Metamorphic Model-based Testing of Autonomous Systems

Mikael Lindvall
Fraunhofer CESE
5825 Univ. Research Ct
College Park, Maryland
mikli@fc-md.umd.edu

Adam Porter
Fraunhofer CESE
5825 Univ. Research Ct
College Park, Maryland
aporter@fc-md.umd.edu

Gudjon Magnusson
Fraunhofer CESE
5825 Univ. Research Ct
College Park, Maryland
GMagnusson@fc-md.umd.edu

Christoph Schulze
Fraunhofer CESE
5825 Univ. Research Ct
College Park, Maryland
cschulze@fc-md.umd.edu

***Abstract*— Testing becomes difficult when we cannot easily determine whether or not the system under test delivers the correct result. Autonomous systems are a case in point because it is difficult to determine whether a safety-critical autonomous system's behavior meets its specifications. To address the problem of testing autonomous drones, we have developed a framework for automated testing of a simulated autonomous drone system using metamorphic testing principles combined with model-based testing. Based on the results from using the framework to test the drone in the simulator using obstacles that do not move during flight, we have determined that this is a cost beneficial solution allowing for comprehensive testing without having to develop complex testing infrastructure to determine detailed test oracles. Our test cases are automatically generated from a set of testing models where each model encodes a certain scenario that can be varied according to metamorphic principles.**

Keywords—Model-based testing; autonomous systems; metamorphic testing.

I. INTRODUCTION

There is an increasing interest in using autonomous drones for various purposes such as surveillance, inspections, search and rescue operations, warfare, etc. Drone technology is relatively mature and one can readily buy and fly a drone without a license. Some drones already have built-in autonomous functions such as sense-and-avoid features that automatically detect obstacles and change course in order to avoid them. Other drones are not equipped with autonomous functionality, but provide control software that the user can modify to add different types of autonomous behaviors.

A great concern is, however, whether or not such autonomous drones are safe. Just because the drone behaves in a safe manner for a set of test scenarios does not mean that it will always behave in a safe manner for other scenarios. Thus, a strategy is needed for how to test autonomous systems for safety. However, autonomous systems are difficult to test in a systematic manner because it is difficult to determine whether an autonomous system behaves as expected. For example, an autonomous drone's mission might be to fly to a certain location without hitting any obstacles along the course, which can be used as a criterion to check that the drone achieved its mission successfully. However, for drones and other safety-critical systems - i.e. systems that can cause harm to human beings or can damage expensive equipment - it is also important to be able to predict and test behavior on a more detailed level. For example, given a certain configuration of obstacles, it is important to be able to predict what actions the autonomous drone will take other than avoiding the obstacles. It is also important to be able to determine that the autonomous system is stable in the sense that it behaves in essentially the same way for essentially the same situations. It is also important that it behaves in the exactly same way when exactly the same situation is repeated. Otherwise the system appears to be random and not stable. There are however many reasons such testing can be expensive. For example, there are endless numbers of ways to configure a course to fly and thus there is an endless need for different test scenarios and test cases. Each test scenario has to be created or found in the real world and then the drone has to physically fly each course, which takes time. In addition, drones can be expensive to buy and since they can get damaged and cause accidents during testing, such testing can get expensive. In addition, certain areas, such as Washington DC, are no-flight zones, which cause problems for testers of drones who reside in such a zone since they would have to travel to run tests.

In this paper, we describe how we are addressing the problem of testing autonomous drones. We started by developing a framework for automated testing of a simulated autonomous drone system using metamorphic testing principles combined with model based testing. The simulated environment serves as a complement to testing in the real world and provides a cost effective starting point for drone testing. In this simulated environment, we can configure the environment by adding obstacles, landing pads, etc. We can use different sense-and-avoid algorithms and we can evaluate which algorithm works best. We can run tests and collect performance data, which reduces cost and time. But probably most importantly, the framework allows us to test the autonomous system in a systematic fashion. This is important because by testing the system systematically, we can gain trust in it when we see that it behaves in a stable and predictable way in a wide range of testing scenarios.

Based on the results from using the framework to test the drone in the simulator, we have determined that using metamorphic testing and model-based testing in a simulated environment is a cost beneficial solution allowing for comprehensive testing without having to develop a complex testing infrastructure to determine the oracle. In our approach, test cases are automatically generated from a set of testing models where each model encodes a certain scenario that can be varied according to metamorphic principles.

II. TESTING IN THE DRONE SIMULATOR

The test framework consists of three components: the drone AI controller (which is the system under test (SUT)), the

978-1-5386-0424-3/17 $31.00 © 2017 IEEE
DOI 10.1109/MET.2017..6

simulation environment (which allows us to visualize the drone operating in a realistic environment and simulate all of its sensors), and the testing API (which allows us to run simulations, configure the system behavior and monitor system state).

In order to test and evaluate our testing strategy we needed to build a realistic implementation of an autonomous system that is simple enough to analyze in detail, but complicated enough to demonstrate the challenges that arise when testing and evaluating autonomous systems. For this purpose, we chose to implement a controller for a quad-rotor, commonly referred to as a drone, with artificial intelligence (AI) behavior that would allow the drone to deliver a package and return to the base without any human intervention. The AI behavior allows the drone to autonomously take off, navigate to a given location while avoiding obstacles and land on a landing pad. In order to create a realistic simulation, we have simulated the following sensors: the Inertial Measurement Unit (IMU), barometer, GPS, cameras, Light Detection and Ranging (lidar- a remote sensing method that uses light in the form of a pulsed laser to measure ranges), and ultrasonic range finder. For actuators, we implemented the simulation of motorized propellers; specifically, we used a quad-copter (four motors) configuration. Since multi-rotors are inherently unstable systems, the drone also requires a flight controller. The AI controller is designed to be agnostic to the vehicle flight controller. This allows us to test the behavior on different types of drones. In this project we have used three complementary flight controllers: ArduPilot Mega (APM - an open source flight controller that is used in a large number of commercially available drones), Parrot AR 2 (an off the shelf controller), and a custom controller called SimpleQuad that Fraunhofer has implemented. APM is the most sophisticated implementation and we use it as a baseline, but it is difficult to integrate with our testing framework. SimpleQuad is a simplified implementation that is optimized to allow us to automatically run a large number of test cases. The Parrot AR 2 is the controller of our physical drone, which we used to evaluate how closely the simulated environment corresponds to a physical drone flying in the real world.

A. The Simulation Environment

The simulation environment is our stand-in for the real world outside. It allows us to fly a drone and analyze its behavior without the risk of damaging expensive hardware, people and properties. It bypasses the need for permit for areas that require one to fly drones, and minimizes the set up time needed for field test. When building the simulator we recognized the requirement that it should be capable of producing realistic environment while at the same time provide flexibility to generate the desired scenarios.

The simulation environment is implemented using Unity, a popular game engine, which is used to create realistic and interactive 3D visualizations. Since the goal is to analyze the decisions made by the AI controller in a given scenario rather than the precise properties of the flight controller, our simulation focuses on sensor fidelity more than physics and aerodynamics. The simulator and the testing framework is decoupled so that our test generation and test monitor tools can be used with other simulation platforms such as Gazebo[9] and the Robot Operating System (ROS)[8]. The testing framework communicates with the simulator over a publish/subscribe message bus which facilitates adding new components independent of language or platform. It should be noted that simulator testing depends on the fidelity of the simulator. We are assuming, based on comparisons to the physical drone, that the simulator is accurate to an acceptable degree.

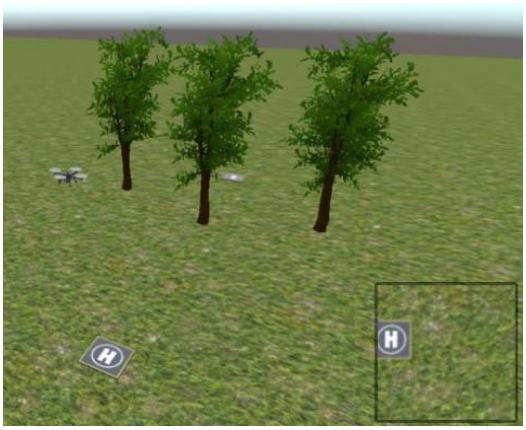

Figure 1 Screenshot from our simulation environment during a test case run

B. The Testing API

To the drone testing framework we added an API that allows us to write JUnit test cases that will spawn a new drone in the simulation environment and run a mission. Each test case defines what type of sensors should be used e.g. lidar or cameras only. The tester can override any of its parameters e.g. max speed, lidar range, camera resolution and much more. Test cases can also construct a specific environment by spawning any number of prefabricated objects such as trees, buildings, power lines, etc. Predefined or random environments can also be used. The test mission is defined as a list of high level commands, such as: ‘takeoff’, ‘fly to location’, ‘land on landing pad’ and more. When the test runs, the drone will execute these commands one after another. During and after the mission, properties related to the vehicle, such as location, velocity, distance from objects, can be verified.

Figure 2 shows a very simple test case written in Java. The test starts by defining a simple environment by placing a landing-pad and three trees to the west. Next it defines a sequence of high level commands that the drone should execute. This test case uses highly simplified helper functions to define the environment. These helper functions are designed to make it easy to model and automatically generate test cases. For more advanced test cases the environment can be parameterized in much greater detail.

```
@Override
public void executeTest() {
	setLandingpadW();
	addTreeW();
	addTreeNW();
	addTreeSW();

	arm();
	takeoff();
	flyToL1();
	precisionLand();
	disarm();
	hold();
	arm();
	takeoff();
	flyHome();
	precisionLand();
	disarm();
}
```

Figure 2 A simple example of a test case for our drone. These test cases are automatically generated and automatically executed.

C. Testing Strategy

Despite having a simulation model of an autonomous vehicle and its environment, it is difficult to run enough test scenarios to gain confidence in its ability to always behave in a safe manner. In addition, it can be difficult to define what correct behavior looks like. Autonomy requirements can be vaguely phrased e.g. "Navigate from A to B without colliding with obstacles". If the system performs well for one scenario, it is unclear how small changes to the scenario may affect the system behavior.

In order to address these difficulties, we defined a testing strategy especially devoted to testing of autonomous systems. Our testing strategy is based on model-based and metamorphic testing. Model-based testing allows us to generate a large number of test cases from a model of the system behavior eliminating the need to hand code test cases for every possible scenario we can think of. Metamorphic testing does not require the tester to specify the detailed expected outcome beforehand. Instead metamorphic testing compares a large number of equivalent scenarios for which the outcome should be equivalent based on some property. Any deviation from the group (say 99 equivalent scenarios of 100 result in the same behavior (e.g. turn to the left of the obstacle) while one does not (it turns to the right of the obstacle) spawns detailed analysis in order to understand why there were unexpected deviations from the majority pattern.

Since we are reasoning about safety for autonomous systems, a testing method for autonomous drones must include testing from a safety perspective. In order to ensure that basic safety requirements are met, we need to be able to test that safety features such as sense-and-avoid algorithms are correctly implemented and that they bring value by avoiding unsafe situations. In addition, we have experimented with a concept called safety cages for additional safety. A safety cage is a last resort that kicks in when an unsafe situation has been detected. For example, one of our safety cages stops the drone when it detects that the drone is too close to an obstacle. The idea comes from how safety related industrial robots have been handled. Instead of trying to guarantee that the industrial robot never would move in such a way that it would strike a human being, the robot was put in a cage. The drone cannot be put in a physical cage but instead we put it into a conceptual cage that puts limits on physical parameters such a proximity to an obstacle. The safety cage is implemented as an extra architectural component that is separate from the regular components. This component monitors sensor readings as well as commands sent from the autonomous "driver" to the actuator components. If the component implementing the safety cage determines that the commands to be sent to the actuator components will cause the drone to enter an unsafe state, then it blocks those commands and instead forces the drone to enter a safe state (e.g. stops).

In addition to proximity to obstacles, we have experimented with a safety cage that stops the drone from ascending further when it reaches a certain altitude (since current laws prevent drones from flying in the air space devoted to air planes). Another safety cage stops the drone from increasing velocity beyond a certain limit (since too high velocity might break the drone). The drone monitors and logs these three variables for later analysis.

D. Metamorphic Testing

Our metamorphic testing for autonomous systems works as follows. We start by defining a model of a scenario of interest and observe the system behavior given this scenario. Next we programmatically generate multiple variations of the original scenario based on metamorphic relations that we have identified. During testing we collect a large amount of data that allows us to compare the drone's behavior in different test runs.

E. Model-Based Testing

Model-based testing (MBT) is a test case design and test case generation technique for test automation. Instead of creating one test case at a time, the tester creates a model of the SUT and lets the computer automatically derive test cases by exploring the model. These test cases are automatically executed as part of the testing process. When a test case fails, a defect may have been detected. The model is designed based on the tester's understanding of the system's requirements and system exploration, for example, by using and experimenting with the system. The model is often expressed using states and transitions. For further information about our other studies of MBT, see [3][4][5]. For information about using MBT in combination with metamorphic testing see for example [1].

In the case of autonomous systems, the model is designed based on the specification of the high-level commands that are used to configure the environment with landing pads and non-moving obstacles as well as the instructions to the drone that defines the mission, e.g. fly to the landing pad and back while avoiding obstacles. In this context, each test case represents one such mission and the comparison of the expected output with the actual output would provide the oracle. However, how to determine the oracle automatically is a problem that will be discussed below.

F. Equivalences

The first step in our use of metamorphic testing is to identify equivalences that are used to create models. Equivalent scenarios are scenarios that are different in at least one aspect but since they are equivalent they should result in the same overall behavior. This means that due to the autonomous nature the behavior can be slightly different within a certain threshold. When comparing different test executions we take into account characteristics such as time to complete the mission, the shape of the path taken, distance from obstacles and more. For two runs to be considered equivalent these variables must match within a defined threshold. The following are a number of equivalences for the autonomous drone that we have identified.

1) Across runs: would the behavior always be the same if exactly the same scenarios was tested several times in a row or on different days or between reboots of the drone and the simulation platform.

2) Rotations: rotating the world geometry such that distances and relative positions are unchanged, should not affect mission performance or behavior. The drone should behave consistently no matter whether it's flying north or south.

3) Translation: translating the world geometry should not affect the overall behavior. The drone should behave the same way on the east and west-coast if the environment looks the same.

4) Obstacle location: after a certain threshold, it should not matter how far from the launch pad the first obstacle is positioned. If there is an obstacle on the drone's path, it should handle it the same way no matter if it is close to the start or end of the mission. Although this might change the overall path, locally around the obstacle we expect the same behavior.

5) Obstacle formations: There are many different ways to place obstacles in a formation. Examples are: random placement, straight line placement, arch placement place, etc. For each type of oracle formation, the behavior should be the same if the geometric equivalences described above are applied. For example, if the obstacles are placed randomly and the entire world is rotated 45 degrees, then the drone should still behave the same way. The same applies to cases when the obstacles are placed in a straight line, arch or other formation and the entire world is rotated or translated and the same test case is repeated immediately or on the next day.

G. Properties to Study

Comparing two test runs to determine if the behavior is equivalent is not trivial. Because of the nondeterministic nature of sensor noise and physical systems, directly comparing sensor values, timing or position is not useful. Instead we want to generalize to see if the overall behavior and high level decisions are equivalent. To do this we have identified a few properties that work well for comparison.

We have found that using the values collected for the safety cages make a good foundation for comparison. Thus we have been experimenting with comparing whether the drone in all equivalent scenarios violate the safety limits (proximity to obstacles, altitude, velocity) the same number of times. We are still working on the analysis of this particular approach.

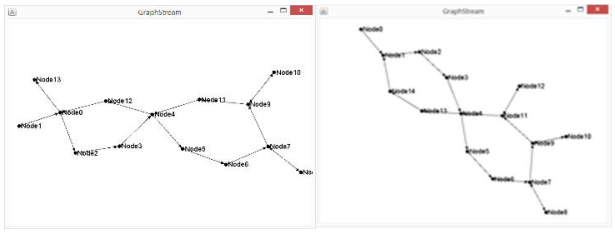

Figure 3 Two discretized state diagrams automatically generated from two test cases

Another idea is to compare the time the drone is in an unsafe state and formulate metamorphic relationships based on that. This would make sense since it would be permitted for the drone to be in an unsafe state for a short while, if it immediately takes an action to move into a safe state. This approach is also at an experimental stage at this time.

The general shape of the path taken by the drone is one property we expect to remain the same for equivalent test cases. After running a large number of test cases we can quickly flip through graphical representations of the flight path in test case and visually compare them to determine if one or more are notably different from the rest. This process can be automated by applying shape matching algorithms with some tolerance.

Another property we have considered is the drones traversal through a state space. Comparing sensor values directly does not work because of sensor noise and the inherent indeterminism of the physical world, but by tracking the trend of certain sensor values over time we can generate discretized states such as ascending, descending, moving, turning, etc. Figure 3 shows an example of two automatically generated state diagrams. The diagrams say nothing about the flight path or mission success but only that the two runs visited the same states in roughly the same order.

H. Generating Test Cases

A test case is composed of a test environment and a test mission. Both can be constructed by hand or generated automatically. The environment contains a list of primitive objects, (such as trees, landing-pads and buildings) and their location. A mission is a sequence of actions that should be executed one after another.

To generate a suite of test cases we first construct a single environment by hand. Next we generate a large number of variations of that environment using the transformations we mentioned before. Missions are constructed by hand to match the environment or can be generated using a model of valid action sequences.

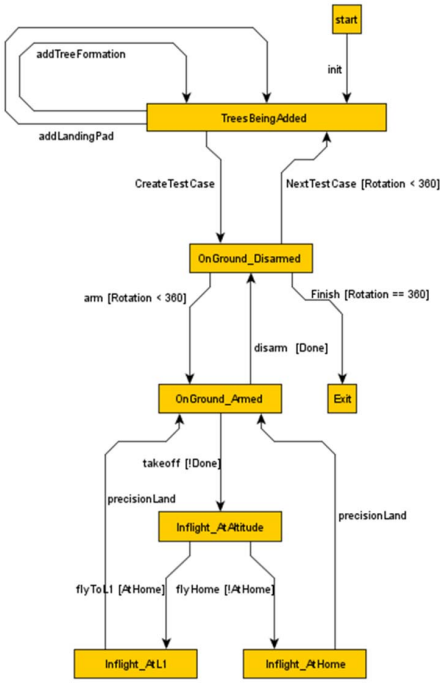

Figure 4 A simplified model for generation of a test suite of equivalent test cases based on rotation equivalence thus every test suite generated from this model is equivalent to every other test case generated from this model.

III. CASE STUDY

A. Test Scenario

For our case study, we defined a simple mission for a delivery drone. In order to complete the mission to deliver a package and return, the drone needs to take off from one landing pad (L1), navigate to another landing pad (L2) while avoiding trees (i.e. obstacles) along the course, land on L2 and then, using the same steps in reverse, fly back to L1. In this case the model of the scenario is the location of the two landing pads and the trees between them as well as the instructions to the drone to take off, land etc. A simplified sample model is provided in Figure 4.

Throughout this study we used the same basic mission but tried various different formations of obstacles and various transformations of the environment. The focus was to see how the environment affected the performance of the navigation and obstacle avoidance of the drone.

To generate variations of the test scenarios, we apply different metamorphic principles by altering the model. For example, we rotate the world by rotating the position of all objects around the origin. In our example L1 is located at the origin so it does not move, but L1 does not have to be fixed.

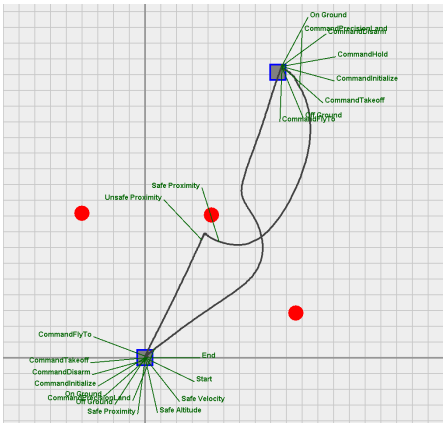

Figure 5 The visualization of one test case where the world has been rotated 25 degrees. The launch pad is represented by the filled square in the lower left corner while the landing pad is in the upper right corner. The drone took off and flew straight against the first obstacle (filled circles), then turned right to avoid it. After having landed, the drone flew back in a similar pattern. The annotations represent the state of the drone. Examples are: armed, ascending, safe, unsafe – too close to obstacle etc.

In order detect metamorphic deviations, we currently use a semi-automatic method that turned out to be very convenient. The results from each test case are recorded and automatically visualized so they are easy to inspect visually. Figure 5 shows a top-down visualization where the blue squares are the landing pads, red circles are the obstacles and the black line shows the path that the drone flew. Each grid line represents one meter in the simulation world. If we rotate the world 360 times, the result is 360 diagrams; Figure 6 shows three test cases representing the variant scenarios. We then automatically create an image sequence from the diagrams, in the order of rotation, that appears as a "video," which makes it easy to spot deviations and inconsistencies just by watching the video. The tester thus launches the test suite, which runs automatically overnight. The morning after, the tester watches the video (which takes 2 minutes to watch) and notes potential issues. The tester then analyzes each issue by analyzing the state transitions (lift off, unsafe proximity etc.), by analyzing the numerical data stored in the log file, and/or by rerunning a selected set of test cases potentially with some variations to better understand what might be the underlying reason (e.g. does the inconsistency persist if the landing pad is moved slightly?). The tester might also analyze the sense-and-avoid algorithm in use in order to shed some light on the root cause of the inconsistency. However, the task to fully debug and fix the issue is not the task of the tester but of the developer so the tester's analysis of the algorithm and associated code is typically relatively brief. Once the developer has fixed the issues, the tester checks that the issue has indeed been fixed by rerunning the test suite the next night, which might reveal other issues that were previously shadowed by the initial issue.

B. Findings

In one scenario we built a model based on three obstacles in an arch formation with the center obstacle in between the launch pad and the landing pad. We then generated test cases by rotating that scenario 360 times in one degree increment. We then compared the results across all degrees of rotation. Since

all testing scenario were equivalent, we expected that the drone's behavior would be exactly the same for each scenario. However, to our surprise, this was not the case. In the first two images in Figure 6 we see two virtually identical scenarios and thus we expect the drone's behavior to be identical. One scenario is rotated 25 degrees and the other 27 degrees. Despite the similarity, the drone picks very different paths; for rotations up to 25 degrees (left diagram), the drone flies to the right of the center obstacle both when flying to the landing pad and flying back. However, for 27 degrees rotation (right diagram), the drone decides to fly to the left side of the center obstacle. Both might be considered valid but we would call this behavior unstable. Unstable and unpredictable behavior might prevent certification and thus the drone developer must investigate and understand why this is the case. In the third image we see a corner case where the scenario failed; instead of avoiding the obstacle, the drone flew straight into it and crashed. This type of failure is highly unintuitive but can happen if the system is not carefully designed and is easily missed if it is not thoroughly tested in a systematic fashion. It turned out that the sense-and-avoid algorithm was sensitive for certain numerical values and therefore misbehaved. This finding allowed the drone developer to improve the algorithm making it more stable and safe. After the change, the drone passed all test cases and confidence in its capability to behave safely increased accordingly.

After running hundreds of test cases to cover different rotations of the environment we discovered another issue in addition to crashing for some scenarios. We noticed the drone had problems landing in some situations. After analyzing the situation carefully, it turned out that we rotated all the objects in the scene but did not rotate the direction of sunlight. This caused a shadow to fall on the landing-pad in some orientations and caused the vision system to fail to recognize the landing pad. This was not the type of problem we expected to find, but is certainly a good thing to catch. In subsequent test we used a more robust vision sensor that is less sensitive to changes in lighting but kept the sun where it was.

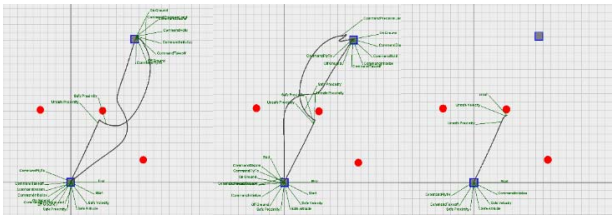

Figure 6 Three test cases rotated at a slightly different angle with very different outcome. Note that the annotations indicate the state of the system including whether it is safe or not at a certain point.

The same approach was used to test safety features implemented as safety cages described above. This case applies to the safety cage that stops the drone when it detects that the drone is too close to an obstacle. In Figure 6, the safety cage was obviously turned off. By comparing the autonomous behavior when safety cages were on or off, it was established that the safety cages did indeed save the drone from crashing in otherwise unsafe conditions.

We also tested a scenario which consisted of two landing pads and three obstacles (trees) forming a straight line as shown in Figure 7. The goal of the drone was again to take off, avoid obstacles, land, take off, avoid obstacles and finally land again where it began. The results from testing this scenario show that the drone can avoid obstacles also for this scenario. If we rotate this scenario by 90 degrees to the right we expect the drone to behave in the 'same' way. So we ran tests where we rotated the scenario in the figure by 90 (east), 180 (south) and 270 (west) degrees and compared the results. All tests succeeded but to our surprise, the drone did not behave in the same way for all of them. For some tests the drone flew to the left of the obstacles and for others it flew to the right in seemingly a random fashion – it was impossible to predict what course it would take.

After running many tests where we have the line setup, as described above, we realized that this type of setup might be a corner case. We decided to change the setup for line tests in such a way that the landing pad would be some degree off to the right or the left. After this change tests became much more predictable. For tests where the offset is 1 degree tests became more predictable than the ones that formed a straight line (no offset). 2 degree offset made the tests even more predictable, and when we tried 3 degrees we could completely predict (99%) whether the drone goes to the left or to the right of the first obstacle encountered. These results confirmed our hypothesis that we had identified a corner case.

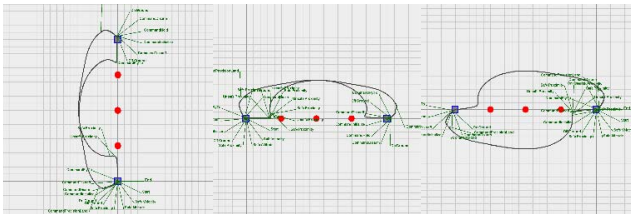

Figure 7 Three test cases rotated 90 degree increments.

IV. RELATED WORK

There are a growing number of technical papers that are related to testing a system in which it is difficult to formulate the test oracle. We discuss those papers from the perspective of our work.

A. Metamorphic Testing

In metamorphic testing [2], the test oracle problem is alleviated by formulating one or more properties about the SUT using so called metamorphic relations between inputs and outputs. The 2016 survey by Segura et al [10] gives a good overview about different domains that metamorphic testing has been successfully applied to and discusses different strategies for creating meaningful metamorphic relations, test cases and test execution strategies for metamorphic testing. The approach presented in this paper applies the idea of metamorphic testing to the properties of an autonomous drone's behavior, its physical characteristics and the environment and scenario it is exposed to during testing.

One of the challenges in metamorphic testing is the problem of identifying so-called metamorphic relations. This work,

proposes metamorphic relations that leverage geometric changes in the simulated environment, such as rotation and translation in combination with different formations of obstacles in the scenarios that the drone exposed to. These relations are encoded into Extended Finite State Machine (EFSM) models. A model-based testing framework [3][4][5] is then leveraged to automatically generate numerous different test cases that result in equivalent behavior. Furthermore, the paper presents measures that can be used to analyze the equivalence of different test executions, such as measuring the timing of different test executions, and analyzing deviations in the path globally and locally around obstacles.

Other authors have applied metamorphic testing to different domains, created metamorphic relations for these domains and used different test generation strategies. In [6], the authors present metamorphic relations for feature models and a process for automated metamorphic testing for feature model analysis tools. They use the metamorphic relations to create variations of a manually created feature model and then compared the outputs of the analysis of the different feature models against each other to identify if they behave equivalently. The models are the inputs for test generation so to say. In this work a model represents all possible variations of equivalent test scenarios and is then used to generate test cases. In [7], the authors combine random testing with metamorphic testing. We have used random testing as a strategy for traversing the EFSM model to create different scenarios from the testing models.

B. Model-Based Testing

In MBT [11] the testing model is used to derive the test cases and also serves as the test oracle for the test execution. In previous work [1] we have used model-based metamorphic testing to test NASA's Data Access Toolkit (DAT). For testing the DAT system, the models were used to generate equivalent queries for DAT's REST and Web UI interface. The test verdict was based on the comparison of the resulting records returned by the two queries. The framework presented in this paper was adapted from our earlier work [1]. The notation of the modeling language, the idea to encode the equivalences into the model and the test generation algorithms are similar to the original approach. However, new metamorphic relations and equivalence measures and a new test execution and analysis framework had to be developed specifically for executing and evaluating the drone simulations.

V. SUMMARY AND FUTURE WORK

We have presented an approach for testing of autonomous systems such as drones. The approach is based on model-based testing combined with metamorphic principles. We have found that the presented testing approach shows potential for identifying corner cases where actual system behavior does not match expected behavior. When the corner cases have been addressed and all test cases pass, the metamorphic testing approach increases our confidence in the autonomous system. We plan to add capabilities to the framework that allows us to test the drone in the simulator using obstacles that move during flight, for example in the form of other drones. We will also add wind and other physical conditions to the testing scenarios. We will also test more of the metamorphic relations outlined in this paper. We plan to automate the comparison of the test runs by applying shape matching algorithms with some tolerance.

ACKNOWLEDGMENT

This work is supported by NAWCAD through the University of Maryland and by NSF Award Number 1446583. Rikard Birgisson developed the simulator, Benedikt Johannesson did most of the testing, Kari Mimisson developed the testing infrastructure, Dharma Ganesan and Bjarki Stefansson developed the infrastructure for random testing.

REFERENCES

[1] M. Lindvall, D. Ganesan, R. Ardal, and R. E. Wiegand, "Metamorphic ´ model-based testing applied on NASA DAT: An experience report," in Proceedings of the 37th International Conference on Software Engineering - Volume 2, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 129–138. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819030

[2] H. Liu, F. Kuo, D. Towey, and T. Y. Chen, "How effectively does metamorphic testing alleviate the oracle problem?" IEEE Trans. Software Eng., vol. 40, no. 1, pp. 4–22, 2014. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2013.46

[3] C. Schulze, D. Ganesan, M. Lindvall, D. McComas, and A. Cudmore, "Model-based testing of NASA's OSAL API - an experience report." in ISSRE, 2013, pp. 300–309.

[4] C. Schulze, D. Ganesan, M. Lindvall, R. Cleaveland, and D. Goldman, "Assessing model-based testing: An empirical study conducted in industry," in Companion Proceedings of the 36th International Conference on Software Engineering, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 135–144. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591180

[5] V. Gudmundsson, C. Schulze, D. Ganesan, M. Lindvall, and R. Wiegand, "An initial evaluation of model-based testing." in ISSRE (Supplemental Proceedings), 2013, pp. 13–14.

[6] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortes, ´ "Automated metamorphic testing on the analyses of feature models," Inf. Softw. Technol., vol. 53, no. 3, pp. 245–258, Mar. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2010.11.002

[7] R. Guderlei and J. Mayer, "Towards automatic testing of imaging software by means of random and metamorphic testing," International Journal of Software Engineering and Knowledge Engineering, vol. 17, no. 06, pp. 757–781, 2007.

[8] M, Quigley, B. Gerkey, K Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, A. Ng, "ROS: an open-source Robot Operating System"

[9] N. Koenig, A. Howard, "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator"

[10] S. Segura, G. Fraser, A. Sanchez, A. Ruiz-Cortes: "A Survey on Metamorphic Testing" IEEE Transactions on software engineering, vol. 42, no. 9, September 2016

[11] Mark Utting , Bruno Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, 2006