Received: 22 September 2023 | Revised: 12 March 2024 | Accepted: 25 May 2024

DOI: 10.1002/spe.3358

RESEARCH ARTICLE

WILEY

An automated model-based testing approach for the self-adaptive behavior of the unmanned aircraft system application software

Zainab Javed[1] | **Muhammad Zohaib Iqbal**[2] | **Muhammad Uzair Khan**[1] | **Muhammad Usman**[2] | **Atif Aftab Ahmed Jilani**[1]

[1]UAV Dependability Lab, National Center of Robotics and Automation (NCRA), Department of Computer Science, National University of Computer and Emerging Sciences, Islamabad, Pakistan

[2]Quest, Islamabad, Pakistan

Correspondence
Zainab Javed, UAV Dependability Lab, National Center of Robotics and Automation (NCRA), Department of Computer Science, National University of Computer and Emerging Sciences, Islamabad, Pakistan.
Email: zainab.javed6@gmail.com

Summary

The unmanned aircraft system (UAS) is rapidly gaining popularity in civil and military domains. A UAS consists of an application software that is responsible for defining a UAS mission and its expected behavior. A UAS during its mission experiences changes (or *interruptions*) that require the unmanned aerial vehicle (UAV) in a UAS to self-adapt, that is, to adjust both its behavior and position in real-time, particularly for maintaining formation in the case of a UAS swarm. This adaptation is critical as the UAS operates in an open environment, interacting with humans, buildings, and neighboring UAVs. To verify if a UAS correctly makes an adaptation, it is important to test it. The current industrial practice for testing the self-adaptive behaviors in UAS is to carry out testing activities manually. This is particularly true for existing UAS rather than newly developed ones. Manual testing is time-consuming and allows the execution of a limited set of test cases. To address this problem, we propose an automated model-based approach to test the self-adaptive behavior of UAS application software. The work is conducted in collaboration with an industrial partner and demonstrated through a case study of UAS swarm formation flight application software. Further, the approach is verified on various self-adaptive behaviors for three open-source autopilots (i.e., Ardu-Copter, Ardu-Plane, and Quad-Plane). Using the proposed model-based testing approach we are able to test sixty unique self-adaptive behaviors. The testing results show that around 80% of the behavior adaptations are correctly executed by UAS application software.

KEYWORDS

model-based testing, self-adaptive behavior, software-testing, UAS application software

Abbreviations: GCS, ground control station; GPS, global positioning system; MBT, model-based testing; OCL, object constraint language; RTL, return to launch; RC, radio control; SUT, system under test; UAS, unmanned aircraft system; UAV, unmanned aerial vehicle; UML, unified modelling language.

1 | INTRODUCTION

The UAS is the Unmanned Aircraft System, which consists of a UAV, a UAV autopilot software, an application that runs on the UAV autopilot software, and a Ground Control Station (GCS).[1] The GCS sends/receives commands to/from the UAV and observes the UAS state during mission execution. The application running on the UAV autopilot software specifies the mission waypoints and the behavior of the UAV. The autopilot software guides a UAV to fly autonomously.[2] In this paper, we use the term *UAS* when referring to the entire system, and we use the term *UAV* when we refer to the aerial vehicle that a UAS consists of Reference 3.

Recently, the applications of UAS have increased a lot due to its ever-growing use in civil and military domains.[4] They are being used for surveillance, precision agriculture, weather forecasting, search and rescue, traffic monitoring, and many other types of missions.[5] The UAS has several benefits such as efficiency, cost-effectiveness, ability to access unreachable areas and to prevent humans from being involved in hazardous tasks.[6]

A UAS, during the execution of an assigned mission, experiences different changes (or *interruptions*). These changes can occur in the environment (in which it operates), in the UAV itself, or in the GCS controlling the UAVs.[7] For example, a safe distance violation between two neighboring UAVs, a GPS disconnection, a battery discharge, and others. For the UAS to handle these run-time interruptions, the application software (as part of UAS) specifies the self-adaptive behavior.

Self-adaptation is the ability of a UAS application software to detect a change, identify possible solutions (i.e., adaptation alternatives), and adapt to the most suitable solution (based on the current situation).[8] In this paper, we consistently use the terms *self-adaptive* and *self-adaptation* to describe UAS application software that can adjust its behavior in real-time based on changes within the system and the changes in the operating environment.[9,10]

The self-adaptation behaviors are different from functional behaviors. Consider an example of traffic monitoring UAS, the functional behaviors would be to monitor and track overspeeding vehicles and signal-violating vehicles, whereas self-adaptive behaviors trigger when there is a UAV's battery discharge while monitoring an accident or an increase in the wind speed. To handle these, a self-adaptive UAS requires a UAV to adapt its behavior.

The application software in UAS is responsible for specifying the self-adaptive behavior based on the UAS self-adaptive behavior requirements. The possible adaptations for every change can be more than one, for example, to handle *battery discharge* and *increased wind speed* change, the UAV can adapt its action to land, to return-to-launch (RTL), or to assign a task to another UAV (in case of a swarm). After the identification of a possible set of adaptation alternatives for a change based on the current situation, a selection is to be made.

The selection of an adaptation alternative depends upon certain factors, where the priority of goals plays a vital role. For example, in the case of battery discharge change, the goals are: *Execute Mission* (i.e., complete the remaining mission) and *Ensure Safety* (i.e., return back to launch/home location). In this scenario, if the priority of the *Execute Mission* is higher than the *Ensure Safety*, and the mission remaining to be completed is less than 25% then the UAV continues its mission. Otherwise, the UAV returns to launch. This is how different factors affect the self-adaptation in a UAS. The application software in UAS executes the self-adaptive behavior based on the self-adaptive behavior requirements.

In the context of UAS swarms, not only do individual UAVs need to adapt their behaviors, but they also must adjust their positions in relation to their neighboring UAVs to maintain the designated UAS formation (e.g., the leader-follower formation), as specified in the UAS software requirements. This becomes particularly crucial when the UAS specifications mandate the UAVs to maintain a specific formation. Adhering to these requirements enhances the overall effectiveness of the UAS operation.[11]

It is important to test the self-adaptive behavior specified in the UAS application software to ensure the safety of the environment (humans and buildings) and the UAS itself.[12]

For the above example of a traffic monitoring UAS, to test a scenario where a UAV adapts its behavior from *Tracking an Over-speeding Vehicle* to *RTL*, it is required to fly the UAV to create a self-adaptive behavior scenario (such as a battery discharge, or an increase in the wind speed) in which the UAV executes this self-adaptive behavior. Similarly, there can be many such scenarios. The identification of the right set of test scenarios to execute the particular self-adaptive behavior is a major challenge. Furthermore, none of the test scenarios are possible without the UAS flight, which in itself is an unavoidable challenge.

Currently, in industry, the testing is carried out manually, which requires a lot of effort, and is very time-consuming. Sometimes only one or two test scenarios are executed in a long day. Furthermore, failure in this case has a high cost (such

as a collision of UAVs (in case of a swarm), a crash of a UAV, and others). This restricts the manual testers from testing self-adaptive behavior scenarios requiring extreme test data. It is important to note that during the development of a UAS, flight tests are conducted to assess its various components, including engines, ailerons, elevators, and rudders, which collectively control a UAV's altitude, speed, and heading. However, the significance of time savings becomes particularly notable in the case of an existing UAS with newly developed application software defining functional and self-adaptive behavior tailored for specific mission requirements.

The solution to this problem is to automate the test case generation, execution, and evaluation to validate the UAS self-adaptive behavior. Testing, in this case, is challenging as (i) each test case involves the generation of a complete mission (such as disarm, arm, takeoff, flying, … , RTL) and (ii) each test case execution requires a complete setup of UAS environment and its flight configurations. According to the existing literature, there are approaches for testing of UAS mission and self-adaptive behavior[12–19] that do not provide an automated way to generate and execute test cases for the self-adaptive behavior of UAS. On the other hand, some approaches such as the work of Schmidt et al.[20,21] proposes an automated approach that has a focus on scenario generation for the testing self-adaptive behavior but the work is limited to safe distance violation from obstacles. Other studies that automate the testing of self-adaptive behaviors in UAS, use techniques such as formal methods,[22,23] utility functions,[24,25] and others.[26,27] These methods are either too complex for stakeholders to use,[28] or do not provide a systematic way to automate the testing process (test case generation, execution, and evaluation).

In this paper, we address the above-mentioned challenges by proposing an automated model-based testing (MBT) approach for the UAS self-adaptive behavior. In the last decade, MBT has been widely used by researchers in various domains for the testing of application software,[29–31] as it provides a systematic way to automate testing process.[32] As part of our approach, we propose a methodology for modeling UAS application software, its mission, and self-adaptive behavior. Our modeling methodology is based on UML, as it is a standard modeling language and is commonly used for modeling concepts of various domains.[30,33] The testing strategy as part of the proposed approach tests the self-adaptive behavior specified in the UAS application software. We rely on the UML models for the generation of test cases. Moreover, a prototype tool is developed to automate the test case generation, execution, and evaluation processes. Note that we do not test the UAS functional behavior, its mission, or the autopilot. Our scope is to test the self-adaptive behavior of the UAS during mission execution. Furthermore, our approach also supports UAS swarms (multiple UAVs) and we apply and demonstrate the proposed approach on an industrial case study of UAS swarm formation flight.

To summarize, the following are the major contributions of this paper:

1. A modeling methodology for the UAS application software, which includes its mission, and self-adaptive behaviors
2. An automated model-based approach for testing self-adaptive behaviors of the modeled UAS swarm
3. A prototype tool to automate the proposed approach
4. Application and demonstration of the proposed approach on an industrial case study of UAS swarm formation flight, further to test twenty-seven unique self-adaptive behaviors for the Ardu-Copter,[*] eighteen for the Ardu-Plane,[†] and fifteen for the Quad-Plane.[‡]

The rest of the paper is organized as follows. Section 2 provides information about our industry partner and the industrial case study. In Section 3, we provide an overview of our proposed approach. Section 4 provides details of the modeling methodology, whereas Section 5 presents details of the model-based testing approach. Section 6 provides details of the developed tool. In Section 7, we provide an evaluation of the proposed approach. The related work is discussed in Section 8.

2 | INDUSTRIAL CONTEXT

The work proposed in this paper is motivated by the problem faced by our industry partner who is working on producing reliable software systems. Currently, the organization is developing a formation flight application for the UAS

[*] https://ardupilot.org/copter/.
[†] https://ardupilot.org/plane/.
[‡] https://ardupilot.org/plane/docs/quadplane-support.html.

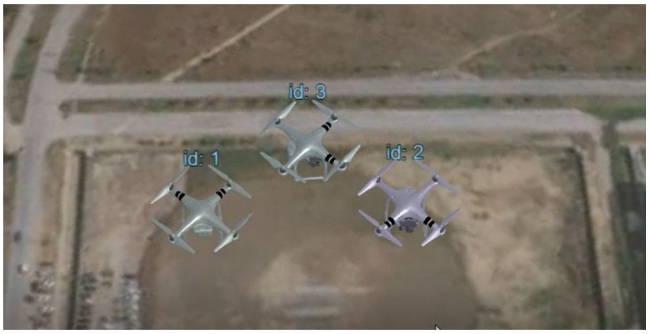


FIGURE 1 Formation flight case study simulation.

swarm. The formation flight application serves as a base for various missions (e.g., traffic monitoring, rout monitoring, area surveillance, and others). The application ensures the leader-follower formation of the UAS swarm. They need to test that the UAS swarm correctly adapts its behavior while staying in a formation. We use this application as a case study.

The formation flight application gives a mission as input to a specific formation of UAVs and they complete it. The UAS swarm consists of three UAVs (Unmanned Aerial Vehicles), each of which is a Quad-copter. The autopilot software used for the simulation of the Quad-copter is Ardupilot.[§] The functional behavior of the UAS swarm is that the swarm has to cover the assigned waypoints while staying in a formation (shown in Figure 1). The formation is in the form of a leader and two followers. The GCS assigns waypoints to the UAVs. Each UAV waits for the others to reach the assigned waypoint. The next waypoint is assigned only when all UAVs reach the previously assigned waypoint. During mission execution, the self-adaptive behavior requires the followers in the UAS swarm to ensure a safe distance from the leader UAV. An increase in the wind speed or a UAV's sharp turn can cause a safe distance violation. Moreover, the swarm should return to launch before the battery drains.

Our industrial partner has to test the UAS swarm formation flight application software. The first challenge is to test if the UAS swarm stays in the specified formation during mission execution, whereas the second challenge is collision avoidance between the UAVs in the swarm and in case of a safe distance violation, whether the UAS swarm adapts to the required self-adaptive behavior or not.

Currently, the testers at the partner organization test the formation flight application by writing, executing, and evaluating test scenarios manually. Each test scenario requires the setup of the UAS swarm formation, the GCS, and the Ardupilot. It came to our knowledge that even after many days of testing, they are only able to test a few scenarios. Moreover, they lost a UAV due to a collision in the UAS swarm during the execution of one of the test scenarios. They have manually observed UAV status during mission execution and found it difficult to verify whether the UAS swarm has correctly executed a self-adaptive behavior or not. The organization is concerned that even after so much effort in terms of time and cost, the testing is incomplete. The formation flight application upgrade is unavoidable, so the testing is required to be repeated whenever the application is upgraded.

In order to address the above-mentioned issues our industrial partner is facing, we propose an automated model-based approach for testing the self-adaptive behavior of the UAS swarm. A model-based solution is chosen as our partner organization has a strong grip over the use of models in an industrial context and most of their solutions use model-based approaches, specifically Unified Modeling Language (UML).[34] UML has been used extensively by researchers to model application software of various domains.[30,33] We have used a UML state machine to capture the self-adaptive behavior of the UAS, which is then used during the generation of test scenarios. The automated generation and execution of various self-adaptive behavioral test scenarios reduces the time and effort required by the testers. Moreover, the automated monitoring of the UAS swarm during mission execution minimizes the manual effort required by mission operators. Lastly, the mission simulator allows testing the UAS swarm on critical test values that may lead to UAV collision or crash if executed in the field. The details of the industrial partner cannot be disclosed due to a non-disclosure agreement.

[§]https://ardupilot.org/copter/.

3 | PROPOSED UAS MODEL-BASED SELF-ADAPTIVE BEHAVIOR TESTING APPROACH

In this section, we briefly present the details of our proposed model-based approach for testing the self-adaptive behavior of the UAS. The proposed approach focuses on system-level testing, that validates whether the UAS satisfies the self-adaptive behavior or not. Figure 2 shows an overview of the proposed approach.

Our proposed approach is divided into two major parts. First is the modeling of the self-adaptive behavior for the UAS, which includes UAS application modeling and its mission modeling (as shown in Figure 2). Second is the automated testing of the required self-adaptive behavior using the modeled UAS application. There are three different roles in our proposed approach, that is, *Approach Provider*, *Software Engineer*, and *Mission Operator*. The *Approach Provider* role is for the developers of the proposed approach, the *Software Engineer* role is for the users of the proposed approach, and the

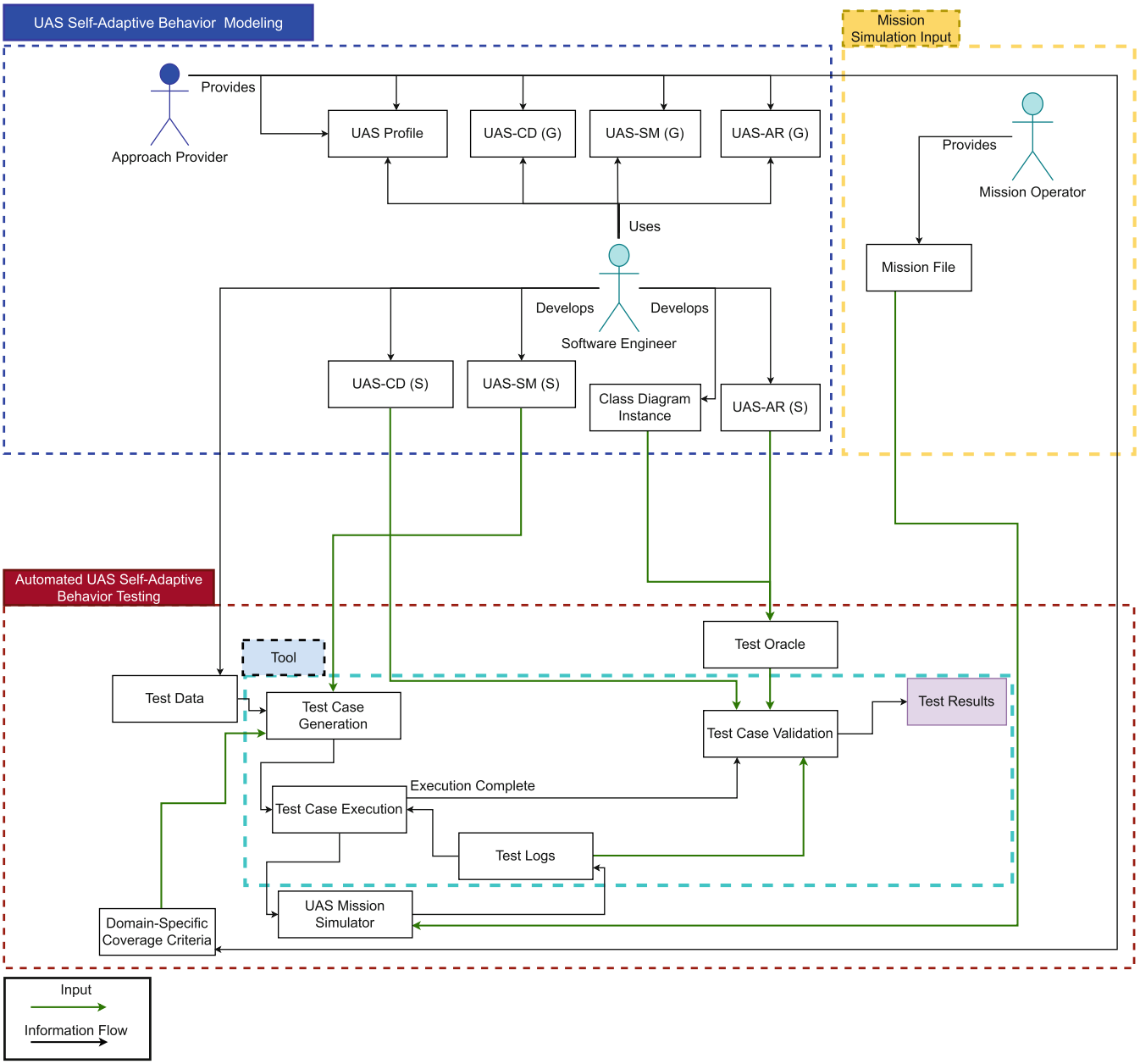


FIGURE 2 Model-based testing approach for UAS.

Mission Operator role is for the provider of the UAS application mission. The *Approach Provider* provides the approach that the *Software Engineer* uses.

3.1 | Modeling UAS application software

This section highlights the modeling details in our proposed approach, which includes the UAS application as well as mission modeling. For the self-adaptive behavior modeling, we propose a UML-based methodology. The *Approach Provider* provides modeling guidelines, whereas the *Software Engineer* is responsible for UAS application modeling. We as *Approach Provider* have developed generalized UML models for UAS application software, that facilitate the *Software Engineer* during UAS application-specific modeling. As part of the proposed UAS application modeling methodology, we as *Approach Provider* have developed: (i) a UML profile that consists of all structural and behavioral concepts of the UAS mission and its self-adaptive behavior (UAS profile as shown in Figure 2), (ii) a generalized class diagram to capture structural details of the UAS application software (*UAS-CD (G)* as shown in Figure 2), (iii) a generalized state machine to demonstrate the self-adaptive behavior common to all UAS applications (*UAS-SM (G)* as shown in Figure 2), and (iv) generalized adaptation rules (*UAS-AR (G)* as shown in Figure 2) for defining various adaptation alternatives for UAS self-adaptive behavior. OCL constraints are used for the definition of adaptation rules and are applied to the generalized UAS class diagram. OCL is a standard language for constraint specification.[35] The *Software Engineer* is assumed as a UAS domain expert and knows all the mission details and self-adaptive behavior requirements of the under-development UAS application software. It is responsible for modeling: (i) a UAS application-specific class diagram (*UAS-CD (S)* as shown in Figure 2), (ii) an instance of the application-specific class diagram, and (iii) a UAS application-specific state machine (*UAS-SM (S)* as shown in Figure 2). The *Software Engineer* also defines adaptation rules for the UAS application-specific self-adaptive behavior (*UAS-AR (S)* as shown in Figure 2) as constraints on the application-specific class diagram. For developing *UAS-CD (S)*, the *Software Engineer* uses *UAS-CD (G)* provided by the *Approach Provider*. The *UAS-CD (G)* is modified for adding UAS application-specific structural details. The *UAS-SM (S)* specifies the expected self-adaptive behavior of the under-development UAS application software, developed by the *Software Engineer* by extending the *UAS-SM (G)*. During the UAS application-specific modeling, the *Software Engineer* uses the proposed UML profile to apply UAS domain-specific concepts on the class diagram and the state machine. The *Mission Operator* is considered an expert in UAS mission modeling. It models the mission (using a mission planning tool) based on the requirements of the under-development UAS application software. The modeled mission contains various details, such as the latitude, longitude, and altitude of the waypoints required by the UAV to cover.

3.2 | Testing UAS application software

Once the modeling of the under-development UAS application is complete, the next step is the automated testing of the self-adaptive behavior for the UAS. For this purpose, our proposed model-based testing approach includes (i) a UAS application-specific test case generation, (ii) Test case execution using a UAS mission simulator, and (iii) Self-adaptive behavior verification. Our proposed testing approach starts with the generation of test cases for the system under test (SUT), that is, the UAS application software. We as *Application Provider* have developed coverage criteria, that are specific to the UAS domain. The coverage criterion is applied to the developed *UAS-SM (S)* to extract the required test paths for UAS self-adaptive behavior testing. As the *Software Engineer* is a domain expert, he is responsible for providing the test data specific to the generated test path. The test path along with the test data completes the generation of test cases.

Next is the execution of the generated test cases using a mission simulator. All the generated test cases are executed one by one. As the execution of a test case demonstrates the execution of a UAS mission, the test logs are generated to record the various details (such as the current waypoint, wind speed, current distance from an obstacle, and others) of the executed UAS mission. In this way, all the generated test cases are executed and test logs are generated separately for each of the test case executions. After all the test cases are executed, the next is to check the self-adaptive behavior of the SUT. In our proposed testing approach, the test oracle consists of SUT-specific adaptation rules (*UAS-AR (S)*) and *UAS-CD (S)* class diagram instance. These are developed by the *Software Engineer* as part of the UAS application software modeling (as shown in Figure 2). The *UAS-CD (S)* instance (developed by *Software Engineer*) only contains the expected values, whereas the actual values are missing, which are to be added after the UAS mission execution (or test cases execution).

For each of the executed test cases, the test logs are populated as actual values in the *UAS-CD (S)* class diagram instance. The *UAS-AR (S)* is applied on the updated *UAS-CD (S)* class diagram instance to validate the self-adaptive behavior of the UAS application. After the verification of all the test cases, a report of the test case verification results is generated. To automate our proposed model-based testing approach, a prototype tool is developed.

4 | MODELING METHODOLOGY FOR UAS MISSION AND SELF-ADAPTIVE BEHAVIOR

In this section, we discuss the details of the proposed UML profile. Our methodology allows to model the UAS mission and self-adaptive behavior requirements specifically for testing. We use UML for the modeling of self-adaptive structural and behavioral aspects as it is a standard[34] and widely known modeling language.[36,37,33] To model UAS domain-specific concepts we extend the standard UML through a UML profile. The proposed UAS UML profile defines structural and behavioral domain-specific concepts of the UAS mission and self-adaptive behavior. To model the structural details, we use the UML class diagram. To model the behavioral details, we use the UML state machine. Our modeling methodology facilitates *Software Engineer* to specify the mission and self-adaptive behavior of the UAS for any mission using the class diagram and state machine. Section 4.1 presents the details for the proposed UAS UML profile, Section 4.2 discusses the details for the UAS structural modeling through the UML class diagram, Section 4.3 provides the details for the self-adaptive behavioral modeling through the UML state machine, Section 4.4 presents the details of the UAS self-adaptive behavior adaptation rules, and Section 4.5 provide details about the UAS application-specific mission modeling.

4.1 | UML profile for UAS mission and self-adaptive behavior

We propose a UAS profile for defining the domain-specific concepts of UAS application software including its mission and self-adaptive behavior. The *Software Engineer* uses this profile to model the UAS domain-specific concepts during the structural and behavioral modeling of the UAS application software. Further, the proposed UAS profile plays a vital role during the model-based testing of self-adaptive behavior for the UAS. For the identification of concepts for the self-adaptive behavior of UAS, we have analyzed the literature on the self-adaptive behavior of UAS systems.[12,25,38] For the concepts of UAS mission, we have analyzed the literature,[39,40] and three open-source mission planning tools, that is, UgCS,[41] MissionPlanner,[42] and QGroundControl.[43] Figure 3A presents the package diagram for the proposed UAS UML profile (shown in Figure 3). The stereotypes in the profile are divided into three major categories, that is, *Mission*, *Adaptation*, and *Change*. The *Mission* category consists of the stereotypes related to the UAS mission. The *Adaptation* category consists of the stereotypes defining the adaptation alternatives possible to handle a change in UAS. The *Change* category consists of the stereotypes related to the different types of changes that occur in a UAS. These are the changes in the environment that require self-adaptive behavior. The Mission stereotypes are extended from the UML standard Class package, whereas the Adaptation and Change stereotypes import the UML standard State Machine package. The UML Primitives package is used to add the UML standard primitive types. The complete details for the *Mission*, *Change*, and *Adaptation* stereotypes are discussed in the following Subsections 4.1.1,4.1.2, and 4.1.3 respectively.

4.1.1 | UAS modeling profile–Mission stereotypes

A UAS mission consists of UAVs, payloads attached to the UAVs, mission goals, waypoints to be covered, and UAS self-adaptive specifications. The aim of UAS mission stereotypes is to allow the *Software Engineer* to specify the structural details of the mission and the self-adaptation during UML class diagram modeling. Figure 3B shows UAS mission stereotypes.

As shown in the figure, the stereotypes defining mission concepts are an extension of the UML metaclass *Class*. A total of twelve stereotypes are defined in the *Mission* package of the UAS UML profile. The mandatory stereotypes in *red* color are the core of the UAS mission, necessary for the *Software Engineer* to specify during the UAS application software class diagram modeling. Following, we explain each of the *Mission* stereotypes.

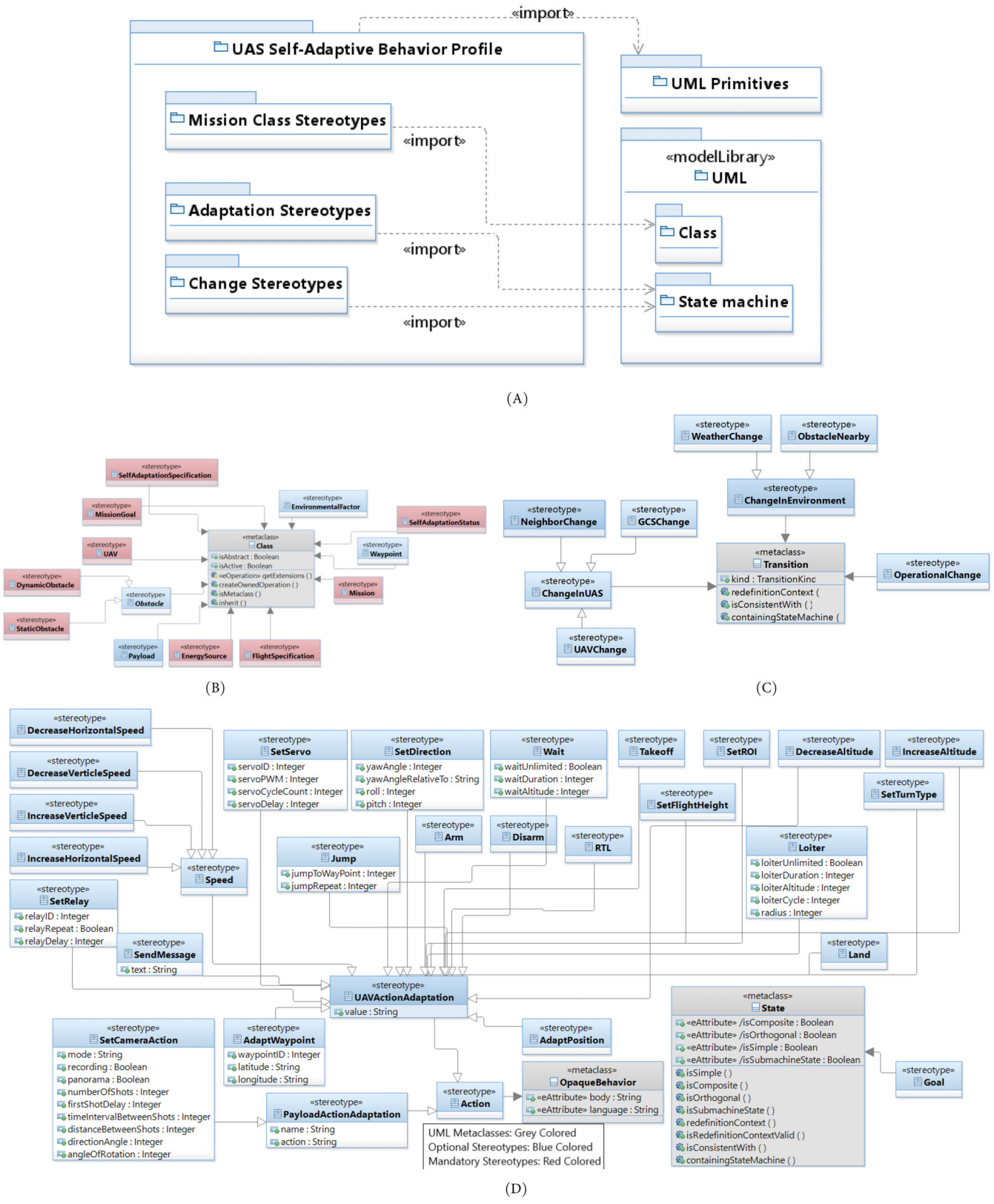


FIGURE 3 UAS modeling profile. (A) Package diagram, (B) mission stereotypes, (C) change stereotypes, (D) adaptation stereotypes.

1. *Mission*: It is a mandatory stereotype, defined to model a class that contains information about a UAS mission. This stereotype is applied to the main class of the UAS mission class diagram. Examples of mission classes are *Traffic Monitoring Mission*, *Fire Fighting Mission*, or *Shark Spotting Mission*.
2. *UAV*: Each mission requires a UAV(s) for its execution. It is a mandatory stereotype, applied to the UAV(s) involved in the UAS mission. Examples of UAVs are *fixed-wing* UAVs or *quad-copter* UAVs.
3. *MissionGoal*: Each mission has a goal(s) that the UAS has to satisfy. It is a mandatory stereotype, applied to classes that define the objectives of the mission. Examples of a UAS mission objectives/goals are to *track traffic signal violating vehicle*, *avoid collision*, or *ensure safety*.
4. *Obstacle*: For each mission, there are physical hurdles that can affect the mission. To model these, we have introduced an abstract stereotype *Obstacle*. It is further categorized into *StaticObstacle* and *DynamicObstacle* based on the nature of the physical hurdle. A *StaticObstacle* stereotype is applied to those physical hurdles that are stationary. For example, a tree or a building. A *DynamicObstacle* stereotype is for the moving physical hurdles. For example, another UAV or a bird. It is mandatory to define *StaticObstacle* and *DynamicObstacle*.
5. *EnergySource*: For each UAV used in the mission, there is an energy source. For example, a battery cell, a solar cell, or a fuel cell. It is a mandatory stereotype, applied to a class defining the energy source of a UAV.
6. *FlightSpecification*: UAVs in the mission have certain characteristics for example the maximum flight time of a UAV, wind resistance of a particular UAV, or maximum altitude at which the UAV flies. It is a mandatory stereotype, applied to a class that defines the characteristics of a specific UAV.
7. *Payload*: Each UAV in the mission has a payload. Examples of payload are a camera, siren, life jacket, or a delivery package. It is an optional stereotype, applied to a class that defines a payload assigned to a UAV.
8. *SelfAdaptationSpecification*: It is a mandatory stereotype, applied to a class that defines the basic self-adaptive behavior in the UAS. The behavior is common in missions and is built into the mission planning tools. Examples of such behaviors are the specification of the action to be taken on battery discharge, GPS loss, or terrain data loss. The possible actions can be *Land*, *RTL* or to continue the mission. These are static values, specified along with the mission definition which is the reason it is extended from *Class* metaclass.
9. *SelfAdaptationStatus*: It is a mandatory stereotype, applied to a class that is defined to capture the run-time status of the UAVs concerning the basic self-adaptive behavior. For example, a class that stores the value of the GPS connection loss of the UAV. The value is *true* or *false*. These values determine if a self-adaptive behavior is required or not.
10. *EnvironmentalFactor*: It is an optional stereotype, applied to a class that defines any known environmental factor that has an impact on the UAS during the mission execution. Modeling this allows the users to check if the UAS is correctly adapting its behavior in case of a weather change or not. Examples of this stereotype are wind, rain, or haze.
11. *Waypoint*: Each mission includes a set of waypoints that the UAV has to cover during the mission. It is applied to a class that defines the waypoints of a mission.

4.1.2 | UAS modeling profile–Change stereotypes

During a UAS mission, there are certain types of changes that can occur. These are the changes that require the UAVs in a UAS to adapt to another behavior. To allow the *Software Engineer* to model these changes, we define UAS change-related stereotypes. These stereotypes refer to the changes in the UAS, in the environment, and the operational needs of a UAS mission.[7] Figure 3C shows the UAS UML profile change stereotypes. These stereotypes allow the *Software Engineer* to model the types of changes during UAS mission behavioral modeling. The stereotypes are an extension of the UML metaclass *Transition*. There is a total of eight stereotypes in the UAS UML profile change package. There are three types of changes that can occur in a UAS mission, that is, changes in the UAS, changes in the environment, and changes in the operational needs.

1. *ChangeInUAS*: It is an abstract stereotype. It is used to model various types of change in the UAS. These changes include changes in the UAV itself, its neighbor, and the GCS controlling the UAV.
2. *UAVChange*: It is an extension of the *ChangeInUAS* stereotype. It is applied to a transition in a UML state machine that is fired in case of a change in the UAV. Examples of changes in UAV are battery discharge, GPS loss, or terrain data loss.
3. *NeighborChange*: It is also an extension of the *ChangeInUAS* stereotype. It is applied to a transition that is fired in case of change in the neighbor of a UAV. Examples of changes in neighbor include failure of the neighbor to perform the assigned task (due to a crash or any other reason)

4. *GCSChange*: It is also an extension of the *ChangeInUAS* stereotype. It is applied on a transition that is fired in case of change in the Ground Control Station (GCS). GCS is responsible for giving commands and controlling the UAS. An example of change in GCS is a disconnection between a UAV and the GCS.
5. *ChangeInEnvironment*: It is an abstract stereotype. It models the changes that occur in the operating environment during the UAS mission. These changes can be an identification of obstacles during flight or changes in the weather.
6. *WeatherChange*: It is applied on a transition that is fired in case of a change in the weather. Weather changes have an impact on the UAS and require self-adaptive behavior. The weather changes include increased wind speed, haze, or rain.
7. *ObstacleNearby*: It is applied on a transition that is fired in case of identification of a nearby obstacle. The obstacles include other UAVs, buildings, or trees.
8. *OperationalChange*: It refers to the changes that occur in the operational needs of a UAS during mission execution and may require a change in mission *Goals*. It is applied on a transition that is fired in case of a change in the operational needs of a UAV. Considering an example of the traffic monitoring mission. A battery discharge requires a UAV to adapt its goal from *Tracking* to *RTL*.

4.1.3 | UAS modeling profile–Adaptation stereotypes

In this section, we describe the stereotypes that are defined to specify the adaptation alternatives for UAS during mission execution. These stereotypes allow the *Software Engineer* to model the solutions (adaptation alternatives) to handle the changes that occur. Once a change is identified, the UAV alters its actions, its payload's action, or its goal. Figure 3D shows the UAS UML profile adaptation stereotypes. The adaptation stereotypes have three main stereotypes: (i) *UAV-Action-Adaptation*, (ii) *Payload-Action-Adaptation*, and (iii) *Goal*. The *UAV-Action-Adaptation* and *Payload-Action-Adaptation* are an extension of the UML metaclass *OpaqueBehavior* and are modeled as actions (effect) on the transition of the UML state machine. The *UAV-Action-Adaptation* stereotype refers to an action that requires adaptation in a UAV. For example, to wait, to land, or to adapt its speed. It has a *value* attribute of data type *String*, which allows specifying the value of an action. For example, if the speed of the UAV is to be increased and the new speed value is 7 m/s then the value attribute contains this value. For the possible UAV actions, we have analyzed three open-source mission planning tools, that is, UgCS,[41] Mission Planner,[42] and QGroundControl.[43] This analysis helped to extract the stereotypes for *UAV-Action-Adaptation*, that is, *Takeoff, Land, RTL, SetFlightHeight, DecreaseHorizontalSpeed, IncreaseHorizontalSpeed, DecreaseVerticleSpeed, IncreaseVerticleSpeed, SetRelay, SetServo, SetDirection, Wait, Loiter, Jump, SetROI, AdaptPosition, AdaptWaypoint, SendMessage, Arm*, and *Disarm*.

1. The *Takeoff, Land*, and *RTL* stereotypes are applied to the takeoff, land, and return-to-launch actions of a UAV. These actions occur as a result of a change transition.
2. The *SetTurnType* allows specifying how the UAV passes through a waypoint. For example, *spline* or *stop and turn*.
3. The *SetFlightHeight* stereotype allows the definition of the adaptation in altitude that the UAV should attain during its mission or at a particular waypoint.
4. The *Speed* stereotype defines the adaptation in the speed of the UAV for the complete mission or after a particular waypoint. This stereotype has four extensions, *DecreaseHorizontalSpeed, IncreaseHorizontalSpeed, DecreaseVerticleSpeed*, and *IncreaseVerticleSpeed*. The stereotypes for horizontal speed allow for defining the increase and decrease adaptation in the horizontal speed of the UAV during the mission. The stereotypes for vertical speed allow for defining the increase and decrease adaptation in the vertical speed of the UAV during the mission.
5. The *SetRelay* stereotype allows adapting the UAV action by using it to send a signal, for example, to broadcast a person's mobile signal. It has three attributes *relayID*, *relayRepeat*, and *relayDelay*. The *relayID* attribute allows specifying the relay number to be switched on. The *relayRepeat* attribute specifies if the relay action is to be set to repeat. The *relayDelay* allows specifying the delay between relay signals. The inherited value attribute allows specifying if a relay is on or off.
6. The *SetServo* stereotype allows defining the adaptation in servo, that is, if a particular servo is required to be set as a result of a change. This stereotype has four attributes as *servoID*, *servoPWM*, *servoCycleCount*, and *servoDelay*. The *servoID* attribute allows specifying the servo number that is to be switched on. The *servoPWM* attribute is the signal sent to the servo. *PWM* specifies times per second, the signal is sent. The *servoCycleCount* attribute allows specifying the number of times a signal is to be sent to a particular servo. The *ServoDelay* attribute allows specifying

the delay between servo signals. The inherited value attribute allows specifying the on or off value of the selected servo.

7. The *SetDirection* stereotype allows specifying adaptation in the direction of the UAV during its flight. This stereotype is defined by specifying the value of any of its four attributes, that is, *yawAngle, yawAngleRelativeTo, roll*, and *pitch*. The *yawAngle* specifies the angle of the UAV nose pointing from left to right, its value is between 0° and 360°. The *yawAngleRelativeTo* attribute specifies the angle of yaw that it is relative to, it is relative to the next waypoint or is relative to the north.[41] The *pitch* attribute specifies the up and down movement of the UAV's nose, it is specified in degrees. The *roll* attribute specifies the UAV's rotation about its axis, it is specified in degrees.
8. The *Wait* stereotype allows defining the waiting action of the UAV as a result of a change. It has three attributes, *waitUnlimited*, *waitDuration*, and *waitAltitude*. The *waitUnlimited* attribute allows specifying if the UAV should wait at a point for an unlimited time or until the next command is received. The *waitDuration* attribute allows specifying the duration in seconds for which the UAV should wait. The *waitAltitude* attribute specifies the Altitude that the UAV should maintain while waiting.
9. The *Loiter* stereotype allows defining the loiter action, that is, the UAV flying over a small area repeatedly, as a result of a change. For example, UAVs loitering over seashore to identify drowning swimmers. It has four attributes as, *loiterUnlimited, loiterDuration, loiterCycle, loiterAltitude*. The *loiterUnlimited* attribute allows specifying if the UAV has to loiter over an area for an unlimited time or until the UAV receives the next command. The *loiterDuration* attribute specifies the duration in seconds or hours for which the UAV should loiter over an area. The *loiterCycle* specifies the number of cycles the UAV should fly for over the specified waypoints. The *loiterAltitude* specifies the altitude that the UAV has to maintain while loitering.
10. The *Jump* stereotype allows defining an adaptation in the *Jump* action of the UAV from a particular waypoint. For example, whenever the UAV reaches waypoint 4, it should jump back to waypoint 2. This stereotype consists of two attributes *jumpToWaypoint* and *jumpUnlimited*. The *jumpToWaypoint* specifies the waypoint at which the UAV should jump when it reaches a particular waypoint let's say 'x'. The *jumpUnlimited* specifies if the UAV should jump to the specified waypoint whenever it reaches the waypoint 'x', for an unlimited time.
11. The *Disarm* stereotype allows modeling the disarming action of the UAV.

The *Payload-Action-Adaptation* stereotype is applied to an action of a transition that requires an adaptation in a payload's action. For example, taking a photo or turning on the fire extinguisher. This stereotype has two attributes, *name* and *action*. The *name* attribute allows specifying the name of the payload, for example, fire extinguisher. The *action* attribute specifies the action required by the payload, for example, to turn on the fire extinguisher.

The *CameraAction* stereotype is an extension of the *Payload-Action-Adaptation* stereotype. This stereotype allows specifying different camera actions in detail. This stereotype has attributes *mode*, *recording*, *panorama*, *numberOfShots*, *firstShotDelay*, *timeIntervalBetweenShots*, *distanceBetweenShots*, *directionAngle*, *angleOfRotation*. The mode attribute allows specifying the mode of the camera, that is, video or photo. The recording attribute allows specifying when the video recording is to be turned on and when it is to be turned off. The panorama attribute allows specifying if a panorama action is required by the payload. Its value is *true* or *false*. Also, if the camera mode is set to video the camera makes a panorama video, otherwise it takes panorama photos. The *numberOfShots* attribute allows specifying the number of shots to be taken for a panorama, in the case of photo mode. The *firstShotDelay* attribute allows to specify the interval between the command given to the payload and the first photo taken, it is specified in seconds. The photos taken are separated by time or by distance. The *timeIntervalBetweenShots* attribute specifies the time in seconds between each photo taken. The *distanceBetweenShots* attribute specifies the distance covered before the next photo is taken. The *directionAngle* attribute is used for missions such as an area scan. This attribute defines the direction of the scanning process, for example, north-south. The *angleOfRotation* attribute specifies the angle with which the camera rotates while making a panorama video or taking panorama photos. The angle is specified in degrees.

The *Goal* stereotype is defined to handle the adaptation that is required when there is a change in the operational need of the UAS mission. It is an extension of the UML metaclass *State*. It is applied to a state that represents an adaptation of a UAV's goals. The adaptation can either be in generic goals or mission-specific goals of the UAS. The generic goals are the ones that are common to any type of UAS mission. For example, to execute the mission, to ensure safety, or to preserve resources. The mission-specific goals are the ones that are specific to a particular mission. For example, to track fire, to spot a shark, or to monitor a car crash. This stereotype allows to model adaptation to the goals of the UAS mission. This adaptation is modeled as states in a UML state machine. For example, a UAV is in a traffic Monitoring state, when an

over-speeding vehicle is identified. The UAV adapts its goal from *Monitoring* to *Tracking* a vehicle. As a result, the UAV moves to the *Tracking* state.

4.2 | UAS structural modeling

This section presents the details of UAS application software, mission, and self-adaptation structural modeling (Figure 4) as part of the proposed modeling methodology. We have used the UML class diagram for modeling the structural details of the UAS application software. We as *Approach Provider* have provided a generalized class diagram (*UAS-CD (G)*) that models the general structural detail required for the UAS application software. Further, the proposed UAS UML profile is also applied to the generalized class diagram. The *Software Engineer* modifies it as per the requirements of the under-development UAS application software.

4.2.1 | UAS generalized class diagram modeling

The standard UML modeling is used for developing the *UAS-CD (G)* for the UAS. Figure 4A shows the generalized class diagram. We have analyzed the UAS domain, by going through various UAS applications, and also the existing literature to identify the concepts, their attributes, and relationships. We have also held meetings with the developers of the formation flight case study to understand the UAS swarm application software requirements.

The various classes in the class diagram are *Mission, UAV, Goal, Waypoint, EnvironmentalFactor, FlightSpecification, EnergySource, StaticObstacle, DynamicObstacle, BasicSelfAdaptationSpecification, BasicSelfAdaptationStatus*, and a *Payload* class. In these classes, we have two categories of attributes. The first category is of attributes that hold values given as input by the *Software Engineer*. The second category is of attributes that hold values, that are set during mission execution. Therefore, the class diagram contains concepts that not only help in UAS mission modeling but also allow to evaluate whether the UAS mission is heading in the right direction or not.

1. The *Mission* class stereotyped as *Mission* allows specifying the mission, for which the self-adaptive behavior of the UAV is required to be defined. The attributes of this class are *totalNumberOfWaypoint, totalDistanceToCover and totalMissionTime, currentUAVCount, UAVCount, safeDistanceFromObstacle*, and *safeDistanceFromTerrain*. These parameters have primitive data types. The *totalNumberOfWaypoint* attribute allows specifying the total number of waypoints in the mission. The waypoints that the UAV has to cover. This attribute allows to keep track of the number of waypoints covered by the UAV during a mission. The *totalDistanceToCover* attribute allows to define the total distance that the UAV has to cover during the mission. This helps in self-adaptive decision making, by keeping track of the distance covered and the distance remaining to be covered during the mission. The *UAVCount* attribute allows to specify the total number of UAVs in case of a UAS swarm. The *currentUAVCount* attribute allows to specify the current number of UAVs in case of a swarm, that is, the number of UAVs active in the swarm during a mission. This helps to identify if a UAV has crashed or is not active due to any malfunctioning. The *safeDistanceFromObstacle* and *safeDistanceFromTerrain* allow to specify the safe distance the UAVs have to maintain from the obstacles and the terrain during a mission. This allows the UAVs to take any action if there is a safe distance violation.
2. The *UAV* class is stereotyped as *UAV*. The purpose of this class is to capture the run-time information, the current status of the UAV when a self-adaptive behavior scenario occurs. It has attributes *currentHeading, currentMode, currentGoal, currentWaypoint, turnRate, currentDistanceToObstacle, targetState* and others. These attributes have primitive data types. The *currentHeading* attribute allows to capture the current heading of the UAVs involved in the mission. The current heading is specified to determine if a UAV is moving in the right direction or not. The *currentMode* is the status of the UAV. For example, a mode is *flying*, *landing*, or *return-to-launch*. The *currentGoal* is the goal that a UAV is currently working towards to satisfy. For example, *Monitoring Traffic* or *AvoidCollision*. The *currentWaypoint* attribute allows to capture the waypoint a UAV is moving towards. The *turnRate* is the rate of turn of the UAV. The *turnRate* has an impact on the distance of the UAV from obstacles. Therefore, this attribute allows to determine if the turning rate is high or not. The *currentDistanceToObstacle* attribute allows to specify the current distance between a UAV and any obstacle. This allows to check if the UAV is at a safe distance from the obstacles or not. The

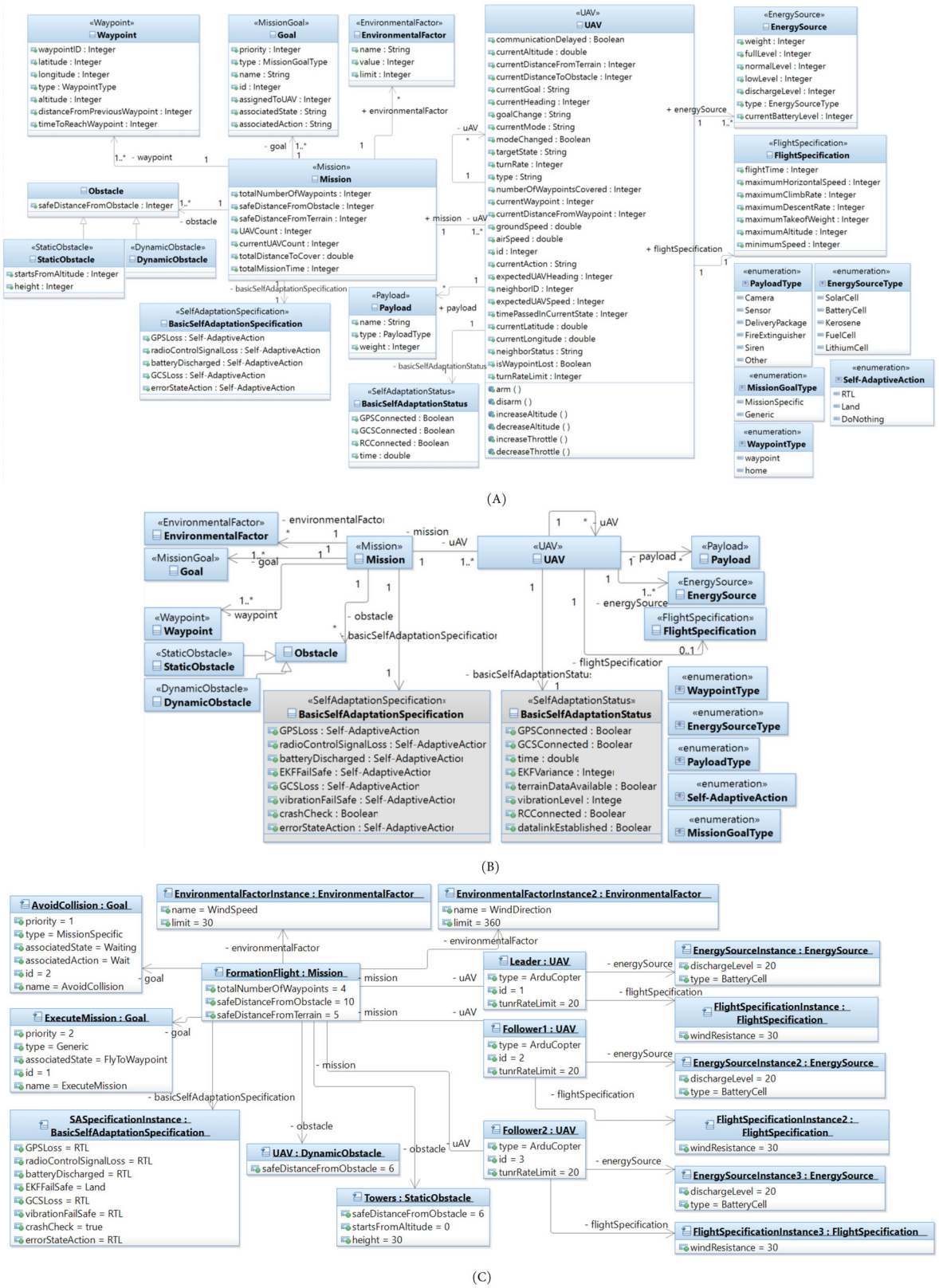


FIGURE 4 UAS structural modeling. (A) Generalized UAS class diagram, (B) Specific UAS swarm formation flight class diagram, (C) UAS swarm formation flight class diagram (*UAS-CD (S)*) instance.

targetState attribute allows to specify the state associated with the current goal of the UAV. Examples of states are *Waiting* or *Flying*.

3. To capture the goals, we have the *Goal* class stereotyped as *MissionGoal*. The goal class has multiple attributes including *priority, type*, and *assignedToUAV*. The purpose of the *priority* attribute is to define the priority of the goal. One goal can have a higher priority than the other. The data type of this attribute is an *Integer*. The purpose of the type attribute is to define the type of goal. The type attribute has *enumeration* data type *MissionGoalType*. This enumeration has two literals, *MissionSpecifc* and *Generic*. The generic goals are the ones that are common to all missions such as execute a mission, ensure safety, and preserve resources. The mission-specific goals are the ones that are specific to a particular type of mission.
4. The *Obstacle* class, stereotyped as *Obstacle* allows the specification of any expected static or dynamic obstacle in the mission area. This helps to identify if a UAV accidentally crosses the safe distance from an obstacle. The *Obstacle* is an abstract class and is implemented by *StaticObstacle* and *DynamicObstacle* classes. A static obstacle is a building, a tree, or a no-fly zone. A dynamic obstacle is a bird or another UAV. The *StaticObstacle* stereotype has two attributes, *startsFromAltitude*, and *Height*. The *startsFromAltitude* specifies the altitude from which the obstacle such as a no-fly zone starts and height specifies the maximum height of the obstacle.
5. The *FlightSpecification* class is stereotyped as *FlightSpecification*. It allows defining the UAV's flight constraints. Its attributes include: *flightTime*, *maximumHorizontalSpeed*, *maximumClimbRate*, *maximumDescendRate*, and *maximumTakeoffWeight*. The *flightTime* attribute allows the specification of the time the UAV stays in the air. This attribute is important to decide if an adaptation takes place or not. The *maximumHorizontalSpeed* allows specifying the maximum speed of the vehicle. Exceeding the maximum speed indicates a disruption in the system. The *maximumClimbRate* and *maximumDescendRate* specify the rate of speed and altitude that the UAV has to maintain during takeoff and landing respectively. Exceeding rates indicate a weather change or a system disruption. The *maximumTakeoffWeight* specifies the weight that the UAV carries.
6. The *EnergySource* class, stereotyped as *EnergySource* allows specifying the source of energy used by the UAV and its related attributes: *weight, fullLevel, normalLevel, lowLevel* and the *type* of energy source. The type of energy source used affects the UAV's performance in terms of its speed, flight time, or maximum altitude. The weight attribute specifies the weight of the energy source. The *fullLevel*, *normalLevel* and *lowLevel* specify the indicators of a full, normal, and low level of the energy source. The type of energy source has an enumeration data type *EnergySourceType*. The enumeration literals are *kerosene, batterycell, fuelcell, lithiumcell*, or *solarcell*.[44]
7. The *BasicSelfAdaptationSpecification* class is stereotyped as *SelfAdaptationSpecification*. The purpose of defining this class is to allow the user to specify expected actions of the self-adaptive behaviors common in missions or the self-adaptive behaviors built-in in the mission simulator (Mission planner,[42] UgCS,[41] QgroundControl[43]) used. It has attributes *GPSLoss, BatteryDischarged, RadioControlSignalLoss, GCSLoss*, and *ErrorStateAction*. Their data type is of type *enumeration Self-AdaptiveAction*. The enumeration *Self-AdaptiveAction* includes literals *RTL, Land*, and *DoNothing*. The *GPSLoss* attribute allows to specify the action of UAV in case there is a GPS signal loss. The *GCSLoss* attribute allows to specify UAV action in case of disconnection of UAV from its GCS. The *ErrorStateAction* attribute allows specification of the UAV's action in case the UAV is unable to move towards its current waypoint. This happens due to weather conditions or UAV malfunctioning.
8. The *BasicSelfAdaptationStatus* class is stereotyped as *SelfAdaptationStatus*. This class has a direct association with the *UAV* class. The purpose of this class is to capture information regarding UAVs. This is done to determine if a self-adaptive behavior is required or not. For example, if the *GPSConnected* attribute has the value *false* and *time* attribute has a value greater than five seconds. This means that a GPS self-adaptive action is required to be taken. The expected action for GPS disconnection is specified in the *BasicSelfAdaptationSpecification* class as *RTL*. This class has attributes *GPSConnected*, *GCSConnected*, and others. These two attributes have *Boolean* data types.
9. The *Payload* class is stereotyped as *Payload*. This class has a direct association with the *UAV* class. Its purpose is to define the payload assigned to a UAV. It has attributes *name, type*, and *weight*. The *name* attribute has a *String* data type. The *type* attribute has an enumeration data type. The enumeration data type is *PayloadType*. Some payloads are specified as literals such as *camera, sensor, deliveryPackage, fire-extinguisher, siren*, and others. The *weight* has data type *Integer*.
10. The *EnvironmentalFactor* is stereotyped as *EnvironmentalFactor*. The purpose of defining this class is to allow the user to capture any known environmental factors that affect the mission. It has attributes *name, value*, and *limit*. *Name* and *limit* attributes are set during design time. While the *value* attribute's value is set during mission execution. The limit attributes allow to define the allowed value that does not affect UAV operation during a mission.

The *Mission* class has a bidirectional relationship with the *UAV* class, with a multiplicity of one to many. A mission has one or more UAVs. *Mission* class has a directed association with the *Goal, Waypoint, BasicSelfAdaptationSpecification, EnvironmentalFactor*, and the *Obstacle* classes. A mission has one or more goals, waypoints, and obstacles. A mission should have one self-adaptation specification class. The bidirectional relationship of *UAV* and *Mission* class shows that each UAV is assigned a mission. The *UAV* class has a direct association with the *FlightSpecification*, *BasicSelfAdaptationStatus*, and the *EnergySource* class. The UAV should have a self-adaptation status. It should have a flight specification and one or more energy sources. The UAV class has a directed relation with itself, to allow specification of the link between UAVs of a UAS swarm.

4.2.2 | UAS application-specific class diagram modeling

In this subsection, we present the details for the modeling of the UAS application-specific class diagram. The *Software Engineer* is responsible for developing *UAS-CD (S)*. This paper uses an industrial case study of UAS swarm formation flight. We discuss the development of *UAS-CD (S)* considering the UAS swarm formation flight application. The *Software Engineer* uses the developed UAS generalized class diagram (*UAS-CD (G)*) for this purpose and integrates the application-specific requirements of the formation flight case study in it.

The class diagram for the UAS swarm formation flight case study is shown in Figure 4B. The newly added or modified classes are shown in *grey* color. The *Software Engineer* reviews the provided *UAS-CD (G)* as per the requirement of the formation flight application, no new classes are required to be added, besides two classes, that is, *BasicSelfAdaptationSpecification* and *SelfAdaptationStatus* are required to be modified. As shown in Figure 4B, the *BasicSelfAdaptationSpecification* class is updated to cater to the self-adaptive behavior details built-in in the Mission Planner simulator. The added attributes are *EKFFailSafe, TerrainDataLossFailSafe, vibrationFailSafe*, and *crashCheck*. The *EKFFailSafe* attribute value specifies the UAV status in case the position and attitude estimator in a UAV fails. The *TerrainDataLossFailSafe* attribute value specifies the UAV status when terrain data is lost as this affects the UAV to maintain a safe distance from the terrain. The *vibrationFailSafe* attribute value specifies the UAV status in case there is a high vibration level to avoid any disturbance in the mission. The *crashCheck* attribute value specifies the UAV status when the UAV is not in control. The *crashCheck* disarms the UAV to reduce damage in case of a crash. The *SelfAdaptationStatus* class is updated in the formation flight class diagram by adding attributes *EKFVariance*, *terrainDataAvailable*, and *vibrationLevel*. The values of these attributes allow to determine if a self-adaptive behavior is required or not.

4.2.3 | UAS application-specific class diagram instance definition

This section presents the details of the UAS application-specific class diagram instance, that is required to specify the expected values for the verification of the self-adaptive behavior requirements of the UAS mission in the under-development application software. The class diagram instance belongs to the *UAS-CD (S)*, therefore the *Software Engineer* is responsible for providing an instance of the developed *UAS-CD (S)*. Here, we discuss the class diagram instance modeled for the UAS swarm formation flight case study.

Normally, the class diagram instance creation is a run-time process, that is initiated at the time of application execution. In our proposed modeling methodology, the application-specific class diagram models the concepts that specify the expected values for the UAS mission and self-adaptive behavior. The *Software Engineer* specifies class diagram instance values for these concepts only. The values of these concepts are not modified during application execution. Figure 4C shows the UAS formation flight class diagram instance. The name of the mission is *FormationFlight* as shown in the instance of *Mission* class. The value of *totalNumberOfWaypoints* attribute is four, to specify that there are four waypoints during formation flight mission execution. There are three instances of the *UAV* class as, *Leader*, *Follower1*, and *Follower2*, to specify that the mission contains three UAVs in the swarm. In the UAV instance, the *id* attribute specifies its number, whereas the *type* attribute specifies the type of UAV, that is, *Ardu-copter* in this case. The instances of *Goal* class are *AvoidCollision* and *ExecuteMission*. The priority of the *AvoidCollision* goal is higher than the priority of *ExecuteMission* as the UAVs in a formation flight must avoid collision with each other in any situation. The obstacles modeled in the case study are *UAV* and *Towers* specify dynamic and static obstacles respectively. The value of the *safeDistanceFromObstacle* attribute is specified as six in both the *UAV:DynamicObstacle* and *Towers:StaticObstacle* class instances. This is the

distance that the follower UAVs (i.e., *Follower1* and *Follower2*) must maintain from the *Leader* UAV during the formation flight mission execution. In the formation flight class diagram instance, each UAV is associated with a *FlightSpecification* class. In this class, we specify the value of the attribute *WindResistance*. This is the speed of the wind that each particular UAV can handle, as different types of UAVs have different wind resistances. The type of *EnergySource* used in the case study is *BatteryCell*. The battery discharge level is specified as *twenty*. This means when the current battery level is equal to twenty, the battery is considered to be discharged. In the *BasicSelfAdaptationSpecifiaction* class, *RTL* is specified for most of the self-adaptive attributes that means whenever there is a *GPSLoss*, or *RadioControlSignalLoss*, or *BatteryDischarge*, or any other situation like this, the UAV should *Return-To-Launch*. The formation flight application has no payload associated with the UAVs in the swarm. There are two environmental factors, wind speed, and wind direction. These factors have an effect on the UAS swarm during mission execution. The wind speed limit is specified as *thirty*. Any value greater than thirty can divert a UAV from its path. The wind direction limit is specified as *360 degrees*, as the wind can be in any direction from 0 to 360 degrees.

4.3 | UAS behavioral modeling

This section presents the details of the UAS self-adaptive behavioral modeling (Figure 5) as part of the proposed modeling methodology. We have used the UML state machine for defining the behavior of the under-development UAS application software. We as *Approach Provider* have developed a generalized UAS self-adaptive behavior state machine (*UAS-SM (G)*). It models the generalized self-adaptive behavior during a UAS mission. The *Software Engineer* uses *UAS-SM (G)* to develop a UAS application-specific state machine (*UAS-SM (S)*).

4.3.1 | UAS generalized state machine modeling

The standard UML state machine modeling guidelines are used to develop a UAS generalized state machine as shown in Figure 5A. For the identification of the concepts, we have got feedback from the domain experts, we have analyzed the mission planning tools, and we have reviewed the existing literature on UAS behavioral modeling. The behavior we have modeled using the identified concepts is common in all UAS missions. For the definition of this behavior, we use the self-adaptive behavior stereotypes, proposed in our UAS UML profile (see details in Section 4.1).

In the *UAS-SM (G)* as shown in Figure 5A, there are five states, *Land, ErrorState, Armed, Disarmed*, and *RTL* besides *initial, final*, and *fork* states. There are a total of sixteen transitions between the states. The starting transition from the initial state leads to the *fork* that facilitates the movement of a UAV in every possible direction. The other various transitions are named *GPS-Loss*, *GCS-Loss, Battery-Discharged, Radio-Control-Signal-Loss*, and others.

In *UAS-SM (G)*, there is a transition from *fork* to *RTL* and *Land* states, named as *RadioControlSignalLoss* and *RadioControlSignalLoss2* respectively. Both transitions specify the loss of radio control signal between the Radio Control Transmitter and the UAV. Each of the transitions fires when the UAV is controlled by a Radio Control Transmitter and there is a loss of radio signal between the transmitter and the UAV. These transitions are stereotyped as *UAVChange* (details available in Section 4.1.2), as the transition represents a change in the UAV. In order to handle this UAV behavioral change, the UAV self-adaptation is modeled as either *fork* to *Land* or *RTL* depending on the condition that would be specified as per the requirements of the application-specific mission details. The end result is that the UAV adapts one of the actions as to *Land* or *RTL*. These actions are modeled as an effect in both the transitions and are stereotyped as *Land* and *RTL* respectively. The *Land* and *RTL* stereotypes are applied as this is an adaptation in the UAV's action (details in Section 4.1.3). Similarly, for the GCS Loss behavioral change of the UAV, we have modeled two transitions, that is, *GCS-Loss* and *GCS-Loss2*. The *GCS-Loss* transition is from *fork* to the *RTL* state and the *GCS-Loss2* is from *fork* to the *Land* state. The selection of transition depends upon the under-development application's mission requirements. These transitions represent a GCS signal loss between the UAV and its GCS (Ground Control Station). To handle this change in the GCS, the transitions are stereotyped as *GCSChange* for modeling self-adaptive behavior. As a result of this change, the UAV is required to adapt its action. The action is either *Land* or *RTL*. The *BatteryDischarged* and *BatteryDischarged2* transitions are modeled to specify the self-adaptive behavior of the UAV, in case the battery level reaches a specific threshold value. This threshold value is specified depending on the requirements of the under-development UAS application's mission requirements. Whenever the threshold value is reached, either the transition to *Land* state or *RTL*

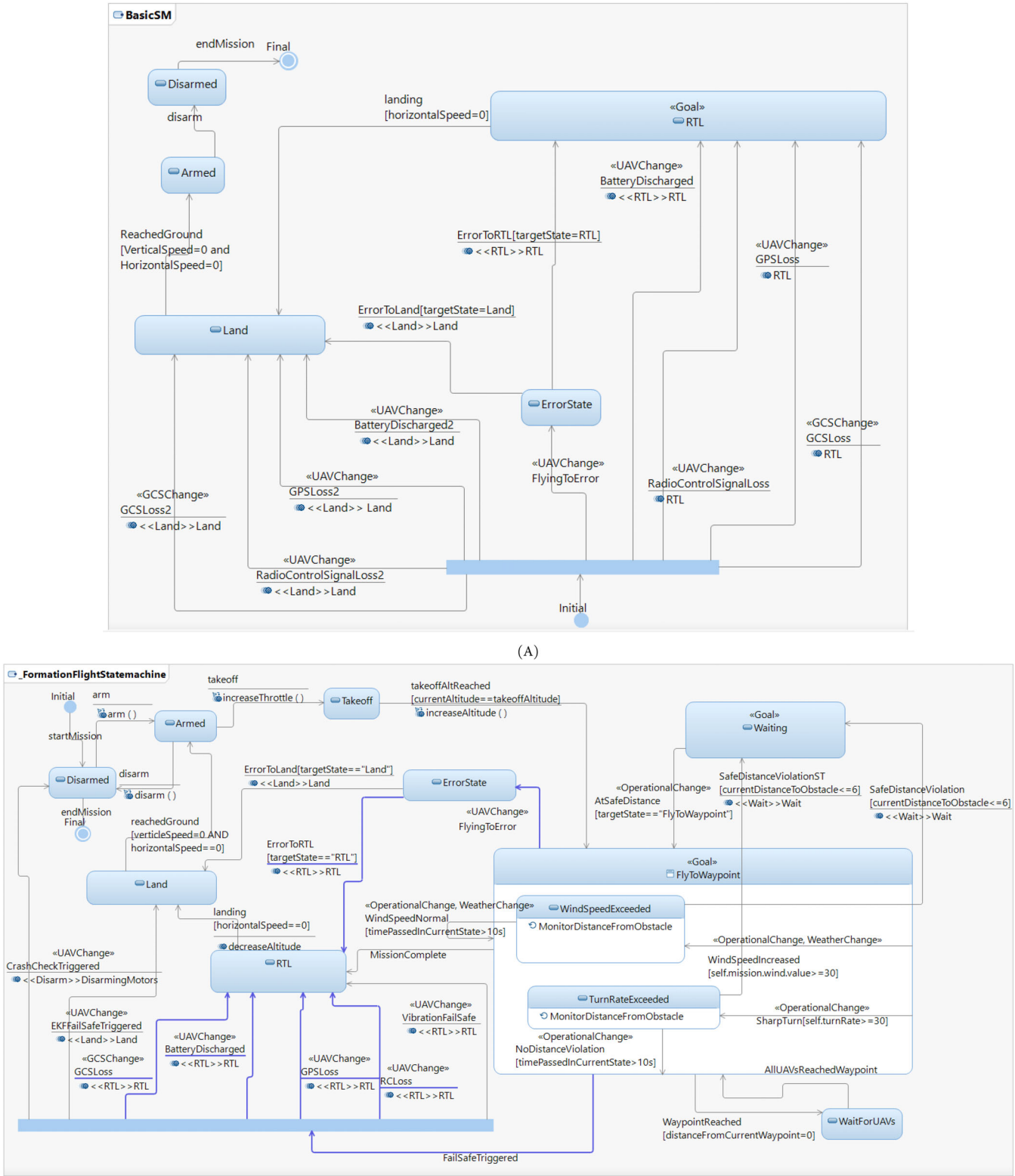


FIGURE 5 UAS self-adaptive behavior modeling. (A) Generalized UAS self-adaptive behavior state machine, (B) UAS swarm formation flight self-adaptive behavior state machine.

state fires depending on the provided condition. Similarly, the *GPS-Loss* and *GPS-Loss2* transitions are modeled to specify the loss of GPS signal during the UAV's flight. To model a possible situation where a UAV in the UAS is unable to move towards its assigned waypoint or loses the record of the assigned waypoint, we have provided a transition *Flying-To-Error* in the developed *UAS-SM (G)*. This is a situation of a change in self-adaptive behavior in the UAS, therefore a *UAVChange* stereotype (details are available in Section 4.1.2) is applied on *Flying-To-Error* transition. The initial state of this transition is a *fork*, whereas the target state of this transition is the *ErrorState* state. From *ErrorState* state, there are two transitions, one is towards *Land* state, whereas the other is towards *RTL* state. Once the UAV is in *ErrorState* state, it can adapt its action to *Land* or to *RTL* based on the condition that would be specified as per the requirements of the application-specific mission details in the application-specific state machine.

4.3.2 | UAS application-specific state machine modeling

This section presents the details of the UAS application-specific state machine (*UAS-SM (S)*) modeling. We discuss this considering the UAS swarm formation flight case study. The *Software Engineer* is responsible for developing *UAS-SM (S)* using the generalized state machine (*UAS-SM (G)*) to model the formation flight case study self-adaptive behavior. As every UAS swarm application software uses a specific mission simulator or planner,[42] so it is also the responsibility of the *Software Engineer* to model the mission planner specific self-adaptive behavior details in the *UAS-SM (S)*.

The modeled *UAS-SM (S)* on behalf of the *Software Engineer* is shown in Figure 5B. From the *UAS-SM (G)* (as shown in Figure 5A), the transitions from the *fork* to the *RTL* state are used as it is because the under-development formation flight case study requires the UAS swarm to *Return-To-Launch* in case of generalized self-adaptive behaviors (i.e., *GCS-Loss, GPS-Loss, RC-Loss*, and *Battery-Dischagred*). The transitions of *UAS-SM (G)* are shown in *blue* color in the figure. The state machine represents the behavior of each UAV in the swarm. In *UAS-SM (S)* as shown in Figure 5B, five new states and eighteen new transitions are added in the generalized state machine (*UAS-SM (G)*) because of the formation flight case study. The newly added states are *FlyToWaypoint, WindSpeedExceeded, TurnRate-Exceeded, WaitForUAVs*, and *Waiting*. Out of the total 18 newly added transitions, 15 transitions are added to model the self-adaptive behavior of UAVs in the UAS swarm formation flight case study, whereas three transitions are added to model the self-adaptive behavior provided by the Mission Planner. The three transitions that belong to the mission planner's behavioral modeling are *VibrationFailSafe, EKFFailSafeTriggered*, and *CrashCheckTriggered*, whereas some of the other transitions are *Wind-Speed-Increased, Sharp-Turn, Safe-Distance-Violation, No-Distance-Violation*, and *Wait-For-UAVs*.

The UAS swarm formation flight behavior requires the UAVs in a swarm to traverse all the waypoints in the mission while keeping the formation. So, the UAS swarm flies from home to the first waypoint, then to the second waypoint, and goes on until all the waypoints are traversed. On reaching each waypoint, the UAV(s) waits for the other UAV(s) to reach this waypoint to maintain the formation and then moves forward.

This behavior is modeled in the formation flight state machine (Figure 5B) as a UAV stays in the *FlyToWaypoint* state, keeping the formation. If a UAV reaches the assigned waypoint, the state changes to *WaitForUAVs*. At this state, the UAV waits for other UAVs to reach the same waypoint. When all the UAVs in the UAS swarm reach the assigned waypoint, the UAV moves to the *FlyToWaypoint* state to fly to the next assigned waypoint. While in the *FlyToWaypoint* state, if the UAV experiences a change in the wind speed that affects the UAV, the UAV moves to the *WindSpeedExceeded* state. If due to this, the safe distance between the leader UAV and the follower UAV is violated, then the UAV moves to the *Waiting* state. Similarly, because of the change in wind speed, if the UAV's turning rate crosses the limit (or threshold value), the UAV moves from the *FlyToWaypoint* state to the *TurnRateExceeded* state. In case the UAV's safe distance is violated while the UAV is in the *TurnRateExceeded* state, the UAV moves to the *Waiting* state. The UAV remains in the *Waiting* state until the safe distance between the leader UAV and the follower UAV is reached. When at a safe distance, the UAV again moves to the *FlyToWaypoint* state. When in the *WindSpeedExceeded* state, if all the UAVs in the UAS swarm remain at a safe distance from each other and there is no safe distance violation until the wind speed is normal, the UAVs move to the *FlyToWaypoint* state again. Similarly, when in the *TurnRateExceeded* state, if all the UAVs remain at a safe distance from each other, then the UAVs move to the *FlyToWaypoint* state again. For the simulator-specific self-adaptive behaviors, a UAV moves from the *FlyToWaypoint* state to the *fork* and then to the *RTL* or *Land* states. The purpose of the *Wind-Speed-Increased* transition in the *UAS-SM (S)* is to model the collision avoidance self-adaptive behavior in the UAS swarm formation flight application. This transition also represents a weather change, therefore it is stereotyped as

the *WeatherChange* stereotype. Moreover, an *OperationalChange* stereotype is also applied to this transition to constantly monitor safe distance violations in the UAS swarm. Furthermore, the states *FlyToWaypoint* and *Waiting* represent an adaptation in the UAS mission goals, therefore they are stereotyped as *Goal*. The *FlyToWaypoint* state represents the *ExecuteMission* goal of the mission and the *Waiting* state represents the *AvoidCollision* goal of the mission. These goals are specified in the *UAS-CD (S)* instance. For the generalized self-adaptive behaviors and self-adaptive behavior provided by the simulator (mission planner), we have a *FailSafeTriggered* transition from the *FlyToWaypoint* state to the *fork*. From the *fork*, the control moves to the required target state depending upon the transition that fires. One of the transitions is the *VibrationFailSafe* transition. The purpose of this transition is to model the self-adaptive behavior required as a result of the high vibration levels of the UAV. This behavior is provided by the Mission Planner. For this transition, the *FlyToWaypoint* state is the source, whereas the target is the *RTL* state. This transition triggers when the level of vibration is above a certain threshold, specified as mission requirements. High vibration levels can affect the climb rate of the UAVs. As this transition represents a change in the UAV, therefore, it is stereotyped as *UAVChange* (see details in Section 4.1.2).

In Figure 5B, various states and transitions are modeled to specify the UAS swarm formation flight self-adaptive behavior. Various stereotypes are applied according to the specific types of behavioral changes and behavioral adaptations. For example, we have a transition *Safe-Distance-Violation* from *WindSpeedExceeded* state to the *Waiting* state. The purpose of this transition is to model the behavior of a UAV when it crosses the safe distance limit from another UAV as a result of an increase in wind speed. As another UAV is an obstacle, therefore this transition is stereotyped as *ObstacleNearBy*. Moreover, we have a transition *CrashCheckTriggered* from the *fork* to the *Disarm* state. The purpose of this transition is to model the behavior of the UAV when it detects a UAV crash. When a crash is detected, the UAV disarms the motors to reduce the damage. This action is stereotyped as *DisarmingMotors* on the *CrashCheckTriggered* transition. The transition is stereotyped as *UAVChange* as it represents a change in the UAV.

4.4 | UAS adaptation rules

This section presents the details of the adaptation rules as part of the proposed UAS modeling methodology. An adaptation rule specifies the actions to be taken by the UAV whenever there is a behavioral change, based on the current situation of the UAS mission. The adaptation rules are specified on the UAS class diagram using OCL, as it is the standard language for specifying constraints (or restrictions).[35,45] The OCL constraints are specified on the modeled class diagram and enforce the UAS self-adaptive behavior in the modeled state machine. We as *Approach Provider* have specified the generalized adaptation rules (*UAS-AR (G)*) for the generalized class diagram (*UAS-CD (G)*) and state machine (*UAS-SM (G)*). The *Software Engineer* uses the *UAS-AR (G)* to specify under-development application-specific adaptation rules (*UAS-AR (S)*) for the application-specific class diagram (*UAS-CD (S)*) and state machine (*UAS-SM (S)*). Following, Sections 4.4.1 and 4.4.2 discuss the details of the generalized and application-specific adaptation rules respectively.

4.4.1 | UAS generalized adaptation rules

This section presents the details of the generalized adaptation rules for the UAS. For this purpose, we use the self-adaptive behaviors in the *UAS-SM (G)* and the attribute values of *UAS-CD (G)*. The generalized adaptation rules (*UAS-AR (G)*) are common to all UAS missions.

The *UAV* class in the *UAS-CD (G)* (as shown in Figure 4A) is the main class in the UAS generalized class diagram. All the other classes are accessible from this class, so all the adaptation rules are applied to the *UAV* class. There is a total of seven adaptation rules. An excerpt of the adaptation rules is shown in Listing 1.

The first adaptation rule specifies the UAV's action in case of a GCS disconnection (i.e., a connection loss between a UAV and its GCS). The context of the constraint is UAV (i.e., UAV class *UAS-CD (G)*). The invariant contains the mission attribute, that is accessible through a relationship between UAV and Mission classes. Further, the *basicSelfAdaptationStatus* attribute represents a relationship between Mission and *BasicSelfAdaptationStatus* classes. If a Boolean *GCSConnected* attribute in *BasicSelfAdaptationStatus* class is *false* and if *time* attribute in *BasicSelfAdaptationStatus* class has value greater than three seconds, then it confirms the GCS disconnection. This fires the transition (either one of *GCS-Loss* and *GCS-Loss2*) in the *UAV* class state machine (i.e., *UAS-SM (G)* as shown in Figure 5A) which leads to the target state that

is equal to the value of *GCSDisconnection* attribute in the *BasicSelfAdaptationSpecification* class. The *GCSDisconnection* attribute specifies the possible self-adaptive behavior actions of the UAV.

```
1 context UAV inv: (self.mission.basicSelfAdaptationStatus.GCSConnected=false and
      self.basicSelfAdaptationStatus.time > 5) implies
      (self.targetState=self.mission.basicSelfAdaptationSpecification.GCSLoss)
2 context UAV inv: (self.mission.basicSelfAdaptationStatus.GPSConnected=false and
      self.basicSelfAdaptationStatus.time > 5) implies
      (self.targetState=self.mission.basicSelfAdaptationSpecification.GPSLoss)
3 context UAV inv :(self.mission.basicSelfAdaptationStatus.RCConnected=false) implies
      (self.targetState=self.mission.basicSelfAdaptationSpecification.radioControlSignalLoss)
4 context UAV inv :(self.energySource.currentLevel=self.energySource.dischargeLevel) implies
      (self.targetState=self.mission.basicSelfAdaptationSpecification.batteryDischarged)
5 context UAV inv: (self.mission.waypoint->exists(w|w.waypointID=self.currentWaypoint and
      w.distanceFromPreviousWaypoint<self.currentDistanceFromWaypoint) or
      (self.currentHeading<>self.expectedUAVHeading and
      self.currentHeading<>self.expectedUAVHeading-5 and
      self.currentHeading<>self.expectedUAVHeading+5)) implies
      (self.targetState='ErrorState')
6 context UAV inv: (self.isWaypointLost=true) implies (self.targetState='RTL')
7 context UAV inv: (self.isWaypointLost=true) implies (self.targetState='Land')
```

Listing 1: UAS Generalized Self-Adaptive Behavior - Adaptation Rules

```
1 context UAV inv : (self.turnRate>self.turnRateLimit and
      self.currentDistanceToObstacle<self.mission.safeDistanceFromObstacle) implies
      (self.targetState='Waiting' and self.currentGoal='AvoidCollision')
2 context UAV inv : (self.turnRate>self.turnRateLimit and
      self.currentDistanceToObstacle>=self.mission.safeDistanceFromObstacle and
      self.timePassedInCurrentState>10) implies (self.targetState='FlyToWaypoint' and
      self.currentGoal='ExecuteMission')
3 context UAV inv : self.mission.environmentalFactor->forAll(e|e.name='WindSpeed' and
      e.value>self.flightSpecification.windResistance) implies
      (self.targetState='WindSpeedExceeded')
4 context UAV inv : (self.mission.environmentalFactor->forAll(e|e.name='WindSpeed' and
      e.value>self.flightSpecification.windResistance) and
      self.currentDistanceToObstacle>self.mission.safeDistanceFromObstacle and
      self.timePassedInCurrentState>10)) implies (self.targetState='FlyToWaypoint' and
      self.currentGoal='ExecuteMission')
5 context UAV inv : (self.mission.environmentalFactor->forAll(e|e.name='WindSpeed' and
      e.value>self.flightSpecification.windResistance) and
      (self.mission.environmentalFactor->forAll(e|e.name='WindDirection' and
      e.value<>self.currentHeading) and
      self.currentDistanceToObstacle<self.mission.safeDistanceFromObstacle and
      self.type='ArduCopter')) implies (self.targetState='Waiting' and
      self.currentGoal='AvoidCollision')
6 context UAV inv: (self.currentDistanceToObstacle>self.mission.safeDistanceFromObstacle)
      implies (self.targetState='FlyToWaypoint')
7 context UAV inv: (self.basicSelfAdaptationStatus.vibrationLevel>30) implies
      (self.targetState = self.basicSelfAdaptationSpecification.vibrationFailSafe)
```

Listing 2: UAS Swarm Formation Flight Self-Adaptive Behavior Adaptation Rules

The third adaptation rule specifies the UAV's action in case there is a radio control signal loss. The target state, as a result, is equal to the value of attribute *radioControlSignalLoss* in *BasicSelfAdaptationSpecification* class. The radioControlSignalLoss attribute value specifies the required UAV action. This rule is specified for the *RCLoss* and *RCLoss2* transitions of the *UAS-SM (G)*.

The fourth rule specifies the UAV's action in case the battery's current level is equal to the battery's discharge level. These values are specified as attribute values in the *EnergySource* class in relationship with the UAV class. As a result of this invariant, the UAV's target state is equal to the value of attribute *batteryDischarged* in the *BasicSelfAdaptationSpecification* class. This rule is specified for the *BatteryDischarged* and *BatteryDischarged2* transitions of the *UAS-SM (G)*.

The fifth rule specifies the UAV's action in case the UAV is moving away from its current waypoint instead of moving towards it. This is determined using the UAV's distance from the waypoint and the difference in the UAV's current and expected heading. As a result of this scenario, the UAV moves to *ErrorState* state. This adaptation rule is specified for the *Flying-To-Error* transition of the *UAS-SM (G)*. The sixth and seventh rules specify the UAV's action on *ErrorState* state. These rules change UAV's self-adaptive behavior during UAS mission execution. When the UAV is in *ErrorState* state, the *isWaypointLost* attribute in the formation flight class diagram instance is set during mission execution. So, if the value of this attribute is *true*, the UAV lands or *Returns-to-launch* (*RTL*). These two rules are specified for the *ErrorToLand* and *ErrorToRTL* transitions of the *UAS-SM (G)*.

4.4.2 | UAS application-specific adaptation rules

This section presents the UAS application-specific adaptation rules (*UAS-AR (S)*). The *Software Engineer* is responsible for defining *UAS-AR (S)*. Further, the *Software Engineer* modifies the *UAS-AR (G)* as per the requirements of the under-development UAS application software. The adaptation rules specific to the UAS swarm formation flight case study are seven and are shown in Listing 2.

The first formation flight adaptation rule specifies the follower UAV's actions in case there is a safe distance violation due to a high turning rate of the leader UAV or any other follower UAV. The context of this constraint is the *UAV* class. The safe distance is specified as a value of *safeDistanceFromObstacle* attribute in the *Mission* class of the *UAS-CD (S)* (as shown in Figure 4B). The current distance to the obstacle is specified as the *currentDistanceToObstacle* attribute value in the UAV class. The *turnRate* and *turnRateLimit* attributes are part of the UAV class.

The formation flight case study specifies in the requirements that if the turning rate of any UAV in the UAS swarm is greater than the *turnRateLimit* then it can cause a safe distance violation between the UAVs. The invariant in the first adaptation rule implements this requirement. In case the invariant evaluates to *true*, the first action of the rule is an adaptation in the UAV action from flying to waiting, whereas the second adaptation is in the change of UAV's goal from *ExecuteMission* to *AvoidCollision*. This adaptation rule is specified for the *SharpTurn* and *Safe-Distance-Violation-ST* transitions of the *UAS-SM (S)* (as shown in Figure 5B).

Similarly, the fifth rule specifies adaptation in the follower UAV's actions in case there is a safe distance violation due to changes in wind speed and wind direction. The context of this constraint is the UAV class. The value of wind speed is specified using the value attribute of the *EnvironmentalFactor* class. The wind resistance of a UAV is specified as the *WindResistance* attribute of the *FlightSpecification* class. The UAV class is associated with the *FlightSpecification* class using the *flightSpecification* attribute. The value of wind direction is also specified using the value attribute of the *EnvironmentalFactor* class. The direction of the wind is compared with the current heading of the UAV. The current heading of the UAV is specified as the *currentHeading* attribute of the UAV class. The safe distance is specified as a value of *safeDistanceFromObstacle* attribute in the Mission class. The UAV class is associated with the Mission class using the mission attribute. The current distance to the obstacle is specified as the *currentDistanceToObstacle* attribute value in the UAV class. The formation flight case study specifies as its requirement that a safe distance violation can occur between the leader and follower UAVs due to a wind speed being greater than the UAV's wind resistance and the wind direction being the same as the UAV's current heading. The invariant in the rule implements this requirement. In case the invariant evaluates to *true*, the first action of the rule is an adaptation in the UAV action from flying to waiting, whereas the second adaptation is in the change of UAV's goal from *ExecuteMission* to *AvoidCollision*. This rule is specified for the *Safe-Distance-Violation* transition of the *UAS-SM (S)*.

In the same way, we specify the adaptation rules for the remaining self-adaptive behaviors of the *UAS-SM (S)*. The invariant in the second rule specifies actions that a follower UAV takes if there is no safe distance violation observed (even after a time of 10 s) after a sharp UAV turn. This rule is specified for the *No-Distance-Violation* transition in the *UAS-SM (S)*. The invariant in the third rule specifies an adaptation to *WindSpeedExceeded* state in case the wind speed increases. This rule is specified for the *Wind-Speed-Increased* transition of the *UAS-SM (S)*. The invariant in the fourth rule specifies adaptation to the *FlyToWaypoint* state and the *ExecuteMission* goal in case no safe distance violation is observed (even after a time of 10 s) after an increase in the wind. This rule is specified for the *WindSpeedNormal* transition of the *UAS-SM (S)*.

4.5 | UAS mission modeling

This section presents the details of the UAS mission modeling. The *Mission Operator* is responsible for mission modeling. The mission modeling is always specific to the under-development UAS application software. The *Mission Operator* models the mission in a Mission Planner as per the requirements of the UAS application software. The modeled mission is represented as a mission file.

Figure 7 presents the mission file for the UAS swarm formation flight case study. This mission file is specific to the Mission Planner simulator.[42] The same mission in the Mission Planner simulator is shown in Figure 6. The first column in the mission file represents the ID of the waypoints. There are a total of four waypoints in the formation flight case study as shown in Figure 7. The ninth and the tenth columns of the mission file represent the latitude and the longitude of the waypoint. The eleventh column represents the altitude of a waypoint. The first waypoint is marked as the home waypoint.

We as *Approach Provider* have developed a mission metamodel for better readability of the modeled mission file. The mission metamodel is shown in Figure 8A. The instance of the mission metamodel is shown in Figure 8B. The mission metamodel instance presents the various waypoints, their IDs, latitudes, longitudes, altitudes, and types. For example, the first waypoint is associated with classes *ID*, *Type*, *Altitude*, *Latitude*, and *Longitude*. The *value* attribute of *ID* class has value 1, The *value* attribute of class *type* has value set as *home*. The *value* attribute of class *altitude* is set as 0.09. The *value* attribute of class *Latitude* is set as 33.681189. The *value* attribute of class *Longitude* is set as 73.036928.

FIGURE 6 Waypoints required by the UAV(s) to cover.

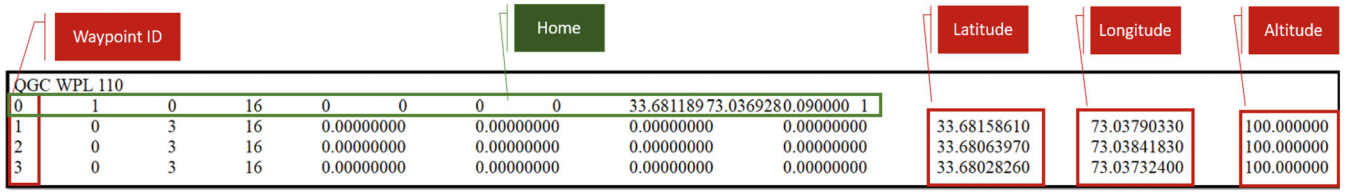


FIGURE 7 UAS swarm formation flight–Mission file.

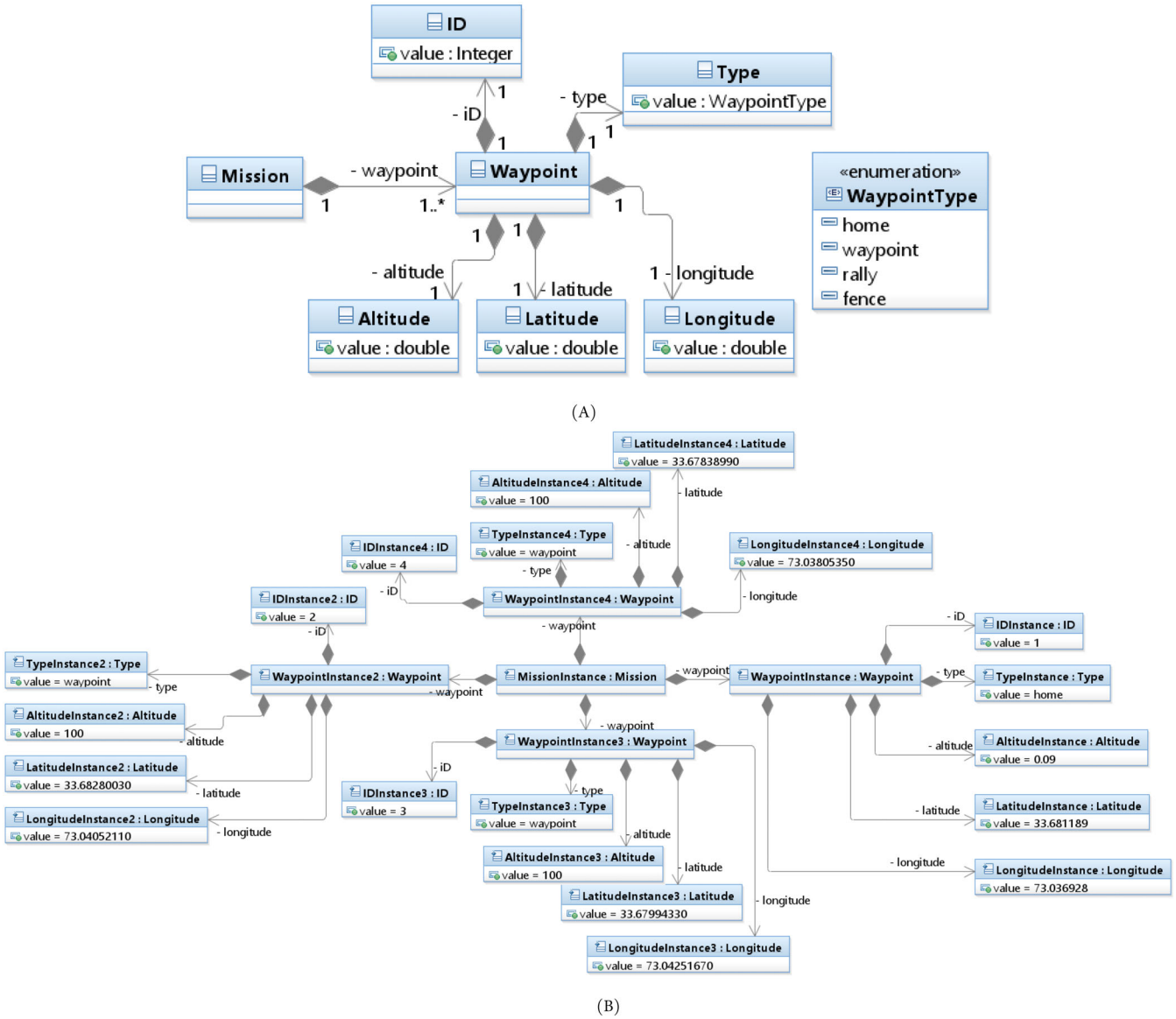


FIGURE 8 UAS mission modeling. (A) UAS mission metamodel, (B) UAS mission metamodel instance.

5 | MODEL-BASED UAS SELF-ADAPTIVE BEHAVIOR TESTING APPROACH

In this section, we discuss the details of our proposed model-based testing approach for the self-adaptive behavior of the UAS. Once the modeling of the mission and self-adaptive behavior for the under-development UAS application software is complete, next is the self-adaptive behavior testing to validate whether the UAS application software autonomously adapts as per the required behavior or not during mission execution. We use model-based testing for this purpose, as it is commonly used for the testing of various application software.[30,33,46] As part of the proposed model-based UAS application software testing approach, we have proposed: (i) a UAS domain-specific coverage criteria, the coverage criteria is used for the automated generation of test cases from the modeled UAS application software UML state machine, (ii) the UAS application testbed to execute the generated test cases in a simulator and logging their execution responses, and (iii) the verification of the test cases using logged responses and test results generation. We discuss the proposed coverage criteria in Section 5.1, Section 5.2 presents the details of automated test case generation, the details of test case execution and verification are discussed in Sections 5.3 and 5.4 respectively.

5.1 | UAS domain-specific coverage criteria

For the generation of test cases, the coverage criteria are an important factor that defines the type of test cases. There are very well-known coverage criteria that facilitate the generation of test cases but there are no criteria that are defined specifically to the UAS domain.

In order to make efficient use of the available resources, it is important to select and generate test cases that are useful in terms of testing goals. When we use a UML state machine for test sequence generation, some of the generated test sequences are infeasible. For example, startMission → Disarmed → arm → Armed → takeoff → Takeoff → takeoffAltReached → FlyToWaypoint. This test sequence does not complete a mission, therefore it is meaningless in the case of a UAS. Similarly, there are test sequences, that do not execute any aspect of the self-adaptive behavior. For example, startMission → Disarmed → arm → Armed → disarm → Disarmed → endMission. Transforming such test sequences into test cases, by identifying the right test data and oracle is not useful in terms of our testing goal, that is, to test the self-adaptive behavior in UAS.

Therefore, we propose three UAS domain-specific coverage criteria, that is, goal, behavioral changes, and behavioral adaptation. The coverage criteria allow the generation of test cases specific to the UAS domain concepts. The details of the proposed coverage criteria are discussed in the following subsections.

5.1.1 | All goals coverage criterion

This coverage criterion is defined to generate the UAS goal-based test cases. A goal is either mission-specific or generic. The mission-specific goal is the one that is specific to a particular mission, for example, *Avoid Collision* is a mission-specific goal in the UAS swarm formation flight case study. The generic goal is the one that belongs to the UAS domain and is common to all the missions, for example, *Execute Mission* or *Ensure Safety*. This coverage criterion allows the selection of those test cases that contain mission-specific or generic goals. The *Goal* stereotype defined in the proposed UAS profile (see details in Section 4.1) plays a vital role in the test case selection through this coverage criteria.

5.1.2 | All behavioral change coverage criterion

This coverage criterion is defined to generate test cases based on the changes that have an impact on the behavior of the UAS. During a UAS mission, there are three types of changes (i.e., changes in the UAS, changes in the environment, and changes in the operational needs of the mission)[7] that affect the behavior of the UAS. This coverage criterion allows to select the test cases that contain any of the three types of behavioral changes. There are various behavioral change stereotypes (e.g., *WeatherChange, ObstacleNearby, UAVChange, GCSChange, OpertaionalChange*, and others) defined in the proposed UAS profile (Section 4.1). These stereotypes play an important role in the test case selection through this coverage criterion.

5.1.3 | All behavioral adaptation coverage criterion

This coverage criterion is defined to generate UAS all behavioral adaptation (or adaptation alternative-based) test cases. The adaptations are in the UAV's action, in the payload's action, or in the goals of the mission. This coverage criterion allows to select the test cases that contain any of the behavioral adaptation actions or alternatives. There are various behavioral adaptation stereotypes (e.g., *AdaptPosition, AdaptWayoint, Land, RTL, Wait, Jump, Loiter*, and others) defined in the proposed UAS profile (Section 4.1.3). These stereotypes play an important role in the test case selection through this coverage criterion.

5.2 | Test case generation

This section provides details of the test case generation strategy as part of our proposed model-based testing approach. A test case includes test data, test sequence, and a test oracle.[47] The test sequence is the series of actions that a UAV(s)

takes during the mission execution. For example, disarm → arm → takeoff → flying → land → disarm is a test sequence. The value of variables involved in a test sequence execution is the test data. For example, the value of a UAV's battery, the value of the distance between UAVs in a swarm, or the value of the current waypoint. The purpose of a test oracle is to provide the expected value of a test case execution. For a UAS, the expected value is the expected UAV's behavior. For example, if the test data for *currentDistanceToObstacle* attribute is set to *less than safe distance*, then the expected UAV's behavior is to *wait*. For an automated generation of test cases, the first step is the generation of state machine paths from the under-development application's *UAS-SM (S)*. It is achieved based on the state transition testing 0-switch coverage criterion.[48]

Next is the transformation of the state machine paths into test sequences. Finally, the required test data is included in the test sequence to form a test case. The complete details of the test case generation are presented in the following sub-sections

5.2.1 | State machine path generation

The *Software Engineer* develops an application-specific state machine to model the self-adaptive behavior of a UAS. For example, a UAV moves to *Waiting* state if there is a safe distance violation between two UAVs. We use the transitions and the states in the state machine to generate test sequences for the testing of UAS self-adaptive behavior. One of the commonly used methods for the generation of test sequences using a state machine is to generate test sequences using the 0-switch coverage criterion.[49] We use multiple 0-switch coverages to cover all transitions. For this, we first flatten the state machine. For state machine flattening, we use the mechanism followed by Iftikhar et al.[31] After this, we generate the state machine paths satisfying the 0-switch coverage.

5.2.2 | State machine path selection

The generated paths from the under-development UAS application software's state machine contain the details of the transitions, states, and the UAS profile stereotypes. As the proposed UAS domain-specific coverage criteria are based on the UAS profile stereotypes, these stereotypes (in the generated paths) play an important role in the selection of the required state machine paths (or test sequence), later to be transformed as test cases. The pseudocode for state machine path selection is shown in Algorithm 1. For the all goal coverage criterion, the *Goal* stereotype applied to the states of the application-specific state machine is considered during state machine path selection. When this coverage criterion is used for the selection of state machine paths, then all those paths that contain the states with *Goal* stereotype are selected. For example, as shown in the formation flight state machine (Figure 5B), *FlyToWaypoint* and *Waiting* states are stereotyped as *Goal*, so all those state machine paths that contain these two states are selected as per all goal coverage criterion.

Similarly, for all behavioral change coverage criterion, the UAS profile stereotypes related to the behavioral change (see details in Section 4.1.2) applied on the transitions of the application-specific state machine are considered during state machine paths selection. For example, as shown in the formation flight state machine (Figure 5B), a *BatteryDischarged* transition from *FlyToWaypoint* state to *fork* to *RTL* state is stereotyped as *UAVChange*, so all those state machine paths that contain this transition are selected as per all behavioral change coverage criterion.

For all behavioral adaptation coverage criteria, the UAS profile stereotypes related to the behavioral adaptation (see details in Section 4.1.3) applied to the actions of the transitions in the application-specific state machine are considered during state machine path selection. For example, as shown in the formation flight state machine (Figure 5B), the *wait* action of the *Safe-Distance-Violation* transition from *WindSpeedExceeded* state to *Waiting* state is an example of the behavioral adaptation, so all those state machine paths that contain this transition are selected as per all behavioral adaptation coverage criterion.

We first generate the state machine paths using the 0-switch coverage criterion for the case study.

Once the paths are generated, all goal coverage criterion is used for state machine path selection. Using this criterion, a total of 348 state machine paths are selected. In the UAS swarm formation flight case study, there are two goals, one is a mission-specific goal (i.e., *AvoidCollision*), and the other is a generic goal (i.e., *ExecuteMission*). From a total of 348 state machine paths for all goal coverage, 144 (out of 348) cover the *ExecuteMission* goal, whereas 204 (out of 348) cover the *AvoidCollision* goal. The mission-specific or generic goals are specified and modeled as part of *UAS-CD (S)*

Algorithm 1. State machine path selection

Input: *SMP*: State Machine Paths, *SM*: *UAS-SM (S)*, *CC*: Required Coverage Criteria
Output:
List < path>SMP − S: Selected State Machine Paths for required coverage criterion

Begin
Load *SM*
SSC ∈ GetCoverageCriterionStereotypes(CC); ▷ this function returns the UAS profile stereotypes specific to the required coverage criterion
Traverse *SMP*
if *CC* == Goal
    Select all *SMP − S* with StateStereotypes ∈ *SSC*
elseif *CC* == BehavioralChange
    Select all *SMP − S* with TransitionStereotypes ∈ *SSC*
elseif *CC* == BehavioralAdaptation
    Select all *SMP − S* with ActionStereotypes ∈ *SSC*
else
Print Error
Endif

Algorithm 2. Test case generation

Input: *UASStereotypesCoverage*: State machine elements and applied stereotypes,
CC: Required Coverage Criteria,
List <path> *SMP − S*: Selected State Machine Paths for a coverage criterion
testData: test data for test cases
Output: *TestCases*

Begin
Traverse *SMP − S* ∈ *CC*
Read initial transition ∈ *SMP − S*
    initial mission test sequence ← initial transitions
    TestCase ← initial mission test sequence
Read self-adaptive behavior transitions ∈ *TTP − S*
    Get transition stereotypes ∈ *UASStereotypesCoverage*
    self-adaptive behavior test sequence ← *transition*
    self-adaptive behavior test sequence ← *testData*
    TestCase ← self-adaptive behavior test sequence
Read final transitions ∈ *SMP − S*
    final mission test sequence ← final transitions
    TestCase ← final mission test sequence

by the *Software Engineer*. Each of the generated state machine paths consists of a set of initial mission states and transitions, final mission states and transitions, and the self-adaptive-behavior states and transitions. For example, for the UAS swarm formation flight application's *UAS-SM (S)* as shown in Figure 5B, the initial-mission-states include: *Disarmed, Armed, Takeoff*, and *FlyToWaypoint*, the initial-mission-transitions include: *arm*, *takeoff*, and *takeoffAltReached*, the final-mission-states include: *Land*, *Armed*, *Disarmed*, and *Final*, and the final-mission-transitions include: *landing*, *reached-ground* and *end-mission*. These states and transitions almost remain similar in all the UAS applications, whereas the self-adaptive behavior states and transitions of a UAS for a mission vary from application to application. For the UAS swarm formation flight application, the self-adaptive behavior states are *FlyToWaypoint* and *Waiting*, whereas the self-adaptive-behavior transitions are: *Wind-Increased*, *Wind-Safe-Distance-Violation, Wind-No-Distance-Violation, Sharp-Turn-Safe-Distance-Violation, Sharp-Turn-No-Distance-Violation, At-Safe-Distance, Flying-To-Error, Error-To-land,*

Error-To-RTL, GPS-Disconnection, GCS-Disconnection, Battery-Discharged, and *Vibration-Failsafe*. All paths have the same initial and final states and transitions. If there is any stereotype applied to the initial and final states or transitions, then they affect the decision of state machine path selection, otherwise, they don't. Each of the generated paths is logged in the text file as: *Path-Id::Source-State, Transition, Target-State*. For example, a generated state machine path selected as per the all goal coverage criterion is: *59:: initial → startMission → Disarmed → arm → Armed → takeoff → Takeoff → takeoffAltReached → FlyToWaypoint → WindSpeedIncreased → WindSpeedExceeded → SafeDistanceViolation → Waiting → AtSafeDistance → FlyToWaypoint → FlyToError → ErrorState → ErrorToRTL → RTL → landing → Land → reachedGround → Armed → disarm → Disarmed → endMission → final* as shown in Figure 9 (highlighted in *red* color). In this path, the safe distance between the leader and the follower is violated due to the wind speed. As a result, the UAV moves to the *Waiting* state. The UAV moves to the *FlyToWaypoint* state when the leader and the follower are at a safe distance from each other. The UAV moves to *ErrorState* state if it is unable to reach the assigned waypoint. From the *ErrorState* state, the UAV moves to *RTL* state and finishes the mission.

5.2.3 | State machine path transformation

After the selection of the required state machine paths as per the required coverage criterion, next is the transformation of state machine paths to test cases. In our proposed testing approach, a test case represents a complete UAS mission, where a state machine path contains states and transitions (i.e., initial, final, and self-adaptive), and a mission model contains the waypoints (see details in Section 4.5). In the state machine path, the initial and final states and the transitions represent the start and end of a UAS mission execution respectively. For the transformation of these states and transitions, the modeled *Mission File* is used that is provided by the *Mission Operator*. For the transformation of self-adaptive behavior transitions and states, the applied self-adaptive behavior stereotypes on the states and transitions are used. Using these stereotypes, the transitions are transformed into their relevant test sequence. In a test sequence, the values of the attributes are provided as part of the test data. A high-level test path's transformation into a test case is shown in Figure 10. The test data is provided by a *Software Engineer*, who is considered a UAS domain and under-development application software expert. The test data is not just a random value, but rather a well-calculated value that would make a self-adaptive behavior occur during the UAS mission execution. The UAS mission is affected by various environmental factors during execution, so it is not that simple to make the self-adaptive behavior occur even with well-calculated test data. Therefore,

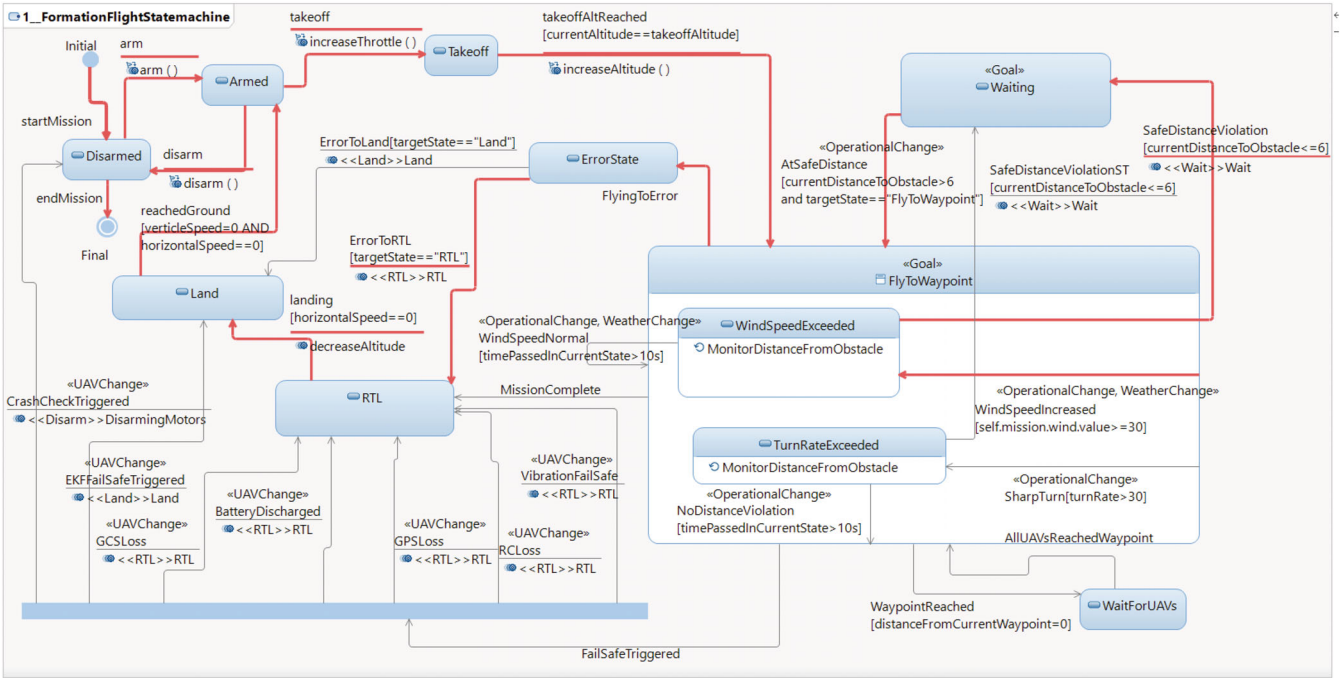


FIGURE 9 Formation flight test sequence.

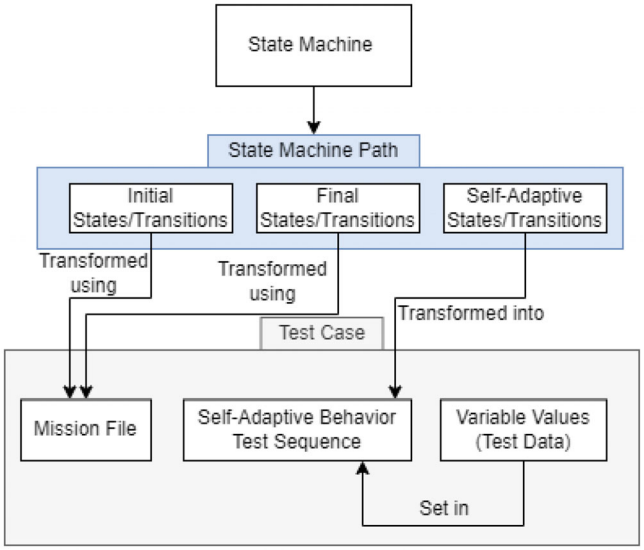


FIGURE 10 UAS self-adaptive behavior test case.

the domain expert also provides some alternative test data for the attributes involved in a test sequence. A pseudocode for the transformation of self-adaptive behavior states and transitions is shown in Algorithm 2. All the generated state machine paths are transformed into test cases. Each test case may belong to one or more self-adaptive behavior categories because of the applied UAS profile stereotypes.

5.2.4 | Test oracle

This section presents the details of the test oracle, which determines whether the UAS is able to correctly adapt the required behavior or not. If the self-adaptive behavior is adapted successfully by the UAVs during UAS application mission execution, then the test case is marked as *Pass*, otherwise it is marked as *Fail*. In our proposed model-based testing approach, a test oracle consists of a UAS application-specific class diagram (*UAS-CD (S)*) instance and adaptation rules (*UAS-AR (S)*). The *UAS-CD (S)* instance contains the actual output of the UAS application software. Actual output is saved as class diagram instances at run-time during mission execution. A class diagram instance is created with run-time simulation values whenever a transition related to a self-adaptive behavior fires. The *UAS-AR (S)*, on the other hand, contains the expected output, that is, the expected self-adaptive behavior of the UAS application software. The *UAS-AR (S)* are provided as OCL constraints by the *Software Engineer* as part of the modeling approach. The constraints are defined on the *UAS-CD (S)*. To reduce the chance of human error, we have verified the *UAS-AR (S)* with domain experts.

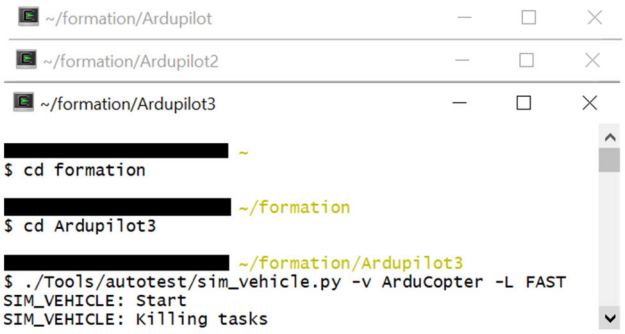
5.3 | Test case execution

In this section, we discuss the execution of the generated test cases.

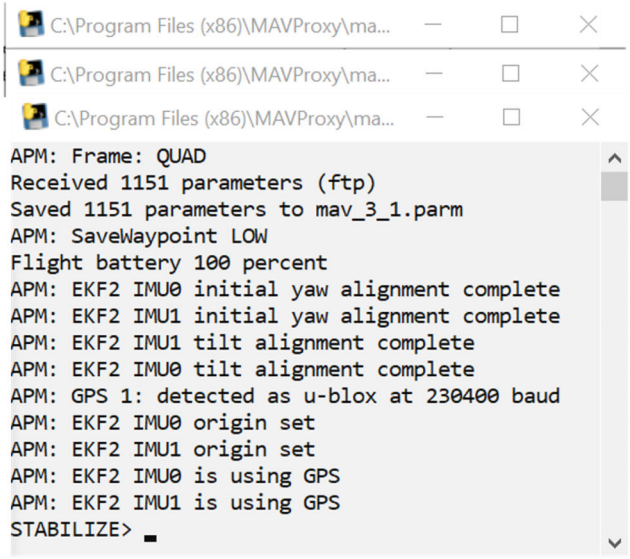
5.3.1 | UAS mission simulator configurations

For the execution of the generated UAS application software test cases, the configuration of the UAS mission simulator is an integral part. It includes a mission file, UAV(s), and simulator's setup. The UAS swarm setup includes the number of UAVs and their types. These are specified in the *UAS-CD (S)* instance developed by the *Software Engineer* as part of our modeling methodology.

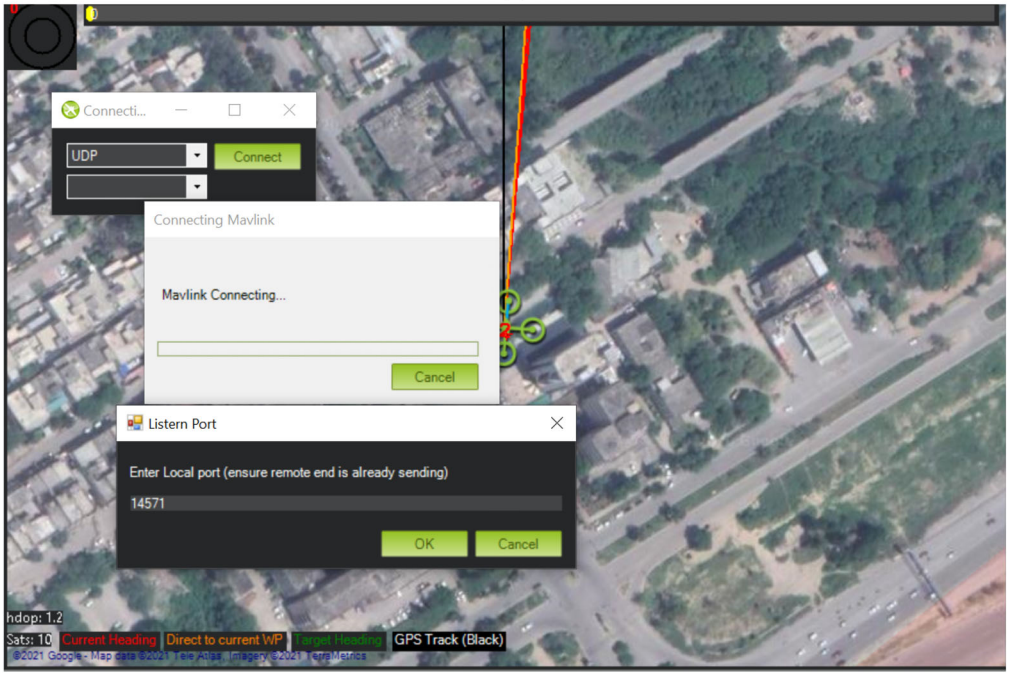
The simulator is required to be setup by making a connection with the UAV(s). For making a connection with the simulator, the Cygwin[50] terminals and MAVLINK[51] communication protocol are used as shown in Figure 11. A UAS



(A)



(B)



(C)

FIGURE 11 UAV connection with simulator (A) Cygwin terminals for three ArduCopters, (B) MAVLINK Windows for three Arducopters, (C) Connecting UAV(s) to the simulator.

swarm of three UAVs for copter type is connected to the simulator. Figure 11A, shows three Cygwin terminals, each representing an Ardu-Copter. Figure 11B shows three MAVLINK windows, one of each Ardu-Copter, appear after Cygwin commands execution, and Figure 11C shows the connection of each UAV with the simulator. When the UAS swarm is connected to the mission simulator, it completes the UAS swarm's setup. Next, the waypoint's details are provided to the mission simulator for the test case execution. One of the challenges of test case execution is reading different types of mission files and extracting the waypoint information from them. Every mission simulator has a mission file in a different format (*.txt* for Mission Planner, *.json* file for UgCS). To address this challenge, we use the mission metamodel (see details in Section 4.5). The waypoint's details are extracted from the Mission File using the UAS mission metamodel. A mission file is provided by the *Mission Operator* as part of our proposed modeling methodology. A mission file contains all waypoints and their relevant details (such as the waypoint's ID, its latitude, its longitude, its altitude, and its type), it is used to execute the mission (see Section 4.5 for reference). The waypoint information allows the execution of a mission on the simulator for test case execution. A UAS swarm's setup includes the setup of the mission and the UAVs. The setup of the mission is specific to the generated test cases. As every test case represents a complete mission, a mission file is re-executed on the simulator for every test case execution. This sets the current state of all the UAVs in the UAV(s) to disarm, sets the current waypoint to 0 (i.e., home), and initializes the default values of all the attributes. The test driver is responsible for the UAS setup and executing test cases one by one.

5.3.2 | Test driver

A test driver plays a very important role in UAS application software test case execution. All the generated test cases are provided to the test driver. The test driver takes one of the generated test cases at a time and executes it on the UAS mission simulator (i.e., Mission Planner) using the Ardupilot as autopilot. Once the execution of a test case is complete, the test driver moves to the next test case. This way the test driver executes all the generated test cases on the UAS mission simulator one by one.

To execute each test case, the test driver first executes the mission using the Mission planner simulator. A mission is executed using the waypoint information provided in the mission file. Next, the test driver has to execute the self-adaptive behavior test sequences in a test case. As a system requirement, the self-adaptive behavior test sequences are executed when the UAS swarm is in *FlyToWaypoint* state, that is, the *FlyToWaypoint* state is part of all the test cases. For the execution of self-adaptive behavior transitions, the test driver executes the self-adaptive behavior test sequences. An example of a self-adaptive behavior test sequence is that if the category of self-adaptive behavior is *Wind-Speed-Increased*, the test driver executes the test sequence to set wind speed and its direction during mission execution. The mission and the self-adaptive behavior transitions are executed in parallel. If a self-adaptive behavior does not occur using one value, a test case with an alternative value is re-executed.

During the UAS mission execution, the test driver constantly monitors the UAS swarm. For this purpose, the UAVs are connected with the mission simulator using MAVLINK communication protocol.[51] This is achieved through adding a port in each of the MAVLINK's screens. After the execution of each test sequence in a test case, the UAS swarm and mission status are observed. For example, Figure 12 shows the run-time status of the UAV 3 after a test sequence execution involving *Wind-Speed-Increased* and *Safe-Distance-Violation*. The UAV detracts from its path as a result of an increase in the wind speed and wind direction value. A purple line shows a detracted path in Figure 12. The effect of wind speed on the current distance of followers from the leader is shown as values of some involved attributes in Figure 12. These values are preserved to be used for test case verification later on.

Test logs

To validate whether the UAS application software adapts its goals or actions during mission execution, the UAV(s) and mission status are observed. The current status represents the current values of the attributes in the *UAS-CD (S)*. If any change occurs, the current state of the UAS application software execution is preserved as a test log. A test log contains a complete *UAS-CD (S)* instance containing the current values of the attribute. It is possible that a single test case contains multiple self-adaptive behaviors, therefore a test log is generated each time a self-adaptive behavior transition in a *UAS-SM (S)* is fired. So, there are multiple test logs for the execution of a single test case.

For example, for the SUT (UAS swarm formation flight application) test case shown in Figure 9 (highlighted in *red* color), there are four self-adaptive behavior transitions, as, *Wind-Speed-Increased, Safe-Distance-Violation,*

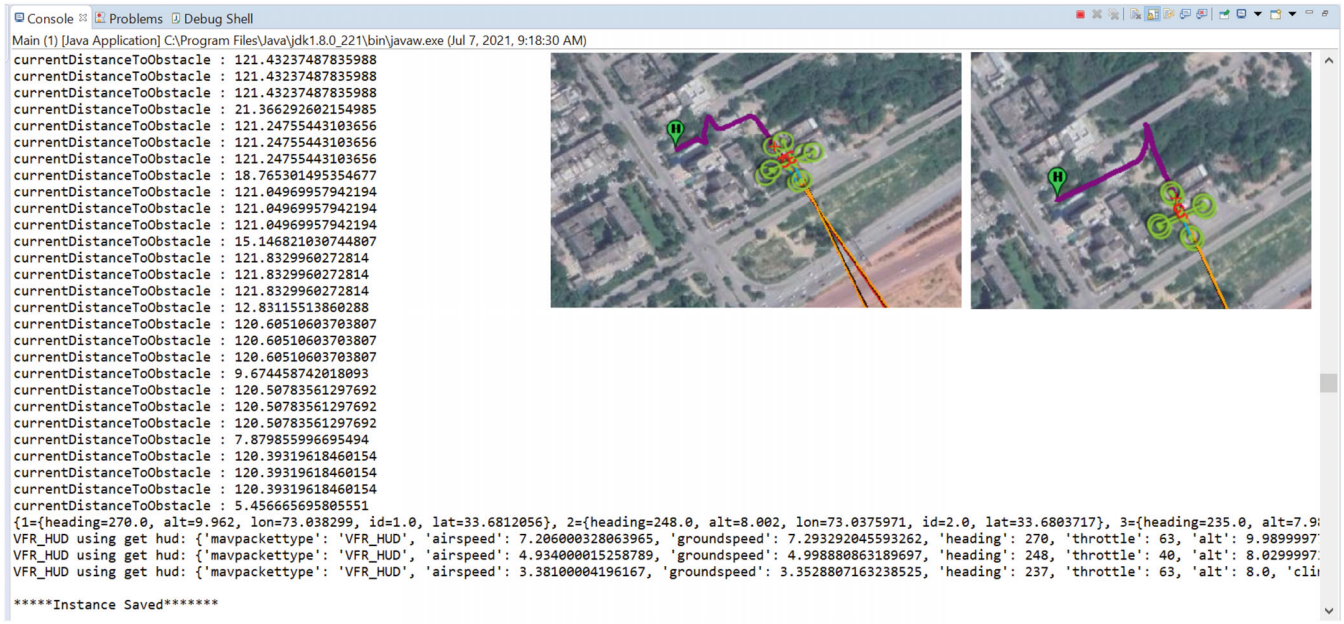


FIGURE 12 Run-time status of UAS after test sequence execution.

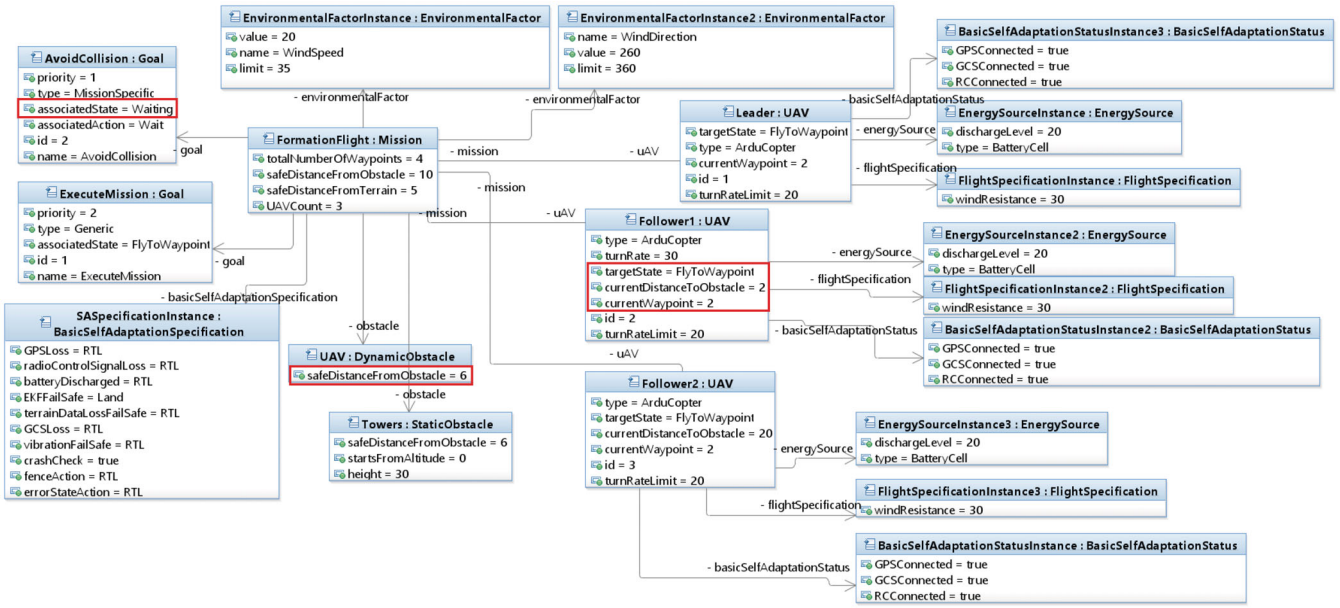


FIGURE 13 UAS swarm formation flight class diagram (*UAS-CD (S)*) updated instance.

Flying-To-Error, and *ErrorToRTL*. A test log is generated for each of these transitions. For the *Safe-Distance-Violation* transition, the *wind-speed, wind-direction*, *UAV's-current-heading*, and *UAV's-current-distance-from-obstacle* are among some of the attributes for which the values are preserved in a test log. The test log represents the *UAS-CD (S)* instance shown in Figure 13. The instance for the formation flight shows the expected values highlighted using *red* boxes. The expected values are given as input by the *Software Engineer*. The actual values (also highlighted using a *red* box on the UAV class) are values saved as test logs during test case execution. These values can also be seen in the sample test log. The expected output in case of *safe-distance-violation-due-to-wind*, is that the follower should move to the *Waiting* state. The actual output is the value of *targetState* attribute of the class UAV. The value is *FlyToWaypoint*.

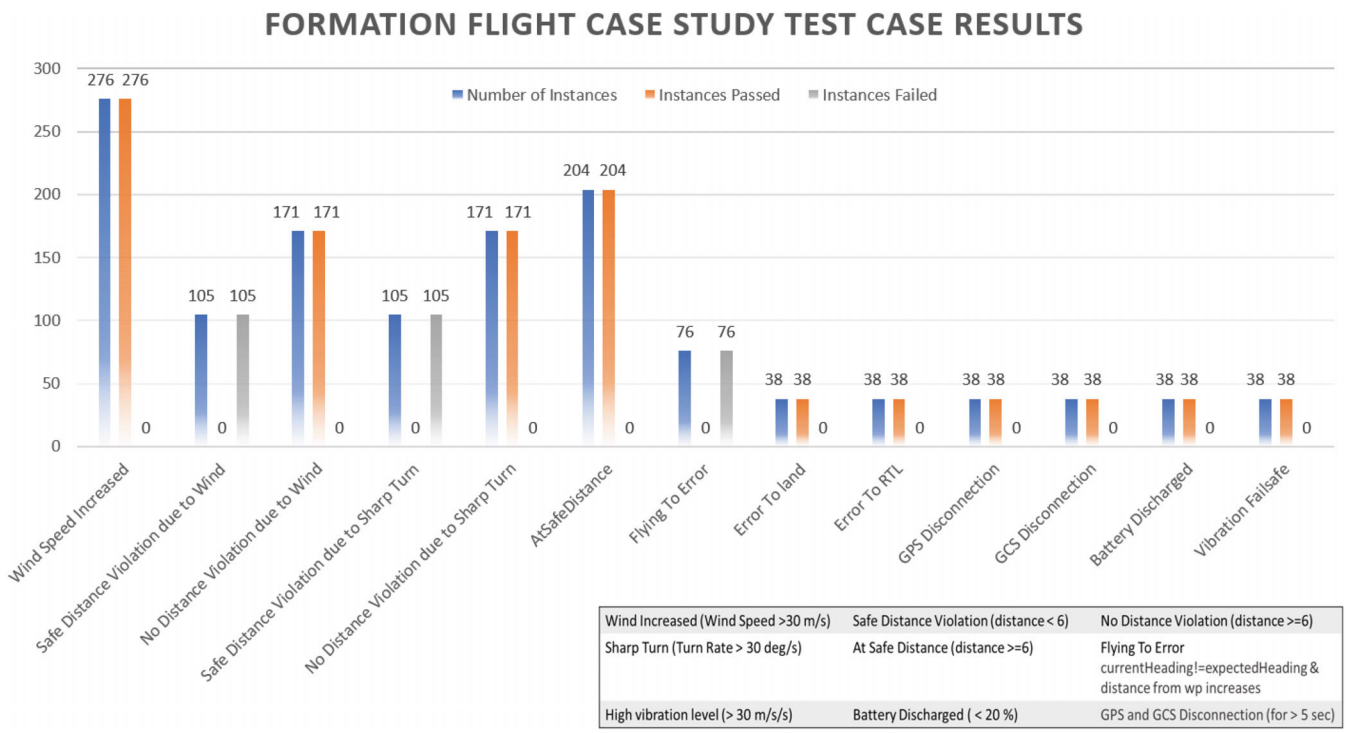


FIGURE 14 Formation flight case study verification results.

```
context UAV inv : (self.turnRate>self.turnRateLimit and
self.currentDistanceToObstacle<self.mission.safeDistanceFromObstacle) implies
(self.targetState='Waiting' and
self.currentGoal='AvoidCollision')
context UAV inv: (self.currentDistanceToObstacle>self.mission.safeDistanceFromObstacle) implies
(self.targetState='FlyToWaypoint')
context UAV inv :(self.energySource.currentLevel=self.energySource.dischargeLevel) implies
(self.targetState=self.mission.basicSelfAdaptationSpecification.batteryDischarged)
```

Listing 3: Adaptation Rules - Test Oracles

5.4 | Test case evaluation

This section presents the details for the verification of the executed UAS application software test cases. We have verified the results of test case execution (stored as UAS-CD (S) instances) against the OCL constraints (defining the UAS-AR (G) and UAS-AR (S)).

As discussed earlier, the test oracle contains a *UAS-CD (S)* instance and *UAS-AR (S)*. To validate OCL constraints on class diagram instance, an existing tool, that is, OclSolver[35] is used. An OclSolver is a model-based testing tool that allows validation of the correctness of the class diagram instance using OCL constraints.

Once all the test cases are executed, the test logs are available for each of the executed test cases. As there is a possibility of multiple test logs for a single test case and each test log contains values of attributes in the *UAS-CD (S)*, therefore a *UAS-CD (S)* instance is updated for every test log. Each instance is related to a particular self-adaptive behavior category. We divide the instances into categories depending on the self-adaptive behavior scenario they are related to. For each self-adaptive behavior scenario, we have a test oracle. This *UAS-CD (S)* instances and test oracle *UAS-AR (S)* are provided to the OclSolver tool for execution of OCL constraints on the *UAS-CD (S)* instance. The result is returned as *true* or *false*, where *true* reflects the correct self-adaptive behavior by UAS for the particular scenario (i.e., a specific transition in a test case).

In the UAS swarm formation flight case study, an example of test-case-16 selected on the basis of all goal coverage criterion. The test case consists of the following self-adaptive behavior categories: *Sharp-Turn*, *Sharp-Turn-Safe-Distance-Violation*, *At-Safe-Distance*, and *Battery-Discharged*. The adaptation rules (*UAS-AR (S)*) and the class diagram (*UAS-CD (S)*) instance as part of the test oracle for this test case are shown in Listing 3 and Figure 13 respectively. The first adaptation rule (Listing 3) when evaluated on the class diagram instance (Figure 13) through OCLSolver returns *false* as the expected target state for sharp turn and safe distance violation is *Waiting* and the expected goal is *Avoid-Collison*. But the actual *targetState* and *currentGoal* attributes are *FlyToWaypoint* and *ExecuteMission* respectively. The second adaptation rule (Listing 3) returns *true* as the expected output for *At-Safe-Distance* is *FlyToWaypoint* which is the same as the actual output for *targetState* attribute. The third rule returns *true* as for *Battery-Discharged* the expected and actual *targetState* attributes are the same, that is, *RTL*. Some of the all goal coverage criterion test cases with the preserved class diagram instances for self-adaptive behavior, the expected values, the actual outputs, and the result of the adaptation rules for each instance are presented in Table A1 of Appendix A.

During UAS swarm formation flight mission execution, there are a total of 1336 class diagram instances for all the 348 all goal coverage criterion test cases. Out of the 1336 instances, 105 instances belong to the *Wind-Speed-Safe-Distance-Violation*. All 105 instances returned *false* as a result of their relevant rules evaluation (see details in Section 4.4). A *false* result shows that the UAVs in a UAS swarm did not correctly adapt their behavior as a result of an operational and weather change. There are 105 instances that belong to the *Sharp-Turn-Safe-Distance-Violation*. All of these instances returned *false* which shows that when there is a safe distance violation due to a sharp turn, the UAVs are not able to correctly adapt behavior and move to *Waiting* state.

Once all the test cases are validated using their instances, a report of results is generated. The report contains *names* and *ids* of all the test case instances, the constraints executed on each test case instance, and the verification result for each instance.

6 | UAS SELF-ADAPTIVE BEHAVIOR TESTING AUTOMATION

To automate the generation and execution of test cases for the under-development UAS application software, a prototype tool named Unmanned-Aircraft-System-Self-Adaptive-Behavior-Testing-Tool (UAS-SBTT) is developed (available at GitHub[¶]). The UAS-SBTT takes the modeled class diagram, state machine, and mission file for the UAS application software as well as the coverage criterion, test data, and the expected class diagram instance with adaptation rules. The UAS-SBTT outputs whether the SUT self-adapts the required behavior or not during mission execution based on the current environment and UAS status. The UAS-SBTT architecture is presented in Figure A1 of Appendix A. The UAS-SBTT consists of two main parts, that is, Test Case Generation and Test Case Execution. There are a total of eleven components, where three components are third-party components and are *grey* colored, and eight components are developed as part of the proposed UAS application software testing approach and are *red* colored. The artifact that represents the input or outputs of a component is *blue* colored. Following, the details of the third-party components are presented briefly, whereas the developed components as part of the UAS-SBTT are discussed in detail.

6.1 | Test case generation

This section presents a description of the steps the developed UAS-SBTT takes for the test case generation.

1. State Machine Path Generator: The purpose of this component is to generate the state machine paths for the model-based testing of the UAS on the basis of the 0-switch coverage criterion. The component takes as input application-specific state machine (*UAS-SM (S)*). The component reads UAS-SM (S) elements (states, transitions, actions) and as an output, it generates state machine paths that satisfy the 0-switch coverage criterion.

[¶]https://github.com/ZainabJaved6/UAS-SBTT.git.

2. State Machine Path Selector: The purpose of this component is to select the state machine paths as per the model-based UAS coverage criteria (see details in Section 5.1). The component takes as input the UAS coverage criteria. The coverage criteria are based on the UAS UML profile. For all goal coverage, the component identifies the states with stereotypes applied and selects paths containing these states. For the all behavioral change coverage, the component identifies transitions with change stereotypes applied and selects paths containing these transitions. For the behavioral adaptations-based coverage, the component identifies actions with adaptation stereotypes applied and selects paths containing these actions. The state machine path selector gives as output the state machine paths based on the desired coverage criterion. The state machine paths selected are saved as text (*.txt*) files.
3. Paths Transformer: The purpose of this component is the transformation of state machine paths into executable UAS missions (test cases). This component takes as input the state machine paths and transforms them into UAS mission test sequences. The component also takes as input the test data provided by the *Software Engineer*. The test data is set in the test sequences to complete test case generation. This component gives as output the test cases.

6.2 | Test case execution

This section presents a description of the steps that the UAS-SBTT takes for the test case execution. One of the major components responsible for the test case execution is *Test Driver*. A test driver takes as input the generated test cases and executes them.

1. Test Case Executor: This is a sub-component of the *Test Driver*. The purpose of this component is to execute the generated test cases. For this, the component first configures the system and develops communication with the UAS mission simulator using the MAVLINK communication protocol. After this, it takes as input a test case and executes it on the UAS mission simulator. The simulator takes the mission file (waypoint information) as input. It is required to be executed as part of the test case execution. The mission file execution requires human input. The component gives the run-time UAS and mission status updates as output.
2. Run-time UAS and Mission Observer: This is a sub-component of the *Test Driver*. The purpose of this component is to observe the run-time situation of the UAS and the mission with the aim of checking if a self-adaptive behavior occurs or not. After a test case is executed, this component starts to receive as input the current UAS and mission status updates through the MAVLINK communication protocol.[51] For this, the component first makes a connection with the UAV(s) connected to the simulator using the MAVLINK. The component observes status updates to check if after executing self-adaptive behavior test sequences, a self-adaptive behavior occurs or not. If the required behavior occurs, this component gets the UAS and mission values at that instant and gives these values as output.
3. Test Log Preserver: This is a sub-component of the *Test Driver*. The purpose of this component is to generate a test log to preserve the outputs of the test case execution. This component takes as input the UAS and mission status values after the occurrence of a self-adaptive behavior and the *UAS-CD (S)* instance. The component updates the attribute values in a *UAS-CD (S)* instance as per the current status values received. The Test Log Preserver gives the updated *UAS-CD (S)* instance (test log) as output. From the Test Log Preserver, the control moves to the *Test Case Executor* component for the execution of the next test case until all test cases are executed.
4. Test Case Validator: The purpose of this component is to validate the test cases to check if the UAS correctly adapts its behavior or not. This component takes the test logs as input for verification. The test case validator makes use of a third-party tool, that is, OCLSolver.[35] The OCLSolver takes as input the application-specific class diagram (*UAS-CD (S)*), the application-specific adaptation rules (*UAS-AR (S)*), and the generalized adaptation rules (*UAS-AR (G)*). The inputs to the OCLSolver are given manually. This tool requires the class diagram (*UAS-CD (S)*) against which it receives the test logs (*UAS-CD (S)* instance). It evaluates the *UAS-AR (S)* and *UAS-AR (G)* on the *UAS-CD (S)* instance and gives as output the test log verification result. After verification of an instance, the Test Case Validator gives the next instance (test log) as input to the OCLSolver until all instances are validated.
5. Test Case Result File Generator: The purpose of this component is the generation of a report of self-adaptive behavior verification results. It takes as input the verification results of instances (test logs) generated against each test case and generates a report as an Excel file. The file contains *name* and *id* of each instance, the constraint executed on the instance, and its result.

7 | EVALUATION

This section presents the evaluation of our proposed model-based approach for testing the self-adaptive behavior of UAS, following the guidelines present in Reference 52. The evaluation has two goals: (i) to find out the ability of the proposed UML profile to cover identified self-adaptive behavior concepts and (ii) to demonstrate if the UAS is able to correctly adapt its behavior at run-time or not. Correct adaptation is defined in terms of *true* outcome of the evaluation of the defined OCL constraints (*UAS-AR (G)* and *UAS-AR(S)*). The following sub-sections present the research questions that provide the basis for the evaluation, the evaluation setup, the evaluation procedure, a discussion of the results, and the threats to the validity of the evaluation and the proposed approach.

7.1 | Research questions

For the evaluation of our proposed approach, two research questions are defined as:

Research Question 1: *Does the proposed UML profile cover the concepts required to model the self-adaptive behavior identified in the selected applications?* With this research question, we aim to increase confidence in the proposed UML profile to allow the modeling of self-adaptive behavior concepts identified in the selected UAS application software. This includes the modeling of UAS domain-specific concepts, mission as well as the required self-adaptive behavior of the UAS application software.

Research Question 2: *How effective is the proposed model-based testing approach in identifying the incorrect adaptation of the self-adaptive behavior for the UAS application software?* With this research question, the aim is to evaluate whether the proposed model-based testing approach is able to determine the incorrect execution of self-adaptive behavior in the UAS application software.

7.2 | Evaluation setup

To answer the first research question, it is required to completely model the structural and behavioral details of the UAS applications. For this, we have selected three open-source autopilot software for UAS: (i) Ardu-Copter,[#] (ii) Ardu-Plane,[‖] and (iii) Quad-Plane.[**] We have applied the proposed concepts of UML profile on seven applications of Ardu-Copter, nine applications of Ardu-Plane, and five applications of Quad-Plane. Out of seven applications of Ardu-Copter, one is the formation flight case study.

As part of the approach, the UAS domain-specific UML profile is provided by the *Approach Provider* for modeling the UAS domain-specific concepts during the UAS application software modeling. Moreover, the generalized class diagram (*UAS-CD (G)*), the state machine (*UAS-SM (G)*), and the adaptation rules (UAS-AR (G)) are provided to facilitate the *Software Engineer* with basic structural and behavioral modeling of the UAS domain.

We have modeled the structural details of all applications as *UAS-CD (S)* using the *UAS-CD (G)* and the UAS UML profile. We have modeled the behavioral details of the selected applications as *UAS-SM (S)* using the *UAS-SM (G)* and the UAS UML profile. Furthermore, we have modeled the adaptation rules for each application as *UAS-AR (S)* using the *UAS-AR (G)*. Modeling various applications has allowed us to improve the proposed UAS profile by adding and modifying concepts wherever needed.

The modeling statistics for the selected applications are shown in Table 1. It includes the details for the structural and behavioral modeling as well as adaptation rules. For all the modeled applications, no additional classes are required to be added in the modeled *UAS-CD (S)* besides *UAS-CD (G)*. The number of classes modeled in the *UAS-CD (G)* is thirteen. Moreover, the total number of applied stereotypes on the *UAS-CD (S)* classes is also the same, that is, 12. As these two details are the same for all applications that is why we have removed these columns from the table. We start the table by presenting the number of attributes added in the *UAS-CD (G)* for the particular application. The *Software Engineer* adds the required attributes in the classes present in the *UAS-CD (G)*. Next, we present the number of attributes in the *UAS-CD (S)* instance. These are the attributes that facilitate during UAS flight. Next, we present the statistics related to

[#] https://ardupilot.org/copter/.

[‖] https://ardupilot.org/plane/.

[**] https://ardupilot.org/plane/docs/quadplane-support.html.

TABLE 1 UAS applications modeling statistics.

Application	Attr added in UAS-CD (S)	Total attr in UAS-CD (S) instance	State (s)	Transition (s)	Action (s)	Guard (s)	Stereo-types in UAS-SM (S)	Total stereo-types applied	UAS-AR (G)	UAS-AR (S)
Ardu-Copter										
Formation-Flight-case-study	7	44	13	28	12	12	9	21	7	7
Shark-Spotting	0	34	10	16	10	3	10	22	3	4
Fire-Tracking	0	34	10	15	8	2	7	19	2	4
Neighbor-Malfunction	0	33	9	14	6	2	8	20	3	2
Neighbor-Malfunction2	0	33	9	15	7	4	8	20	3	3
Package-Delivery	0	40	11	17	10	2	8	20	3	6
Traffic-Monitoring	0	35	10	15	7	3	7	19	3	3
Ardu-Plane										
Fire-Tracking	0	34	10	15	6	1	7	19	3	3
Neighbor-Malfunction	0	33	9	14	5	0	8	20	3	2
Neighbor-Malfunction2	0	33	9	15	6	2	9	21	3	3
Safe-Distance-Violation	0	33	9	12	4	0	7	19	2	2
Safe-Distance-Violation-Wind	0	36	9	12	4	1	8	20	2	2
WaypointMission1	0	37	9	13	4	1	6	18	2	2
WaypointMission2	0	37	9	14	5	2	7	19	2	3
Traffic-Monitoring	0	35	10	14	5	1	7	19	2	3
Vibration-Level-Increased	2	33	8	12	5	2	5	17	2	2
Quad-Plane										
Neighbor-Malfunction	0	33	9	12	3	2	5	17	0	3
Safe-Distance-Violation	0	33	9	11	3	2	6	18	0	3
Traffic-Monitoring	0	35	10	13	5	1	6	18	0	4
WaypointMission1	0	37	9	13	4	2	7	19	1	3
Vibration-Level-Increased	2	33	8	13	6	4	5	17	1	4

the behavioral modeling of the selected applications. These include the states, transitions, actions, guards, and the total number of applied stereotypes on the *UAS-SM (S)*. Then we present the total number of applied stereotypes for a particular application. This includes a total of applied stereotypes on the *UAS-CD (S)* and the applied stereotypes on the *UAS-SM (S)*. In the end, we present the number of UAS-AR (G) and UAS-AR (S) defined for each application. These are the adaptation rules defined as OCL constraints. Next, we discuss the modeling statistics for the UAS formation flight case study in detail.

For structural modeling of the formation flight case study, the modeled *UAS-CD (S)* reused thirteen classes from the UAS-CD (G), where two of these classes were modified by adding attributes specific to the formation flight. There are a total of seven attributes added in the *UAS-CD (S)* specific to the case study. These attributes are added in the *BasicSelfAdaptationSpecification* and the *BasicSelfAdaptationStatus* classes. The total number of unique attributes in the *UAS-CD (S)* instance is 44. There is a total of 12 stereotypes applied on various classes in the *UAS-CD (S)*. For behavioral modeling of the case study, the modeled *UAS-SM (S)* contains a total of 13 states, 28 transitions, 12 actions, and 12 guards. The number of stereotypes applied to the *UAS-SM (S)* is nine.

For the verification of the test cases, there is a total of 14 OCL constraints. Out of the 14, seven belong to the self-adaptive behavior specific to formation flight, and seven belong to the generalized self-adaptive behavior. We specify one constraint for each self-adaptive behavior in the *UAS-SM (S)*. These constraints are modeled as Adaptation Alternatives by the *Software Engineer* (see details in Section 4.4). To answer the second research question, the test cases for each selected application are generated using the modeled *UAS-SM (S)* for each of the proposed model-based UAS coverage criteria. As discussed earlier in Section 5.2, the coverage criterion is applied on the state machine paths for the selection of test paths, that are later on transformed to test cases.

7.3 | Evaluation procedure

For the first research question, the UAS applications under test are modeled as *UAS-CD (S)* for structural details, *UAS-SM (S)* for behavioral details, *UAS-CD (S)*-instance for expected values, and *UAS-AR (S)* for self-adaption behavioral constraints. For this, a number of meetings were held with *Software Engineer* from our industrial partner organization. The details of the procedure followed are provided in Section 4.

For the second research question, the testing approach is applied to the modeled industrial UAS swarm formation flight case study, as well as the applications of Ardu-Copter, Ardu-Plane, and Quad-Plane. Firstly, the *UAS-SM (S)* is used for the generation of test cases based on the required UAS coverage criterion. Secondly, the test driver is written as well as the configuration of the UAS mission simulator for the execution of the test cases. Next, the generated test cases are executed as a UAS mission one by one and the test logs for each of the executions are preserved. Finally, the *UAS-CD (S)*-instance is updated for each of the test logs, and then every *UAS-CD (S)*-instance is verified through *UAS-AR (G)* and *UAS-AR (S)* for incorrect self-adaptive behavior adaptation. The details of this procedure are discussed in Section 5. The test case results are verified by the domain experts of our industrial partner organization.

7.4 | Results and discussion

This section provides the answers to the research questions and presents a discussion of the collected results.

Research question 1: *Does the proposed UML profile cover the concepts required to model the self-adaptive behavior identified in the selected applications?* **Answer**: To answer this research question, we have modeled 21 UAS applications. After improving the UAS UML profile (as discussed in Section 7.2), we are able to capture all required self-adaptive behavior concepts in the selected applications using the proposed UML profile. In all the modeled UAS applications, there are a total of 1399 applications of various concepts. These concepts include the classes in the *UAS-CD (G)* (such as *Mission, Goal, UAV, Obstacle*, and others), the attributes in the UAS-CD (S) instances (such as *currentDistanceFromObstacle, currentHeading, turnRate*, and others), the stereotypes applied on the *UAS-CD (S)* (such as *Mission, MissionGoal, EnvironmentalFactor*, and others), the stereotypes applied on the *UAS-SM (S)* (such as *NeighborChange, OperationalNeedChange, ObstacleNearby, and others*), and the adaptation rules in UAS-AR (G).

> The proposed UML profile was successfully used to model the self-adaptive behavior concepts identified in one industrial case study and twenty UAS applications.

Research question 2: *How effective is the proposed model-based testing approach in identifying the incorrect adaptation of the self-adaptive behavior for the UAS application software?* **Answer**: To answer this research question, an experiment is conducted by applying the proposed model-based testing approach to the industrial UAS swarm case study and the selected applications. Due to space limitations, we provide the results in Table A2 of Appendix A. The table includes the number of test cases generated for each application using the goal-based coverage criterion, the number of instances generated against each self-adaptive behavior occurrence in a test case, the number of self-adaptive behavior instances passed, and the number of self-adaptive behavior instances that failed after being evaluated on the defined OCL constraints. The details of the self-adaptive behaviors tested and the instances generated for each self-adaptive behavior for each of the selected applications are presented in Appendix A Table A3. Using the selected UAS applications, we have been able to test 27 unique self-adaptive behaviors for the Ardu-Copter, 18 unique self-adaptive behaviors for the Ardu-Plane, and 15 unique self-adaptive behaviors for the Quad-Plane. We are able to successfully automate the generation, execution, and evaluation of 512 test cases. For these test cases, we have generated a total of 1752 instances. These instances are generated for each transition in the state machine related to the self-adaptive behavior. Out of the 1752 instances, 1409 instances are evaluated to *pass*, whereas 343 instances are evaluated to *fail*. We have found out that in most of the failed instances, there is a major increase in the wind speed (i.e., greater than 25 m/s), or an increase in vibration levels (i.e., greater than 30 m/s), or an increase in the turn rates (i.e., greater than 30 deg/s). When the wind speed, vibration levels, and turn rates are higher, the UAV(s) are unable to correctly adapt their behavior. Following, we discuss the results of the industrial case study in detail. For the formation flight case study, a total of 348 test cases are generated for each of the UAS coverage criteria. The generated test cases cover both the goals in the case study (i.e., mission-specific and generic), therefore the test suite has 100% goal coverage. The generated test cases cover six out of eight types of behavioral changes modeled in the case study; therefore, the test suite has 75% behavioral change coverage. There are 25 total adaptation alternative stereotypes and five are used in the formation flight case study. Therefore, the generated test suite has 20% behavioral adaptation coverage. The behavioral changes and the adaptation alternatives coverage are calculated based on the number of stereotypes (in the proposed UAS profile) used by the case study. This is shown in Table 2.

There are a total of 1336 *UAS-CD (S)*-instances generated for all the 348 test cases. These instances belong to various self-adaptive behaviors in the formation flight UAS swarm, such as *Wind-Increased, Safe-Distance-Violation-due-to-Wind, No-Distance-Violation-due-to-Wind, Safe-Distance-Violation-due-to-Sharp-Turn, No-Distance-Violation-due-to-Sharp-Turn, At-Safe-Distance, Flying-To-Error, Error-To-land, Error-To-RTL, GPS-Disconnection, GCS-Disconnection, Battery-Discharged*, and *Vibration-Failsafe*. Each of the generated test cases covers more than one self-adaptive behavior, which is the reason there are multiple *UAS-CD (S)*-instances that belong to a single test case. Each instance (self-adaptive behavior category) is evaluated for its relevant OCL constraint, that is, *UAS-AR (S)* (shown in Appendix A Table A4) to form a test case result. The results are graphically shown in Figure 14. Following, the details for each of the self-adaptive behaviors in context to answer this research question are discussed.

1. The *Wind-Speed-Increased* self-adaptive behavior contains all those instances that are generated as a result of an increase in the wind speed during UAS swarm mission execution. As per the domain experts of the UAS domain, the increased wind value is greater than 30 m/s. Any wind speed value beyond this value is considered as increased wind

TABLE 2 Test case coverage count of formation flight case study.

Goal	Test cases	Behavioral change	Test cases	Adaptation alternative	Test cases
ExecuteMission	144	UAVChange	100	Wait	76
AvoidCollision	204	OperationalChange	6	Goal	6
		ObstacleNearby	166	Disarm	38
		ChangeInEnvironment	19	RTL	190
		WeatherChange	38	Land	38
		GCSChange	19		
Total test cases	348	Total test cases	348	Total test cases	348

because this wind speed is able to detract a UAV from its path and cause a safe distance violation between UAVs. The number of *UAS-CD (S)*-instances generated for *Wind-Speed-Increased* is 276. After evaluating these instances for *UAS-AR (S)* adaptation rules, the result returned is *true* (for all 276 instances). This means that when the wind speed increases, the follower UAV moves to a *WindSpeedExceeded* state, where it monitors the wind speed and its distance from the leader. The result reflects that the *Wind-Speed-Increased* self-adaptive behavior is adapted correctly by the UAS swarm in the formation flight case study.

2. The *Safe-Distance-Violation-due-to-Sharp-Turn* self-adaptive behavior contains all those instances that are generated as a result of a safe distance violation between the leader and the follower UAVs due to the sharp turning rate of the UAVs during formation flight. According to the UAS domain experts, any turn rate value greater than 30 deg/s is considered a sharp turn because this can cause the distance between UAVs to reduce. As per the UAS swarm formation flight application requirement, a distance of six meters between the UAVs in a UAS swarm is considered to be safe. The number of *UAS-CD (S)*-instances generated for this self-adaptive behavior is 105. On the evaluation of these instances for *UAS-AR (S)* adaptation rules, the result for all (105) instances is *false*. This means that when there is a safe distance violation due to a sharp turn, the follower UAV does not move to the *Waiting* state, rather it remains in *Fly-to-Waypoint* state. The result reflects that the *Safe-Distance-Violation-due-to-Sharp-Turn* self-adaptive behavior is not adapted correctly by the UAS swarm in the formation flight case study.
3. The *Safe-Distance-Violation-due-to-Wind* self-adaptive behavior contains all those instances that are generated as a result of a safe distance violation between the leader and the follower UAV due to wind. The safe distance is violated when the value of distance becomes less than six. As a result of this change, the follower UAV is expected to move to the Waiting state (specified in the relevant adaptation rule). The number of *UAS-CD (S)*-instances generated for this self-adaptive behavior is 105. On the evaluation of these instances for *UAS-AR (S)* adaptation rules, the result for all (105) instances is *false* as the follower UAV after a safe distance violation did not move to the Waiting state, rather it remains in *FlytoWaypoint* state. The result reflects that the *Safe-Distance-Violation-due-to-Wind* self-adaptive behavior is not adapted correctly by the UAS swarm in the formation flight case study.
4. The *At-Safe-Distance* self-adaptive behavior contains all those instances that are generated when the distance between leader and follower becomes safe (i.e., greater than or equal to six). The number of *UAS-CD (S)*-instances generated for this self-adaptive behavior is 204. On the evaluation of these instances for *UAS-AR (S)* adaptation rules, the result for all instances is *true*. This means that when the distance is safe, the follower UAVs move towards the *FlyToWaypoint* state. The result reflects that the *At-Safe-Distance* self-adaptive behavior is adapted correctly by the UAS swarm in the formation flight case study.
5. The *No-Distance-Violation-due-to-Wind* and *No-Distance-Violation-due-to-Sharp-Turn* self-adaptive behaviors contain all of those instances that are generated when there was an increase in the wind speed or when there was a sharp turn and the UAV kept on monitoring for 10 s but there was no safe distance violation. The ten second time is specified by the domain experts. As a result of this change, the UAV is expected to move back to the *FlyToWaypoint* state. The number of *UAS-CD (S)*-instances generated for both categories in 171 each. On the evaluation of these instances for *UAS-AR (S)* adaptation rules, the result for all (171) instances is *true*. This means that when there is no distance violation the UAVs correctly adapt their behavior and move back to *FlyToWaypoint* state. The result reflects that the No-Distance-Violation-due-to-Wind and the *No-Distance-Violation-due-to-Sharp-Turn* self-adaptive behaviors are adapted correctly by the UAS swarm in the formation flight case study.
6. The *Flying-To-Error* self-adaptive behavior contains all those instances that are generated as a result of UAV not being able to reach the assigned waypoint. In this case, the UAV is expected to move to the *ErrorState* state, according to the adaptation rule. The number of *UAS-CD (S)*-instances generated for this is 76. On the evaluation of these instances for *UAS-AR (S)* adaptation rules, the result for all (76) instances is *false*. This means the UAVs did not move to the *ErrorState* state when they are unable to move towards the waypoint, rather they keep on trying to move towards the assigned waypoint. The result reflects that the *Flying-To-Error* self-adaptive behavior is not adapted correctly by the UAS swarm in the formation flight case study.
7. The *Error-To-Land* and *Error-To-RTL* self-adaptive behaviors contain all those instances that are generated when a UAV is at the *ErrorState* state and to ensure a UAV's safety it is supposed to either land or RTL. The number of *UAS-CD (S)*-instances generated for *ErrorToLand* is thirty-eight and for *ErrorToRTL* are also 38. On the evaluation of these instances for *UAS-AR (S)* adaptation rules, the result for all instances is *true*. This means that the

UAVs correctly move to *Land* or *RTL* states when it is at the *ErrorState* state. The result reflects that the *ErrorToLand* and the *ErrorToRTL* self-adaptive behaviors are adapted correctly by the UAS swarm in the formation flight case study.

8. The *GPS-Disconnection*, *GCS-Disconnection*, *Battery-Discharged*, and *Vibration-Failsafe* self-adaptive behaviors contain instances that are generated as a result of GPS signal loss for five seconds, a GCS signal loss between UAV and GCS for 5 s, a discharged battery (<20%), and a high vibration level (>30 m/s/s) respectively. The threshold value for each self-adaptive behavior is specified by the domain experts. The number of *UAS-CD (S)*-instances generated for each of these self-adaptive behaviors is 38. On the evaluation of these instances for *UAS-AR (S)* adaptation rules, the result for all instances is *true*. The result reflects that the *ErrorToLand* and the *ErrorToRTL* self-adaptive behaviors are adapted correctly by the UAS swarm in the formation flight case study.

We have verified the above self-adaptive behavior verification results with our industry partners as well. The results show that using the proposed model-based testing approach, we are able to generate and execute test cases that test the SUT in extreme scenarios, that is, scenarios that may result in swarm collision if executed in the field. Furthermore, using the proposed approach we are able to automate the generation, execution, and evaluation of 348 case study test cases. This allows our industry partner to have confidence in the SUT. Moreover, the verification results *false* (shown in Listing 5) for the case study reflect that the UAS swarm formation flight application is unable to execute the required self-adaptive behavior in case where there is a safe distance violation between the UAVs. This shows a possible fault in the self-adaptive behavior definition of the SUT for the safe distance violation change. Furthermore, the UAS swarm is unable to transit to an *ErrorState* state when a UAV fails to reach its waypoint. This behavior shows a possible fault in the self-adaptive behavior of the *ErrorState* state of the SUT. The verification results *true* (shown in Listing 4) show the correct run-time self-adaptation of the SUT. The UAS swarm is able to correctly adapt (i) when it is required to monitor the current situation as a result of a change, (ii) when the generalized self-adaptive behavior changes occur, and (iii) when the UAVs are required to adapt to *FlyToWaypoint* state as a result of *AtSafeDistance* self-adaptive behavior.

The results conclude that the proposed model-based testing approach is able to identify incorrect execution of the self-adaptive behavior in UAS application software.

```
1  context UAV inv: (self.turnRate>self.turnRateLimit and
       self.currentDistanceToObstacle>=self.mission.safeDistanceFromObstacle and
       self.timePassedInCurrentState>10) implies (self.targetState='FlyToWaypoint' and
       self.currentGoal='ExecuteMission')
2  context UAV inv: self.mission.environmentalFactor->forAll(e|e.name='WindSpeed' and
       e.value>self.flightSpecification.windResistance) implies
       (self.targetState='WindSpeedExceeded')
3  context UAV inv : (self.mission.environmentalFactor->forAll(e|e.name='WindSpeed' and
       e.value>self.flightSpecification.windResistance) and
       self.currentDistanceToObstacle>self.mission.safeDistanceFromObstacle and
       self.timePassedInCurrentState>10)) implies (self.targetState='FlyToWaypoint' and
       self.currentGoal='ExecuteMission')
4  context UAV inv: (self.isWaypointLost=true) implies (self.targetState='Land')
5  context UAV inv: (self.isWaypointLost=true) implies (self.targetState='RTL')
6  context UAV inv: (self.currentDistanceToObstacle>self.mission.safeDistanceFromObstacle)
       implies (self.targetState='FlyToWaypoint')
7  context UAV inv: (self.basicSelfAdaptationStatus.vibrationLevel>30) implies
       (self.targetState = self.basicSelfAdaptationSpecification.vibrationFailSafe)
8  context UAV inv: (self.mission.basicSelfAdaptationStatus.GCSConnected=false and
       self.basicSelfAdaptationStatus.time > 5) implies
       (self.targetState=self.mission.basicSelfAdaptationSpecification.GCSLoss)
9  context UAV inv: (self.mission.basicSelfAdaptationStatus.GPSConnected=false and
       self.basicSelfAdaptationStatus.time > 5) implies
       (self.targetState=self.mission.basicSelfAdaptationSpecification.GPSLoss)
10 context UAV inv :(self.energySource.currentLevel=self.energySource.dischargeLevel) implies
       (self.targetState=self.mission.basicSelfAdaptationSpecification.batteryDischarged)
```

Listing 4: Adaptation Alternatives - Returned True after Evaluation

```
1 context UAV inv : (self.turnRate>self.turnRateLimit and
      self.currentDistanceToObstacle<self.mission.safeDistanceFromObstacle) implies
      (self.targetState='Waiting' and self.currentGoal='AvoidCollision')
2 context UAV inv : (self.mission.environmentalFactor->forAll(e|e.name='WindSpeed' and
      e.value>self.flightSpecification.windResistance) and
      (self.mission.environmentalFactor->forAll(e|e.name='WindDirection' and
      e.value<>self.currentHeading) and
      self.currentDistanceToObstacle<self.mission.safeDistanceFromObstacle and
      self.type='ArduCopter')) implies (self.targetState='Waiting' and
      self.currentGoal='AvoidCollision')
3 context UAV inv: (self.mission.waypoint->exists(w|w.waypointID=self.currentWaypoint and
      w.distanceFromPreviousWaypoint<self.currentDistanceFromWaypoint) or
      (self.currentHeading<>self.expectedUAVHeading and
      self.currentHeading<>self.expectedUAVHeading-5 and
      self.currentHeading<>self.expectedUAVHeading+5)) implies
      (self.targetState='ErrorState')
```

Listing 5: Adaptation Alternatives - Returned False after Evaluation

7.5 | Threats to validity

This section presents the threats to the validity of the proposed work.

The first threat is related to the generalization of the proposed modeling approach in self-adaptive behavior modeling of any type of UAS mission. To mitigate this threat, we have applied the approach to a large-scale industrial case study and on other 20 open-source UAS applications. In these, we have successfully modeled structural concepts of the selected applications as 273 instances of classes, 724 instances of attributes, and 252 instances of stereotypes on classes. Furthermore, we successfully modeled 150 instances of self-adaptive behavior stereotypes. The second threat is related to manually modeling the artifacts (class diagram, state machine, adaptation rules) as part of our approach. As humans are involved there is always a chance of error in modeling, which can further have an impact on the test case generation and the self-adaptive behavior verification. To mitigate this threat we verified the developed models with UAS domain experts and fixed the identified issues.

8 | RELATED WORK

Recently, the UAS has been an active research area. This section presents the existing literature on (i) mission and self-adaptive behavior specification of UAS, (ii) mission and self-adaptive behavior specification of any unmanned vehicle, and (iii) testing of mission or self-adaptive behavior in the UAS.

First, we discuss literature on *UAS mission and self-adaptive behavior specification*. Dui et al.[22] proposed a mathematical modeling approach for the self-organization of the UAS swarm, intending to improve mission reliability. The proposed approach models the UAS swarm structure but no support for UAS swarm behavioral modeling is available. Moreover, the approach only focuses on self-organization. It does not allow modeling of other aspects of self-adaptive behavior (such as adaptations in UAS goals or actions) that occur as a result of a varied set of changes specific to the UAS swarm (such as changes in the UAV, its operational needs, and its environment). Cybulski et al.[23] proposed a formal modeling approach for defining the UAS swarm mission and its behavior. The normal behavior of UAS is considered during a mission. The approach does not support self-adaptive behavior modeling. Findeis et al.[53] focused on one aspect of the self-adaptive behavior, that is, the changes in weather (specifically wind) which may affect the swarm during its mission. For this, the paper proposed a modeling approach to support the organization of the UAS swarm in a way that minimizes the impact of wind on the swarm as a whole. The approach does not provide support for modeling UAS mission and other possible adaptations (adaptation in UAV actions, and goals). It also does not cater to other types of changes (changes in operational needs, changes in UAS) that can occur in the UAS swarm during mission execution. Madni et al.[7] proposed a design for resilient systems using formal modeling. For the evaluation of the approach, they have used a UAS swarm. They have modeled the architecture and behavior of resilient UAS swarm considering UAV crashes. The paper also discusses the various types of changes that can occur in a UAS swarm during a mission requiring the UAS to adapt.

These changes and the required UAS actions are discussed in the text. The paper does not provide any modeling notations for these concepts. Only describing in the text is ambiguous, whereas modeling benefits for better understandability, reuse, and automation using models. Revill[54] proposed the use of a specific formal modeling methodology for UAS swarm self-adaptive behaviors. The behaviors are modeled as all possible interactions between agents of the UAS. To determine if the UAS can adapt its behavior as the modeled interaction specifies, the approach requires the users to manually analyze and trace the interactions. Manual analysis is labor-intensive. There is no simulation or field execution of the behaviors. Also, no run-time adaptation is shown. The approach is not evaluated using any case study. Ordoukhaian et al.[25] proposed rules for self-adaptive behavior decision-making. The self-adaptive behavior considered focused only on the adaptation of generic goals, such as mission completion, UAV's safety, or hardware resources preservation. It does not define rules for the adaptation in mission-specific goals (such as *track-traffic-signal-violating-vehicle* to *RTL*, or *monitor-vehicle-accident* to *Land*, and others) or UAS actions (such as *to wait*, *increase-speed*, or *increase-altitude*). Moreover, the approach allows to model the architecture of the UAS but it does not provide any support for behavioral or structural modeling. Maia et al.[38] proposed an aspect-oriented programming (AOP) approach for the self-adaptive behavior of a single UAV. The paper models the behavior of the UAV and generates wrappers using AOP for the self-adaptive behavior. The paper does not provide support for modeling the structural details of a UAS. Also, does not provide concepts to model the self-adaptive behavior of UAS. Alves et al.[55] proposes a meta-model to capture concepts of aspect-oriented programming to enable developers to write self-adaptive behaviors in the form of aspects in a drone-based application. The meta-model concepts are dependent on the code-level details and are only applicable if aspect-oriented programming is used. Our modeling methodology, on the other hand, is generic and is not specific to code-level details. Araujo-de-Oliveira et al.[9] discusses the problem of evaluating self-adaptive systems at design time. According to them, a possible solution is to have a flexible architecture for self-adaptive software systems that includes flexibility in requirement specification, system, and behavior modeling. Building upon previous work, the authors provide support for the simulation of the executable specification of a self-adaptive system. The work enhances simulation capabilities by allowing dynamic adaptations to better understand the effects of adaptations on the overall system model. The paper proposes generic concepts for modeling the architecture of a self-adaptive software system and does not capture concepts specific to the UAS domain. Roy et al.[56] proposes a domain-specific language (DSL) for autonomous and robotic systems. The authors address the problem of conflicting non-functional requirements (NFR) in autonomous and robotic systems therefore, the proposed DSL captures the concepts of the operational contexts of the robots required to specify the non-functional requirements. On the other hand, the focus of our modeling methodology is on capturing concepts related to the mission and self-adaptive behavior of the UAS domain.

Next, we discuss literature that is not necessarily specific to the UAS domain but proposes generic approaches for mission definition that apply to any *unmanned vehicle*. We also present approaches for self-adaptive behavior specification that are not specific to the UAS but apply to any self-adaptive system. Dhouib et al.[39] proposed a domain-specific language (DSL) for mission specification of cyber-physical systems such as autonomous cars, unmanned water vehicles, and unmanned aerial vehicles. The language allowed modeling of the architecture and behavior of a system. It does not allow to model the structural details of the mission. Also, the language does not provide support for self-adaptive behavior. Similarly, Ruscio et al.[40] proposed a DSL for mission modeling of unmanned systems, such as unmanned ground, air, and water vehicles. It allowed modeling of the structural aspects of the mission. The language does not provide support for the behavior modeling and the definition of self-adaptive behavior. Carter et al.[57] proposed the use of SysML[58] for modeling a cyber-physical system's mission for security vulnerabilities. The approach only allows to model mission for security-related concepts, and not the whole mission (e.g., modeling waypoints, goals, UAVs, and others). Moreover, there is nothing on modeling UAS self-adaptive behavior. Islam et al.[59] proposed the use of feature models to adapt the generic goals of an autonomous car at run-time. The feature models were created for the car's operative context, physical elements of the cars, and concepts for different types of car actions. The work does not provide details about modeling and testing the self-adaptive behavior of the cars. The modeling concepts specific to UAS (changes in UAS, adaptations in UAS behavior) are not considered. Han et al.[60] proposed a modeling approach for any self-adaptive system (such as a computer system or a UAV). The approach provides concepts to model the architecture of such a system and also provides concepts to model self-adaptiveness in such systems. As the approach is generic to any self-adaptive system, it does not provide support to model concepts specific to the structure or behavior of a UAS mission (such as waypoints, goals, involved UAVs) and its self-adaptive behavior (i.e., the changes and adaptations specific to a UAS). Petrovska et al.[61] highlights the importance of understanding the definition of self-adaptiveness of a system to engineer it. Therefore, the paper provides a formal definition of the self-adaptiveness in cyber-physical systems (CPS). Using the formally defined concepts the paper also proposes an architecture for the engineering of self-adaptive CPS. As the focus

of the paper is on any CPS, the concepts defined in the proposed architecture are at an abstract level. The concepts specific to the UAS domain are not defined. Yang et al.[11] address the challenge of adaptation in the formation control of UAV swarms, considering uncertainties in their mass and inertia during mission execution. The authors discuss adaptiveness in terms of maintaining formation during flight, emphasizing its critical role in the accuracy of path-following algorithms. Consequently, they propose a model and implementation for adaptive formation control specifically tailored for fixed-wing UAVs. Similarly, Rosa et al.[62] also discusses adaptive formation control of multi-agent systems. The paper describes the multi-agents as Euler-Lagrange systems and proposes an adaptive formation control algorithm utilizing the concept of distributed synchronization between the agents for coordination. To test the proposed algorithm, the authors use a formation of fixed-wing UAVs. In contrast, our work not only encompasses self-adaptive behavior concerning UAS swarms but also extends to any behavior adaptation that an individual UAV can make in its application software to adapt to changes in itself or the environment. Moreover, the focus of the work proposed by the authors is on testing path-following algorithms or formation control algorithms while our work aims to test mission-specific self-adaptive behaviors.

Finally, we discuss literature on *testing of UAS mission and self-adaptive behavior*. Ma et al.[12] proposed a UML profile for the self-healing behavior of cyber-physical systems. The profile was to be used for the automated testing of the self-healing property of the system. Self-healing is the property of self-adaptiveness. Its focus is only on faults in a system and adaptations to heal the faults. The generation of test cases is automated. The testing is simulation-based. The approach allows to model concepts specific to self-healing. Self-healing occurs when there is a fault. But self-adaptive behavior is not only required in case of fault. Rather, self-adaptive behavior is also required when there is a change in the UAS. The change can be in the UAS environment, in the UAV's system, in the UAV's neighbor, or the UAS mission operational needs. The approach allowed modeling the structural and behavioral concepts of the UAS considering only the self-healing behavior. It does not provide support to model a UAS mission and requirements of the self-adaptive behavior. Wang et al.[13] proposed a test-bed for the validation of UAS swarm collaborative algorithms. The authors performed field testing as well as simulation-based testing. Testing is limited to the collaborative behavior of the UAS swarm. The evaluation showed only the initial collaborative behavior of the quadrotor. It does not show collaboration during mission execution. Moreover, the approach does not validate other self-adaptive behaviors, such as adaptation in goals of the UAS, or adaptations in UAS actions. Zhao et al.[14] proposed a test-bed for the basic behaviors of a UAS swarm, such as hovering or following the defined trajectory. It does not allow to test the self-adaptive behavior of the UAS. Arefin et al.[15] proposed a system-level model-based and search-based testing approach for the behavior provided by the MicroUAV's autopilot (such as following trajectories and responding to environmental factors). A behavioral model is developed and a code-based coverage criterion multiple condition decision coverage (MC/DC) is used. The coverage criteria used are at a concrete level (specific to the used autopilot). Similarly, the self-adaptive behaviors tested are specific to the considered Autopilot. The approach does not allow testing of mission-specific self-adaptive behavior (self-adaptive behavior required in response to changes in operational needs). Cwiakala et al.[63] proposed an approach to test if a single UAV can maintain flight control while following the specified path in different weather conditions and on different types of terrains. The approach does not support the testing of the self-adaptive behavior of the UAS. Khatiri et al.[16] addresses the problem of correct representation of testing scenarios in the simulation-based environment. The proposed approach is limited to the testing of the functional behavior of the UAVs. Similarly, Sarkar et al.[17] proposes a framework that is limited to the testing of the functional behavior (the mission) of the UAVs. Agrawal et al.[18] propose a platform that allows the software engineers to specify application-specific requirements of the UAVs, write test cases, and simulate missions considering the written tests. The focus of the work is not on testing the self-adaptive behavior of the UAV. Chilkunda et al.[19] propose a scenario builder that generates scenarios for testing autonomous cars. The proposed work is limited to the generation of scenarios for the functional behavior of autonomous cars. Schmidt et al.[20,21] propose a tool for scenario-based testing of the UAVs. The tool generates concrete test cases from logical scenarios using optimization algorithms. The proposed work is limited to testing the behavior related to obstacle detection and avoidance and does not allow for testing of mission-specific self-adaptive behaviors.

To conclude, the approaches for UAS mission and self-adaptive behavior specification[7,9,22,23,53] either only allow specification of UAS mission or do not allow specification of all aspects (such as types of changes and types of adaptation alternation alternatives) of the UAS self-adaptive behavior. The approaches on unmanned vehicles mission and self-adaptive behavior specification,[39,40,60] do not provide concepts that would allow a modeler to model the structural and the behavioral details specific to the UAS mission and its self-adaptive behavior. The approaches on testing of UAS swarm mission and self-adaptive behavior,[12–19] do not provide an automated way to generate and execute test cases for the self-adaptive behavior of UAS. Schmidt et al.,[20,21] does have a focus on scenario generation for the testing self-adaptive

behavior but the work is limited to safe distance violation from obstacles. Currently, the industry requires concepts to model the different types of changes (changes in the UAS system, in the UAS environment, mission operational needs), different types of adaptation alternatives (UAV action, payload action, and goal adaptation), and various UAS mission details, such as goals, their priorities, their assignments, and their types. Therefore, we propose a model-based approach that firstly allows to model the UAS mission and its self-adaptive behavior and then test the UAS application for incorrect execution of the self-adaptive behavior automatically. The UML profile defines concepts that are specific to the UAS domain. This further helps in testing UAS domain-specific behavior.

9 | CONCLUSION

Self-adaptation in a UAS is of utmost importance to ensure the safety of the environment and the UAS. Adaptation becomes difficult when flying in the real world with various environmental factors. The major drawback in the manual testing of UAS application software for self-adaptive behavior adaptation is that it is time-consuming, laborious, and allows testing only a few scenarios. It becomes very tough for the tester to test the self-adaptive behaviors, especially on extreme values, where there are chances of UAV collision. This paper addresses this challenge by proposing an automated model-based approach for testing the self-adaptive behavior in the UAS. The proposed approach includes a UML profile for capturing UAS application software including its mission and self-adaptive behavior requirements, and a model-based testing approach for verifying whether the UAS has made a correct adaptation in its behavior during mission execution.

As part of the modeling methodology, the *Software Engineer* models the UAS application software including the mission using a UML class diagram and self-adaptive behavior using a UML state machine. A UML profile is developed by the *Approach Provider* to facilitate the inclusion of UAS domain-specific concepts during structural and behavioral modeling. Further, a generalized UAS class diagram and state machine are developed to give a starting point for the UAS application software modeling. Next, the model-based testing approach is used for the automated generation of self-adaptive behavior test cases using a UAS application-specific state machine. The coverage criteria are proposed for generating UAS domain-specific test cases. The test case generation, execution, and evaluation are automated through a prototype testing tool. The proposed modeling and testing approach is demonstrated and evaluated on an industrial case study of UAS swarm formation flight application software and 20 UAS open source applications. The evaluation results for the proposed UML profile show that it is very capable of modeling the UAS application's mission and self-adaptive behavior. A total of 512 test cases are generated using proposed UAS coverage criteria (i.e., goal, behavioral change, and behavioral adaptation). All these generated test cases are executed through the Mission Planner simulator. The test case executions are logged as UAS application-specific class diagram instances. A total of 1752 class diagram instances are verified for UAS application-specific adaptation rules (written as OCL constraints) using a third-party OCL Solver tool. A total of 1409 instances are evaluated to be *true*, whereas 343 instances are evaluated to be *false* by the OCL Solver tool. The *true* result means that a self-adaptive behavior is correctly executed by the UAS application software during the mission, whereas the *false* result means a self-adaptive behavior is not executed. The verification results show that the proposed model-based testing approach is very proficient in identifying the incorrect execution of the self-adaptive behavior in the UAS application software.

AUTHOR CONTRIBUTIONS

Zainab Javed: Worked on the research idea provided by Muhammad Zohaib Iqbal and Muhammad Uzair Khan; Responsible for drafting the manuscript; including all sections; Conducted a comprehensive literature review; Developed the approach with the guidance of Muhammad Zohaib Iqbal and Muhammad Uzair Khan; Conducted the evaluation of the proposed approach; Analysed and interpreted the results; Created all figures and tables; Performed statistical analysis; Developed the prototype tool. **Muhammad Zohaib Iqbal:** Provided the initial research idea for the study; Provided guidance in the formation of the approach; Contributed to reviewing the manuscript for clarity and coherence. **Muhammad Uzair Khan**: Collaborated with Muhammad Zohaib Iqbal to provide the initial research idea for the study; Provided guidance in the formation of the approach; Contributed to reviewing the manuscript for clarity and coherence. **Muhammad Usman:** Provided guidance in structuring the manuscript; Contributed to reviewing and refining the

manuscript for clarity and coherence. **Atif Aftab Ahmed Jilani**: Contributed to reviewing the manuscript for clarity and coherence.

ACKNOWLEDGMENTS

This research work is supported through a research grant titled 'Establishment of National Centre of Robotics and Automation (NCRA)' by the Higher Education Commission (HEC), Pakistan. We would like to acknowledge anonymous reviewers for their valuable suggestions, which greatly contributed to the significant improvement of this paper.

CONFLICT OF INTEREST STATEMENT

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in UAS-SBTT at https://github.com/ZainabJaved6/UAS-SBTT.git.

ORCID

Zainab Javed https://orcid.org/0000-0001-9472-6256

REFERENCES

1. Suraj G, Gupta DG, Jawandhiya PM, et al. Review of unmanned aircraft system (UAS). *Int J Adv Res Comput Eng Technol*. 2013;2:1646-1658.
2. Oktay T. Simultaneous small UAV and autopilot system design. *Aircr Eng Aerosp Technol*. 2016;88:818-834.
3. Singh KK, Frazier AE. A metaanalysis and review of unmanned aircraft system (UAS) imagery for terrestrial applications. *Int J Remote Sens*. 2018;39(15-16):5078-5098.
4. Skorobogatov G, Barrado C, Salamí E. Multiple UAV systems: a survey. *Unmanned Syst*. 2020;8(2):149-169.
5. Shakhatreh H. Unmanned aerial vehicles (UAVs): a survey on civil applications and key research challenges. *IEEE Access*. 2019;7:48572-48634.
6. Otto A. Optimization approaches for civil applications of unmanned aerial vehicles (UAVs) or aerial drones: a survey. *Networks*. 2018;72(4):411-458.
7. Madni AM. Formal methods in resilient systems design: application to multi-UAV system-of-systems control. *Disciplinary Convergence in Systems Engineering Research*. Springer; 2018:407-418.
8. D'Angelo M, Napolitano A, Caporuscio M. CyPhEF: a model-driven engineering framework for self-adaptive cyber-physical systems. Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings. 2018 101-104.
9. Araújo-de-Oliveira P, Durán F, Pimentel E. A procedural and flexible approach for specification, modeling, definition, and analysis for self-adaptive systems. *Softw Pract Exp*. 2021;51(6):1387-1415.
10. Pekaric I. A systematic review on security and safety of self-adaptive systems. *J Syst Softw*. 2023;203:111716.
11. Yang J. A software-in-the-loop implementation of adaptive formation control for fixed-wing UAVs. *IEEE/CAA J Automat Sin*. 2019;6(5):1230-1239.
12. Ma T, Ali S, Yue T. Modeling foundations for executable model-based testing of self-healing cyber-physical systems. *Softw Syst Model*. 2019;18(5):2843-2873.
13. Wang S. CK-YAN: a quadrotor UAV swarm testbed for cooperative algorithms. 2020 Information Communication Technologies Conference (ICTC). IEEE. 2020 272-276.
14. Zhao W, Hooi CS. UWB system based UAV swarm testbed. 2020 16th International Conference on Control, Automation, Robotics and Vision (ICARCV). IEEE. 2020 67-72.
15. Hemmati H, Arefin SS, Loewen HW. Evaluating specification-level MC/DC criterion in model-based testing of safety critical systems. 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). IEEE. 2018 256-265.
16. Khatiri S, Panichella S, Tonella P. Simulation-Based Test Case Generation for Unmanned Aerial Vehicles in the Neighborhood of Real Flights. 2023 IEEE Conference on Software Testing, Verification and Validation (ICST). IEEE. 2023;281-292. doi:10.1109/ICST57152.2023.00034
17. Sarkar M. A framework for testing and evaluation of operational performance of multi-UAV systems. Intelligent Systems and Applications: Proceedings of the 2021 Intelligent Systems Conference (IntelliSys) Volume 1. Springer. 2022;355-374. doi:10.1007/978-3-030-82193-7_24
18. Agrawal A. A requirements-driven platform for validating field operations of small Uncrewed aerial vehicles. 2023 IEEE 31st International Requirements Engineering ConferAn Automated Model-Based Testing Approach for the Self-Adaptive Behavior of the UAS Application Software 43 Ence (RE). IEEE. 2023;29-40. doi:10.1109/RE57278.2023.00013
19. Chilkunda A. UAV-based scenario builder and physical testing platform for autonomous vehicles. 2023 6th Conference on Cloud and Internet of Things (CIoT). IEEE. 2023;77-84. doi:10.1109/CIoT57267.2023.10084885

20. Schmidt T, Hauer F, Pretschner A. Understanding safety for unmanned aerial vehicles in urban environments. 2021 IEEE Intelligent Vehicles Symposium (IV). IEEE. 2021;638-643. doi:10.1109/IV48863.2021.9575755
21. Schmidt T, Pretschner A. StellaUAV: a tool for testing the safe behavior of UAVs with scenario-based testing (tools and Artifact track). 2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE). IEEE. 2022;37-48. doi:10.1109/ISSRE55969.2022.00015
22. Dui H. Mission reliability modeling of UAV swarm and its structure optimization based on importance measure. *Reliab Eng Syst Safe*. 2021;215:107879.
23. Cybulski P, Zieliński Z. UAV swarms behavior modeling using tracking bigraphical reactive systems. *Sensors*. 2021;21(2):622.
24. Ordoukhanian E, Madni AM. Engineering resilience into multi-UAV systems. *Procedia Comput Sci*. 2019;153:9-16.
25. Ordoukhanian E, Madni AM. Resilient operation of autonomous unmanned aerial vehicles. AIAA Scitech 2019 Forum. 2019 222.
26. Wolf S. Secure and resilient swarms: autonomous decentralized lightweight UAVs to the rescue. *IEEE Consum Electron Mag*. 2020;9(4):34-40.
27. Wubben J. Providing resilience to UAV swarms following planned missions. 2020 29th International Conference on Computer Communications and Networks (ICCCN). IEEE. 2020 1-6.
28. Serna E, Serna A. Power and limitations of formal methods for software fabrication: thirty years later. *Informatica*. 2017;41:3.
29. Sartaj H, Iqbal MZ, Khan MU. Testing cockpit display systems of aircraft using a model-based approach. *Softw Syst Model*. 2021;20(6):1977-2002.
30. Usman M, Iqbal MZ, Khan MU. An automated modelbased approach for unit-level performance test generation of mobile applications. *J Softw: Evol Process*. 2020;32(1):e2215.
31. Iftikhar S. An automated model based testing approach for platform games. 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS). IEEE. 2015 426-435.
32. Ibrahim K, Whittaker JA. Model-based software testing. *Encyclopedia on Software Engineering*. Wiley; 2001.
33. Iqbal MZ. A model-based testing approach for cockpit display systems of avionics. 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS). IEEE. 2019 67-77.
34. Douglass BP. *Real Time Uml: Advances in the Uml for Real-Time Systems, 3/E*. Pearson Education India; 2004.
35. Ali S. A search-based OCL constraint solver for model-based test data generation. 2011 11th International Conference on Quality Software. IEEE. 2011 41-50.
36. Imtiaz J, Iqbal MZ. An automated model-based approach to repair test suites of evolving web applications. *J Syst Softw*. 2021;171:110841.
37. Usman M, Iqbal MZ, Khan MU. A product-line model-driven engineering approach for generating feature-based mobile applications. *J Syst Softw*. 2017;123:1-32.
38. Maia PH. Dragonfly: a tool for simulating self-adaptive drone behaviours. 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). IEEE. 2019 107-113.
39. Dhouib S. Robotml, a domain-specific language to design, simulate and deploy robotic applications. International Conference on Simulation, Modeling, and Programming for Autonomous Robots. Springer. 2012 149-160.
40. Di Ruscio D, Malavolta I, Pelliccione P. A family of domain-specific languages for specifying civilian missions of multi-robot systems. MORSE@ STAF. 2014 16-29.
41. Universal Ground Control Software-UgCS. https://www.ugcs.com/.
42. ArduPilot. Mission Planner. https://ardupilot.org/planner/
43. QGroundControl. Mission Planning Tool. http://qgroundcontrol.com/
44. Vergouw B. Drone technology: types, payloads, applications, frequency spectrum issues and future developments. *The Future of Drone Use*. Springer; 2016:21-45.
45. Ali S. Improving the performance of OCL constraint solving with novel heuristics for logical operations: a search-based approach. *Empiric Softw Eng*. 2016;21(6):2459-2502.
46. Utting M, Legeard B. *Practical Model-based Testing: a Tools Approach*. Elsevier; 2010.
47. Shirole M, Kumar R. UML behavioral model based test case generation: a survey. *ACM SIGSOFT Softw Eng Notes*. 2013;38(4):1-13.
48. Rechtberger V, Bures M, Ahmed BS. Overview of test coverage criteria for test case generation from finite state machines modelled as directed graphs. 2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE. 2022 207-214.
49. O'Regan G. Test case analysis and design. *Concise Guide to Software Testing*. Springer; 2019:117-132.
50. ArduPilot. Cygwin. https://ardupilot.org/mavproxy/
51. Mavlink. Comm Protocol. https://mavlink.io/en/
52. Wohlin C. *Experimentation in Software Engineering*. Springer Science & Business Media; 2012.
53. Findeis DE. Modeling and simulation of UAV swarm formation control in response to wind gusts. AIAA Scitech 2019 Forum. 2019 1571.
54. Revill MB. *UAV Swarm Behavior Modeling for Early Exposure of Failure Modes*. Tech. rep. Naval Postgraduate School Monterey United States; 2016.
55. Alves L. DRESS-ML: a domain-specific language for modelling exceptional scenarios and self-adaptive behaviours for drone-based applications. Proceedings of the 2022 ACM/IEEE 44th International Conference on Software Engineering: Software Engineering in Society. 2022 56-66.
56. Roy M. SCARS: suturing wounds due to conflicts between non-functional requirements in autonomous and robotic systems. *Softw Pract Exp*. 2023;54:759-795.
57. Carter BT. Cyber-physical systems modeling for security using SysML. *Systems Engineering in Context*. Springer; 2019:665-675.

58. Holt J, Perry S. *SysML for Systems Engineering*. Vol 7. IET; 2008.
59. Islam N, Azim A. Assuring the runtime behavior of self-adaptive cyber-physical systems using feature modeling. Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering. 2018 48-59.
60. Han D, Yang Q, Xing J. Extending uml for the modeling of fuzzy selfadaptive software systems. The 26th Chinese Control and Decision Conference (2014 CCDC). IEEE. 2014 2400-2406.
61. Petrovska A. Defining adaptivity and logical architecture for engineering (smart) selfadaptive cyber-physical systems. *Inf Softw Technol*. 2022;147:106866.
62. Rosa MR. Adaptive hierarchical formation control for uncertain Euler–Lagrange systems using distributed inverse dynamics. *Eur J Control*. 2019;48:52-65.
63. Paweł Ćwiąkała. Testing procedure of unmanned aerial vehicles (UAVs) trajectory in automatic missions. *Appl Sci*. 2019;9(17):3488.

How to cite this article: Javed Z, Iqbal MZ, Khan MU, Usman M, Jilani AAA. An automated model-based testing approach for the self-adaptive behavior of the unmanned aircraft system application software. *Softw: Pract Exper*. 2024;1-53. doi: 10.1002/spe.3358

APPENDIX A

TABLE A1 Self-adaptive behavior verification.

Test case	Instance categories	Expected output	Actual output	Result
Test Case 16	Sharp Turn Safe Distance Violation due to Sharp Turn	targetState=Waiting, currentGoal=AvoidCollision	targetState=FlyToWaypoint, currentGoal=ExecuteMission	False
	At Safe Distance	targetState=FlyToWaypoint, currentGoal=ExecuteMission	targetState=FlyToWaypoint, currentGoal= ExecuteMission	True
	Battery Discharged	targetState=RTL	targetState=RTL	True
Test Case 59	Wind Speed Increased	targetState=WindSpeedExceeded	targetState= WindSpeedExceeded	True
	Safe Distance Violation due to Wind	targetState=Waiting, currentGoal=AvoidCollision	targetState=FlyToWaypoint, currentGoal=ExecuteMission	False
	AtSafeDistance	targetState=FlyToWaypoint	targetState=FlyToWaypoint	True
	FlyingToError	targetState=Waiting	targetState=FlyToWaypoint	False
	ErrorToRTL	targetState=RTL	targetState=RTL	True
Test Case 72	Wind Speed Increased	targetState=WindSpeedExceeded	targetState= WindSpeedExceeded	True
	No Distance Violation due to Wind	targetState=FlyToWaypoint	targetState=FlyToWaypoint	True
	Sharp Turn Safe Distance Violation due to Sharp Turn	targetState=Waiting, currentGoal=AvoidCollision	targetState=FlyToWaypoint, currentGoal=ExecuteMission	False
	At Safe Distance	targetState=FlyToWaypoint, currentGoal=ExecuteMission	targetState=FlyToWaypoint, currentGoal= ExecuteMission	True

TABLE A2 Self-adaptive behavior verification results.

Application	No. of test cases	Total no. of self-adaptive behavior instances generated	Total no. of self-adaptive behavior instances passed	Total no. of self-adaptive behavior instances failed
Ardu-Copter				
Formation-Flight-Case-Study	348	1336	1050	286
Shark-Spotting	11	31	28	3
Fire-Tracking	7	18	16	2
Neighbor-Malfunction	7	13	13	0
Neighbor-Malfunction2	11	27	27	0
Package-Delivery	17	60	60	0
Traffic-Monitoring	7	16	16	0
Ardu-Plane				
Fire-Tracking	8	18	18	0
Neighbor-Malfunction	7	13	11	2
Neighbor-Malfunction2	11	27	27	0
Safe-Distance-Violation	4	8	8	0
Safe-Distance-Violation-Wind	4	8	5	3
WaypointMission1	4	5	5	0
WaypointMission2	5	7	5	2
Traffic-Monitoring	5	11	11	0
Vibration-Level-Increased	4	4	2	2
Quad-Plane				
Neighbor-Malfunction	4	8	8	0
Safe-Distance-Violation	3	4	2	2
Traffic-Monitoring	3	7	7	0
WaypointMission1	6	13	11	2
Vibration-Level-Increased	36	118	79	39

TABLE A3 UAS applications evaluation results.

Autopilot	UAS application	Self-adaptive behaviors tested	Number of instances	Instances passed	Instances failed
ArduCopter	Shark-Spotting	ReachedWaypointWithWind	8	8	0
		SharkSpottedWithWind	4	4	0
		LocationSentWithWind	4	4	0
		NoSharkSpottedWithWind	6	3	3
		GPSDisconnection	3	3	0
		GCSDisconnection	3	3	0
		BatteryDischarged	3	3	0
	Fire-Tracking	FireFoundWithWind	4	4	0
		BroadcastedFireLocWithWind	4	4	0
		FireExtinguishedWithWind	4	4	0
		GPSDisconnectionWithWind	2	0	2
		GCSDisconnection	2	2	0
		BatteryDischarged	2	2	0
	Neighbor-Malfunction	ReachedWaypointWithWind	4	4	0
		NeighborMalfunction	3	3	0
		GPSDisconnection	2	2	0
		GCSDisconnection	2	2	0
		BatteryDischarged	2	2	0
	Neighbor-Malfunction2	NeighborMalfunction2	4	4	0
		ReachedWaypointWithWind	8	8	0
		NeighborMalfunction	6	6	0
		GPSDisconnection	3	3	0
		GCSDisconnection	3	3	0
		BatteryDischarged	3	3	0
	Package-Delivery	ReachedPackageLocWithWind	11	11	0
		ReachedDeliveryLocWithWind	5	5	0
		SafeDistanceViolation	10	10	0
		PackageGrippedWithWind	9	9	0
		PackageDeliveredWithWind	5	5	0
		AtSafeDistance	10	10	0
		GPSDisconnection	5	5	0
		BatteryDischarged	5	5	0
	Traffic-Monitoring	GroundPerArrivedWithWind	3	3	0
		SignalViolatedWithWind	3	3	0
		GPSDisconnection	2	2	0
		GCSDisconnection	2	2	0
		BatteryDischarged	2	2	0
		ReachedMonitoringWaypointWind	4	4	0

(Continues)

TABLE A3 (Continued)

Autopilot	UAS application	Self-adaptive behaviors tested	Number of instances	Instances passed	Instances failed
ArduPlane	Fire-Tracking	FireFoundWithWind	4	4	0
		BroadcastedFireLocWithWind	4	4	0
		FireExtinguishedWithWind	4	4	0
		GPSDisconnection	2	2	0
		GCSDisconnection	2	2	0
		BatteryDischarged	2	2	0
	Neighbor-Malfunction	ReachedWaypointWithWind	4	2	2
		NeighborMalfunction	3	3	0
		GPSDisconnection	2	2	0
		GCSDisconnection	2	2	0
		BatteryDischarged	2	2	0
	Neighbor-Malfunction2	NeighborMalfunction2	4	4	0
		ReachedWaypointWithWind	8	8	0
		NeighborMalfunction	6	6	0
		GPSDisconnection	3	3	0
		GCSDisconnection	3	3	0
		BatteryDischarged	3	3	0
	Safe-Distance-Violation	SafeDistanceViolation	2	2	0
		AtSafeDistance	2	2	0
		GPSDisconnection	2	2	0
		BatteryDischarged	2	2	0
	Safe-Distance-Violation-Wind	SafeDistanceViolationWind	2	0	2
		AtSafeDistance	2	1	1
		GPSDisconnection	2	2	0
		BatteryDischarged	2	2	0
	WaypointMission1	WindSpeedIncreased	1	1	0
		GPSDisconnection	1	1	0
		BatteryDischarged	1	1	0
		ReachedWaypointWithWind	2	2	0
	WaypointMission2	WindSpeedIncreased	1	1	0
		WindSpeedIncreased2	1	0	1
		GPSDisconnection	1	1	0
		BatteryDischarged	1	1	0
		ReachedWaypointWithWind	3	2	1
	Traffic-Monitoring	GroundPerArrivedWithWind	2	2	0
		SignalViolatedWithWind	2	2	0
		GPSDisconnection	2	2	0
		BatteryDischarged	2	2	0
		ReachedMonitoringWaypointWind	3	3	0
	Vibration-Level-Increased	VibrationFailSafe2	1	0	1
		VibrationFailSafe	1	0	1
		GPSDisconnection	1	1	0
		BatteryDischarged	1	1	0

(Continues)

TABLE A3 (Continued)

Autopilot	UAS application	Self-adaptive behaviors tested	Number of instances	Instances passed	Instances failed
QuadPlane	Neighbor-Malfunction	NeighborMalfunction2	2	2	0
		ReachedWaypointWithWind	4	4	0
		NeighborMalfunction	2	2	0
	Safe-Distance-Violation	SafeDistanceViolationWithWind	1	0	1
		AtSafeDistance	2	1	1
		SafeDistanceViolation	1	1	0
	Traffic-Monitoring	GroundPerArrivedWithWind	2	2	0
		SignalViolatedWithWind	1	1	0
		VehicleOverspeedingWithWind	1	1	0
		ReachedMonitoringWaypointWind	3	3	0
	WaypointMission1	WindSpeedIncreased	2	2	0
		WindSpeedIncreased2	2	0	2
		GPSDisconnection	3	3	0
		ReachedWaypointWithWind	6	6	0
	Vibration-Level-Increased	VibrationFailSafe2	17	0	17
		VibrationFailSafe	32	10	22
		VibrationLevelNormal2	18	18	0
		VibrationLevelNormal	24	24	0
		GPSDisconnection	27	27	0

TABLE A4 Formation flight case study self-adaptive behavior verification results.

Self-adaptive-behavior-category	Test-cases-containing-the-category	OCL-constraint	Upper-limit-of-related-variables	Expected-output	Actual-output	Result
Wind Speed Increased	TC1-TC12, TC22-TC39, TC49-TC94	Constraint # 2 (listing 4)	windResistance=30 m/s	targetState= *Wind-SpeedExceeded*	targetState= *Wind-SpeedExceeded*	True
Safe Distance Violation due to Wind	TC22-TC30, TC49- TC67	Constraint # 2 (listing 5)	windResistance=30 m/s, windDirection= UAV heading, safeDistanceFromObstacle = 6m	self.targetState = *Waiting* self.currentGoal = *AvoidCollision*	targetState= *FlyToWaypoint*, currentGoal= *ExecuteMission*	False
No Distance Violation due to Wind	TC4-TC12, TC31-TC39, TC68-TC94	Constraint # 3 (listing 4)	windResistance=30 m/s, safeDistanceFromObstacle=6m, timePassedInCurrentState>10 sec	targetState= *FlyToWaypoint*	targetState= *FlyToWaypoint*	True
Safe Distance Violation due to Sharp Turn	TC1-TC19, TC68-TC76	Constraint # 1 (listing 5)	turnRateLimit=30 deg/sec, safeDistanceFromObstacle=6 m	targetState= *Waiting*, currentGoal= *AvoidCollision*	targetState= *FlyToWaypoint*, currentGoal= *ExecuteMission*	False
No Distance Violation due to Sharp Turn	TC22-TC46, TC50-TC58, TC77-TC85	Constraint # 1 (listing 4)	turnRateLimit=30 deg/sec, timePassedInCurrentState>10 sec	targetState= *FlyToWaypoint*	targetState= *FlyToWaypoint*	True
AtSafeDistance	TC1-TC19, TC22-TC30, TC49-TC76	Constraint # 6 (listing 4)	safeDistanceFrom Obstacle = 6 m	targetState= *FlyToWaypoint*, currentGoal= *ExecuteMission*	targetState= *FlyToWaypoint*, currentGoal= *ExecuteMission*	True
Flying To Error	TC1, TC2, TC4, TC5, TC20, TC21, TC22, TC23, TC24 TC31, TC32, TC47, TC48, TC50, TC51, TC59, TC60, TC68, TC69, TC77, TC78, TC86, TC87	Constraint # 3 (listing 5)	currentHeading != expectedUAVHeading currentDistanceFromWaypoint increases rather than decreasing	targetState= *Waiting*	targetState= *FlyToWaypoint*	False
Error To land	TC2, TC5, TC21, TC23, TC32, TC48, TC51, TC60, TC69, TC78, TC87	Constraint # 4 (listing 4)	isWaypointLost= true when currentState = *ErrorState*	targetState= *Land*	targetState= *Land*	True
Error To RTL	TC1, TC4, TC20, TC22, TC31, TC41, TC50, TC59, TC68, TC77, TC86	Constraint # 5 (listing 4)	isWaypointLost= true when currentState = ErrorState	targetState= *RTL*	targetState= *RTL*	True
GPS Disconnection	TC8, TC15, TC26, TC35, TC42, TC54, TC63, TC72, TC81, TC90, TC97	Constraint # 9 (listing 4)	GPSConnected= false for time 5 sec	targetState= *RTL*	targetState= *RTL*	True
GCS Disconnection	TC10, TC17, TC28, TC37, TC44, TC56, TC65, TC74, TC83, TC92, TC99	Constraint # 8 (listing 4)	GCSConnected= false for time 5 sec	targetState= *RTL*	targetState= *RTL*	True
Battery Discharged	TC9, TC16, TC27, TC36, TC43, TC55, TC64, TC73, TC82, TC91, TC98	Constraint # 10 (listing 4)	currentLevel > 20 %	targetState= *RTL*	targetState= *RTL*	True
Vibration Failsafe	TC6, TC13, TC24, TC33, TC40, TC52, TC61, TC70, TC79, TC88, TC95	Constraint # 7 (listing 4)	vibrationLevel > 30 m/s/s	targetState= *RTL*	targetState= *RTL*	True

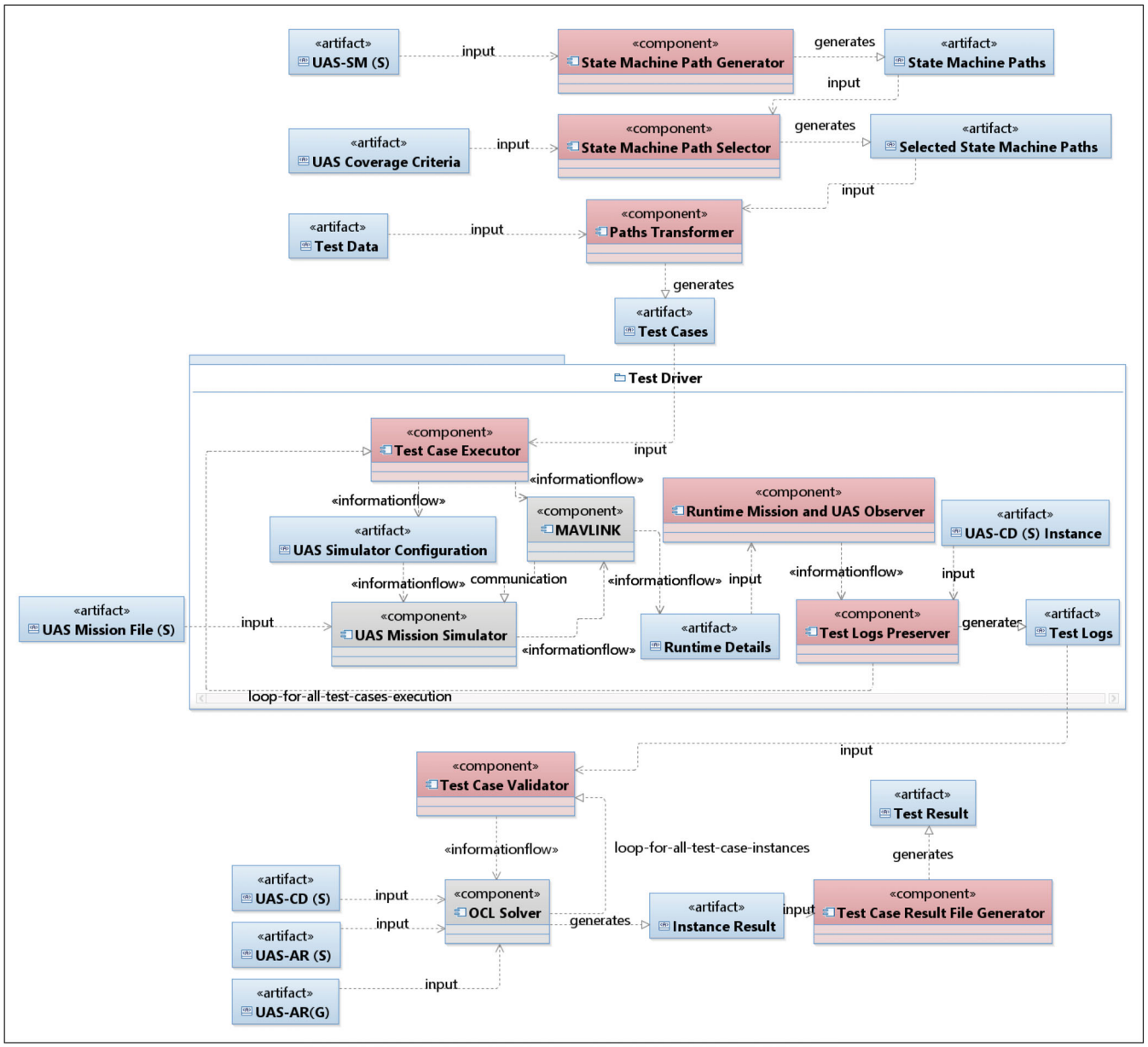


FIGURE A1 UAS-SBTT tool architecture.