StellaUAV: A Tool for Testing the Safe Behavior of UAVs with Scenario-Based Testing (Tools and Artifact Track)

PrePrint for 33rd IEEE International Symposium on Software Reliability Engineering, 2022
Published version: https://doi.org/10.1109/ISSRE55969.2022.00015

Tabea Schmidt
Department of Informatics
Technical University of Munich
Munich, Germany
tabea.schmidt@tum.de

Alexander Pretschner
Department of Informatics
Technical University of Munich
Munich, Germany
alexander.pretschner@tum.de

***Abstract*—When we allow Unmanned Aerial Vehicles (UAVs) to perform their missions autonomously in the near future, we need to ensure their safe behavior. To generate relevant test cases that can reveal potential faults in the tested UAVs, we propose to leverage scenario-based testing from the automotive domain. For a systematic application of this methodology, we present *StellaUAV*, a tool for testing the safe behavior of UAVs with scenario-based testing. With our proposed tool, we can describe relevant test situations, generate test cases for these situations that can reveal potential faults in the tested UAV, and evaluate the performance of different optimization algorithms and their combinations. To demonstrate its applicability, we apply StellaUAV to generate test cases for various situations and discover several safety distance violations of the tested exemplary UAV in the presence of dynamic obstacles. These experimental results indicate that the given system under test can handle situations with only static obstacles rather well, while it encounters problems when facing dynamic ones. Further, we detect that a combination of optimization algorithms can find safety distance violations for a logical scenario that the widely used algorithm NSGAII deemed safe for the tested system. Overall, our results show that StellaUAV can effectively detect potential faults in the tested UAV.**

***Index Terms*—unmanned aerial vehicles, scenario-based testing, test case generation, safety**

I. Introduction

Unmanned Aerial Vehicles (UAVs) are used in various use cases such as monitoring areas, delivering packages, or search & rescue operations [1], [2], [3]. As an industrial example, Zipline has effectively supplied medicine in Rwanda with UAVs since 2016 [4]. As autonomously operating UAVs are on the rise and will perform most of these use cases in the future, we need to ensure that they behave safely and do not harm anybody or anything while operating.

When testing the safe behavior of autonomously operating UAVs, we need to inspect their behavior in various situations and ensure that they behave safely, even in the most challenging situations, called worst-case situations. To tackle these challenges, we propose to leverage scenario-based testing [5] from the automotive domain, where the method has provided valuable insights into the safe behavior of autonomous cars [6], [7], [8]. In scenario-based testing, we test the System Under Test (SUT) in typical situations that it might encounter in the real world. An example of such a typical situation for a UAV is the following: the UAV has the mission to autonomously fly to a target point in foggy conditions and light rain while avoiding two dynamic obstacles. For each of these situations, we then search for challenging situations for the SUT to find test cases that can reveal *potential* faults in the SUT. Note that we cannot apply the concepts from the automotive domain without modifications and extensions. This adaptation is needed as the environment of cars is structured in a much more rigid and fine-grained way: by geometric constraints such as possibly curved streets, lanes, and crossings; by temporal constraints imposed by the movements of other vehicles; and by regulations such as traffic rules. This information helps define and restrict the search space for relevant test cases for autonomous cars. Without this additional information, specifying test situations and the UAVs' safe behavior is more challenging, as spelled out in [9]. Moreover, we focus on *missions* for UAVs, whereas one concentrates on *maneuvers* for autonomous cars, which present more fine-grained descriptions of the systems' expected behavior. Thus, we need to define different test situations and adapt the methodology to find challenging situations for UAVs. For the systematic application of scenario-based testing, we need a tool that enables us to (A) define typical situations in which we aim to test the UAVs' behavior, (B) generate test cases that represent worst-case situations for the SUT, and (C) evaluate the quality of the

generated test cases.

In related work, the authors of [10] and [11] present frameworks focusing on testing the reliability of UAVs and path planning algorithms. The authors of [12] concentrate on generating test cases based on belief state machines and minimizing test suites for testing the reliability of cyber-physical systems. In contrast with these works, we focus on testing the *safe behavior* of UAVs as another essential aspect of testing these autonomously operating cyber-physical systems. Further, we can not directly apply tools from the automotive domain [13], [14] to test the *safe behavior of UAVs* since different situations are challenging for these two kinds of systems. In [15], the authors provide a framework for testing the stability of a controller for UAVs with metamorphic and model-based testing. When applying the presented methodology, one needs to define correct and complete metamorphic relations, which is a challenging task in practice. In contrast, our focus is on scenario-based testing and the definition of relevant typical situations *for the entire SUT*. The authors of [16] apply fault-injection methods to test the safe behavior of their UAV autopilot system in their framework. However, such fault-injection techniques assume that we already know all relevant faults, which is not the case when testing the safe behavior of UAVs. In contrast with their work, we aim to also find unknown faults in our systems by searching for challenging situations for the SUT. To summarize, existing tools and frameworks focus on testing the *reliability of UAVs* or the *safe behavior of autonomous cars*. To the best of our knowledge, there is no tool for testing the *safe behavior of UAVs* with scenario-based testing that allows for the specification of test scenarios for UAVs and the evaluation of their safe behavior.

The **contribution of this paper** is StellaUAV, a tool for testing the safe behavior of autonomously operating UAVs using scenario-based testing. With StellaUAV, we can specify test scenarios and generate worst-case situations for each of them with the help of optimization algorithms. As StellaUAV uses implementations of various such algorithms, we can further compare the test cases created by different algorithms with StellaUAV. Using StellaUAV, we have found that the results of different optimization algorithms differ by as much as 20%, a result of the heuristic nature of these algorithms and a hint to the practitioner that, in general, it is not sufficient to rely on just one such algorithm.

Organization. In Section II, we demonstrate the process of testing the safe behavior of UAVs with scenario-based testing. Section III describes the methodology, and Section IV presents the architecture of StellaUAV. In Section V, we show experimental results of applying StellaUAV. Section VI discusses related work; Section VII concludes.

II. Scenario-Based Testing

When applying scenario-based testing to assess a UAV's safe behavior, we distinguish between *logical* and *concrete* scenarios, as proposed by the authors of [17]. Logical scenarios describe typical situations in which we aim to test the UAV's behavior. We describe these logical scenarios with n parameters $P = \{p_1, p_2, \ldots, p_n\}$ and their respective parameter ranges. Table I contains an exemplary logical scenario that describes the UAV's environment with light rain, moderate ambient temperature, medium cloud coverage, and fog. In this example, parameter p_1 represents the ambient temperature in degrees Celsius in the presented logical scenario. Parameter p_2 specifies the range for the light rainfall in centimeters per hour, and p_3 describes the moderate cloud coverage between 10% and 50%. Finally, p_4 introduces fog that affects the visibility of the UAV in the logical scenario with reduced visibility of 200.0 to 1000.0 meters. Table I presents a simplified, incomplete, and exemplary logical scenario that we may enrich with additional parameters, such as the wind force and direction or parameters of the included obstacles. By selecting concrete values from the ranges of each parameter, we create a concrete scenario that represents a test case for our SUT. Table II presents two concrete scenarios that instantiate the logical scenario defined in Table I. In the first exemplary concrete scenario, we, e.g., set the ambient temperature to 15.2 degrees Celsius, the rainfall to 0.21 centimeters per hour, the cloud coverage to 44.8%, and the reduced visibility to 430.0 meters. The derivation of such concrete scenarios is the purpose of the test case generation procedure described below.

TABLE I
Example of a logical scenario for testing the safe behavior of UAVs with moderate ambient temperature, light rainfall, moderate cloud coverage, and fog.

Parameter P		Value Range (Min, Max)
p_1	temperature [°C]	(10.0, 30.0)
p_2	precipitation [cm/h]	(0.1, 0.25)
p_3	cloud coverage [%]	(10.0, 50.0)
p_4	reduced visibility [m]	(200.0, 1000.0)

TABLE II
Two exemplary concrete scenarios for the logical scenario presented in Table I.

Parameter P		Test Case 1	Test Case 2
p_1	temperature [°C]	15.2	27.1
p_2	precipitation [cm/h]	0.21	0.13
p_3	cloud coverage [%]	44.8	31.9
p_4	reduced visibility [m]	430.0	873.5

Let us now turn our attention to the general process of testing the safe behavior of UAVs with scenario-based testing. As depicted in Fig. 1, we first define the test situations for the SUT as logical scenarios ①. As explained above, we define these logical scenarios with parameters that describe the UAV and its environment. Next, we need to ensure that the derived list of logical scenarios represents all relevant test situations ⑤ to enable certification of these systems. How to achieve this goal is a very active research area in the scenario-based testing community [18], [19], [20]. Even if we cannot be sure that we have defined a complete list of logical scenarios, we still need to gain an intuition if our list is representative of relevant test situations for the SUT. Once we are sufficiently satisfied with the completeness of the list of logical scenarios for testing the SUT, we proceed to generate test cases for each

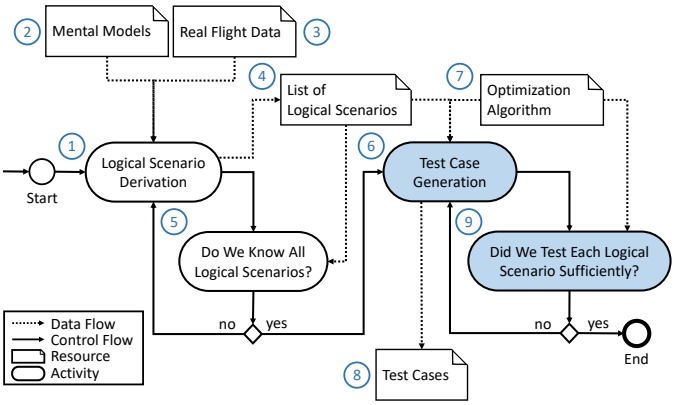


Fig. 1. Methodology of testing the safe behavior of UAVs with scenario-based testing. Our tool provides a solution for the highlighted activities.

of them ⑥. Otherwise, we return to step ① and collect more logical scenarios before creating test cases. Finally, we need to ensure that the UAV behaves safely in all instances of a logical scenario ⑨ to achieve a thorough safety argumentation about the UAV's behavior. As it is infeasible to test all instances of a logical scenario, we instead aim to derive the most challenging ones and inspect the UAV's safe behavior in these. We can find these challenging concrete scenarios by casting the test case generation problem as an optimization problem, as we explain below.

One way to identify logical scenarios for testing the safe behavior of UAVs is based on "mental models" ② that reflect experts' intuition about challenging and relevant situations for UAVs. This intuition stems from literature, specifications, and experts' knowledge of challenging situations for the SUT, as presented in [17], [21]. The use of these (implicit) mental models arguably is the predominant approach for manually deriving logical scenarios today. As we cannot easily state whether a manually derived list of logical scenarios presents all relevant logical scenarios for a SUT, literature suggests applying data-driven approaches to complement the manually derived logical scenarios ③. One example of these approaches is [22], which applies clustering techniques to collected real-world data of the SUT. With these techniques, we automatically group the gathered concrete scenarios from the real world that show similar behavior. These clusters of concrete scenarios represent potential relevant logical scenarios for the SUT. The manual and the data-driven methods both yield a list of logical scenarios ④ to test the SUT.

In the test case generation step ⑥, we aim to create "good" test cases as introduced in [23] instead of randomly sampling the search space for test cases. These "good" test cases represent challenging worst-case situations for the SUT and, thus, can reveal *potential* faults in the SUT. We can detect unsafe behaviors of an incorrect system in such "good" test cases while a correct system presents a safe behavior by, e.g., keeping specified safety distances to all obstacles. The search for "good" test cases can then be formulated as an optimization problem with the goal of, for instance, minimizing the distance

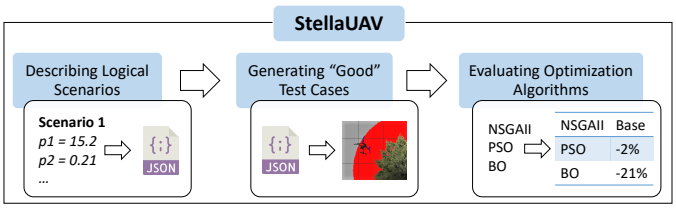


Fig. 2. The three fundamental use cases of StellaUAV as a tool for testing the safe behavior of UAVs with scenario-based testing.

to obstacles in the scenario and hopefully going below an acceptable safety margin. Optimization algorithms ⑦ such as Non-dominated Sorting Genetic Algorithm II (NSGAII) [24] or Particle Swarm Optimization (PSO) [25] perform this search and derive "good" test cases ⑧ for the SUT, including the worst-case situation for the tested logical scenario, as a result.

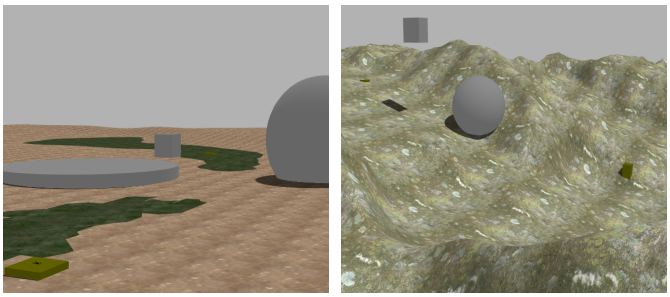
In this work, we present a tool for testing the safe behavior of UAVs with scenario-based testing. Thus, we focus on the test case generation step ⑥. Further, the tool provides implementations of various optimization algorithms and their combinations to allow for their evaluation in step ⑨.

III. Methodology

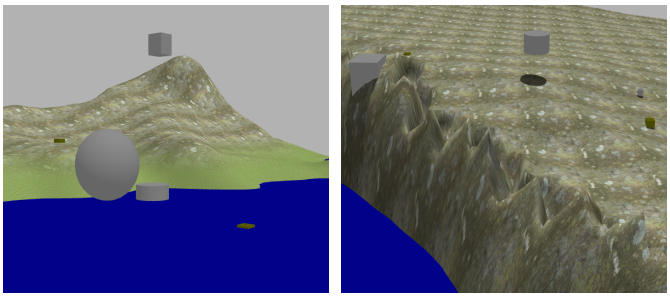
In this section, we describe the fundamental use cases of our tool StellaUAV and present the methodology we apply for each of them. The three essential use cases of StellaUAV are: (A) describing relevant logical scenarios for the SUT, (B) generating "good" test cases that depict worst-case situations for the SUT, and (C) evaluating the performance of various optimization algorithms for our use case. Fig. 2 presents an overview of these use cases and their relationship.

A. Describing Logical Scenarios

Before we can evaluate the behavior of a UAV in a logical scenario, we first need to describe the characteristics of this logical scenario. Remember that logical scenarios are defined by parameters and their ranges, e.g., p_1 as the ambient temperature with a range of 10.0 to 30.0 degrees Celsius. We propose to use the JavaScript Object Notation (JSON) format to provide a machine-readable description of a logical scenario. We selected JSON since it offers a simple syntax, has easy and fast parsing possibilities, and has the option to present various data types. In addition, the JSON syntax allows us to specify a general schema that all logical scenarios need to comply with. In such a JSON schema, we can define necessary and optional dimensions of logical scenarios and possible parameter values such as *cold*, *moderate*, or *hot* temperature. Note that such categories are system-specific and that we need to define the corresponding value ranges for each of these categories specifically for the SUT. Such a system-specific description enables us to search for challenging situations for our SUT since, e.g., the cold temperature might be represented by a temperature of 5.0 to 10.0 degrees Celsius for UAVs operating near the ground and a temperature of -10.0 to 0.0 degrees Celsius for UAVs flying in high altitudes.



(a) flat/land (b) depression/land



(c) elevation/water (d) steep transition/water

Fig. 3. Exemplary worlds that we can simulate with StellaUAV with different landforms, surface natures, and obstacles of various sizes and forms.

```
{
    "name": "Test Scenario #1",
    "system": {
        "UAVs": [{"maneuvers": ["move to waypoint"]}]
    },
    "environment": {
        "flight area": {
            "landform": "flat",
            "surface nature": "land"
        },
        "obstacles": [
            {
                "kind": "dynamic",
                "size": "medium",
                "form": "sphere"
            },
            {
                "kind": "static",
                "size": "large",
                "form": "cuboid"
            }
        ],
        "weather": {
            "wind force": "moderate",
            "reduced visibility": "heavy_fog",
            "lighting": "normal",
            "temperature": "moderate",
            "precipitation": "none",
            "lightning": "none",
            "cloud cover": "none"
        }
    }
}
```

Fig. 4. Example of a logical scenario specified in JSON.

StellaUAV provides a JSON schema that includes various landforms, surface natures, obstacle kinds, obstacle sizes, obstacle forms, wind forces, and reduced visibilities. Fig. 3 presents examples of simulation worlds built with StellaUAV with these options. In each logical scenario, StellaUAV automatically sets the mission of the UAV to fly to a specified waypoint dependent on the landform in which it operates. Fig. 4 shows an example of a logical scenario specified in JSON for StellaUAV. All chosen parameter values represent parameter ranges, e.g., moderate wind force stands for a wind force of $4.0 - 8.0$ kilometers per hour, and medium-sized obstacles for obstacles of a size of $5.0 - 10.0$ meters. We need to define these value ranges manually and system-specifically as different situations present challenging situations for various kinds of UAVs. If necessary, the list of parameters can be extended by the user.

B. Generating "Good" Test Cases

To evaluate the safe behavior of UAVs, we aim to find challenging situations in the provided logical scenarios. In order to do so, StellaUAV first derives the search space for each logical scenario. We define the dimensions of this search space with the logical scenario's parameters P and their corresponding value ranges. In addition, we include the initial obstacle positions and their velocities in the search space in StellaUAV. To find "good" test cases in this search space, we specify a fitness function f that encodes the notion of challenging situations. StellaUAV uses the fitness function described in [9], which is used to minimize the difference between the minimal distance $d(cs, t)$ that the UAV keeps in a concrete scenario cs to any of the obstacles at time $t \in T$ and a specified safety distance $s(cs, t)$:

$$f(cs) = min(\{t \in T : d(cs, t) - s(cs, t)\}) \quad (1)$$

This fitness function provides an automatic oracle about the safe behavior of the UAV: If the fitness function evaluates to a negative value, the UAV violated the specified safety distance. Otherwise, the UAV exhibited safe behavior in the evaluated concrete scenario as it kept a sufficient distance from all obstacles: $\forall t \in T : d(cs, t) \geq s(cs, t)$. After the definition of the search space and the fitness function, the tool searches for "good" test cases following the process depicted in Fig. 5.

The input to the test case generation step ⑥ from Fig. 1 is a logical scenario Ⓘ in which we aim to evaluate the UAV's behavior and an optimization algorithm Ⓘ to perform the search for "good" test cases. The optimization algorithm first picks an initial set of concrete scenarios from the search space to evaluate the UAV's behavior. In step Ⓘ, the tool builds for each concrete scenario the simulation environment Ⓘ for the applied simulator according to the parameter values of each concrete scenario. In step Ⓥ, StellaUAV starts the simulation in the built environment, positions the SUT Ⓘ in it, and sends the SUT on its mission to fly to the specified destination point using an external interface (the MAVSDK-Python library). Note that we define the mission as part of the evaluated logical scenario. Throughout the simulation, the tool

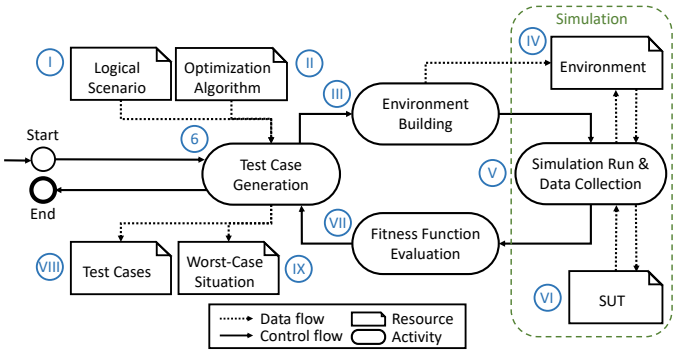


Fig. 5. A detailed look at the test case generation process of the tool StellaUAV, which describes step (6) in Fig. 1.

collects data about the position of the SUT and the obstacles. Finally, after the simulation has finished, StellaUAV computes the fitness value of the evaluated concrete scenarios (VII). In this step, the tool applies (1) to calculate the remaining distance until the SUT violates the specified safety distance at each time step. For this computation, StellaUAV calculates the minimal distance that the SUT keeps to any obstacle at each time step and compares it with the specified safety distance. After assessing the initial set of concrete scenarios, the selected optimization algorithm generates additional concrete scenarios with the goal to create "better" test cases that depict worst-case situations for the SUT. Results from the test case generation step are the created test cases (VIII) and the worst-case situation found in the evaluations (IX). The behavior of the SUT in this worst-case situation shows us its safe or unsafe behavior in the provided logical scenario.

To ensure the realism of the generated test cases, StellaUAV checks whether the created concrete scenarios represent instances of the evaluated logical scenarios. The tool assigns a poor fitness value to a concrete scenario that does not conform to the logical scenario as it, e.g., does not perform the specified mission. Secondly, we exclude concrete scenarios with dynamic obstacles with very high velocities to enable the SUT to cope reasonably with these obstacles. If the obstacles move too fast and the SUT cannot react to them, we cannot blame the SUT when violating any safety distances. Thus, we need to ensure that the SUT can theoretically keep the specified safety distance and test whether it does so. In StellaUAV, we prevent these unavoidable situations by limiting the velocity of the dynamic obstacles and not placing them next to the starting or landing area. Finally, note that we use approximations of real-world obstacles and include only specific environmental effects due to the limitations of our simulation setup that we will discuss in the next section, which limits the representativeness of real-world situations. Nonetheless, we depict various relevant environmental effects for UAVs, such as wind and fog, and include static and dynamic obstacles.

C. Evaluating Optimization Algorithms

If the presented test case generation procedure results in an unsafe behavior of the tested UAV, we need to fix the corresponding fault to develop a safely behaving UAV. However, if the UAV shows a safe behavior in all test cases we derived for a logical scenario — even in the most challenging one — there are two potential causes for this behavior: (1) the UAV indeed always behaves safely in the provided logical scenario, even in the most challenging situation; or (2) due to its heuristic nature, the optimization algorithm did not discover the most challenging situation. In the second case, we hence do not know whether the UAV will behave safely in this missing worst-case situation. Thus, to build a thorough safety argumentation about the UAV's behavior, we need to assess the quality of the generated test cases. We propose to evaluate this quality by applying various optimization algorithms to gain an understanding of the most suitable one for finding worst-case situations for the SUT.

StellaUAV allows for such considerations by providing various optimization algorithms for generating "good" test cases, namely NSGAII, PSO, and BO and their collaborative combinations, as explained in [26]. In these collaborative combinations, the optimization algorithms share information between *sequential executions* of the different algorithms. Roughly, the best solutions found by one algorithm are used as initial solution candidates for another algorithm, as suggested by related work on seeding strategies [27], [28], [29]. In addition, we add to the initial candidates of the succeeding optimization algorithm random candidates from the areas of the search space in which the algorithm combination produced the fewest candidates so far. StellaUAV uses permutations of three-fold combinations of the optimization algorithms NSGAII, PSO, and BO without repeating the same algorithm in two subsequent executions. The resulting worst-case situation of a combination is the concrete scenario with the lowest fitness value found in all algorithms of this combination. When evaluating the performance of various optimization algorithms and their combinations, we encourage test engineers to perform several runs of each algorithm for a specific logical scenario to gain reliable results.

IV. Architecture

Fig. 6 depicts the architecture of StellaUAV and highlights connections with the presented methodology in Fig. 5 by referencing essential steps. StellaUAV consists of four main packages that implement the optimization process, the optimization algorithms, the forms of the obstacles, and additional utility classes. In the *optimization_process* package, StellaUAV includes classes to build the simulation world in the simulator Gazebo [30], formulate the search problem as a FloatProblem of the jmetalpy framework [31], and communicate with the UAV, the SUT, via the MAVSDK-Python library [32]. The open-source simulator Gazebo offers the possibility to simulate various systems such as UAVs, submarines, and robots in environments with different landforms, obstacles, and environmental effects. The simulator further presents an extensive set of

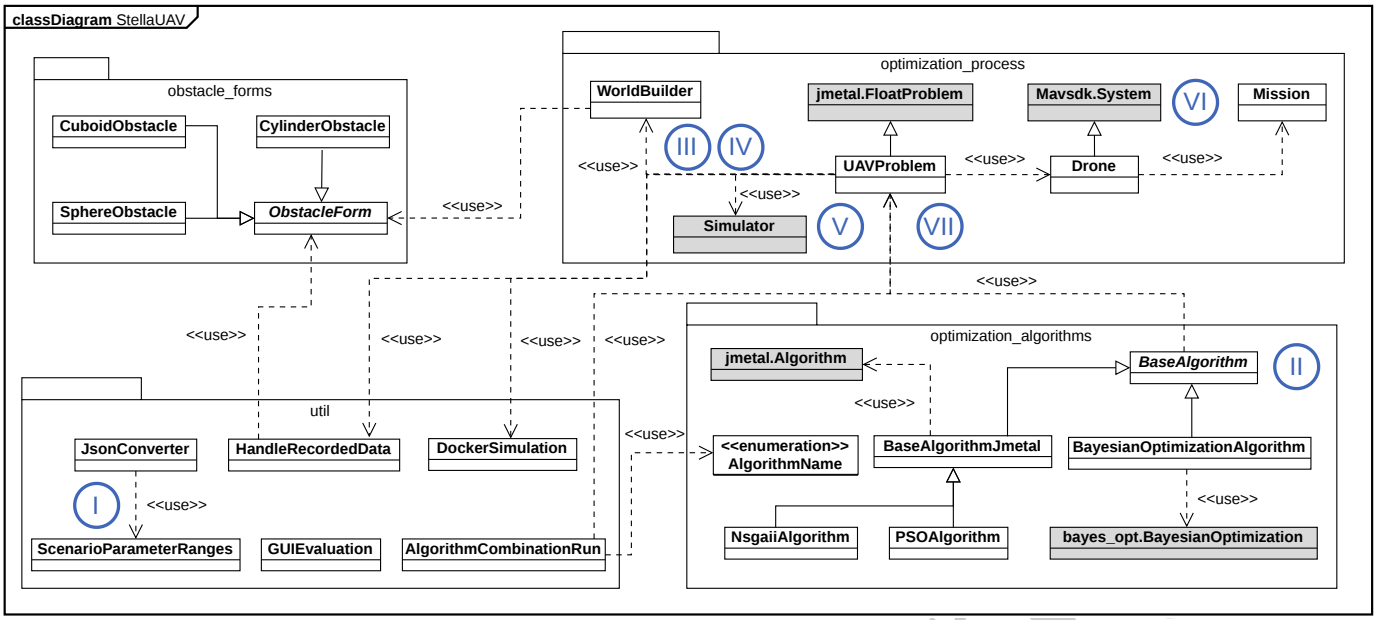


Fig. 6. The architecture of our tool StellaUAV for testing the safe behavior of UAVs. To visualize the connections of the implementation with the methodology, we additionally present steps from Fig. 5. Finally, we highlight third-party libraries and frameworks in gray.

sensors and plugins to enrich the features of the environment. We specify the simulation environment (IV) in Gazebo in a *.world* file. We generate this file in the WorldBuilder class of StellaUAV (III). Currently, StellaUAV includes the landform, the water level, the obstacles with their behavior, the wind force from one direction, and fog to reduce the visibility of the UAV in this file. To include additional elements in the simulation environment, we only need to extend the *adapt()* function of the WorldBuilder class. In the UAVProblem class, we define the search problem by specifying the search space and the fitness function. An optimization algorithm calls the *evaluate()* function of this class to simulate the UAV in the chosen concrete scenario and implemented simulator (V) and evaluate its fitness value (VII). The Drone class connects to the simulated UAV (VI) via MAVSDK to upload and start its mission. MAVSDK provides APIs for controlling and communicating with MAVLink systems, such as UAVs, by supplying them with missions or movement controls.

We implement the various optimization algorithms (II) used in StellaUAV in the *optimization_algorithms* package, in which all algorithm implementations inherit from the abstract BaseAlgorithm class. For the implementation of NSGAII and an enhanced version of PSO, called Speed-constraint Multi-objective PSO, we use the jmetalpy framework [31]. This framework provides implementations of various multi-objective optimization algorithms and for describing search problems that these algorithms evaluate. For the implementation of Bayesian Optimization (BO), we apply the BayesianOptimization framework [33], which presents an implementation of this algorithm that models the fitness function with a Gaussian process. We can easily extend the list of optimization algorithms by adding an implementation of the new algorithm that inherits from the BaseAlgorithm class and includes methods to define and run the chosen optimization algorithm.

In the *obstacle_forms* package, StellaUAV includes different forms of obstacles used in the simulations. As we currently use approximations of real-world obstacles, the tool provides implementations of cuboid, spherical, and cylindrical obstacles that inherit from the abstract ObstacleForm class. To add new obstacle forms, we can extend this ObstacleForm class by implementing methods for computing the distance to these new obstacle forms and updating their position.

Finally, StellaUAV includes utility classes in the *util* package. This package incorporates a class for converting the JSON files we use to describe specific logical scenarios (I) into parameter ranges, which specify the search space. It further contains the class HandleRecordedData that assesses the UAV's behavior in the simulation by inspecting the recorded data of the simulation and computing the SUT's distance to the obstacles during the flight. The DockerSimulation class runs the simulation in a Docker container and copies the log files to the host system. The AlgorithmCombinationRun class implements sequential combinations of the optimization algorithms. Finally, the class GUIEvaluation implements a GUI for the parameter selection of the tool, including the logical scenarios to evaluate, the maximum number of evaluations to perform per optimization algorithm, the size parameter of the algorithm, and the optimization algorithms to use for the evaluation of the safe behavior of the SUT.

Limitations and Extensibility. StellaUAV currently uses the open-source simulator Gazebo to represent the simulation world. The capabilities of this simulator limit the environmental effects we can depict in the tested logical scenarios. Gazebo

can display different landforms, surface natures, obstacles, wind forces, and various degrees of fogginess. However, it lacks the possibility to model, e.g., rain, ambient temperature, or cloudiness. To extend StellaUAV with another simulator, we need to adapt the step in which we build the simulation world in the class WorldBuilder (III) and launch this other simulator instead of Gazebo in the roslaunch files. StellaUAV tests the safe behavior of UAVs by generating "good" test cases for the SUT (VI). As a prerequisite, the tested UAV must provide the possibility to communicate via the Robot Operating System (ROS) [34] and MAVSDK to use StellaUAV. To change the SUT in StellaUAV, we need to adapt the roslaunch files to launch the new SUT and the script for running the simulation of this new SUT. Note that no further changes to the source code are necessary to apply another SUT. In our experiments, we test the PX4 autopilot [35], which we explain in more detail in Section V-A. Note that we demonstrate our methodology with Software-in-the-Loop testing in this work but can also apply it for Model-in-the-Loop or Hardware-in-the-Loop testing. Finally, note that the user needs to specify the logical scenarios in which they aim to test the SUT and the optimization algorithm with its size parameter and a maximum number of evaluations to search for worst-case situations.

V. Evaluation

In this section, we present logical scenarios for testing the safe behavior of the PX4 autopilot [35] in a Software-in-the-Loop simulation and generate "good" test cases for them with StellaUAV and various optimization algorithms. We provide the source code for our tool StellaUAV in [36]. This source code also allows replicating the presented experiments in this section.

A. System Under Test

In our experiments, we investigate the safe behavior of the obstacle detection and avoidance extension of the open-source PX4 autopilot [35] in StellaUAV. Note that a large open-source community works on this autopilot and that the authors of this work did not implement it by themselves. The used version of the PX4 autopilot autonomously plans its trajectory after receiving a mission such as flying to a specified waypoint. When encountering static or dynamic obstacles, the autopilot can adapt its trajectory to avoid these obstacles. In addition, it adjusts its planned trajectory based on other environmental effects such as moderate wind conditions. Finally, the PX4 autopilot tries to keep a defined safety distance from all obstacles, which is set to 1.0 meters in its specification. When assessing the UAV's safe behavior, StellaUAV executes the PX4 autopilot software, simulates its environment with Gazebo in the derived concrete scenarios, and analyzes the recorded simulation data. Finally, we run each simulation in a Docker container to enable the parallel execution of several simulations and decrease unwanted side effects.

B. Logical Scenarios

Before we apply optimization algorithms to generate "good" test cases for testing the safe behavior of UAVs, we need to define the logical scenarios in which we aim to evaluate the behavior of our SUT. As described in Section II, we derive these logical scenarios manually from literature, specifications, and experts' knowledge of challenging situations; or automatically by, e.g., clustering real-world data. Further, we emphasize the need for a system-specific definition of the relevant parameters P and corresponding value ranges of these logical scenarios in Section III-A. In the experiments described in this paper, we manually derive logical scenarios from experts' knowledge and literature that describe the UAV's environment. To this end, we first collect relevant parameters and all reasonable value ranges, e.g., for the parameter *wind*, we gather the value ranges *none*, *light*, *moderate*, and *strong* that represent appropriate value ranges for our SUT. For example, light wind conditions might be winds of $1.0 - 4.0$ kilometers per hour. We use the following relevant parameters and value ranges in our experiments: landform (flat, depression, elevation, steep transition), surface nature (land, water, mixture), obstacle kinds (static, dynamic), obstacle sizes (small, medium, large), obstacle shapes (cuboid, sphere, cylinder), wind force (none, light, moderate, strong), and reduced visibility (none, fog, heavy fog, thick fog). We are aware that these characteristics do not represent all relevant parameters for testing the safe behavior of UAVs but that we need to limit ourselves to these due to the limitations of our simulation setup, as discussed in Section IV.

After collecting relevant parameters and their reasonable value ranges, we can select specific logical scenarios based on a defect hypothesis about what constitutes challenging situations for our SUT. However, defining such a defect hypothesis is still an open research challenge. For simplicity of presentation and the purpose of this paper, we select specific logical scenarios for our experiments based on the simple defect hypothesis that a pair-wise combination of the value ranges [37] is sufficient to provoke all relevant failures in the SUT. However, we are aware that we might miss faults in the SUT if this defect hypothesis is inadequate and, e.g., several faults only occur when combining the value ranges in a triple-wise manner. Table III presents the resulting logical scenarios for testing the safe behavior of UAVs in our experiments. Note that these logical scenarios solely represent situations based on the used defect hypothesis and, thus, do not represent a complete list of relevant logical scenarios for testing our SUT. For manually finding such a comprehensive list, we need to ensure that we know all parameters of relevant situations for the SUT and apply a correct defect hypothesis that presents all challenging situations for the SUT.

C. "Good" Test Cases

When generating "good" test cases for testing the safe behavior of UAVs for the provided logical scenarios, we need to define the search space and the fitness function. The parameters P of each logical scenario present the dimensions

TABLE III

LOGICAL SCENARIOS FOR TESTING THE SAFE BEHAVIOR OF UAVS THAT FOCUS ON THE ENVIRONMENT-RELATED DIMENSIONS. THE KIND OF THE OBSTACLES O IS EITHER STATIC ST OR DYNAMIC DY, THEIR SIZE IS EITHER SMALL S, MEDIUM M, OR LARGE L, AND THEIR FORM IS EITHER A CUBOID CU, A SPHERE SP, OR A CYLINDER CY.

#	Landform	Nature	# O	Obstacle Kinds				Obstacle Sizes				Obstacle Forms				Wind	Red. Visibility
1	flat	mixture	4	DY	DY	DY	DY	S	S	S	S	CY	SP	CY	CY	strong	thick fog
2	depression	land	3	ST	ST	ST	–	L	M	M	–	SP	CY	CU	–	light	heavy fog
3	elevation	water	1	DY	–	–	–	M	–	–	–	CU	–	–	–	moderate	fog
4	steep transition	mixture	4	ST	DY	ST	ST	M	L	L	M	CU	CU	SP	CU	none	none
5	flat	land	2	ST	ST	–	–	S	L	–	–	CY	CU	–	–	strong	none
6	elevation	water	4	DY	ST	DY	ST	L	M	L	L	SP	SP	CY	SP	none	fog
7	depression	water	4	DY	DY	DY	DY	M	L	M	M	CY	CY	CU	SP	moderate	heavy fog
8	steep transition	land	4	ST	ST	ST	DY	L	S	S	L	CU	CY	SP	CY	moderate	thick fog
9	flat	water	4	DY	DY	ST	DY	S	M	M	S	SP	CU	SP	CU	light	fog
10	flat	mixture	1	ST	–	–	–	L	–	–	–	SP	–	–	–	none	thick fog
11	steep transition	land	1	DY	–	–	–	S	–	–	–	CY	–	–	–	strong	heavy fog
12	depression	land	4	ST	ST	DY	ST	M	S	L	S	CU	SP	CU	CU	strong	heavy fog
13	elevation	mixture	4	ST	ST	ST	ST	S	M	S	M	CY	CY	CU	CY	light	fog
14	steep transition	water	3	ST	DY	DY	–	M	S	S	–	SP	SP	CY	–	moderate	none
15	flat	land	4	ST	DY	DY	ST	M	L	M	L	CU	CU	CY	SP	light	thick fog
16	depression	mixture	1	DY	–	–	–	S	–	–	–	CU	–	–	–	light	none
17	depression	land	2	DY	DY	–	–	L	S	–	–	CY	CY	–	–	none	fog
18	depression	water	4	ST	ST	ST	DY	L	L	L	M	SP	CY	CY	CY	strong	none
19	elevation	mixture	4	DY	DY	DY	ST	S	S	M	L	CY	CU	SP	CU	moderate	heavy fog
20	elevation	land	4	ST	DY	ST	DY	L	L	S	S	CY	CU	CU	SP	none	none
21	flat	mixture	3	DY	ST	ST	–	S	L	L	–	CY	SP	SP	–	strong	fog
22	steep transition	water	2	ST	ST	–	–	M	M	–	–	CU	SP	–	–	light	thick fog
23	flat	land	4	DY	ST	DY	ST	S	S	M	M	SP	SP	CY	CY	none	heavy fog
24	steep transition	mixture	4	DY	ST	DY	DY	S	M	M	L	SP	SP	CU	SP	strong	none
25	flat	land	4	ST	DY	DY	DY	L	M	L	S	CU	CY	CU	CU	moderate	thick fog
26	depression	water	4	ST	ST	DY	ST	S	S	L	M	CY	CU	SP	SP	light	thick fog
27	depression	land	4	DY	DY	DY	DY	S	M	S	L	CY	SP	CY	CU	light	heavy fog
28	elevation	mixture	2	DY	ST	–	–	S	M	–	–	SP	SP	–	–	moderate	thick fog
29	elevation	water	3	DY	ST	DY	–	L	S	S	–	CU	CU	CY	–	strong	thick fog
30	steep transition	water	4	DY	DY	DY	DY	M	L	M	S	CU	CU	CY	CY	light	fog
31	flat	land	2	DY	DY	–	–	M	M	–	–	SP	CY	–	–	moderate	heavy fog
32	steep transition	mixture	3	DY	ST	ST	–	L	L	S	–	SP	SP	CU	–	none	none

of our search space. For the fitness function, let us assume the existence of regulations that allow us to specify a safety distance that the UAV should keep to all obstacles. To test the safe behavior of our SUT, we then use the fitness function described in Equation (1) above. That function minimizes the remaining distance between the SUT and any obstacles before violating the specified safety distance. For simplicity's sake, we use a fixed safety distance $s(cs, t) = 1.0$ meters in our experiments, which is the default value of the corresponding parameter in the PX4 autopilot. Using this fitness function as an optimization goal, we search for challenging situations for the UAV in which the UAV comes close to obstacles. If the fitness value of a concrete scenario is negative, we know that the UAV behaved unsafely and violated the specified safety distance.

1) Experimental Setup: In these first experiments, we aim to show the general applicability of StellaUAV for generating "good" test cases. Thus, we first restrict ourselves to the widely used optimization algorithm NSGAII [6], [9], [38] and perform only three repetitions of our experiments. We will, however, present additional experimental results in the subsequent subsection for all optimization algorithms of StellaUAV. We use the implementation of the jMetalPy framework for NSGAII and search for "good" test cases for 500 evaluations for each logical scenario, as proposed in [39]. Following the suggestion of the authors of [8], we use the default settings of jmetalpy and utilize SBX Crossover, Binary Tournament Selection, and polynomial mutation operators. We apply a crossover rate of 0.9 and a mutation rate of $1/NV$ with NV denoting the number of variables in the search space to find the best candidates for the subsequent population after evaluating the 100 candidates of each population. Since the applied optimization algorithm uses a heuristic search for finding "good" test cases, the results are not deterministic. Therefore, we repeat our search for "good" test cases several times for each logical scenario to gain more reliable results, as proposed by the authors of [40], [41]. Finally, note that the dynamic obstacles in our experiments perform only the simple maneuver of moving in a straight line between two points. We are currently working on expanding our tool to present options for more complex trajectories of dynamic obstacles.

2) Results: Table IV displays the results of the experiments for generating "good" test cases for testing the safe behavior of UAVs. We repeat our search for "good" test cases three times for each logical scenario. We have set the optimization algorithm to create 500 concrete scenario instances in each run, and each concrete scenario represents one test case. The table shows the number of test cases in which the SUT violates

TABLE IV
THE NUMBER OF SAFETY DISTANCE VIOLATIONS THAT WE DISCOVER IN OUR EXPERIMENTS FOR EACH LOGICAL SCENARIO IN THREE RUNS. FURTHER, WE DEPICT THE AVERAGE AND MEDIAN VALUE OF ALL RUNS.

Scenario	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Run 1	104	0	97	12	0	112	18	34	209	0	37	9	0	10	128	7
Run 2	116	0	69	22	0	163	7	25	222	0	31	18	0	12	124	18
Run 3	61	0	16	5	0	151	14	25	234	0	31	2	0	14	133	37
Average	94	0	61	13	0	142	13	28	222	0	33	10	0	12	128	21
Median	104	0	69	12	0	151	14	25	222	0	31	9	0	12	128	18
Scenario	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**
Run 1	58	6	198	153	23	0	177	24	123	36	12	167	61	38	229	126
Run 2	17	3	211	180	1	0	177	39	185	132	8	180	53	42	260	140
Run 3	50	1	169	135	10	0	143	10	149	83	6	160	12	20	239	111
Average	42	3	193	156	11	0	166	24	152	84	9	169	42	33	243	126
Median	50	3	198	153	10	0	177	24	149	83	8	167	53	38	239	126

the specified safety distance of 1.0 meters for each logical scenario and each of these three runs.

3) Discussion: In our search for "good" test cases, we create challenging situations for the SUT in which a safely behaving UAV still keeps the specified safety distance. In our experiments, the SUT behaves safely in this way in five logical scenarios and violates the specified safety distance in the remaining twenty-seven. An inspection of the challenging situations that force the UAV to exhibit unsafe behavior reveals that all these situations include dynamic obstacles. Symmetrically, those situations in which the SUT exhibits safe behavior contain only static obstacles. These observations indicate that the SUT can handle situations with only static obstacles rather well, while it has problems when encountering dynamic ones. Finally, the experimental results show the fault detection effectiveness of our tool StellaUAV.

4) Threats to Validity: In our experiments, we generate "good" test cases for one system and test its safe behavior in logical scenarios relevant to this system. Thus, the results might not generalize to other UAVs and their relevant logical scenarios. To decrease threats to external validity, we test the SUT in 32 different logical scenarios that represent various challenging situations for the SUT. However, we do not claim to present a complete list of relevant logical scenarios with the presented ones. To decrease the bias of our experiments, we base our tool on existing libraries and test the safe behavior of an open-source UAV instead of implementing our own. Finally, we run all simulations of the UAV in isolated Docker containers to reduce unwanted side effects.

D. Optimization Algorithms

In the above experiments, we discovered that the tested PX4 autopilot shows a presumably safe behavior for logical scenarios with only static obstacles. However, as the applied heuristic optimization algorithm might not find the global optimum, i.e., the worst-case situation for a given logical scenario, we need to investigate the performance of various algorithms for testing the safe behavior of UAVs in these logical scenarios. In a further set of experiments, we thus, focus on five logical scenarios with only static obstacles. We use logical scenarios 2, 5, 13, and 22 from Table III and one with one static cuboid obstacle in a depression landform with moderate wind and fog in our experiments to depict a decent mix of various environmental effects in the evaluated logical scenarios with only static obstacles.

1) Experimental Setup: We evaluate the performance of the optimization algorithms and their sequential combinations mentioned in Section III-C: NSGAII (abbreviates as N), PSO (P), and BO (B) alone; then, the combinations NPN, NPB, NBN, NBP, PNP, PNB, PBN, PBP, BNP, BNB, BPN, BPB. We follow the suggestion of [39] and run 500 evaluations for each evaluated algorithm, as in the previous experiments. We set the population size of NSGAII, the swarm size of PSO, and the size of the set of initial candidates for BO to an equal value of 50 to enable a fair comparison. As we aim to investigate the performance of various optimization algorithms in this second set of experiments, we execute more than three runs this time to more reliably explore their performance, as proposed by the authors of [40], [41]. Note that even though we perform fifteen runs for each algorithm and logical scenario, we do not execute 30 runs as proposed by the authors of [40] due to limited computation time and power: 15 algorithm combinations times 5 logical scenarios times 500 evaluations times 2 minutes for each simulation results in 52 days per run. In our experiments, we focus on the median of the detected best fitness values over all runs for each optimization algorithm, which depicts the average performance of the algorithms and a common objective for such evaluations [42], [43]. Thus, an optimization algorithm with a lower median value of the found minimal fitness values overall runs shows a better performance than one with a higher median value.

2) Results: In Table V, we present the performance of the optimization algorithms compared to NSGAII, which literature widely applies and has shown insightful results when generating "good" test cases for various autonomous systems [6], [8], [9], [38]. To understand the significance of these results, we also provide the results of a Mann-Whitney U test and Vargha and Delaney's A_{12} measure [44]. Two algorithms perform significantly differently for a 95% confidence interval if the resulting P-value of a Mann-Whitney U test is smaller than 0.05. The authors of [44] further present guidelines for interpreting the effect size of their A_{12} measure by classifying

it as negligible, small, medium, or large depending on the reached value.

3) Discussion: The combination PNP shows the best performance with an increase of 20% compared to NSGAII, which is significant for a 95% confidence interval and has a small effect size. Thus, the results of our experiments indicate that the sequential combination PNP is a reasonable choice for reliably generating "good" test cases for testing the safe behavior of our SUT. With PNP, we even discover a safety distance violation in a logical scenario with only static obstacles that NSGAII did not detect. Thus, we found a worst-case situation with PNP in which the SUT behaves *unsafely*, whereas NSGAII only discovered challenging situations in which the SUT behaved *safely*, which suggested that the SUT behaves safely in this logical scenario. Furthermore, even though the algorithm combination PNP performed best in our case study, this still does not ensure finding all worst-case situations. Indeed, when inspecting the discovered worst-case situations of the algorithms during all runs, we noticed that PNP still misses several worst-case situations detected by other optimization algorithms for some of the logical scenarios. This finding indicates that we need to perform multiple runs *and* execute multiple optimization algorithms and combinations for all logical scenarios when testing the safe behavior of autonomous systems with scenario-based testing to gain reliable results. Note that this crucial problem is not limited to testing UAVs but also applies to other autonomous systems. We feel that this insight and its consequences are currently not considered in the scenario-based testing community.

4) Threats to Validity: In our experiments, we assess the performance of different optimization algorithms for creating worst-case situations for the PX4 autopilot only. Thus, the resulting performance values of the algorithms might not generalize to other UAVs. Nevertheless, we present the general problem of finding all worst-case situations when using these algorithms for detecting "good" test cases for autonomous systems. In addition, we perform several runs to gain more robust results, even though we do not perform 30 runs, as proposed by the authors of [40], due to limited computation time and power. To decrease the bias of our experiments, StellaUAV uses existing open-source libraries to execute the optimization algorithms and tests an open-source UAV. In addition, StellaUAV runs all simulations in Docker containers to reduce the threats from confounding variables.

VI. Related Work

Several frameworks for testing autonomous systems have been described in the literature. The authors of [10] and [11] present frameworks that focus on testing the reliability of UAVs and path planning algorithms with metamorphic testing. The authors of [12] focus on generating test cases for testing the reliability of cyber-physical systems and minimizing the resulting test suites based on their cost, effectiveness, and uncertainty. In their test case creation, they consider the uncertainty of cyber-physical systems and base their process on belief state machines. In contrast to their works, our framework focuses on testing *the safe behavior of UAVs* as another essential aspect of testing these autonomous systems. In the automotive domain, the authors of [13] and [14] present tools for testing the safe behavior of autonomous cars that use fuzzing techniques to find safety distance violations or to cover relevant behaviors of the SUT. Since different situations are challenging for autonomous cars and UAVs, a direct application of the methodology and tools from the automotive domain for testing the safe behavior of UAVs is not possible. In [15], the authors use metamorphic and model-based testing to evaluate their self-implemented AI controller for UAVs. Their framework concentrates on assessing the stability of the UAV's behaviors and presents one methodology for testing the safe behavior of UAVs. However, it requires the definition of correct and complete metamorphic relations, which is not easily accomplished. Thus, we decided to instead focus on scenario-based testing and finding worst-case situations for relevant logical scenarios, as the definition of these logical scenarios is more intuitive and more often performed. The authors of [16] focus on developing a UAV simulation model and a Hardware-in-the-Loop simulation test platform to ensure the credibility of their simulation. They apply fault injection to test the safe behavior of their UAV autopilot system. However, such fault-injection techniques assume that we already know all relevant faults and only need to detect them if they are present in our tested systems. In contrast to their work, we aim to find additional faults that we do not know beforehand by searching for challenging situations for the SUT.

In sum, tools and frameworks in the literature address testing the reliability of UAVs, testing autonomous cars, and testing the safe behavior of UAVs with metamorphic testing or fault-injection techniques. In contrast, our tool allows us to assess the safe behavior of UAVs as another crucial aspect of testing UAVs and apply scenario-based testing to test typical test situations.

VII. Conclusion

When testing the safe behavior of UAVs with scenario-based testing, we face the following challenges: (A) defining logical scenarios in which we aim to test the behavior of our SUT, (B) generating "good" test cases that can reveal potential faults in the SUT, and (C) evaluating the quality of the generated test cases. In this work, we present a tool, StellaUAV, for testing the safe behavior of an autonomously operating open-source UAV with scenario-based testing that tackles the presented challenges. StellaUAV provides the option to specify various relevant logical scenarios for the SUT in a JSON format, including different landforms, surface natures, obstacles, wind forces, and reduced visibilities. In addition, StellaUAV uses optimization algorithms to generate "good" test cases that represent worst-case situations for the SUT. Finally, the tool offers the possibility to execute various optimization algorithms and their sequential combinations to enable an evaluation of the quality of their generated test cases.

To demonstrate the applicability of the tool, we present derived logical scenarios for our SUT that are selected on

TABLE V

PERFORMANCE OF THE OPTIMIZATION ALGORITHMS COMPARED TO NSGAII, WHERE POSITIVE VALUES INDICATE BETTER PERFORMANCE. BELOW, WE PRESENT THE P-VALUE OF A MANN-WHITNEY-U TEST FOR THIS COMPARISON AND ITS VARGHA AND DELANEY'S A_{12} MEASURE. WE HIGHLIGHT SIGNIFICANT DIFFERENCES FOR A 95% CONFIDENCE INTERVAL IN THE P-VALUES AND ALL EFFECT SIZES FOR THE A_{12} MEASURE THAT ARE NOT NEGLIGIBLE.

	NSGAII	**PSO**	**BO**	**NPN**	**NPB**	**NBN**	**NBP**	**PNP**	**PNB**	**PBN**	**PBP**	**BNP**	**BNB**	**BPN**	**BPB**
Performance	Base	-2%	-21%	+10%	-12%	+12%	+5%	+20%	+5%	+16%	+9%	+11%	-5%	+8%	-11%
P-value	Base	0.20	**0.00**	0.17	0.34	0.40	0.48	**0.01**	0.32	0.11	0.17	0.22	0.26	0.33	**0.04**
A_{12}	Base	0.46	**0.26**	0.55	0.49	0.52	0.51	**0.57**	0.53	**0.56**	0.54	0.51	0.44	0.50	**0.39**

the simple defect hypothesis that a pair-wise combination of dimensions is sufficient to provoke all relevant failures in the SUT. We chose this defect hypothesis for simplicity of presentation and are aware that it does not necessarily represent an adequate defect hypothesis for all relevant logical scenarios related to testing the safe behavior of UAVs. For each of the derived logical scenarios, we generate worst-case situations with our tool StellaUAV. In our experiments, we discover various safety distance violations of the given SUT for all logical scenarios that include dynamic obstacles. These results indicate that the SUT can handle logical scenarios with static obstacles rather well, while it encounters problems when facing dynamic ones. As the optimization algorithms are heuristics, we evaluate the performance of various optimization algorithms for several logical scenarios. These experiments indicate that one of the combinations performs 20% better than NSGAII and that we need to execute multiple optimization algorithms to increase the chance of finding worst-case situations. In fact, a combination of algorithms found safety distance violations that NSGAII alone did not detect. Finally, the experiments show how StellaUAV effectively generates "good" test cases that can reveal potential faults in the SUT.

In future work, we will investigate more complex defect hypotheses for deriving logical scenarios for our SUT that describe all challenging situations for this system. In addition, we aim to extend our tool to incorporate further types of obstacles, more environmental effects, and more complex maneuvers for the dynamic obstacles to investigate the UAV's behavior more thoroughly.

REFERENCES

[1] P. Doherty and P. Rudol, "A uav search and rescue scenario with human body detection and geolocalization," in *Australasian Joint Conference on Artificial Intelligence*. Springer, 2007, pp. 1–13.

[2] H. Shakhatreh *et al.*, "Unmanned aerial vehicles (uavs): A survey on civil applications and key research challenges," *Ieee Access*, vol. 7, pp. 48 572–48 634, 2019.

[3] A. R. Vetrella, G. Fasano, A. Renga, and D. Accardo, "Cooperative uav navigation based on distributed multi-antenna gnss, vision, and mems sensors," in *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2015, pp. 1128–1137.

[4] J. W. Rosen, "Zipline's ambitious medical drone delivery in africa," *MIT Technology Review, June*, vol. 8, p. 2017, 2017.

[5] J. Cem Kaner, "An introduction to scenario testing," *Florida Institute of Technology, Melbourne*, pp. 1–13, 2013.

[6] F. Hauer, A. Pretschner, and B. Holzmüller, "Fitness functions for testing automated and autonomous driving systems," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2019, pp. 69–84.

[7] J. Wegener and O. Bühler, "Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system," in *Genetic and Evolutionary Computation Conference*. Springer, 2004, pp. 1400–1412.

[8] A. Calò *et al.*, "Generating avoidable collision scenarios for testing autonomous driving systems," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 375–386.

[9] T. Schmidt, F. Hauer, and A. Pretschner, "Understanding safety for unmanned aerial vehicles in urban environments," in *2021 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2021, pp. 638–643.

[10] R. Li *et al.*, "Metamorphic testing on multi-module uav systems," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1171–1173.

[11] J. Zhang *et al.*, "Testing graph searching based path planning algorithms by metamorphic testing," in *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2019, pp. 158–167.

[12] M. Zhang, S. Ali, and T. Yue, "Uncertainty-wise test case generation and minimization for cyber-physical systems," *Journal of Systems and Software*, vol. 153, pp. 1–21, 2019.

[13] G. Li *et al.*, "Av-fuzzer: Finding safety violations in autonomous driving systems," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 25–36.

[14] R. Majumdar *et al.*, "Paracosm: A test framework for autonomous driving simulations," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, Cham, 2021, pp. 172–195.

[15] M. Lindvall, A. Porter, G. Magnusson, and C. Schulze, "Metamorphic model-based testing of autonomous systems," in *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*. IEEE, 2017, pp. 35–41.

[16] X. Dai, C. Ke, Q. Quan, and K.-Y. Cai, "Rflysim: Automatic test platform for uav autopilot systems with fpga-based hardware-in-the-loop simulations," *Aerospace Science and Technology*, vol. 114, p. 106727, 2021.

[17] T. Menzel, G. Bagschik, and M. Maurer, "Scenarios for development, test and validation of automated vehicles," in *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2018, pp. 1821–1827.

[18] E. de Gelder, J.-P. Paardekooper, O. Op den Camp, and B. De Schutter, "Safety assessment of automated vehicles: how to determine whether we have collected enough field data?" *Traffic injury prevention*, vol. 20, no. sup1, pp. S162–S170, 2019.

[19] F. Hauer, T. Schmidt, B. Holzmüller, and A. Pretschner, "Did we test all scenarios for automated and autonomous driving systems?" in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. IEEE, 2019, pp. 2950–2955.

[20] W. Wang, C. Liu, and D. Zhao, "How much data are enough? a statistical approach with case study on longitudinal driving behavior," *IEEE Transactions on Intelligent Vehicles*, vol. 2, no. 2, pp. 85–98, 2017.

[21] G. Bagschik, T. Menzel, and M. Maurer, "Ontology based scene creation for the development of automated vehicles," in *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2018, pp. 1813–1820.

[22] F. Hauer, I. Gerostathopoulos, T. Schmidt, and A. Pretschner, "Clustering traffic scenarios using mental models as little as possible," in *2020 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2020, pp. 1007–1012.

[23] A. Pretschner, "Defect-based testing." *Dependable Software Systems Engineering*, vol. 84, 2015.

[24] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[25] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-International Conference on Neural Networks*, vol. 4. IEEE, 1995, pp. 1942–1948.
[26] J. Puchinger and G. Raidl, "Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification," in *International Work-Conference on the Interplay Between Natural and Artificial Computation*. Springer, 2005, pp. 41–53.
[27] C. Aranha and H. Iba, "Modelling cost into a genetic algorithm-based portfolio optimization system by seeding and objective sharing," in *2007 IEEE Congress on Evolutionary Computation*. IEEE, 2007, pp. 196–203.
[28] T. Chen, M. Li, and X. Yao, "On the effects of seeding strategies: a case for search-based multi-objective service composition," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018, pp. 1419–1426.
[29] G. Fraser and A. Arcuri, "The seed is strong: Seeding strategies in search-based software testing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 121–130.
[30] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *International Conference on Intelligent Robots and Systems*, vol. 3. IEEE, 2004, pp. 2149–2154.
[31] A. Benítez-Hidalgo *et al.*, "jmetalpy: a python framework for multi-objective optimization with metaheuristics," *Swarm and Evolutionary Computation*, vol. 51, p. 100598, 2019.
[32] J. Oes, "Mavsdk-python," 2019. [Online]. Available: https://github.com/mavlink/MAVSDK-Python
[33] F. Nogueira, "Bayesian Optimization: Open source constrained global optimization tool for Python," 2014. [Online]. Available: https://github.com/fmfn/BayesianOptimization
[34] M. Quigley *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5.
[35] L. Meier, D. Honegger, and M. Pollefeys, "Px4: a node-based multithreaded open source robotics framework for deeply embedded platforms," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 6235–6240.
[36] T. Schmidt and A. Pretschner, "Supplementary material for 'StellaUAV: A Tool for Testing the Safe Behavior of UAVs with Scenario-Based Testing'," 2022. [Online]. Available: https://doi.org/10.6084/m9.figshare.19666311
[37] S. R. Dalal *et al.*, "Model-based testing in practice," in *Proceedings of the 21st international conference on Software engineering*, 1999, pp. 285–294.
[38] X. Zou, R. Alexander, and J. McDermid, "Testing method for multi-uav conflict resolution using agent-based simulation and multi-objective search," *Journal of Aerospace Information Systems*, vol. 13, no. 5, pp. 191–203, 2016.
[39] F. Klück, M. Zimmermann, F. Wotawa, and M. Nica, "Performance comparison of two search-based testing strategies for adas system validation," in *IFIP International Conference on Testing Software and Systems*. Springer, 2019, pp. 140–156.
[40] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 1–10.
[41] V. Beiranvand, W. Hare, and Y. Lucet, "Best practices for comparing optimization algorithms," *Optimization and Engineering*, vol. 18, no. 4, pp. 815–848, 2017.
[42] M. H. Moradi and M. Abedini, "A combination of genetic algorithm and particle swarm optimization for optimal dg location and sizing in distribution systems," *International Journal of Electrical Power & Energy Systems*, vol. 34, no. 1, pp. 66–74, 2012.
[43] M. Song and D. Chen, "A comparison of three heuristic optimization algorithms for solving the multi-objective land allocation (mola) problem," *Annals of GIS*, vol. 24, no. 1, pp. 19–31, 2018.
[44] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.