Integration of Test Generation Into Simulation-Based Platforms: An Experience Report

Luca Vittorio Sartori*
Jérémie Guiochet*
Hélène Waeselynck*
Aizar Antonio Berlanga Galvan
{firstName.surname}@laas.fr
University of Toulouse, LAAS-CNRS
Toulouse, France

Simon Hébert-Vernhes
Naïo Technologies
Toulouse, France

Magnus Albert
SICK AG
Waldkirch, Germany

ABSTRACT

Field-testing is costly and time-consuming, hence, simulation-based testing is becoming more and more important to validate autonomous systems. Since autonomous systems can be deployed in diverse environments, a significant amount of diversified test cases has to be created. TAF (Testing Automation Framework) is a test generation tool we developed to serve this purpose. It produces the test cases from a data model that specifies the virtual environments of interest. This paper presents a practitioner's view of the integration of TAF into simulation-based test platforms, through two industrial case studies. The first one is for testing an agricultural robot developed by Naïo Technologies, and the second one for a static perception system by SICK AG that surveils a road crossing to support connected vehicles with tracking data in complex urban scenarios. We report on our experience in the design of the data models, as well as in the automation of the execution, logging, and analysis of the generated tests. We conclude with lessons learned.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

software testing, automation, software engineering, testing framework, industrial case study, autonomous systems, test case generation, simulation, software-in-the-loop (SIL) simulation, autonomous robot, agricultural robot, test oracle, dynamic agents

ACM Reference Format:
Luca Vittorio Sartori, Jérémie Guiochet, Hélène Waeselynck, Aizar Antonio Berlanga Galvan, Simon Hébert-Vernhes, and Magnus Albert. 2022. Integration of Test Generation Into Simulation-Based Platforms: An Experience Report. In *IEEE/ACM 3rd International Conference on Automation of Software Test (AST '22), May 17–18, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3524481.3527236

AST '22, May 17–18, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9286-0/22/05…$15.00
https://doi.org/10.1145/3524481.3527236

1 INTRODUCTION

Complex autonomous systems are now deployed in real applications, but trusting them is still an unsolved issue. Such systems would face an enormous number of scenarios, and their validation using testing is a hot topic. Test in the field is resource-consuming in terms of time, money, and can even lead to hazardous situations for the users, the environment, or the system itself. For these reasons, test in simulation is becoming popular in robotics and automotive. It also allows testing the system in many more scenarios than the ones experienced during tests in the field. New simulators with realistic animations and images (e.g., from the video game industry, like Unity), or with more realistic simulation of physics (e.g., gravity, friction, etc. in the robotics domain, like Gazebo), or with specific behavior already integrated (e.g., in the automotive domain, like CARLA), are a promising technology: they allow for the deployment of more and more tests in simulation without spending resources. We particularly focus on software-in-the-loop (SIL) simulation, where the real software is running in a simulated environment.

However, testing in simulation raises classic concerns in system testing, like how to generate and select test cases. Manually developing test cases is indeed a tedious and non-efficient approach, and in many software projects it has been replaced by automatic generation, selection, execution, and analysis of tests. This is particularly efficient in the context of continuous integration. However, in the context of SIL testing of autonomous system, current approaches for test generation are not appropriate (mainly due to the absence of formal specification of the inputs, and their wide diversity).

The objective of this study is to assess what are the benefits and limits of the integration of automation, and particularly test case generation, into current test architectures for autonomous systems. We explore this issue by integrating an automatic test case generation in two experiments with industrial applications. We chose to integrate the tool TAF (Testing Automation Framework [15], which was originally designed for the generation of 3D worlds and missions for testing autonomous robots. This tool allows the user to enter a test data model (an abstract model of what could be a test case), the constraints between the data, and it produces test cases (files or scripts) that could be directly used by the simulators. The two case studies integrate TAF, but are different because the first one, the simulation of an agricultural robot in a static environment, includes a system under test and its simulation, whereas the second one does not simulate a system, but is limited to the generation of tests scenarios with dynamic agents (pedestrians and vehicles).

For each case study, we gather all benefits and limits, in order to produce a list of lessons learned that might be valuable for any team interested in developing or integrating automatic test generation for test in simulation.

After outlining in Section II the body of literature relevant to this work, we present TAF in Section III. Section IV describes the first case study, about an agricultural robot, followed by the second case study, the perception system, in Section V. The lessons learned are summarized in Section VI. We conclude in Section VII.

2 RELATED WORK

This section presents first the work related to testing in simulation, and then focuses on the work regarding the generation of test cases.

Testing in simulation is an approach for software testing that can be used to validate software in virtualized conditions. There are various types of simulation that can be used, based on the test objective, like Model-in-the-Loop (MIL) simulation, in which a model of the system is used, or Software-in-the-Loop (SIL), in which the real software of the system is running. For SIL, depending on the domain, different simulators with a varying level of physical fidelity were deployed: Gazebo [8] for robotics, Unity [4] for video games, CARLA [7] for automotive, etc. Generating test cases and scenarios to run in these simulators requires time, with many cases generated manually. CARLA, through its scenario_runner, offers support for ASAM OpenSCENARIO [1] for the description of the evolution of scenarios for automotive simulators, but this approach offers a restricted constraints' management. More in general, for test generation, there are very few published works in the context of test in SIL simulation, but there are many works about automatic test case generation in the testing community, with a survey [5] available on the most prominent techniques.

For the test case generation, in summary [13], there are three main categories of approaches for test case generation: 1) based on *generate-then-filter*, which consists in producing a candidate test case and discarding it if it does not respect the criteria. The process is repeated until there are enough feasible test cases. This has the drawback of generating a lot of invalid solutions if there are constraints, hence, it is not efficient for complex simulation data models. 2) Based on *tuning the generation*, using application-specific knowledge on how to build the data. The disadvantages are that it requires high development effort for each application and that it is not generic. 3) Based on *constraints solving*, where the generation effort is delegated to a constraint solver. This approach is not suitable for rich data structures, and there are few contributions for generating diverse solutions.

To mitigate the disadvantages of the last approach, LAAS-CNRS developed a tool: TAF (Testing Automation Framework) [12]. TAF, and the present paper, are inserted in an ongoing work at LAAS-CNRS regarding simulation-based testing [11–14]. The work of LAAS-CNRS focused on the generation of virtual worlds, on finding bugs in simulators with low physical fidelity for the robot Mana [14], on addressing non-determinism, and lately on developing TAF for aiding practitioners in generating test cases from rich data structures with constraints. TAF is a tool that mixes random sampling and resolution of constraints to generate test cases with enforced diversity. This tool was developed starting from an issue encountered in the Oz robot industrial case study [13], and generalizing the solution to tackle the bigger issue of generating size-varying data with constraints.

Even if TAF has been shown to manage well constraints, diversity, and rich structures [12], no paper has been published to assess the benefits and limits of its integration in a test suite for industrial case studies. This paper presents this assessment. Previously, with the agricultural weeding robot Oz [13], we encountered non-determinism related to the decisions of the system, and we proposed a statistical approach to address it [11]. In another case study, we encountered another type of non-determinism, related to the toolchain and controllability in simulation [9]. These works reinforce the idea that simulation-based testing is not a substitute of field-testing, and has its own advantages and drawbacks, hence the need for more experience reports with industrial case studies, to help the practitioners.

3 TAF: TESTING AUTOMATION FRAMEWORK

TAF (Testing Automation Framework) is an open-source tool developed at LAAS-CNRS to generate concrete test cases with added diversity and that respect constraints. TAF works by mixing random sampling and resolution of constraints to enforce diversity. The main algorithm uses the Z3 solver [6] for constraint solving. TAF was initially created to generate test cases of worlds and missions for robotic simulation, but it was expanded to solve the more general problem of generating size-varying data with semantic constraints.

As presented in Figure 1, TAF transforms an abstract model of the test data (called the Template), containing constraints, into instantiated test cases that respect the constraints (defined as "Test cases" before the export in Figure 1). Moreover, by using an export function/file customized by the user, TAF converts these test cases to output files that are specific to the testing architecture, e.g. maps/environments for the simulator, scripts, etc. An example is given in Figure 1, where the generated files are in JSON and in a specific format for a robotic application (explained later in the agricultural robot in Section 4). TAF uses a hierarchical XML tree structure to model the test data (the Template) with dedicated functions. This model contains the constraints, also expressed with XML, but extended with test generating features (e.g., with normal distribution for random sorting). Based on the model, TAF will create a skeleton of an export function in Python. This export function has to be customized by the user.

The concepts presented in this subsection (TAF template, constraints, and export) are explained in the following paragraphs.

3.1 TAF XML template

The TAF XML template is the input file of the TAF framework. It is on the left of Figure 1. It contains a hierarchical tree structure—the model of the test case—with its elements and parameters. A TAF test case template includes four types of XML elements: Root, Node, Parameter, and Constraint. Every element must have a "name" attribute. Elements are nested to form a tree starting from the root. The root and nodes are composite data structures with child elements, while a parameter is not composite. The parameters can be of type boolean, string, real, and integer. Both the parameters and

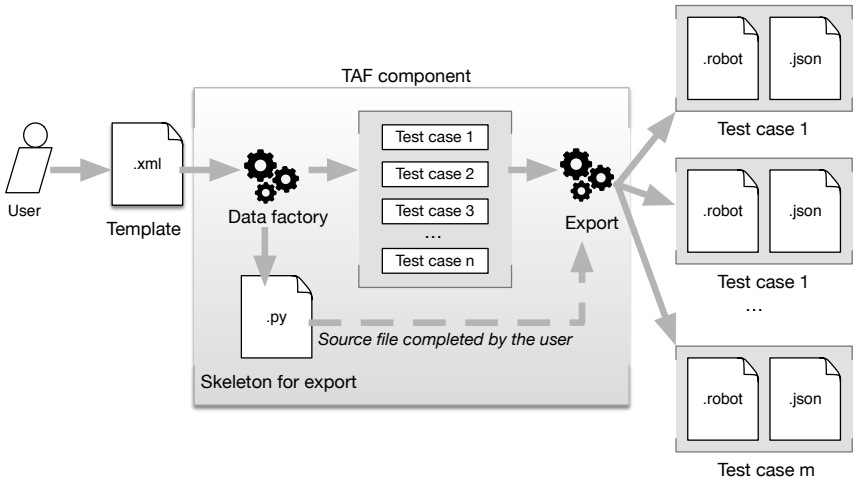

Figure 1: Overview of TAF.

nodes have a "nb_instances" (number of instances) meta-attribute that allows for multiplicity.

A snippet of an XML template can be seen below. This example specifies a world (which is a field for agriculture), composed of rows (of vegetables) (with a multiplicity between 2 and 10 rows, in L3). Each row has a length, which is between 10 and 100 (L4), and an inclination between -2 and 2 (L5). Only one constraint is presented here (L7-L11), which is explained in the next subsection.

```
1  <node name="world">     <!-- file .geojson-->
2    <parameter name="len_between_rows" type="real" min="1.5
       " max="2.5"/>
3    <node name="row" min="2" max="10">
4      <parameter name="row_length" type="real" min="10" max
       ="100"/>
5      <parameter name="row_inclination" type="integer" min=
       "-2" max="2"/>
6      <!-- rows constraints -->
7      <constraint name="interval" types="forall"
8        expressions="row[i]\length INFEQ 1.1*row[i-1]\
       length; row[i]\length SUPEQ 0.9*row[i-1]\length"
9                   quantifiers="i"
10                  ranges="[1, row.nb_instances-1]"/>
```

Listing 1: Snippet of the TAF XML template for the robot Dino

More examples of templates can be seen in the TAF paper [12] or on the TAF repository [15].

3.2 TAF model constraints

The TAF template include constraints, which consist of expressions that specify semantic properties in a descriptive way. The expressions may involve logical (not, and, or, implies), arithmetic (+, -, *, /), and relational (==, !=, <, <=, >, >=) operators. The structure of the expressions follows the Z3 solver syntax [6]. In Listing 1, the constraint in L7-L11 specifies that two consecutive rows (row[i-1] and row [i]) have lengths that cannot differ more than 10% (using 1.1 and 0.9 multiplication). The quantifier "i" (an iterator) is used to enumerate the rows, and its interval is specified with the keyword "ranges" (L11).

TAF uses the capabilities of the Z3 solver to construct valid test cases. TAF can create valid test cases from size-varying data structures with numerical constraints. It uses layered generation, meaning that layer "k" is generated respecting the constraints, then TAF moves to the generation of layer "k+1". If it is not possible to find values for the parameters that respect the constraints for layer "k+1", because there are constraints between values at layer "k+1" and layer "k", TAF uses backtracking to go back to layer "k" and generates a new different solution, then tries again to generate layer "k+1".

3.3 TAF export facilities

The role of the TAF export is to transform the instantiated model (the genotype, like the DNA of a person, which contains the value of the parameters) in files for the simulation (the phenotype, like the external traits, features, and appearance of the person).

The first time TAF is executed with a new template, it creates an export skeleton that is different for each template and that shows to the user how to access the various data elements in the instantiated test case. The export file is on the bottom-left of Figure 1. The export functions are initially empty, because they have to be completed with the code and customizations of the user. When customized, this export script creates output files based on the user needs. On the right of Figure 1 there are examples of the output files, in the form of test cases files .robot and .JSON.

This and the previous subsections have presented how TAF works, from input (template) to output (test cases), explaining the inner workings of TAF. The following subsection moves to a zoomed-out general view and shows how TAF can be integrated in a testing framework.

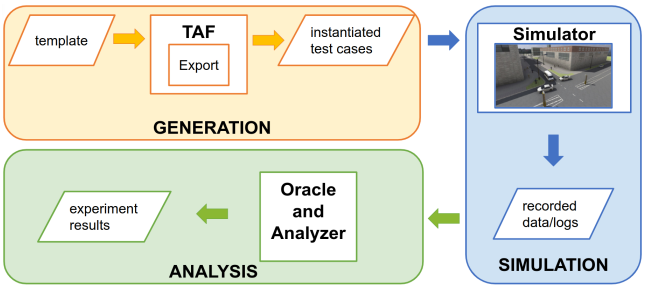

Figure 2: TAF integrated in a testing framework with generation, simulation, and analysis phases.

3.4 TAF integration in a testing framework

The goal of a testing framework is to assure that a system is thoroughly tested before going to production. It uses a repeatable procedure divided in phases, making it easy to test if changes to the code of the system have introduced faults, as repeating the testing procedure will show new failures. The testing framework is usually composed by three phases: generation, simulation, and analysis, visible in Figure 2. During the generation phase, the test cases are modeled and created. They can be generated using tools, like TAF, or can be composed by hard-coded test cases created by the developers. The test cases are then run in the simulation phase, using the simulator, which will record data and logs. The recorded data is used during the analysis phase, in which they are analyzed to understand what led to the failures. TAF can be inserted in the generation phase, as on the upper left of Figure 2, where TAF generates the test cases, starting from the template, as described in the previous subsections. As there are different ways to generate test cases, TAF was compared to other approaches [12], e.g., generate-then-filter, showing that TAF, for the four case studies, can generate diverse test cases (while other approaches can not), and that the performance of the generation time is competitive with respect to a similar generation tool, while obtaining a much better coverage of the data space.

4 THE AGRICULTURAL ROBOT CASE STUDY

This section introduces a case study that permits us to illustrate the integration of TAF into a continuous integration pipeline. We present an overview of the system, the type of test cases that are generated, and our contribution, in order to frame the insights gained during the integration, which are shown in Section 6.

This case study is in the context of validating the weeding mission of an agricultural robot, Dino, visible in Figure 3, where the System-Under-Test (SUT) is the software of the robot. The focus of our work is on the integration of the generation of test cases in the existing toolchain and continuous integration pipeline of the company, with a simulator in development. To assess our work on the testing automation and new generation process, we performed a testing campaign comparing simulation-based testing against field-testing, which showed bugs in the stable version of robot software and simulator.

Figure 3: The Dino robot for vegetable weeding on large-scale vegetable farms. Source Naïo Technologies [10]

4.1 System overview, environment, mission

Dino is an autonomous robot developed to mechanically weed vegetable crops on large-scale vegetable farms. It is designed by Naïo Technologies in Toulouse, France. Dino eliminates the need for chemical weed control, weeds all by itself, and works as follows. First, it memorizes and plots a map of the field of crops. Second, it uses GPS-tracking to work autonomously throughout the rows. Third, it uses a camera vision system to detect crops and to position its tools as closely as possible to the crops.

A typical **Environment** for Dino is a static flat field composed by multiple parallel rows (or vegetable beds) with similar length and with a similar distance between rows. The structure of the fields changes based on the vegetables, e.g., lettuce, carrots, onions, etc. Of course, there can be less typical fields with much more variety.

In the environment, a typical weeding **mission** for Dino consists in loading the map of the field, aligning itself with the row, regulating the position of the weeding tools before entering the row, weeding the row, raising the tools when exiting the row, performing a U-turn to align itself with the next row, repeating the previous steps until the last row has been weeded, and displaying on the remote a message to confirm that the mission has been completed.

The new Dino missions and environments are generated by TAF, which is integrated in the existing architecture as follows.

4.2 Test architecture

Naïo had already in place a continuous integration pipeline with a test architecture. This means that when new code or fixes are added to the robot software (the SUT), a set of hard-coded and predefined test cases are run in simulation, the observed robot behavior is compared to the expected behavior, and a test report is produced. With this process, if a failure is reported, the developers can understand what went wrong and fix the issue with the next code commit.

The test architecture of Naïo is comprised by many phases, e.g., build testing, unit testing, code analysis, acceptance testing. For this case study, we focused on the generation, simulation, and analysis, for a subset of the possible missions of the acceptance testing.

The company gave us two examples of predefined test cases used in their pipeline: one with one straight row and a second with two straight rows. Every time, the same two test cases were run without diversity. We extended and generalized this type of missions to an arbitrary number of rows, with added diversity. Due to the structure of the testing architecture, our TAF generation is done offline, detached (and before) the continuous integration of new code.

Regarding the integration of TAF in the testing architecture of Naïo, it is depicted in Figure 4. If there are just predefined test cases, there would be no generation phase, because the simulation would run the same tests every time. Instead, we introduced the generation phase with generic test cases, which is the first phase of the testing procedure. More specifically, the TAF template file is the starting point. TAF is in the **generation phase** and, starting from the template—the model of the test cases—, provides the instantiated test cases, and, through the TAF export, provides as outputs the files needed to start the simulation phase, i.e., map.geojson and mission.robot. TAF generates test cases containing the world, mission, and some properties for the expected behavior (like the maximum weeding time for the rows, commands to send to the robot, and expected keywords returned by the GUI of the robot). These test cases are then run in simulation.

To automate the **simulation phase**, the company uses Robot Framework [2], a keyword-driven test automation framework. In the test architecture of Naïo, Robot Framework reads the mission file, launches the SUT, the simulation, connects the SUT and the simulator using a messaging system, sends the mission commands to the robot, provides a runtime oracle, interrupts the simulation if needed, and writes a test report. Please note that an oracle is a mechanism to check if the robot behaves as expected, and gives a pass or fail verdict to the test, depending on if there are violations. During the simulation, the SUT writes a log with its perception. With our contribution, now the simulator also records a log—the ground truth—which can be compared to the SUT log.

After the simulation is finished, the **analysis phase** starts, in which the simulator and SUT logs are compared to check for possible misbehavior or wrong perception regarding the position, orientation, speed, etc. If a failure is detected during the comparison or at runtime by the oracle of Robot Framework, the test report is also used to analyze what happened to the SUT, since it contains information about the failure. The log of the SUT can also be replayed with a 2D offline replayer, different from the simulator, that shows a simplified version of the actions of the SUT. With continuous integration, when a bug is detected and a fix attempt is pushed as a code commit, the testing procedure is started again with the same test case to test if the bug has been fixed.

It should be noted that the SUT—Dino-core—is in C++, but the simulator is developed using Python in combination with Bullet for real-time physics simulation.

Test suite management. Since a new test suite is generated each time, a python scheduler manages its creation and other operations, but we did not include it in Figure 4.

Our experience and contribution regarding the design of the data models, the automation of the execution, logging, and analysis of the generated tests is explained in the following subsections.

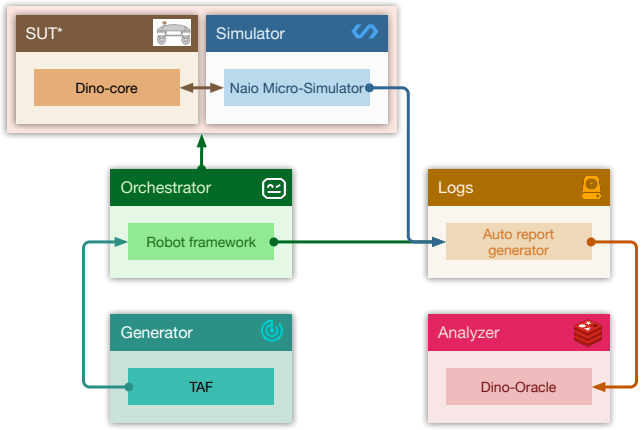

Figure 4: Testing architecture for Dino.

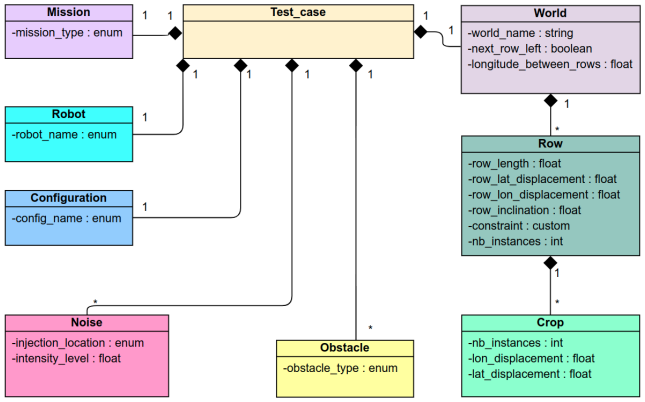

Figure 5: The UML class diagram of the Dino robot.

4.3 Design of the data model

To create instantiated test cases, TAF needs a template containing the model of the **world** and **mission**. We initially started the **modeling** by adapting a model of a previous agricultural robot case study of the same company [13]. The model was already complex, so we removed and changed parameters. Later we started with a new simple model, and we added parameters iteratively. We started the modeling from the two test cases provided by the company, i.e., one row and two rows, and the corresponding Robot Framework script. From there, we created the data model and the export for TAF. We modeled the test cases in order to create generic fields with an arbitrary number of rows and with more diversity. The **model** was enriched with new parameters, like the inclination of the rows, the longitude offset of the crops, etc.

The model uses a tree structure to represent the test case elements. As shown in Figure 5, the root node is named *Test_case* and is composed be the *Robot* and the *World* (the vegetables field). This *World* is composed by a *Row* element, which has multiplicity (the "*"), meaning that there could be multiple *Row* elements. *Row* is composed by the *Crop* element. Each element has its own parameters, e.g., *row_length*. **Constraints** linking elements of the test case

are also modeled, like the maximum length difference between one row and the next. This model has been translated in XML inside the TAF template.

After this brief description of the structure, it is worth reiterating that creating the template is an iterative process. Discussions are needed with the developers, and the execution of new tests can give more insight, which is used to improve the model and the UML class diagram, which can be one starting point for writing the template. Transposing a UML class diagram to a TAF template is straightforward. The difficulties are having a UML diagram that respects the specification requirements, and properly modeling the constraints. For this case study, the final template has 63 lines, 9 nodes, and 2 constraints. Its complexity is manageable, as the structure is still easy to visualize, the constraints are few and not complex to write. For the comparison of the testing campaign, a restricted template has been chosen, which focuses on the fields in possession of the company, where it is possible to test the robot and compare the execution to the simulated executions. More information about how to build the template are described in the TAF paper [12].

We then focused on the **TAF export**, which transforms the instantiated test cases in input files for the simulation, i.e., map.geojson and mission.robot. The company uses Robot Framework with a system of keywords to manage the test cases, so, to interface with Robot Framework, we modified the TAF export to create mission.robot files with the proper keywords.

From the export, TAF generates a part of the **expected behavior**, like the commands to send to the robot, the GUI keywords returned by the robot, and the timeouts checks for the weeding time, which have to be included in the mission.robot. We modeled the timeouts as constraints, which are linked to the test case, and more specifically, to the robot speed, and number and length of the rows.

4.4 Automation of the execution

The Robot Framework procedure managing the simulation was hard-coded, so we modified its library and keywords system to accept new and diverse test cases.

To pass from a predefined number of test cases to a test suite, a python scheduler was created to manage the creation of the test suite, the connection with the existing architecture and Robot Framework routine, and the moving and storing of the simulation files for reproducibility. We created the specification requirements and the first versions of the python scheduler, while subsequent versions were developed by a third party company.

4.5 Logging and analysis

Logging. The company relied on the Robot Framework report and the SUT log for diagnostic. The log of the simulator, the ground truth, was absent. Since it is necessary for a more complete analysis, we added logging capabilities to the simulator.

We designed the simulator log to include the same elements that were perceived by the SUT, like position (x, y, z), orientation, speed, etc., and to synchronize the recordings with the Unix timestamp, since the simulator timestamp would be relative to the initial time of the simulation.

Analysis and the oracle. There can be an online oracle, which checks the behavior during the runtime, and a post-processing oracle, which operates on the logs after the simulation is finished.

The online oracle relies on the properties inside the mission.robot file to check the expected behavior, thus, we worked with Naïo engineers to define the expected behavior of the SUT for the new test cases. We then developed the TAF export to create a mission.robot that would update the expected behavior for each new test case.

The online oracle usually checks fewer properties than the post-processing oracle, so it can be useful to have a post-processing oracle. Since the company did not have a post-processing oracle, we wrote its specification requirements, which will be used in the future for deeper analysis. The specification requirements take into account the information on properties, triggers, data, thresholds, and detection. Examples: Dino does not exit a specific area during a U-turn mission phase, or the maximal angle covered by U-turns does not exceed an angle threshold.

The results of the additions described in this and the previous subsections are narrated in the next subsection.

4.6 Results

We performed a comparison campaign in April 2021 to see the effect of our contribution to the testing architecture. We compared the results of the simulation-based testing and field-testing in two fields of the company, with the SUT being the production version of the robot software, running on 200 robots. The generation was restricted to be able to generate environments just like these two fields of the company, i.e., one field of 8 straight rows of 60m and one field of 14 sinuous rows of 20m. The simulation-based testing consisted of 500 runs. The results were: 445 successful missions, giving a success rate of 89%. For the 55 missions that failed, there are two types of errors. The first type are errors due to the Robot Framework implementation, which was expected, since the architecture is still in development. The last type is due to a bug in the guidance system of Dino, which goes in a fail-safe state and stops the mission early during a U-Turn. This bug has been confirmed in the field-testing. This is an encouraging result to push for the automation of the generation of more diverse test cases, that could save time and lead to finding bugs harder to find in a physical test.

Regarding the performance of simulation-based testing, the generation of 125 test cases with TAF, with the restricted template, took 7 seconds. This time depends on the complexity of the template, meaning that a field with more constraints could require more time to find suitable test cases. The test execution of a single test of the campaign, with 8 long rows, takes ≈18 minutes, comparable to a field-testing run, since it runs in real-time. In comparison, field-testing took one day to run 20 missions, since the robot has to be transported to the field and set up for the tests. The time needed for the tests in simulation of this campaign is greater than the few minutes needed for the original tests provided by the company, that have maximum 2 short rows. Nonetheless, the company sees the outcome of the campaign as a positive result and wants to integrate more diverse test cases in the Continuous Integration in the future. The company is interested in continuing the improvement of the simulation-based testing due to its advantages, because it can: be

parallelized, run alone and automatically (without developers being physically present), go faster than real-time, run 24/7, run a variety of different fields, with every possible condition, with execution time proportional to the complexity of the instantiated field.

Additionally, even if the testing campaign was restricted to the generation of virtual fields that had a comparable physical field usable by the company, with the TAF template, it is possible to generate completely different fields, in order to test other situations and corner cases. Preliminary results with the extended template have shown promise in finding new bugs, and the generation of more diverse test cases will be explored in future works.

This subsection concludes the description of the first case study, whose lessons learned are available in Section 6. The next section will present the second case study.

5 PEDESTRIAN PERCEPTION CASE STUDY

This case study was provided by SICK AG, a global manufacturer of sensors and sensor solutions for industrial applications, with headquarters in Waldkirch, Germany. This section presents an overview of this second case study, and our work on the integration of the generation of test cases with parametrizable behavior of dynamic agents. Compared to the previous case study, this case study cannot be presented with a similar level of detail, due to the Non-Disclosure Agreement. An important difference is also that we focus on scenario generation with dynamic agents (pedestrians and vehicles), and not on simulating the SUT. The lessons learned are shown in Section 6.

5.1 Overview

This case study is about an infrastructure-based perception system, designed by SICK AG, with the objective of monitoring a complex urban road intersection and tracking the agents idling or transiting on the roads, sidewalks, and pedestrian crossing. An example of the simulated intersection is visible in Figure 6. Compared to the previous case study, the **environment** is dynamic, due to the pedestrians and vehicles on the intersection. The object data potentially supports connected vehicles, e.g., autonomous vehicles (AVs), that will traverse the intersection. This enables the connected vehicles to avoid undesirable situations or cross at the maximum speed, if the intersection is free. The study was conducted in simulation, which mimics a real life intersection with various agents, e.g., pedestrians, cars, busses, or bicycles. The goal of our work was to integrate the generation of diverse test cases with dynamic agents, in order to generate more varied sensor data within the simulation. From the point of view of the perception system, the dynamic agents (vehicles and pedestrians) in front will occlude the line of sight, and consequentially, hide the dynamic agents in the back, making the detection and tracking more difficult. Thus, stressing the perception system, which is the goal of the testing.

5.2 The testing architecture

The company provided a simulation environment with four predefined test cases. Compared to the previous case study, the SUT—the tracking software—was not provided, thus, the analysis phase was not performed. The AVs were also not simulated. Regarding the simulator, SICK AG used a customized simulation framework, with own

Figure 6: Simulation of the road intersection. Source SICK AG [3]

modules, API, and modules for sensor data recording. The company records the ground truth of the agents inside the simulation, and compares it to the perception of the tracking system, to assess the correctness of the tracking. They also perform statistical analysis to assess the tracking performance. The testing procedure was requiring manual activities performed by the user, like launching the simulations, data recording, starting the tracking, or starting the analysis process.

For the generation phase, the company had its predefined test cases. We started from their framework, and we added the generation of test cases with parametrized dynamic agents, using TAF, as shown in Figure 7. TAF transforms the test cases in JSON files. In addition, C# scripts were created manually, but in the future, the plan is for TAF to generate them automatically. The JSON and C# scripts are the output of the generation phase, and the inputs for the simulation phase, as shown in Figure 7. Regarding the **simulation phase**, the SUT (not used in this case study) will track the dynamic agents and send the data to the autonomous vehicles, which will use the data to modify their speed. The AVs were also not simulated in this case study. The simulation framework, to add features, like dynamic agents assets, uses third-party modules and customized modules, which make use of simulation objects. To interface with the existing framework, we had to take into account the existing objects and modules inside the simulation framework. Thus, we wrote C# scripts that the SICK AG simulation framework used to control the agents' new behavior, made sure that the agents were instantiated correctly, and that the scenarios were executed properly. The scripts were connected to third party modules available for the simulation framework. During the simulation, the framework of the company records the object data, which will be used during the analysis.

The **analysis phase** was not performed for this case study, but it consists in comparing the effectiveness of the tracking software to the ground truth, which contains the agents instantiated by the simulator and their parameters (position, speed, etc.). Additionally, statistical analysis is performed to characterize the confidence interval and distribution of the tracking. This is important to understand if the SUT is suitable for the Operational Design Domain, if the

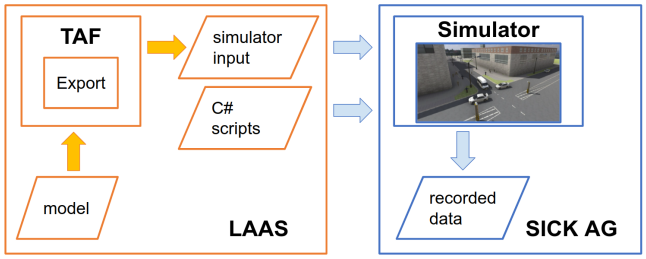

Figure 7: The testing procedure. The orange part on the left is the addition of LAAS-CNRS (TAF and C# scripts), while the blue part on the right is the part of SICK AG.

scenarios are inside the specifications, and if there are failures in the SUT.

5.3 Design of the data model

SICK AG provided four test cases with predefined parameters value for the agents, like the path to take, the starting time, speed, etc. From the provided case studies, we started investigating the generation phase by **modeling** a class diagram of the test case and the agents' parameters, which were included in a TAF template (of which, a simplified snippet is in Listing 2). The TAF **template** is the starting point on the left of Figure 7. To manage the dynamic agents at runtime, it was necessary to create a parametrizable behavior that could be defined during the generation with TAF. The behavior includes the selection of the path to take, direction, starting time, starting speed, etc., related to the various dynamic agents, e.g., pedestrians, vehicles, cyclists, and idle pedestrians. While iterating the modeling and executing the test cases, we observed that some vehicles and pedestrians spawned too close to each other, resulting in collisions, like in Figure 8. For this reason, we modeled **constraints** to not have spawning times too close for the agents. An example of this type of constraints for pedestrians is visible in Listing 2, starting from line 6. Apart from constraints related to the starting time, the other constraints are related to the starting path and position. The result, for this case study, is that the final template has 54 lines, 4 nodes, and 5 constraints. Compared to the agricultural robot case study, this template has a simpler structure, but more constraints, in order to avoid unwanted scenarios. We then created a TAF **export** function, which would create a JSON file containing the parametrized behavior of the agents. After being created by the TAF export, the test cases instantiate pedestrians and vehicles that take the possible paths, at different times, along the sideways and road lanes.

```
1  <root name="test_case">
2    <node name="pedestrian" min="5" max="15">
3    <parameter name="path" type="string" values="PathA;
       PathB;PathC;PathD"/>
4    <parameter name="starting_time" type="integer" min="1"
       max="30"/>
5    <parameter name="starting_speed" type="real" min="1.5"
       max="2.0"/>
6    <constraint name="spawn_pedestrian" types="forall;
       forall"
7          expressions="IMPLIES(i DIF j, pedestrian[i]\
       starting_time DIF pedestrian[j]\starting_time)"
8          quantifiers="i;j"
9          ranges="[0, pedestrian.nb_instances-1];[0,
       pedestrian.nb_instances-1]"/>
10   </node>
11 </root>
```

Listing 2: snippet of the TAF XML template for the SICK AG case study

5.4 Results

The integration of the generation phase with TAF was successful. It generated more diverse test cases, created scenarios to occlude the view of the SUT with agents, and made the tracking more difficult. The test cases, managed by the scripts, respect the constraints. The test cases were tested with a maximum of 20 walking pedestrians, 10 vehicles, and 8 idle pedestrians. Quantitative measurements of the performance are not available for disclosure.

6 LESSONS LEARNED AND TAKEAWAYS

This section lists the lessons learned, the takeaways, and the examples and experiences that led us to the recommendations. They are framed in three topics: the test data models (generation), the automation of the execution (simulation), and analysis. To ease the reading, we will refer to Naïo for the robot case study and to SICK AG for the pedestrian perception case study.

6.1 Lessons learned in designing test data models

Data models are intertwined. We started with the idea that the generation of the environment, mission, and expected behavior could be decoupled, but in practice, they are all coupled. For Naïo, they are coupled inside the Robot Framework files. There can be case studies in which they can be separated, but it is not a safe general assumption.

Iterative and incremental building of test data models. The building of the model should be iterative and incremental, as it is done in a classical or agile development process. It is particularly important to build prototypes of test data models and validate them with runs in simulation, because of effects that are difficult to foresee before running the test case. For instance, there can be effects of the modeling that require adding more constraint to make the test cases run as expected in the simulation. It was the case for the SICK AG case study, where the identification of physical constraints is not obvious and has an impact on the spawning time of vehicles. An illustrative example is that the vehicles have a physical volume, so it is not possible to spawn two cars in the same starting position with a small delay, as shown in Figure 8. To avoid such situation, it is necessary to add a constraint in the model about the spawning time.

Take into account simulator features in data models. During the modeling, a first abstract model could be done (as it is done with Operational Design Domains—ODD—) but some ideas and elements may not be implementable due to simulation features and limits. For Naïo, we can model, but not generate some missions, because they require features that are not available in the simulator, e.g., we cannot initialize the robot at an arbitrary position in the

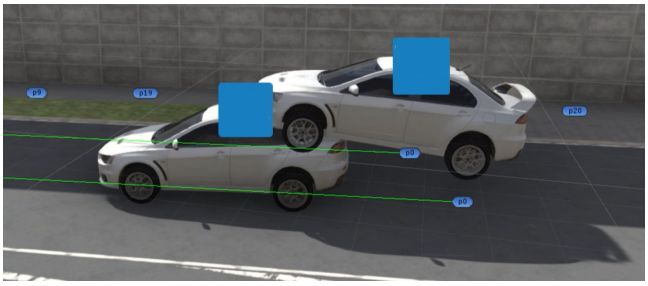

Figure 8: Cars collision due to missing time constraints.

middle of the field, because the company is still developing the simulator and the robot has to start outside the field. For SICK AG, some objects from the simulator framework impose how some objects should be modelled. For instance, pedestrians' paths and the modules that provide the path-related functionalities are only run when paths are predefined, thus, the paths have been integrated in the test model.

Constraints should be handled in the data models. The constraints should always be first considered to be integrated in the template, where they are handled by the solver and connected directly to the other parameters. For Naïo, we inserted one constraint for the maximum weeding time, a timeout, in the export, which coupled instantiation and export. This was due to a restricted view of the model, that looked just at the inputs. In retrospect, we should have considered the outputs too, and we should have integrated it in the template.

Data models constraints should consider expected outputs. For the data model, it is restrictive to consider just the inputs, as some expected outputs of the simulation should be considered and should have corresponding constraints included in the model. For instance, for Naïo we needed to generate weeding timeouts (which are not inputs) for the oracle. The timeouts were related to the expected behavior and were created using constraints that connected the robot speed, row length, and rows number.

Use simple scenarios, not complex ones. Even if it sounds obvious, most of the faults can be found with many simple scenarios and not with complex ones. Besides the fact that simpler tests are easier to develop, maintain, and analyse, they also are more reliable and deterministic.

Generation is not limited to simulator input files. In most of the cases, the generation has to produce static simulator-specific input files, but also additional scripts when a behavior is required for the test input. For Naïo, this means adapting the TAF export to create a Robot Framework mission.robot file containing the mission, commands for the robot, and timeouts. For SICK AG, C# scripts were developed to be used by the simulation framework to load the JSON file and instantiate and manage dynamic agents.

Automatic generation requires development resources. Adding the generation phase is still not plug and play. Moving from predefined test cases to new automatic test suites requires a test suite management system and a scheduler to coordinate the various steps of the testing. Even if simulators already integrate such features (e.g., scenario_runner in CARLA simulator [7]), they are usually not integrating facilities like constraint solving. For both of our experiment, no test input generator was available on the market or in the research community to be directly used.

6.2 Lessons learned for the automation of the execution

Simulation-based testing brings non-determinism. Simulation-based testing is complementary to field-testing and presents its own issues and limits. It can present non-determinism related to the toolchain [9], to the decisions taken by the SUT [11] or to the design of the tests. We have observed that the toolchain can introduce spurious failures, like in the simulation campaign for Naïo, where, out of the 500 tests run, 51 tests failed due to simulator or framework related issues. To diagnose and avoid these issues, a trend is to use the same exact configuration files, virtual environments, and use containers like Docker (to avoid dependency hell). A related issue is the failing of the test due to the wrong synchronization between the SUT and the simulator, since the testing architecture and simulator are in development.

We also observed failures due to phantom processes or memory not being cleaned properly, which would manifest after running more than 100 test cases or with TAF not being able to instantiate test cases. In the last case, a reboot solved the issue.

We have also observed flaky (unstable) tests for this SIL testing, even if they are usually a classic problem for other testing fields. If the test environment is stable, rerunning the unstable tests and redesigning them solved the issues.

Use test objectives to select the right simulator. Since the available simulators differ greatly regarding the complexity, the physical fidelity, and how the execution can be automated, the selection of the toolchain and the simulator depends on the needs of the company and on the test objectives. Naïo wants simple physics and fast iterations (hence Bullet), because they use the simulation for non-regression testing and acceptance testing. Previously, they experimented with a more complex simulator, but decided to select a simpler simulator that allowed them to fail fast, still find bugs, and iterate fast with the CI. A simple simulator can be useful, because, even with low physical fidelity, it has been demonstrated that bugs can be found [14]. SICK AG, on the other hand, needs realistic sensor data and good support to manage and integrate sensor modules, hence they use a more mature simulator with better commercial support.

6.3 Lessons learned for logging and analysis

Use different oracles for different needs. The oracle can be used at runtime or for post-processing. Use an online oracle when the interest is to stop the test as soon as a violation is detected. Use a post-processing oracle to easily change the properties without relaunching the simulation, and to have a more complex analysis. The online oracle is suitable for Naïo because they want to interrupt a test as soon as there is a failure, to save resources and time. Their online oracle is also simpler than the post-processing oracle and based on the commands sent to the GUI (graphical user interface) of the robot, which could also lead to failures.

It is also possible to do the analysis just in post-processing, like in the case of the SICK AG. SICK AG has no interest in stopping the simulation early, as they want to gather a lot of sensor data with each simulation, and their SUT is tested after the simulation is over, using the recorded data. They use the post-processing to analyze complex properties and the tracking over long periods of time.

Record the ground truth for better analysis . The post-processing oracle needs to compare the SUT log and the simulator log. For the ground truth, it is necessary to log the data corresponding to the subjective perception of the robot, like the position, orientation, and speed. The logs need to have comparable coordinate systems and timestamps. If this is not the case, like with Naïo's framework, then a conversion is needed. A small tolerance has to be considered when comparing an event in the SUT and simulator logs, since the two timestamps will never have the same value. Also, triggers have to be inserted, in order to not record unnecessary information before the SUT is connected to the simulator.

For dynamic agents, like with SICK AG, a list of all the objects and agents inside the simulation has to be recorded, so it is possible to check if they are tracked correctly later.

Do not limit the analysis to simulation and oracle. For the case study of Naïo we could use three types of diagnostic: online, post-processing (RF report or ground truth comparison), and a simplified replayer for the events, which is manual and out of the CI. This replayer is a separate tool, which is 2D and lacks some features of the simulator, but is faster to use than a simulation, and complements well the written reports, giving an additional option to understand the events leading to the failure.

7 CONCLUSION

This paper presented our work and experience in integrating an automated test case generation, provided by the tool TAF, in two industrial simulation-based testing frameworks, with a focus on the generation, simulation, and analysis phases. The two testing frameworks are related to two case studies. The first one is about an agricultural robot, by Naïo Technologies, that performs weeding missions in a static field. The second one is about a perception system, by SICK AG, that surveils a road crossing with dynamic agents. The two case study provided two very different types of test cases for us to model, and for TAF to generate. The tool TAF is introduced with its features and capabilities, and the two case studies are presented with a focus on their testing process and how the addition of the automated generation changes the testing architecture.

In both case, the integration of test case generation (with TAF), was a success to explore more test cases, and to automatize a tedious and limited manual process. We abstracted the experience gained with the case studies, and we presented the lessons learned on the design of the test data models, the automation of the execution, logging, and analysis of the generated test cases, in order to provide insight to practitioners of simulation-based testing.

Our future work will explore the integration of the automatic generation in other testing frameworks with a post-processing oracle and an analysis phase. Another direction that we plan to follow is the addition of search-based and combinatorial algorithms, and the transformation of the testing frameworks in testing loops, to optimize the model parameters of the next iteration, based on the results of the test cases previously ran, to increase the probability of encountering failures. The last direction is to use TAF to model and generate more complex test cases and scenarios with dynamic agents, in which the generation of the events has to be created before the runtime.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 812.788 (MSCA-ETN SAS). This publication reflects only the authors' view, exempting the European Union from any liability. Project website: http://etn-sas.eu/.

COVR has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 779.966.

REFERENCES

[1] [n. d.]. ASAM OpenSCENARIO. https://www.asam.net/standards/detail/openscenario/ Accessed: 2022-01-11.
[2] [n. d.]. *Robot Framework*. https://github.com/robotframework/robotframework Accessed: 2022-01-11.
[3] [n. d.]. *SICK Germany*. https://www.sick.com/de/en/ Accessed: 2022-01-11.
[4] [n. d.]. Unity. https://unity.com/ Accessed: 2022-01-11.
[5] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, Jenny Li, and Hong Zhu. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978 – 2001. https://doi.org/10.1016/j.jss.2013.02.061
[6] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary*. 337–340.
[7] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. *arXiv:1711.03938 [cs]* (Nov. 2017). http://arxiv.org/abs/1711.03938 arXiv: 1711.03938.
[8] N. Koenig and A. Howard. 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) (IEEE Cat. No.04CH37566)*, Vol. 3. 2149–2154 vol.3. https://doi.org/10.1109/IROS.2004.1389727
[9] Mohamed El Mostadi, Hélène Waeselynck, and Jean-Marc Gabriel. 2021. Seven Technical Issues That May Ruin Your Virtual Tests for ADAS. In *2021 IEEE Intelligent Vehicles Symposium (IV)*. 16–21. https://doi.org/10.1109/IV48863.2021.9575953
[10] Naïo Technologies 2013. https://www.naio-technologies.com/. Accessed: 2022-01-11.
[11] Clément Robert, Jérémie Guiochet, and Hélène Waeselynck. 2020. Testing a non-deterministic robot in simulation - How many repeated runs ?. In *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*. 263–270. https://doi.org/10.1109/IRC.2020.00048
[12] Clément Robert, Jérémie Guiochet, Hélène Waeselynck, and Luca Vittorio Sartori. 2021. TAF: a tool for diverse and constrained test case generation. https://hal.laas.fr/hal-03435959
[13] Clement Robert, Thierry Sotiropoulos, Helene Waeselynck, Jeremie Guiochet, and Simon Verhnes. 2020. The virtual lands of Oz: testing an agribot in simulation. *Empirical Software Engineering (EMSE)* 25, 3 (2020), 2025–2054.
[14] Thierry Sotiropoulos, Hélène Waeselynck, Jérémie Guiochet, and Félix Ingrand. 2017. Can robot navigation bugs be found in simulation? An exploratory study. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*.
[15] Testing Automation Framework 2019. Testing Automation Framework. https://www.laas.fr/projects/taf/. Accessed: 2022-01-11.