the result of tests may be not relevant due to the non-determinism. In this paper, we propose an approach to assess if repeated runs are more efficient than diversity of test cases execution, in the context of debugging. To do so, we use a complete testing framework, from the test generation to the oracle analysis. We then consider 400 runs (80 tests cases, executed 5 times), and analyse the results. This analysis relies on two methods: the first one shows the evolution of the number of run needed so that every property violation occurs at least once. The second one focuses on the evolution of the failure distribution regarding the number of repeated runs and the number of different test cases. Surprisingly, our study shows that for our case study it would be more efficient to favor diversity over repeated run. A future direction will be to apply the same method to a new case study on another agricultural robot from Naïo. This time we will use a custom simulator from Naïo with a lower level of fidelity but far more performant. It will allow us to generate and execute more test cases.

Acknowledgment

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644400 (CPSE Labs project). The authors want to acknowledge the help of colleagues at Naïo Technologies during the study: Simon Vernhes, Gaëtan Séverac, Pascal Schmidt, Marc Jambert.

References

[1] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, Singapore*, pages 63–74, 2016.

[2] Anneliese Amschler Andrews, Mahmoud Abdelgawad, and Ahmed Gario. World model for testing urban search and rescue (usar) robots using petri nets. In *4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), Rome, Italy*, pages 663–670, 2016.

[3] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of Computer Programming*, 16(2):103 – 149, 1991.

[4] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19-2:87–152, 1992.

[5] Jean-Jacques Borrelly, Eve Coste-Manière, Bernard Espiau, Konstantinos Kapellos, Roger Pissard-Gibollet, Daniel Simon, and Nicolas Turro. The orccad architecture. *The International Journal of Robotics Research*, 17(4):338–359, 1998.

[6] Gilberto Echeverria, Nicolas Lassabe, Arnaud Degroote, and Séverin Lemaignan. Modular open robots simulation engine: Morse. In *IEEE International Conference on Robotics and Automation (ICRA 2011), Shanghai, China*, pages 46–51, 2011.

[7] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79:1305 – 1320, 10 1991.

[8] Félix Ingrand and Malik Ghallab. Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 247:10–44, 2017.

[9] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004), Sendai, Japan*, volume 3, pages 2149–2154, 2004.

[10] Yihao Li, Jianbo Tao, and Franz Wotawa. Ontology-based test generation for automated and autonomous driving functions. *Information & Software Technology*, 117, 2020.

[11] E. Marchand, E. Rutten, H. Marchand, and F. Chaumette. Specifying and verifying active vision-based robotic systems with the signal environment. *The International Journal of Robotics Research*, 17(4):418–432, 1998.

[12] Lindvall Mikael, Porter Adam, Magnusson Gudjon, and Christoph Schulze. Metamorphic model-based testing of autonomous systems. In *Proceedings of the 2nd International Workshop on Metamorphic Testing (MET), Buenos Aires, Argentina*, pages 35–41, 2017.

[13] https://www.naio-technologies.com/, 2018. Accessed: 2019-09-19.

[14] Cu D. Nguyen, Anna Perini, Paolo Tonella, Simon Miles, Mark Harman, and Michael Luck. Evolutionary testing of autonomous software agents. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary*, volume 1, pages 521–528, 2009.

[15] https://www.oktalsydac.com/, 2018. Accessed: 2019-09-19.

[16] https://tass.plm.automation.siemens.com/prescan, 2018. Accessed: 2019-09-19.

[17] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software, Kobe, Japan*, volume 3, 2009.

[18] Christophe Reymann, Mohammed Foughali, and Simon Lacroix. Repeatable decentralized simulations for cyber-physical systems. In *International Conference on Software Quality, Reliability and Security (QRS 2019), Sofia, Bulgaria*, pages 240–247, 2019.

[19] C. Robert, T. Sotiropoulos, H. Waeselynck, J. Guiochet, and S. Verhnes. The virtual lands of oz: testing an agribot in simulation. *in Empirical Software Engineering (EMSE), yet to appear. DOI: 10.1007/s10664-020-09800-3*.

[20] T. Sotiropoulos, H. Waeselynck, J. Guiochet, and F. Ingrand. Can robot navigation bugs be found in simulation? an exploratory study. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS 2017), Prague, Czech Republic*, pages 150–159, 2017.

[21] Thierry Sotiropoulos, Jérémie Guiochet, Félix Ingrand, and Hélène Waeselynck. Virtual worlds for testing robot navigation: a study on the difficulty level. In *IEEE 12th European on Dependable Computing Conference (EDCC 2016), Iasi, Romania*, pages 153–160, 2016.

[22] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018), Gothenburg, Sweden*, pages 303–314, 2018.

[23] 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*, 3:172–186, 2011.

[24] https://unity.com/, 2019. Accessed: 2019-09-19.

[25] https://www.unrealengine.com/, 2018. Accessed: 2019-09-19.

[26] http://www.mscsoftware.com/product/virtual-test-drive, 2018. Accessed: 2019-09-19.