



The virtual lands of Oz: testing an agribot in simulation

Clément Robert¹ · Thierry Sotiropoulos¹ · Hélène Waeselynck¹ · Jérémie Guiochet¹ · Simon Vernhes²

Published online: 25 February 2020

© Springer Science+Business Media, LLC, part of Springer Nature 2020

Abstract

Testing autonomous robots typically requires expensive test campaigns in the field. To alleviate them, a promising approach is to perform intensive tests in virtual environments. This paper presents an industrial case study on the feasibility and effectiveness of such an approach. The subject system is Oz, an agriculture robot for autonomous weeding. Its software was tested with weeding missions in virtual crop fields, using a 3D simulator based on Gazebo. The case study faced several challenges: the randomized generation of complex 3D environments, the automated checking of the robot behavior (test oracle), and the imperfect fidelity of simulation with respect to real-world behavior. We describe the test approach we developed, and compare the results with the ones of the industrial field tests. Despite the low-fidelity physics of the robot, the virtual tests revealed most software issues found in the field, including a major one that caused the majority of failures; they also revealed a new issue missed in the field. On the downside, the simulation could introduce spurious failures that would not occur in the real world.

Keywords Testing · Software in the loop (SIL) simulation · World generation · Autonomous robot · Agriculture robot · Industrial case study

Communicated by: Hadi Hemmati

✉ Hélène Waeselynck
Helene.Waeselynck@laas.fr

Clément Robert
clement.robert@laas.fr

Thierry Sotiropoulos
thierry.sotiropoulos@gmail.com

Jérémie Guiochet
jeremie.guiochet@laas.fr

Simon Vernhes
simon@naio-technologies.com

¹ LAAS-CNRS, Université de Toulouse, CNRS, UPS, Toulouse, France

² Naïo Technology, Toulouse, France

1 Introduction

Autonomous robotic systems have decisional capabilities allowing them to accomplish complex missions without human intervention. Self-driving cars and unmanned aerial vehicles are emblematic examples of systems behaving with a high degree of autonomy. More generally, there is a huge potential for autonomous robots in many different application domains like space exploration, manufacturing, personal assistance, rescue operation or agriculture, to name a few. The robot studied in this paper, Oz, comes from the agriculture domain and is deployed in vegetable crops for autonomous weed control. It is representative of the many innovations currently introduced by the agricultural high-tech industry. Tractica (a market intelligence firm in US) forecasts that the agricultural robot market will increase exponentially from \$3 billion in 2015 to \$16 billion in 2020 and then \$73 billion in 2024 (Tractica 2016).

The rise of autonomous robots in many domains creates new challenges for their validation. In particular, the mission-level validation typically involves test campaigns in the field, which are costly and may also be risky in case of misbehavior. Dangerous tests must be avoided as much as possible, or stopped before the robot causes any harm to itself, people or property. In the case of agricultural robots, additional constraints come from the seasonal characteristics of the missions, which further limit the situations that can be practically tested at a given time of year. For example, a crop field in January does not resemble the same field in May in terms of weed and crop growth.

Given these issues, a pragmatic approach is to consider simulation-based testing. The robot is immersed in a virtual world, and can be tested in a more flexible, safer and less costly way than in the real world. Such potential benefits motivated Naïo Technologies – the French small tech company developing Oz and other agricultural robots – to introduce mission-level simulation in their validation process. They implemented a software-in-the-loop (SIL) simulation, where the real software is tested but the hardware and physical components of the robot are simulated. Interestingly, their initial experience was disappointing. The engineers faced simulation performance issues, imposing a simplification of the simulated physics and a limitation of the complexity of the virtual environments. As a result, when the study reported in this paper started, Naïo made limited use of simulation-based testing. They ran few virtual test cases (most often just one) as a smoke test prior to field testing. The aim was to ensure that the software can successfully execute a mission before proceeding with field testing. Naïo did not consider more intensive virtual testing beyond those few cases. There was a perception that realistic test conditions were essential for a proper validation of the robotic software, and that the simulation was too far from reality.

Joint work between Naïo and LAAS researchers questioned this perception. We experimentally studied the revealing power of more intensive testing of Oz in simulation. Oz was tested in a wide variety of virtual crop environments, which was made possible by using a randomized test generation procedure. The results were then compared to the ones of the industrial field tests. The main outcome of the study is that simulation-based testing can be effective even if it imperfectly reproduces real-world conditions:

- The virtual tests could find several software issues that surfaced during the field tests, including the issue that caused the majority of failures;
- They also revealed a new issue not exposed by the industrial field tests;
- On the downside, the simulation introduced spurious failures that would not occur in the real world.

A side outcome of the study concerns the key challenges and lessons learnt in the design of the virtual tests. One of the challenges was to capture the key characteristics of crop fields and weeding missions in a test generation model. Another one was the definition of the test oracle procedure, in order to detect misbehavior. Both of these challenges necessitated several interactions between Naö and LAAS. The exercise illustrated the real difficulties associated with specifying the environment and expected behavior of an autonomous system. The paper reports on our experience and extracts some recommendations.

The structure of the paper is as follows. Section 2 discusses related work. Section 3 introduces the case study: the Oz robot and its simulation platform, the experimental approach for studying tests in simulation. Test design is detailed in Sections 4 and 5, respectively addressing the randomized generation of virtual crop fields and the automated checking of test traces. Section 6 presents the misbehavior patterns revealed by the random tests in simulation. Section 7 performs the comparison with software issues found by the field tests. Section 8 provides an overview of the experimental outcomes and discusses threats to validity. Finally, Section 9 concludes.

2 Related Work

In this section, we focus on the mission-level validation of autonomous systems in simulated environments. We discuss three challenges: the fidelity of the simulation, the generation and selection of test cases, and the specification of the test oracle.

2.1 Fidelity of the Simulation

The situation regarding the availability of realistic simulation platforms for autonomous systems is contrasted.

On the one hand, systems like self-driving cars or Advanced Driving Assistance Systems (ADAS) benefit from the critical mass of the automotive sector. Dedicated simulation platforms with a high level of fidelity have been developed (e.g. see Virtual Test Drive (2018), PreScan Simulation platform for ADAS (2018) or Okdal Sydac (2018)). They provide vehicle dynamics models, realistic sensor models and facilities to create complex virtual driving environments. The environments include static elements such as the road infrastructure but also dynamic elements like pedestrians, other vehicles and global traffic. Camera-based systems are used to detect lanes, obstacles or traffic signs and require an accurate representation of the real world.

On the other hand, there are no such dedicated simulators for the many existing robotic applications, due to the diversity of robots, architectures and environments. The system providers have to develop their own environment and system models, integrated into generic simulation platforms. For example, Gazebo (Koenig and Howard 2004), MORSE (Echeverría et al. 2011) or even Unreal (Unreal Game Engine 2018) (originally developed for video games), are more and more used for building software-in-the-loop simulators. Those simulators may not be as sophisticated as the ones benefiting from a dedicated technology (like in the automotive domain). The achievable fidelity is typically limited by the amount of development effort that can be put on the simulation and by the computational resources available for running the tests.

Fortunately, some studies concur that a low-fidelity simulation can still be relevant for testing purposes. Arnold and Alexander (2013) tested a simple robot controller, consisting in a path-following algorithm with collision avoidance. A basic simulation in 2D obstructed

environments sufficed to reveal several issues. Our previous work (Sotiropoulos et al. 2017) performed an in-depth analysis of faults in the navigation software of Mana, a rough-terrain experimental robot. Out of the 33 bugs extracted from the code commits, only one requires a high fidelity to be replicated in simulation (the bug is related to mechanical vibration that was not simulated). Timperley et al. (2018) came to a similar conclusion for bugs in the open-source ArduPilot system: the majority of them surface under simple conditions that can be easily reproduced in software-based simulation.

In this paper, the simulator was developed by Naïo Technologies based on the Gazebo generic platform. It can be considered as a low-fidelity simulator regarding the physics of the robot. Still, it is more elaborated than the simulators used by many of the robot testing work discussed in the next subsection. Naïo wanted sufficient degree of detail to study the full integration of the perception, decision and motion-control functions.

2.2 Generation and Selection of Test Cases

We identified two broad categories of test approaches for autonomous systems: approaches that reproduce real-world tests in simulation, and model-based generative approaches that create new synthetic tests.

The first category is mostly studied in the automotive domain. Car constructors have collected large volumes of data from real drives, and can leverage them to test ADAS and autonomous driving functions in simulation. But the reproduction of real test cases is not straightforward: it requires advanced post-processing of the recorded data. For example, (Bach et al. 2017) reconstruct the geometry of a road section based on the steering wheel angles and visual lane recognition, while others use the DGPS position data to retrieve the road model from a digital map database (Nentwig and Stamminger 2010; Lamprecht and Ganslmeier 2010). Image processing techniques are used to identify relevant objects (other cars, pedestrians) and reconstruct their trajectory relative to the ego-vehicle (Nentwig and Stamminger 2010). Since the recorded data set may yield a large amount of redundant test cases, some authors extract a small subset that covers abstract situations of interest or classes of parameter values (Bach et al. 2017). Other authors have worked on increasing the diversity of the reconstructed cases, by mutating them to produce variants (Zofka et al. 2015). For example, a spatial translation is applied to the original trajectory of a car.

In the second category, the generative approaches do not require the availability of real data sets and offer flexibility to ensure test diversity. However, they require the definition of a generation model, which can be very challenging. Think of a test case as a virtual world in which the autonomous system is asked to perform a mission: the set of relevant worlds and missions is infinite and difficult to characterize. In practice, the model is derived from domain-specific knowledge. For example, a world model for an ADAS is expected to include intersecting road segments, stationary and mobile obstacles on the road, weather conditions, etc. Several modeling approaches have been proposed to provide a structured view of the world elements and their relations: ontologies (Geyer et al. 2014; Ulbrich et al. 2015; Klueck et al. 2018), UML structure diagrams (Micskei et al. 2012; Sotiropoulos et al. 2016; Andrews et al. 2016), or XML-based decompositions of the domain (Zendel et al. 2013). The dynamic aspects are often included as parameters of the structural elements, e.g., a mobile object has attributes to parametrize its trajectory. However, for some authors, a behavioral model explicitly supplements the structural one: UML Sequence Diagrams for Micskei et al. (2012), Petri nets for Andrews et al. (2016). Model-based test criteria classically consist in covering (combinations of) classes of parameter values, and possibly sequences of events in a behavioral model.

Rather than using model coverage criteria, some authors apply search-based testing to find fail scenarios. For example, the aim is to find the collision scenarios that may be generated from the model. The used techniques range from a simple random search over the parameter space (Arnold and Alexander 2013) to advanced techniques combining an evolutionary search with learning algorithms (Ben Abdessalem et al. 2016, 2018). Several challenges are faced when applying techniques beyond a random search. First, the simulation time of a test case may be long (e.g., it takes minutes), which severely constrains the number of iterations allowed for the search. To alleviate the problem, Ben Abdessalem et al. (2016) use neural networks to predict the fitness values without running the actual simulations. Only the test cases with sufficiently high predicted values are executed, the other ones are skipped. A second challenge is the non-deterministic behavior of autonomous systems. It may be due to the non-determinism of the decision algorithms, or simply to the execution non-determinism that is common in highly concurrent robotic software. To the best of our knowledge, Nguyen et al. (2009) were the only ones to account for non-determinism in their evolutionary testing of an autonomous agent. The fitness value was calculated from 5 repeated runs of the test case, which was empirically found to ensure the stability of the evaluation. Note how the need for repeated executions exacerbates the problem of the long simulation time. Finally, a third challenge is the effective degree of control provided by the high-level generation parameters defined in a model. There may be a considerable gap between the abstract world model and the concrete elements fed into the simulator. Such is the case if, as proposed by Arnold and Alexander (2013), the generation of the concrete worlds uses procedural content generation (PCG) techniques developed in the domain of video games (see, e.g., the survey by Togelius et al. (2011)). For example, Arnold and Alexander (2013) demonstrated the generation of 2D maps by means of a Perlin noise process and some filter effects. The model parameters were the size of the map, the obstruction rate and the settings for the filters. These are very high-level parameters compared to the concrete map contents. Similarly, our work on the Mana robot retained the principle of PCG and generated 3D maps using facilities from the Blender game environment (Sotiropoulos et al. 2016). The parameters of the model (smoothness of the terrain, obstruction rate) were found to provide a coarse control of the difficulty of navigating in the concrete maps.

In this paper, we adopt a generative approach. The world model is specified in an UML structure diagram, augmented with an attribute grammar for the description of semantic constraints. The parameter space is explored by a random search, using PCG techniques to produce concrete worlds from the generated parameter values. We leave more advanced search techniques for future work, given the difficulties in terms of simulation time, non-deterministic behavior, and degree of control of the search.

2.3 Specification of the Test Oracle

For autonomous systems, the oracle problem is specifically difficult due to the decisional aspects. They make the specification of expected behavior quite challenging. For example, consider an autonomous driving system. As noted by Tian et al. (2018), creating detailed specifications for such a system would essentially involve recreating the logic of a human driver. To circumvent the problem, a practical approach is to check for a limited set of properties, like metamorphic properties (involving related test executions) or invariant properties (that should hold for any test execution).

In metamorphic testing (Chen 2015), the properties relate a test case and some follow-up cases. The approach has been used by Zhang et al. (2018) to test driving models based

on Deep Neural Networks. The follow-up cases consist in modifying the weather conditions and checking whether the driving behavior remains as in the original test. Similarity is assessed within some tolerance in the steering angle variation. In the same vein, Tian et al. (2018) check that the steering angle does not change significantly when certain transformations are applied to the input images. For testing a drone AI controller, Lindvall et al. (2017) consider follow-up cases like rotations and translations of the world geometry. For example, if the world is rotated 180 degrees, so that the drone is flying South instead of North, the behavior should still be similar. Note that metamorphic testing may be impractical if the system behavior is highly non-deterministic. In Sotiropoulos et al. (2016), we tested the navigation of Mana, an academic all-terrain robot, and observed completely different trajectories in repeated runs of exactly the same test case. Then, finding metamorphic properties appears to be problematic.

Regarding invariant properties, the test oracle typically checks that some critical failures never occur. For example, collisions are critical when testing drones (Zou et al. 2014), autonomous cars (Ben Abdessalem et al. 2018) and various types of mobile robots (Nguyen et al. 2009; Arnold and Alexander 2013). When specifying the oracle, the retained set of properties should not be too narrow. There may be a wide variety of misbehaviors other than collisions, as shown by the history of navigation bugs affecting Mana (Sotiropoulos et al. 2017). From the analysis of these bugs, we identified at least five aspects that would be worth considering in the oracle: (i) requirements attached to mission phases, (ii) threshold-based invariants related to robot movement, (iii) absence of critical events (like collisions), (iv) requirements attached to error reports, and (v) good perception requirements.

Rather than reporting a Pass/Fail verdict, the oracle may report a degree of satisfaction or violation of the checked requirements. Such a grading approach has been formalized by recent work on cyber-physical systems (Menghi et al. 2019), for properties expressed in a fragment of the Signal First Order logic. For example, if an output signal should remain below a threshold, a slight overshooting is graded as less severe than a large one. Generally speaking, a quantitative oracle may be convenient to flag the most critical tests to the attention of engineers (Arnold and Alexander 2013; Hallerbach et al. 2018).

In this paper, the test oracle for Oz delivers a Pass/fail verdict based on invariant properties. The retained set of properties refers to the five aspects identified by our previous work on navigation bugs.



Fig. 1 Oz in operation. Left: front view; right: back view with tools

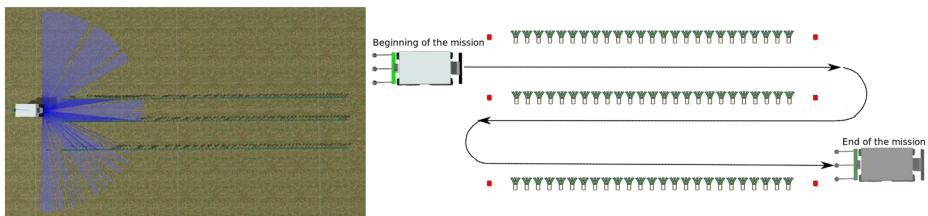


Fig. 2 Oz in simulation. Left: virtual crop field using Gazebo. Right: expected mission

3 Case study and Experimental Approach

3.1 The Oz Robot and its Simulator

Oz is an autonomous weeding robot for fields of vegetables. Figure 1 shows Oz in operation. Its small size ($75\text{ cm} \times 45\text{ cm} \times 55\text{ cm}$) allows it to navigate between the crop rows composing a field. The weeding is performed mechanically, using specific tools attached to the rear of the robot and pulled by it. When there are several rows to weed, the robot must make a U-turn at the end of a row to go to the next. It also has to decide whether the weeding is done in one or two passes, depending on the distance between rows. This is illustrated by the mission in Fig. 2: the interspace between the first two rows can be weeded in one pass, but the next interspace is a bit wider and needs two passes.

Oz perceives its environment using a laser sensor (LiDAR 2D) at the front, as well as two cameras. In the version under study, line tracking along rows of vegetables relies on the lidar only. The cameras are used during U-turns. Red-colored stakes delimitate the extremities of each row. They are placed at about 50 cm of the first and last vegetable in the row. The robot identifies them and uses their visual perception for its U-turn maneuvers. The cameras also allow the use of stereo visual odometry techniques to detect possible skidding of the robot during U-turns. Finally, contact sensors (bumpers) detect collisions and trigger an emergency stop. These safety devices prevent human injury, given the low velocity of Oz (0.4 m.s^{-1}), its moderate weight (120 kg with tools) and the fact that the weeding tools are not dangerous by themselves. Still, the operation of the robot is not without risk. Consider an ill-controlled trajectory: crop plants could be damaged, or the robot could reach a dangerous area outside of the crop field (e.g., a road).

Naïo technologies has developed SIL simulation for testing the navigation of Oz. Figure 3 gives an overview of the corresponding test architecture. The platform is based on Gazebo, a simulator widely used in robotics research. The software under test, OzCore, receives simulated sensor data and produces actuator commands, the effects of which are simulated to update sensor data. OzCore is written in C and C++, for a total of about 151 KLOC. This code includes some Oz functions that are not handled by the platform. The effect of the weeding tools is not simulated, hence the corresponding output commands are ignored. As shown in Fig. 3, the simulator only receives the wheel speed commands controlling the robot motion. Also, in the real world, the farmer parametrizes the mission via a user interface. In the simulation platform, the mission parameters directly come from a .json file.

The simulator uses three inputs to instantiate a virtual crop field: one image .jpg encoding the 3D terrain and two .sdf files for the other elements. Figure 2 is representative of

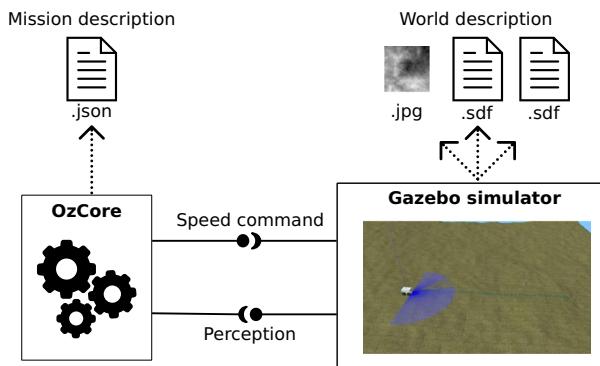


Fig. 3 Simulation architecture

the complexity of the inputs that can realistically be managed given the simulator's performance. The virtual field has three rows of vegetables, low weed density between them, and the length of the rows is small compared to the one in real world fields. Despite the limited scale of the field, this test case manages to exercise many important aspects of a mission, like crop row perception in the presence of weeds, line tracking, end of row detection, two U-turns with different interspace configurations, and one-pass/two-passes decision. Indeed, this specific case is the one that is the most often used as a smoke test before field testing.

Performance issues not only limit the complexity of the test cases, they also force us to compromise on the simulated physics. Initially, Naío implemented an accurate model of the interaction between the wheels and the ground. It was deemed important to reproduce the effect of mud, stones, ground sliding, etc. However, the accurate model proved too demanding in terms of computing resources. To give an idea, a PC with 2 Quad core Intel Xeon E5-2623 v3s CPU at 3.5GHz, and 64GB of RAM is not sufficient to run the test case in Fig. 2 (which has a flat terrain and none of the above-mentioned stressful elements). An alternative and much-simplified model of the wheels is thus provided for the tests: the wheels are abstracted as mere cylinders and the physics ignores friction at the ground. Our study of simulation-based testing uses this low-fidelity version of the platform.

3.2 Experimental Approach

Our research question concerns the revealing power of simulation-based testing compared to testing in real-world conditions:

RQ: What are the benefits and drawbacks of simulation-based testing with respect to:

- Real issues found by the field tests,
- Real issues not found by the field tests,
- Spurious issues coming as simulation artifacts?

In order to answer this question, we developed an automated approach for testing Oz with a large sample of virtual test cases, and then compared the results with the industrial field tests. LAAS researchers made the virtual test development, with support from Naío Technologies for the initial test design and the final comparison. The Ozcore version under

test was an R&D version that became ready for field testing during the study. Its potential faults were unknown and could not bias the test design. Moreover, LAAS researchers were kept uninformed of the results of field testing until the final comparison.

The study involved three phases, to be described in greater details in the next sections:

- Define the test generation model (Section 4) and oracle procedure (Section 5). The generation model should ideally capture all possible crop fields and weeding missions in these fields that may be worth testing. Such a model did not exist at the beginning of the study and had to be specified. Likewise, there was no formalized definition of expected behavior for Oz: the properties to check had to be crafted with the help of Naïo experts.
- Run the virtual test experiments and analyze the fail cases, in parallel with the test campaigns by Naïo (Section 6). A key driver was to prepare the final comparison with field testing. The concrete fail cases were categorized into high-level misbehavior patterns, likely to indicate different software issues. This was done by manual inspection of the detailed test traces, with the help of data visualization facilities. Note that the analysis remained black box: the LAAS researchers did not access the source code.
- Compare the results of the simulation runs with the field tests results (Section 7). A key driver was to determine whether the software issues identified by the test experiments at LAAS matched the set of issues found by Naïo. In cases they did not, it had to be determined what was missed, what was new and whether the potentially new issues were real or spurious.

All outcomes of the study are finally gathered in Section 8 to answer the research question. We additionally draw some lessons regarding the development of automated simulation-based testing of Oz.

4 Test Design: World and Mission Generation

The simulator must be fed with the description of a crop field and a weeding mission. The corresponding input artifacts are low-level, providing any necessary details about the virtual environment in a format understood by the simulator. In such cases, it is convenient to approach the generation problem at a higher level of abstraction. It yields a two-step process that generates abstract test case descriptors from a world model and then produces the concrete artifacts. The world model, upon which the generation process is based, was a key part of the test design. It took several meetings between LAAS and Naïo to discuss its contents.

After a brief overview of the two-step generation process, this section focuses on how we designed the world model. Some implementation aspects are also presented.

4.1 A Two-Step Generation Process

To manage the low-level test artifacts, we propose to implement techniques derived from the Procedural Content Generation of worlds (see Section 2.2 and Togelius et al. (2011)). The basic idea of PCG is to use a set of high level parameters (called the *genotype*) to control the production of concrete world content (*phenotype*). The process is usually based on randomized procedures, and thus may produce different phenotypes from a given value of the genotype.

Figure 4 shows the use of PCG in a two-step generation approach, where the parameters in the genotype come from a world model. To illustrate the process, let us take the example of the generation of an uneven terrain. The world model has a terrain element with a high-level parameter characterizing its roughness. The first step is to randomly select the roughness value as part of the genotype. Then, the second step generates a concrete terrain artifact, namely an image encoding the 3D relief (e.g., the image in the rightmost part of Fig. 4). A noise function may produce the content of this image, the noise level being tuned at the desired roughness value.

The two-step approach allows keeping the world model at a high level of abstraction. The modeling can concentrate on identifying the major elements of the world and their macroscopic characteristics (e.g., a terrain element and its roughness degree), without being overwhelmed by the complexity of microscopic details (e.g., the precise shape of the terrain at every point) or the idiosyncrasies of the simulation platform (e.g., the image format to produce).

4.2 World Model Definition

The modeling first created a structured view of the candidate world elements and their high-level parameters, with the support of UML diagrams. The visual presentation with diagrams was useful to discuss the world model with experts in the system, who could suggest extensions or simplifications. Then, the formalization of the genotypes was finalized by switching to a grammar-based representation. In this representation, the genotypes are strings that encode the parameters of each world element to generate. The use of an attribute grammar formalism was found convenient to specify these genotype strings, accounting for both structural concerns (how the complete genotype aggregates the parameters of the world elements) and semantic ones (the domains of the parameters, the dependency constraints between them).

Applied to the Oz case study, this modeling approach started by the elicitation of structural relations, such as:

- A Field is composed of :
 - Crop_rows composed of
 - * Crops which could be of type Cabbage or Leek
 - * Red Stake at the beginning and end of rows
 - Weed_areas which contain the wild grass to eliminate.

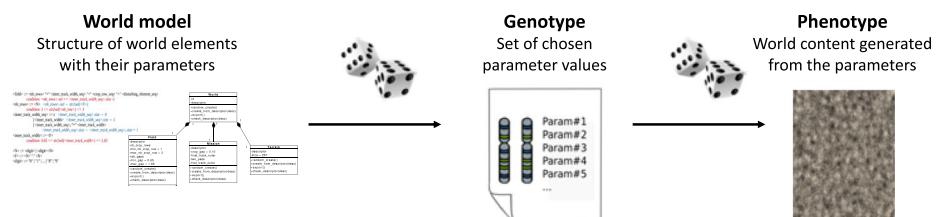


Fig. 4 Two-step generation process. The dice symbol indicates randomized procedures at both steps: (1) generation of values to instantiate the world model parameters, (2) generation of a concrete world content from the parameter values

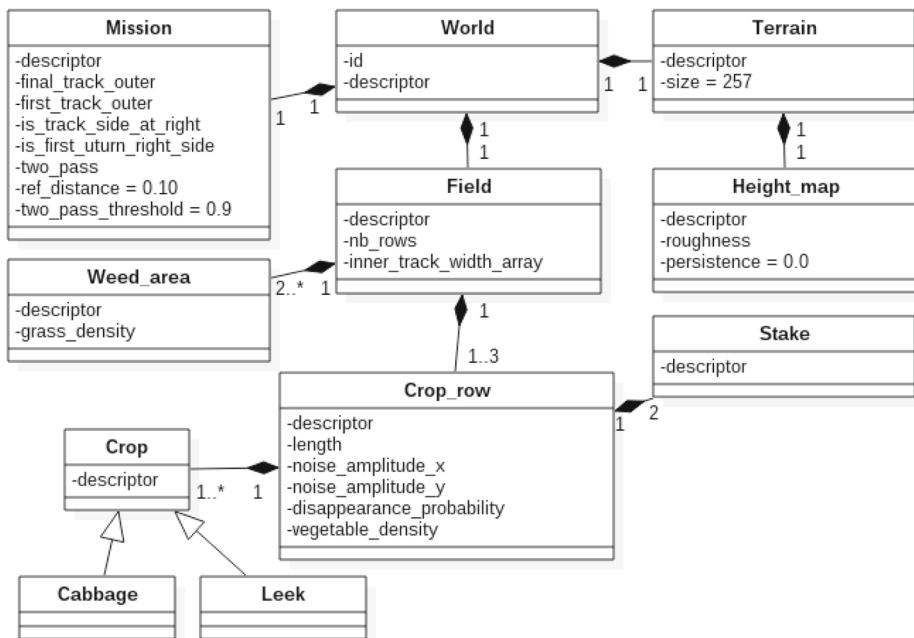


Fig. 5 UML class diagram of the World model

The complete structure is captured by the UML class diagram in Fig. 5. The model aggregates world elements (classes) related to the crop field, the 3D terrain and the weeding mission. The decomposition was refined until the level of detail was deemed sufficient. Each element has attributes representing generation parameters. The first version of the model had more than 30 parameters, but several simplifications were suggested by Naïo. For example, the initial model had parameters to allow for misplaced or missing red stakes. It was judged as too stressful for Oz and this possibility was removed. The final model has 15 parameters, including 14 class attributes plus the choice of the crop plant type. The diagrammatic representation facilitated the discussion with the engineers, being easy to understand and modify.

Figure 6 illustrates some of the retained parameters. The space between two rows (inner_track_width) and the density of vegetables in each row pertain to the crop field, while the other parameters characterize the mission. For example, `first_track_outer` indicates whether the robot starts at the external side of the field. Parameter `final_track_outer` indicates whether the robot is allowed to weed the other external side reached at the end of the mission (the farmer can forbid this when entering the mission).

The set of parameters defines the genotype. To explicitly model this concept, each class has a `descriptor` attribute that reifies its genotype as a string. It encodes the values of all parameters from this class and the contained classes. Hence, the root world descriptor exposes the complete genotype of a test case.

The parameters in the genotype are not independent from each other. If the crop field has N rows, we must consider $N-1$ row spacing values. If it has a single row then `first_track_outer` must be true. The rows must have nearly the same length, and should all contain the same type of vegetable (these were simplifications suggested by

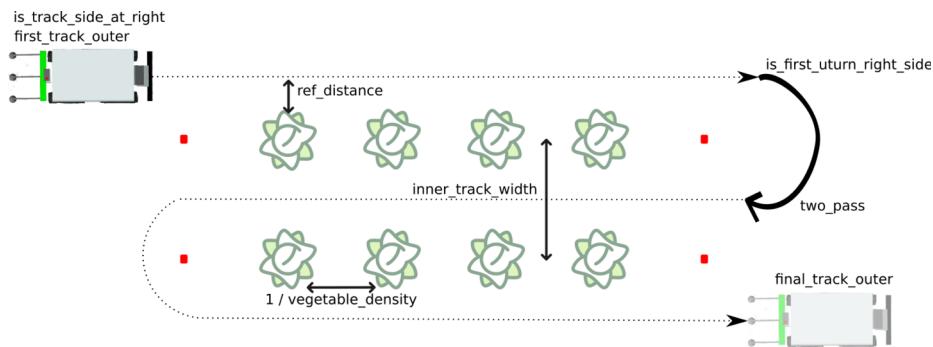


Fig. 6 Schematic view of a subset of parameters

Naïo). In a string-oriented view of genotypes, such constraints may be formalized by an attribute grammar. A genotype is then valid if it is a word of the grammar.

Figure 7 shows an extract of the grammar. The syntactic rules (in black) encode the structure of the class diagram into the strings. For example, a world descriptor has a substring to describe its field element. Attributes are associated with elements, like *val* associated with `<nb_rows>`. Assignments (in blue) and conditions (in red) specify their value, adding semantic information to the syntax. The used language includes all classical operators (Boolean, relational, arithmetic) plus some utility functions like `str2val()` converting a numerical string into its value. In Fig. 7, the shown extract focuses on how we specify the constraint relating the number of rows (`<nb_rows>.val`) and the one of row spacing values (`<inner_track_width_seq>.size`). Constraints on multiple attributes are introduced in the first common ancestor in the decomposition hierarchy, hence this constraint appears as a condition attached to `<field>`. Its truth value is determined by the assignments performed in the child elements. Note how the value assigned to the *size* attribute of `<inner_track_width_seq>` is defined recursively, depending on how many `<inner_track_width>` elements are produced.

Conceptually, the grammar-based specification of descriptors makes the genotype a first-class citizen of the model. The implementation of the test generator kept the same line: it creates and manipulates world element descriptors, as described next.

```

<world> ::= <terrain> "-" <field> "-" <mission>
<field> ::= <nb_rows> "+" <inner_track_width_seq> "+" <crop_row_seq> "+" <weed_area_seq>
           condition: <nb_rows>.val == <inner_track_width_seq>.size -1
<nb_rows> ::= <N> <nb_rows>.val ← str2val(<N>)
           condition: 1 <= str2val(<nb_rows>) <= 3
<inner_track_width_seq> ::= ε <inner_track_width_seq>.size ← 0
                           | <inner_track_width> <inner_track_width_seq>.size ← 1
                           | <inner_track_width_seq>₂ ":" <inner_track_width>
                             <inner_track_width_seq>₂.size ← <inner_track_width_seq>₂.size + 1
<inner_track_width> ::= <F>
           condition: 0.65 <= str2val(<inner_track_width>) <= 1.65

<N> ::= <digit> | <digit><N>
<F> ::= <N> ." " <N>
<digit> ::= "0" | "1" | ... | "8" | "9"

```

Fig. 7 Extract of the grammar of genotypes

4.3 Implementation of a Test Generator

As presented in Fig. 4, the test generation has two steps, each involving randomness: (1) the production of a valid genotype, (2) the production of a phenotype from this genotype. They have been implemented by a custom generator, developed in Python in an object-oriented style. The structure of the Python program is the same as the one of the world model in Fig. 5, with the addition of methods to each class. The distribution of the code across the hierarchy of world elements, each offering a unified interface, is intended to facilitate the co-evolution of the world model and its generator. Indeed, there were several preliminary versions to demonstrate the generation functions. It was possible to accommodate the addition or removal of parameters, with limited impact on the overall code.

A world element has the following interface to manage its genotype:

- `random_create()`: randomly generates a descriptor, that is to say randomly picks a valid configuration for the parameters of the class and its child classes. If this method is called in the world class (the root of the structure), it recursively calls the same method of every child class to randomly construct the complete world genotype. The structure is instantiated such that each parameter has the value encoded by the descriptor.
- `create_from_descriptor(desc)`: also instantiates the structure, but offers more control on the values of parameters. The descriptor given in input can carry two types of elements: terminals (to force a value) and operators (to select a generation method). Operators can be seen as wildcards that replace a valid substring in a descriptor. Operator r is simply a random generation by `random_create()`. For instance, `desc="0.0-r-r"` at the root level (`|terraini-|fieldi-|missioni`) means that we want to create any valid genotype with a flat terrain (the terrain roughness is forced to 0.0, the mission and crop field parameters are random). Other operators m and M have also been implemented. They respectively select the minimum and maximum value of a numerical parameter, e.g., we want any field with the maximum number of rows. The combination of terminals and operators offers high flexibility to explore regions of the parameter space.
- `check_descriptor(desc)`: parses a given descriptor and checks if it is valid. It analyzes both the syntactic and semantic constraints specified in the grammar. For example, the method in `field` checks that the number of `|inner_track_widthi` substrings is consistent with the number of rows.

The test experiments presented in this paper only used `random_create()`. But the management of genotypes is designed to allow for future work beyond a random search. The generator provides a basic service for a more controlled exploration of the parameter space, via the `create_from_descriptor()` facility and the list of operators that can be extended.

Once a valid genotype has been produced, the method for generating a phenotype (a concrete test case) is the following:

- `export()`: the method attached to the root world class generates several files in the right format for the simulation: a `json` file for the mission, and `sdf` and `jpeg` files for the 3D world for Gazebo (see Fig. 3). The same method attached to child classes generates the concrete contents to put in the files. For example, the export of `Height_map` uses Perlin noise to generate a matrix of pixels, where the grey level of a pixel encodes the z coordinate of the corresponding point in the map. Crop plants and red stakes are picked in a library of predefined geometric objects (*meshes*). Crop row generates concrete (x, y) coordinates of the plants, based on their density, some noisy alignment, and

a disappearance probability (the vegetable did not grow, leaving a gap in the row). The mission parameters are used to determine the initial position of the robot relative to the field.

Note that the phenotype is strongly dependent on the simulator. In this respect, Gazebo may be less convenient than video game environments. For example, the simulator used in previous work on Mana was based on Blender, and could benefit from a rich API for the creation and manipulation of 3D scenes. The built-in possibilities were more limited in Gazebo.

5 Test Design: Oracle Procedure

For field tests like for the smoke tests performed at Naïo, verdicts are determined by testers who visually check the robot behavior. Such a manual solution is no longer tractable when exercising the navigation software with a large number of randomly-generated cases: we need an automated oracle. However, the specification of the oracle faced the difficulty that there was no specific list of requirements to check against. We first introduce the approach we used to overcome this difficulty, before presenting the specific oracle designed for Oz.

5.1 Guidelines for Specifying the Oracle Checks

We consider the specification of a property-based oracle (see Section 2.3). The oracle is based on a set of properties that the test traces should satisfy, any violation being reported as a failure. When there is no clear definition of failure, as in the case of Oz, the properties have to be specified from scratch. In order to guide the specification process, we may take inspiration from the fault history of Mana (Sotiropoulos et al. 2017). Our previous work analyzed the effects of Mana navigation bugs, and identified five broad categories of detectors to catch them. They correspond to diverse aspects of the behavior of an autonomous robot that may be worth consideration. These broad classes of requirements include:

1. **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 at each phase.
2. **Thresholds related to robot movement.** Here, the aim is to detect abnormal values of kinematic or kinetic variables.
3. **Critical events**, like collisions.
4. **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.
5. **Perception requirements**, focusing on unacceptable mismatches between the ground truth and its perception by the robot.

We used them to structure the discussion of candidate properties to check.

5.2 The Oracle for Oz

Accordingly, Table 1 shows the list of properties proposed for Oz. A mission is viewed as a sequence of weeding and U-turn phases. Wrong mission execution (e.g., the robot weeds a wrong row) yields a Fail verdict. But full mission completion is not required: the robot may decide to abandon the mission at any time. It then reports an error and must safely stops (P8).

Table 1 Oracle checks for Oz

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	$Velocity < V_{max}$
Critical 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

The most critical failures are causing damage to the environment (P5) and escaping from the crop field (P6). For two of the other properties (P2, P7), there was a discussion whether they pertain to performance or correctness. In particular, maintaining a reference distance to the vegetables corresponds to an ideal weeding trajectory, but Naö does not consider it as a strong requirement as long as there is no collision. Likewise, self-localization is not strongly required, since the robot is guided by its perception of the lines of vegetables and of the red stakes. Misperception of the lines and stakes would be more relevant for consideration, but it was judged too heavy to add the missing instrumentation and analysis facilities into the test platform.

Following the discussion, it was decided to implement the eight checks of Table 1, but to exclude P2 and P7 from the elaboration of the test verdict. These two checks could be useful for future experiments.

Table 2 displays the timestamped data collected by the test platform and made available for the implementation of the checks. Part of the data is logged at the robot interface: it includes the outputs (commands to actuators, error and success reports) plus some internal data made observable by the logging facilities of Ozcore (perceived position and yaw). Another part is logged by the simulator to trace the actual – rather than perceived – position and 3D orientation of the robot. Based on the available data, the implementation of P4 and P7 was straightforward. The other checks required more effort because they have to analyze the raw data in relation with the environment of the robot and the prescribed mission. For example, the raw position of the robot may be inside an area to weed, or inside an end-of-row U-turn area, or outside the perimeter of the field. The areas of interest are automatically pre-calculated for each concrete test case provided as an input. The check for P3 consists in verifying that the actual positions of the robot traverse the areas to weed in the right order. The P1 check focuses on the U-turn areas, and counts the number of changes of direction of the robot. The P2 check calculates distance to a relevant line of vegetables. Collisions (P5) are detected in relation with the position of the objects of the environment.

Table 2 Timestamped data logged during the tests

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 3 Proportion of test cases with n Fail verdicts over the five repeated runs

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

For simplification purposes, all checks abstract the robot by the movement of its center point.

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 low-fidelity simulation. Indeed, the simulation ignores the engine braking force and overestimates velocity on downward slopes arising from terrain irregularities. We decided to deactivate 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.

6 Test Results in Simulation

This section presents the results of the simulation-based tests: first a quantitative overview and then a detailed analysis of the fail cases.

6.1 Overview of the Results

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 account for the non-determinism of test executions. On average, a run takes about 3.5 minutes. Hence, the 400 runs of the test campaign take about 24 hours.

Due to non-determinism, we observed a high variability in the test verdicts. Table 3 gives the proportion of test cases yielding from $n = 0$ to 5 Fail verdicts over the five repeated runs. 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. In what follows, we thus provide the test results in terms of failing runs rather than in terms of failing test cases.

As shown in Table 4, 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 5 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 8 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 line of leeks represented by green

Table 4 Test verdicts for the 400 runs

	Fail	Pass
192		208

Table 5 Failure type counts

P1	P3	P5	P6	P8
0	70	142	57	14

circles. While performing the U-turn at the end of the line, 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 P2 (collision). Overall, collision is the most frequent failure type, observed in 35.5% of the test runs.

Smoke tests at Naö missed these failures: the software was found ready for field testing. In contrast, the randomly generated cases obtain a high failure rate of the software. They explore diverse missions in terms of rows to weed, uneven terrain, noise in the alignment of the plants, etc.

6.2 Qualitative Analysis of the Failure Cases

In order to prepare the comparison with the industrial field tests, LAAS researchers went through the 192 identified failing runs. For each property violation, information was extracted to characterize the context and propose a grouping into similar violation subcases. The obtained subcases are listed in Fig. 9.

From these subcases, it is obvious that the U-turn is the most prevalent context of failure. This is quite understandable as the U-turns are the most difficult parts of a mission, compared to line tracking. As much as 75% of the failing runs involve at least one property violation in relation with the U-turn. It suggests a major issue in the management of the corresponding maneuvers, which would be the main explanation for the overall high failure rate we obtain. During the U-turn, the robot may bump into a red stake (P5_2). Also, the maneuvers often leave the robot in a final position and orientation that are inadequate for entering the next row. As a result, the robot enters the wrong row (P3_1, P3_2, P3_3) and/or collides with the stake and crop plants upon entrance (P5_1). The run previously shown in Fig. 8 illustrates this type of misbehavior. When the U-turn takes place at the extremity of the field, the inadequate orientation occasionally yields an escape trajectory outside of the crop field (P6_1). An exemplary run is shown in Fig. 10a. In such runs, the robot always raises an error and stops, but after having crossed the specified field limits. The limits are at 1.5 meters after the red stakes and extremal rows on each side.

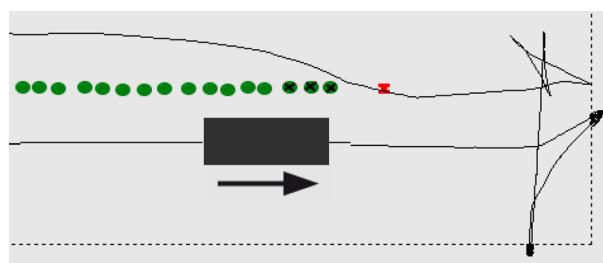


Fig. 8 Visualization of a run with multiple failure types. The robot (black rectangle) has the trajectory indicated by the solid line. The crop plants appear as green circles, and the red stake as a red cross. The perimeter of the crop field is materialized by a dotted line. In this run, two end points of the U-turn maneuvers are outside of the authorized perimeter, and the trajectory collides with the red stake and some crop plants

P3: Wrong sequence of rows (70 runs).
– P3_1: The robot re-enters the same row after the U-turn (55/70)
– P3_2: The robot re-enters a previous row after the U-turn (4/70)
– P3_3: The robot skips a row after the U-turn (9/70)
– P3_4: The robot leaves its row while weeding (2/70)
P5: Collisions with red stakes or crop plants (142 runs).
– P5_1: After the U-turn (91/142)
– P5_2: During the U-turn (18/142)
– P5_3: At the end of a row (11/142)
– P5_4: In the middle of a row (22/142)
P6: The robot exceeds the field limits (57 runs).
– P6_1: After the U-turn (6/57)
– P6_2: During the U-turn (49/57)
– P6_3: At the end of a row (2/57)
P8: The robot does not stop straight after raising an error (14 runs).
– P8_1: After a <i>blind course</i> recoverable error (7/14)
– P8_2: After a <i>steersman</i> recoverable error (1/14)
– P8_3: After an <i>invalid markers spacing</i> fatal error (6/14)

Fig. 9 Property violation subcases

Again in relation with the U-turn, a different subcase of P6 violation (P6_2) is illustrated by Fig. 10b. The U-turn may be successful (it may align the robot to the right row) but takes more space than allowed. The end point of a maneuver slightly exceeds the limits of the field, yielding a temporary out-of-field position. Each time, the oracle check reports one or two centimeters over the limit. But the measurement is for the central point of the robot, hence the overshooting is actually in the tens of centimeters.

The remaining property violations are not directly related to the U-turn. An example is stopping too late after issuing an error (P8_1, P8_2, P8_3). There are three possible errors raised during testing, all of which would necessitate a human intervention in real-world operation. They consist in recoverable errors, allowing the operator to resume the mission after the stop, and fatal errors requiring a reboot of the robot. The *blind course* recoverable error is raised when the robot has traveled about 3 meters without perceiving the line of crop plants it is supposed to follow. The *steersman* error is raised when the robot considers its yaw angle as suspicious in line tracking mode: the current path direction differs too much from the one in the past 10 meters (the angle is greater than 15 degrees). The fatal *Invalid*

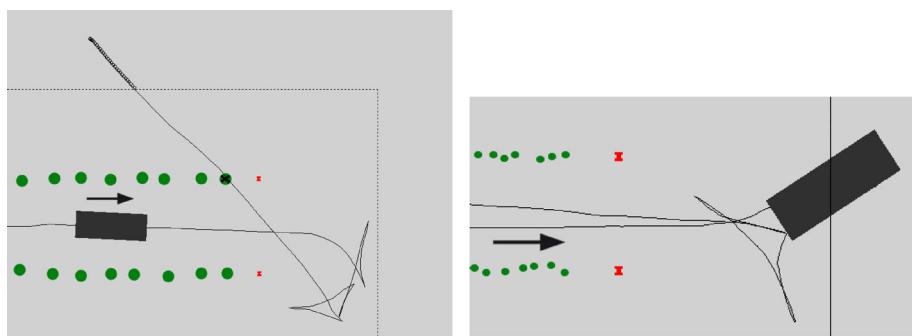


Fig. 10 Out-of-field failures in relation with the U-turn

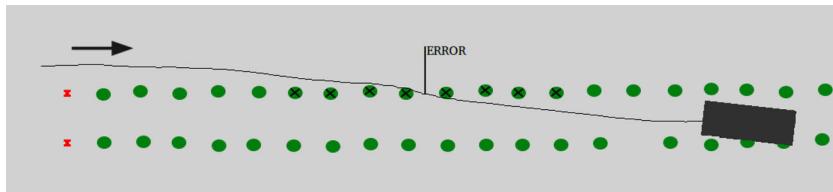


Fig. 11 The robot leaves its row while weeding

markers spacing error is raised when the robot detects a missing or misplaced red stake. In the generated test cases, the stakes are always at the right location. The fatal error is raised when the robot weeds a wrong row at the extremity of the field and does not see the expected number of stakes at the end of this row. As can be seen from the subcases P8_1, P8_2, P8_3 in Fig. 9, the late stop is not tied to a specific error type: it occurs for the three errors. Moreover, the failure is not systematic: in the complete set of runs, there are also many correct stops after each type of error (respectively 46, 3 and 26 correct stops). It suggests a timing/concurrency issue, which nondeterministically affects the generic handling of errors.

Finally, collisions and out-of-field failures may happen outside the context of U-turns. As regards collisions (P5_3, P5_4), the robot gets too close to the line of crop plants it follows in a weeding phase. In two of the failing runs, the robot even traverses the line, yielding the two P3_4 violations we observed. One of these runs is shown in Fig. 11. Interestingly, robot raises a blind course error around the time of the collision. Such is also the case in about 25% of the mid-row collisions. This suggests that part of these collisions could be due to a row perception loss while weeding. In the run of Fig. 11, a P8_1 violation adds up so that the robot pursues its blind course in the neighboring row instead of immediately stopping. As regards out-of-field failures, subcase P6_3 at the end of a row also suggests a misperception problem. The two failing runs (see Fig. 12) respectively raise a blind course and steersman error more than 1.5 meters after the red stakes have been passed. It indicates that the robot has not correctly perceived the end-of-row situation: it is still trying to perform line tracking long after it exited the row.

In summary, the qualitative analysis of the failing runs gives the following hypothesized issues:

- A major U-turn issue, affecting the ability to safely perform a mission when there is more than one row to weed.
- A concurrency/timing issue delaying the stop of the robot in cases of errors.
- A line tracking issue, possibly in relation with a row perception loss.
- An end-of-row detection issue occurring in rare cases.

7 Industrial Feedback

The final phase involved joint meetings and a week visit of the first author to Naö. It gave him the opportunity to meet key persons in charge of the testing or the development of Ozcore, to consult the notes reporting from the field tests, and to discuss the various examples of test fails in detail, with the help of a replay tool developed by Naö for diagnosis purposes.

In parallel to the simulation-based tests, the field tests also found issues in the Ozcore software. The reports of five test sessions were shared for the study. Each session requires

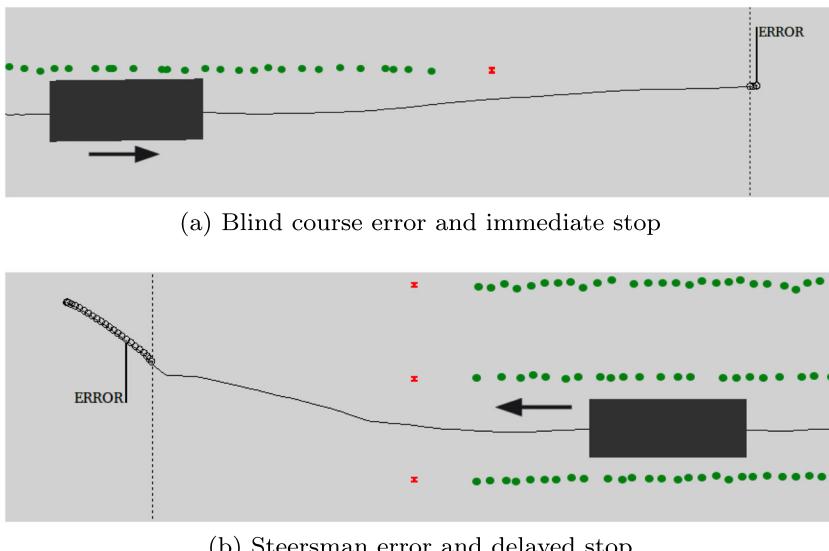


Fig. 12 Two runs for which the robot does not perceive the exit of the row

some time to load the robot into the van, travel to the experimental site, unload the robot, initialize the experiments and do the same in reverse for way back. The effective test time is typically one or two hours for a half-day session. Table 6 lists the navigation failures observed during these tests, from the notes taken by the test operators. The test control and oracle procedures are manual. For example, the operator enters a weeding mission via the user interface. At some point, she visually determines that a row entrance trajectory is inadequate and stops the robot before a collision occurs.

The comparison with the simulation-based tests started about four months after the fifth test session. Naïo had thus hindsight on the diagnosis of software issues and their resolution. This was useful to determine whether the failures observed in simulation revealed the same issues or different ones.

Due to the large number of fails in simulation, it was not possible to have a joint review of each of them. Rather, the comparison relied on the preparatory analysis performed at LAAS with a deeper examination of a few exemplary test runs.

Table 6 Navigation failures observed during the field tests

Failure description	# occurrences
F1 - Bad row entrance trajectory after the U-turn	11
F2 - Bad row exit trajectory before the U-turn	2
F3 - The robot skips a row after the U-turn	2
F4 - The robot loses direction during line tracking	2
F5 - Deviated trajectory when there is a large gap in the line	2
F6 - The robot unexpectedly stops upon red stake detection	2
F7 - Large detour at the beginning of a row, leaving an unweeded area	1
F8 - The robot stops out of the field limits at the end of the mission	1

The final comparison results are given in Table 7. They include real, spurious and undiagnosed issues. We discuss each of them in turn.

Issue I1 Like the simulation-based tests, the field tests revealed a major U-turn issue causing the majority of failures. There is a good match between the failures observed by each type of test: the robot enters the wrong row (F3 vs. P3_1, P3_2, P3_3), or has an inadequate positioning and angle to enter its row (F1 vs. P5_1, P6_1). From Naïo's analysis, the U-turn issue also causes bad row exit trajectories (F2). Indeed, when the robot perceives the red stakes, it starts preparing the U-turn and the trajectory for approach may collide with the end of the line. It is thus plausible that part of the end-of-line collisions observed in simulation (P5_3) are also due to the U-turn and not only to perception problems. This could however not be specifically demonstrated. The whole U-turn issue was considered as critical by Naïo and a lot of effort has been spent to solve it. Being unable to spot a specific fault, they re-developed the functionality from scratch.

Issue I2. The exceeding space taken by the U-turn maneuvers in simulation (P6_2) was not found by the field tests. The operator did not check the precise amplitude of the maneuvers, hence an overshooting could have happened but remained unnoticed. After analysis, the issue was confirmed by Naïo. The planning of the maneuvers does not take a sufficient margin to avoid exceeding the 1.5m limit in all cases. Requiring shorter maneuvers would make the U-turn even more difficult. Rather, Naïo has revised the conditions of use of the robot to provision more space for the U-turn.

Issue I3. Failures F4 and F5 reveal an issue in the handling of line perception losses. In the case of F4, the loss was due to a sensing alea, while in the case of F5 there was a real gap in the line (many consecutive plants did not grow). When the row is no longer seen, the robot heuristically considers a likely direction based on its past behavior. The aim is to keep the robot on track until it perceives the row again or it raises a blind course error. But the heuristics may be wrong, sending the robot to the crop plants. This diagnosis is consistent with LAAS' observation of mid-row collisions associated with blind course errors. To further confirm the issue in simulation, an exemplary run with a mid-row collision but no blind course error was analysed, using a replay tool developed at Naïo. The chosen run corresponds to an uneven terrain, which is challenging for the LIDAR-based perception: the laser beams have erratic inclinations as the robot goes upwards and downwards. The replay of the run confirmed a transient perception loss of the neighboring crop plants, upon

Table 7 Comparison of issues revealed by field tests and simulation-based ones: '✓' = revealed, '–' = not revealed, '?' = no conclusion for lack of diagnosis

Issues	Field tests	Simul. tests	Real / Spurious
I1 - U-turn functionality	✓	✓	Real
I2 - Space margin for U-turn	–	✓	Real
I3 - Heuristics for transient perception losses	✓	✓	Real
I4 - Processing of red stake images	✓	–	Real
I5 - Alignment at the beginning of a row	✓	(✓) (with P2')	Real
I6 - Skidding/odometry	✓	–	Real
I7 - End-of-row detection	?	?	?
I8 - Simulated stop upon error	–	✓	Spurious
I9 - Simulated velocity on downward slopes	–	✓	Spurious

which the direction of the trajectory changes. The conclusion is that the heuristic issue is found by the simulation-based tests. However, these tests only reproduced scenarios with sensing aleas in uneven terrains, not ones with a large gap in a row. The generated rows had gaps of 2 or 3 missing plants, which was not stressful enough. In contrast, the virtual terrain irregularities were more stressful than in the real world, inducing many sensing aleas. Naïo currently studies alternative heuristics to improve the tolerance of transient perception losses, but none of them is integrated into the Oz platform yet. Regarding large gaps in a row, farmers are advised to put substitute elements (e.g., stakes) to repair the line.

Issue 14. Failure F6 is caused by the processing of images for red stake detection. Artifacts in the images may induce the double vision of a single stake, which leads the robot to raise a spurious invalid markers spacing error. Naïo has reworked the image processing code to fix the issue. In simulation, the test oracle does not judge the relevance of errors: it only checks the safe stop upon errors. LAAS researchers manually inspected all runs ending upon an invalid markers spacing error to determine whether false positives occurred. But such was not the case: in all these runs, the robot had entered a wrong row after a U-turn, and correctly detected a wrong number of red stakes at the end of the row. We conclude that the double vision is not reproduced in simulation. Indeed, the simulated images are too clear and crisp compared to the real ones. The red stakes stand out perfectly against the unicolor ground and the sky, while the real-world vision may suffer from misleading effects.

Issue 15. Failure F7 exposes a difficulty in the initial alignment of the weeding trajectory with the crop row. The robot may take time to rejoin the correct alignment and the beginning of a row is not adequately weeded. The oracle used in simulation does not detect this kind of misbehavior. Remember that, in the initial design of the study, we disabled the P2 check on the reference distance to the vegetables. In order to determine whether the issue was missed by the oracle but in fact reproduced, LAAS researchers introduced a modified version of P2. The detection threshold was changed to focus on large distances and the variation rate was also monitored. With this check P2', the initial alignment issue is indeed revealed in simulation. Figure 13a is a virtual scenario close to the one observed in the field: the robot initially deviates away from the line and takes time to change its course, leaving the first 2–3 meters unweeded. The virtual tests also found new misalignment scenarios. An interesting one concerns two-passes weeding (Fig. 13b): the robot occasionally starts to weed the wrong side during 3 meters before rejoining the correct side. There was also a case with oscillations over the first 4 meters before the trajectory becomes stable. Note that the length of a row is typically greater than 100 meters in real fields. Naïo considers the occasional misalignment in the first few meters as a minor issue and did not investigate a fix.

Issue 16. Failure F8 corresponds to an out-of-field normal stop: the robot is not in error, it stops at the end of its mission but more than 1.5 meters after the red stakes. From Naïo's analysis, the failure was due to excessive skidding and erroneous odometry. A fix has been introduced to better handle skidding. In simulation, the interaction of the wheels with the ground is not accurate with respect to skidding, slippage or sliding. In any case, the specific failure F8 was not obtained.

Issue 17. In simulation, two out-of-field stops seemed to be due to a misperception of the end of the row (P6.3 and Fig. 12). This was never observed during the five field test sessions we analyzed. However, since that time, Naïo has observed rare cases where the robot does not detect the end of a row. The issue being undiagnosed so far, it is not possible to conclude on the agreement of the tests.

Issues 18 and 19. The last entries in Table 7 are issues causing spurious failures in simulation only. The delayed stop upon an error (P8 violations) was never observed in the field. After analysis, the failure is due to an issue in the simulation scripts, not in the software

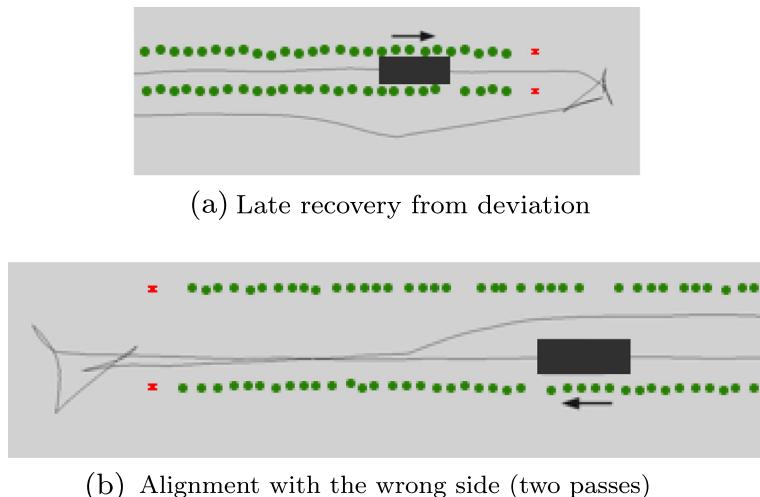


Fig. 13 Alignment issue at the beginning of a row in simulation

under test. For the sake of completeness, Table 7 also mentions the spurious overspeed we observed on downward slopes. It was an artifact of the low-fidelity physics and yielded the disabling of P4 checks.

8 Overview of Outcomes and Threats to Validity

We are now able to answer the question explored by the study, and identify lessons learned. The study could be affected by threats to validity, which we also discuss.

8.1 Answers to the Research Question

RQ: What are the benefits and drawbacks of simulation-based testing with respect to:

- *Real issues found by the field tests,*
- *Real issues not found by the field tests,*
- *Spurious issues coming as simulation artifacts?*

The study confirms that the fault revealing power of simulation-based testing is under-exploited. It finds real issues that are currently caught by field testing. The major U-Turn issue is a striking example. It was missed by the smoke tests and caused the majority of failures during field testing. Still, the issue was not hard to reveal by the random tests, yielding a high failure rate in simulation. The simulation also managed to reproduce improper reactions to transient perception losses, and difficulties in the initial alignment to a row.

The issue requiring accurate reproduction of skidding seems hard to find with a simplified physics. The image processing issue was also missed in simulation, but it would be possible to add disturbance effects to the simulated images (see e.g., related work by Zendel et al. (2013, 2017)).

When revealing issues, the simulation-based tests provided diverse examples of misbehavior caused by them. This is useful since the issues found by the field tests are not

low-level bugs. They question the adequacy of the core algorithms and heuristics underlying the perception, decision and motion control abilities of the robot. There may be no obvious fix or improvement. Then, it is helpful to quickly obtain an overview of the various undesirable effects of the chosen algorithms. For example, the simulation gave U-turn failures that were not observed during the five field test sessions, like coming back to a previously weeded row or having collisions with red stakes during the maneuvers. Similarly, the difficulty in initial row alignment did not only surface as a large detour, but also as an oscillation over the first meters or an initial alignment to the wrong side of the two-passes route. In simulation, those diverse failures can be obtained at once, by running a bunch of tests. In the field, the tests extend over a longer period of time, involving several sessions on different days.

The simulation-based tests even uncovered an issue missed by the field tests: the insufficient space margin for the U-turn. This was not a necessary objective, as the tests were not designed to explore corner cases difficult to reach in the field. They involved mere random sampling over a permissible domain of use of the robot. It is impossible to say whether the space margin violation did not occur at all in the field, or occurred but was not seen by the test operator. In any case, it occurred in simulation and was detected by the oracle checks.

On the downside, the simulation can introduce spurious issues. In the study, they were of two types. First, the low fidelity simulation induced a behavior that is impossible in the real world, the transient overspeed on downward slopes. Second, the simulation code was faulty, non-deterministically delaying the effective stop of the robot.

The need to develop the simulator must be added as a drawback. The Oz simulator is rather complex in its own right. Developments on top of Gazebo are technically nontrivial. Indeed, it is not surprising that the simulation code can have issues. Beyond the initial development effort, maintenance also proved a concern due to the dependency on an external and unstable technology. As an anecdote, just before the visit of the LAAS researcher to Naïo, an update of Gazebo broke the simulator, which was no longer working with the new version.

8.2 Lessons Learned

Based on our experience, some insights and recommendations can be derived for the development of automated simulation-based testing.

1. **Design test generator and oracle for evolvability.** Specifying the virtual environments and oracle checks is hard, and the test design will most probably have shortcomings. It should be continuously improved as more experience is gained on the system or on similar ones. For Oz, the feedback from the field tests provided insights on relevant improvements. The P2 oracle check has been turned into P2' based on the detection condition of a misalignment problem. Note that P2' takes inspiration from, but is more general than, the specific problem detected in the field. The generation model needs improvements as well, if one wants to consider large gaps in a row and visual hazards affecting images. The current generation may also be a bit too stressful with respect to the terrain irregularities, compared to real fields. To support continuous improvements, the test generator and test oracle should be designed for evolvability from the outset.
2. **Specify a well-structured world model.** Test generation has to be based on a well-structured world model, which facilitates the addition, removal or modification of elements. In the study, we had an UML-based structure model. The implementation kept the same structure with a uniform interface for each element. From the experience

with Oz, the modeling approach also has to accommodate constraints on the parameters attached to world element. In the study, the structural model was supplemented by an attribute grammar specifying the valid configurations of parameter values. Constraints relating several parameters were introduced at the level of the closest ancestor element in the structure. The modeling can be kept at a high level of abstraction by using world content generation procedures to bridge the gap between the abstract test cases and the concrete test inputs.

3. **Clearly separate data logging (on-line) and analysis (off-line).** The test oracle is property-based. It is composed of a set of checks to detect misbehavior. If the oracle specification starts from scratch, the five aspects of behavior we identified can serve as a guide to explore candidate properties to check. In order to facilitate the evolution of the oracle, we recommend an off-line analysis of the test traces. There should be a clear separation between raw data recording, which is done during the simulation runs, and the potentially complex data analysis to be done afterwards. In this way, checks can be deactivated, revised or added without changing the simulation code. Also, alternative versions of the oracle (e.g., with P2' instead of P2) can be studied without having to re-execute the tests. The raw data recording should be as complete as possible, beyond the minimal dataset required by the current checks. It should ideally consider as much data as possible to cover future needs. The instrumentations should be placed on both the software under test and the simulator. The aim is to capture both the “subjective” point of view of the robot (e.g., its perceived positions, the commands it sends, the errors it raises) and the objective situation of the simulation (e.g., real positions, collision events). The complexity of some checks comes with the need for situation-awareness, when the raw data must be analyzed in relation with the robot’s environment and prescribed mission.
4. **Manage simulator fidelity and complexity.** The feedback on the simulation platform used by the study is mixed. On the one hand, the Gazebo-based simulation has demonstrated its effectiveness with respect to real issues, including a major one. It thus holds out the prospect of alleviating the costly field tests. On the other hand, the development and maintenance of the simulator proved heavy for a small company like Naïo. If many issues do not require an accurate reproduction of real-world conditions, one may wonder whether a lighter and easier-to-maintain simulation platform would not perhaps be sufficient. Lighter simulation would also be faster, allowing for a higher number of tests in shorter time.

8.3 Threats to Validity

The case study reported in this paper is exploratory, based on a qualitative analysis (the types of failures, the issues causing them).

Construct validity concerns the correct identification of which issues are found by each type of tests.

As regards issues found by the field tests, a risk comes from the fact that the software version was not exactly the same from one test session to the other, due to patches in the code. The comparison with the simulation-based tests is thus based on the hindsight that Naïo has on the diagnosis of issues. For example, the first session reported failures that are retrospectively assigned to the major U-turn issue and to the initial alignment issue. The other issues surfaced later but it is judged that they were there from the beginning. Regarding the major U-turn issue, the same failures persisted in all test sessions despite tentative fixes, until the redevelopment was decided.

As regards the issues from the simulation-based tests, it was not possible to review all the failing runs with Naïo. While there is a good confidence that part of the fails are caused by the issues in Table 7, some other part could have unknown causes. If these were spurious, it would decrease the benefit from the simulation-based tests. We cannot exclude unknown issues, but took care that all failures cases are consistent with the effects of the issues identified by the study, as far as they are understood.

The risk for having a different understanding of the issues was mitigated by the interactions established during the study. The visit of the LAAS researcher at Naïo allowed for in-depth discussions of the issues. It was followed by a peer debriefing with other researchers. Then, there were several subsequent interactions with Naïo on pending or unclear points. And finally the list of issues was jointly reviewed and re-discussed with Naïo.

Measures for *internal validity* concerned the avoidance of bias in the design of the simulation-based tests, in particular bias due to some prior knowledge of issues. At the time of the design, issues I1 to I9 were unknown to LAAS and Naïo. LAAS researchers were also unaware of the results of field tests when they analyzed the failing cases to extract a set of hypothesized issues.

External validity concerns the generalization of outcomes beyond the case of Oz. The context of the study is clearly a robotic software developed by a small company. The simulation technologies used by Naïo are representative of the context: the Gazebo platform and ROS middleware are widespread in robotics. The difficulties experienced during the study (specification challenges, unstable simulation technology, spurious failures due to the simplified physics or issues in the simulation code) are not specific to the Oz example. The outcomes regarding test effectiveness may be more application-dependent. However, we consider it useful to study effectiveness for a real-scale industrial example, which supplements other empirical results on academic examples, e.g., Sotiropoulos et al. (2017) and Timperley et al. (2018). Finally, the approaches adopted for generating tests and detecting misbehavior are general, their principle can be reused outside the case study.

9 Conclusion

In the face of the rising complexity of autonomous systems development, testing becomes one of the biggest challenges. This article addresses mission-level validation. We studied the benefits and drawbacks of simulation-based tests compared to field tests, and also provided some practical recommendations for the deployment of the simulation-based tests. The Oz robot, developed by the French company Naïo, served as a real-scale example to support investigation. The work was carried out both in the lab and in the company, with interactions to design the case study and interpret the results.

The generation of the test cases (including 3D crop fields and weeding missions) was based on a structural model in UML and an attribute grammar to express constraints. The valid words of the grammar represent the valid configurations of model parameters (the genotype), from which the concrete contents of test cases are produced (the phenotype). The test oracle was designed using five general classes of properties and implemented to automatically analyze test traces. The non-determinism of test results was anticipated, which yielded several runs of the same test cases. The main outcome of the study was to show that major software issues could be revealed in simulation rather than in the field; the randomly-generated tests even discovered a new issue that was not identified during the field tests. On

the negative side, some spurious failures and simulation maintenance problems must be put in the balance.

An open question is the degree of fidelity of the simulator. It is expected to have a high impact on the cost-effectiveness of the tests. For instance, a high-fidelity may be required to catch issues related to complex image processing or physical interaction. But computational time and resources may grow prohibitively (as experienced with the Oz simulator), and complex simulation code may be buggy and difficult to maintain. Conversely, a lighter simulator may leave much more issues uncaught before field testing (and induce spurious failures), but at the same time requires less resources, accelerates simulation and is easier to maintain. The question of the simulation fidelity has no simple answer, and it is interesting to mention how the strategy of Naïo has evolved since the starting point of the study. At that time, their effort was on making the simulation as realistic as possible given the resource constraints. But eventually, they have decided to abandon the Gazebo-based simulator for a much lighter platform, developed in-house. The simulation-based tests are also more systematic than at the time of the study, and now involve a set of diverse cases.

From an academic perspective, current work at LAAS elaborates on the generation of test cases. It retains the principles of the custom generator used for Oz (well-structured genotypes as first-class citizens, checking and manipulation of elements of the genotype, generation mixing fixed and free elements) in order to develop a more generic and reusable generation framework. There is also a plan to extend the framework to integrate more elements such as mobile obstacles or noise on sensors. The framework will be used to explore test selection strategies, which were not addressed by the study (only based on random testing). Continuing collaboration with Naïo will provide access to the lightweight simulator for experimentation purposes. The faster simulation is expected to make the test selection problem more amenable to advanced search-based techniques guided by fitness functions.

Acknowledgements 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 during the study: Gaëtan Séverac, Pascal Schmidt, Marc Jambert.

References

- Andrews AA, Abdelgawad M, Gario A (2016) World model for testing urban search and rescue (USAR) robots using petri nets. In: Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016), Rome, Italy, pp 663–670
- Arnold J, Alexander R (2013) Testing autonomous robot control software using procedural content generation. In: Computer Safety, Reliability, and Security (SAFECOMP 2013), Toulouse, France, pp 33–44
- Bach J, Langner J, Otten S, Sax E, Holzapfel M (2017) Test scenario selection for system-level verification and validation of geolocation-dependent automotive control systems. In: 2017 International conference on engineering, technology and innovation (ICE/ITMC 2017), Madeira Island, Portugal, pp 203–210
- Ben Abdessalem R, Nejati S, Briand LC, Stifter T (2016) Testing advanced driver assistance systems using multi-objective search and neural networks. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, Singapore, pp 63–74
- Ben Abdessalem R, Nejati S, Briand LC, Stifter T (2018) Testing vision-based control systems using learnable evolutionary algorithms. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE 2018), Gothenburg, Sweden, pp 1016–1026
- Chen TY (2015) Metamorphic testing: a simple method for alleviating the test oracle problem. In: Proceedings of the 10th International Workshop on Automation of Software Test, Florence, Italy, pp 53–54

- Echeverria G, Lassabe N, Degroote A, Lemaignan S (2011) Modular open robots simulation engine: Morse. In: IEEE international conference on robotics and automation (ICRA 2011), Shanghai, China
- Geyer S, Kienle M, Franz B, Winner H, Bengler K, Baltzer M, Flemisch F, Kauer M, Weißgerber T, Bruder R, Hakuli S, Meier S (2014) Concept and development of a unified ontology for generating test and use-case catalogues for assisted and automated vehicle guidance. *IET Intelligent Transport Systems* 8(3):183–189
- Hallerbach S, Xia Y, Eberle U, Koester F (2018) Simulation-based identification of critical scenarios for cooperative and automated vehicles. Tech. rep., SAE International in United States. <https://doi.org/10.4271/2018-01-1066>
- Klueck F, Li Y, Nica M, Tao J, Wotawa F (2018) Using ontologies for test suites generation for automated and autonomous driving functions. In: 2018 IEEE international symposium on software reliability engineering workshops (ISSREW 2018), Memphis, TN, USA, vol 00, pp 118–123
- Koenig N, Howard A (2004) 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, vol 3, pp 2149–2154
- Lamprecht A, Ganslmeier T (2010) Simulation process for vehicle applications depending on alternative driving routes between real-world locations. *Advanced Microsystems for Automotive Applications 2010*, pp 377–386
- Lindvall M, Porter A, Magnusson G, Schulze C (2017) Metamorphic model-based testing of autonomous systems. In: 2nd IEEE/ACM International Workshop on Metamorphic Testing (ICSE 2017), Buenos Aires, Argentina, pp 35–41
- Menghi C, Nejati S, Gaaloul K, Briand LC (2019) Generating automated and online test oracles for simulink models with continuous and uncertain behaviors. In: 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, pp 27–38
- Micskei Z, Szatmári Z, Oláh J, Majzik I (2012) A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In: Agent and Multi-Agent Systems. Technologies and Applications, Dubrovnik, Croatia, pp 504–513
- Nentwig M, Stamminger M (2010) A method for the reproduction of vehicle test drives for the simulation based evaluation of image processing algorithms. In: 13th International IEEE Conference on Intelligent Transportation Systems, Madeira Island, Portugal, pp 1307–1312
- Nguyen CD, Perini A, Tonella P, Miles S, Harman M, Luck M (2009) Evolutionary testing of autonomous software agents. In: Proceedings of The 8th international conference on autonomous agents and multiagent systems (AAMAS 2009), Budapest, Hungary, vol 1, pp 521–528
- Okdal Sydac (2018) <https://www.oktalsydac.com/>, Accessed 2019-09-19
- PreScan Simulation platform for ADAS (2018) <https://tass.plm.automation.siemens.com/prescan>, Accessed 2019-09-19
- Sotiropoulos T, Guiochet J, Ingrand F, Weaselynck H (2016) Virtual worlds for testing robot navigation: a study on the difficulty level. In: IEEE 12th European on Dependable Computing Conference (EDCC 2016), Iasi, Romania, pp 153–160
- Sotiropoulos T, Waeselynck H, Guiochet J, Ingrand F (2017) 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, pp 150–159
- Tian Y, Pei K, Jana S, Ray B (2018) Deepertest: Automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of the 40th International Conference on Software Engineering (ICSE 2018), Gothenburg, Sweden, pp 303–314
- Timperley CS, Afzal A, Katz DS, Hernandez JM, Goues CL (2018) Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST 2018), Västerås, Sweden, pp 331–342
- Togelius J, Yannakakis GN, Stanley KO, Browne C (2011) Search-based procedural content generation: a taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3):172–186
- Tractica (2016) Agricultural robots – executive summary. Research Report. <https://www.tractica.com/research/agricultural-robots/>, Accessed 2019-09-19
- Ulbrich S, Menzel T, Reschka A, Schuldt F, Maurer M (2015) Defining and substantiating the terms scene, situation, and scenario for automated driving. In: 2015 IEEE 18th International Conference on Intelligent Transportation Systems, Gran Canaria, Spain, pp 982–988
- Unreal Game Engine (2018) <https://www.unrealengine.com/>, Accessed 2019-09-19
- Virtual Test Drive (2018) <http://www.mcssoftware.com/product/virtual-test-drive>, Accessed 2019-09-19
- Zendel O, Herzner W, Murschitz M (2013) Vitro - model based vision testing for robustness. In: IEEE International Symposium on Robotics (ISR 2013), Seoul, Korea, pp 1–6

- Zendel O, Murschitz M, Humenberger M, Herzner W (2017) How good is my test data? introducing safety analysis for computer vision. *Int J Comput Vis* 125(1):95–109
- Zhang M, Zhang Y, Zhang L, Liu C, Khurshid S (2018) Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, pp 132–142
- Zofka MR, Kuhnt F, Kohlhaas R, Rist C, Schamm T, Zöllner JM (2015) Data-driven simulation and parametrization of traffic scenarios for the development of advanced driver assistance systems. In: 2015 18th International Conference on Information Fusion (Fusion 2015), Washington, DC, USA, pp 1422–1428
- Zou X, Alexander R, McDermid J (2014) Safety validation of sense and avoid algorithms using simulation and evolutionary search. In: Computer Safety, Reliability, and Security, (SAFECOMP 2014), Florence, Italy, pp 33–48

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Clément Robert received an Engineer degree in automation and industrial computing in 2017 at ENSEEIHT. He is currently doing his PhD at LAAS-CNRS in the dependable computing and fault tolerance group. His subject is to evaluate the potential of simulation for the testing of autonomous robot. This topic involves software testing, dependability, autonomous robots and simulation.



Thierry Sotiropoulos received a master degree in Artificial Intelligence and Robotics in 2014 at University of Toulouse III, France, and a PhD in computer science in the dependable computing and fault tolerance group at LAAS-CNRS, France, in 2018. His doctorate subject aims to ensure dependability of autonomous systems by testing the software in simulation. He is currently working as a post-doctoral fellow at ICAM Toulouse to design and develop intelligent, connected and social helping objects.



Hélène Waevelynck received the Engineer degree from the National Institute of Applied Sciences of Toulouse, France, in 1989, and the Ph.D. degree in computer science from the National Polytechnic Institute of Toulouse, in 1993.

She is a Senior Research Scientist (Directrice de Recherche) with the LAAS-CNRS, University of Toulouse, Toulouse. Her current research interests include software testing and verification, with an emphasis on dependable computing systems.



Jérémie Guiochet received the Ph.D. degree in model-based safety analysis of service robotics from INSA Toulouse, Toulouse, France, in 2003.

He is an Assistant Professor of computer science with the University of Toulouse III, Toulouse and with the Dependable Computing and Fault Tolerance Group at LAAS-CNRS, France. His current research interests include safety assessment, fault removal, and fault-tolerance in safety-critical autonomous systems.



Simon Vernhes received an Engineer degree in 2011 from INSA Toulouse, France. He received his PhD in Computer Science from ISAE Toulouse and ONERA in 2014, where he worked on automated planning and scheduling. Since 2016, he is currently working at Naïo Technologies as an R&D engineer on the guidance software of weeding robots for vegetable farms and vineyards.