The Journal of Systems & Software 197 (2023) 111543

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Configuring mission-specific behavior in a product line of collaborating Small Unmanned Aerial Systems ☆

Md Nafee Al Islam [a], Muhammed Tawfiq Chowdhury [a], Ankit Agrawal [a], Michael Murphy [a], Raj Mehta [a], Daria Kudriavtseva [a], Jane Cleland-Huang [a,*], Michael Vierhauser [b], Marsha Chechik [c]

[a] Department of Computer Science and Engineering, University of Notre Dame, IN, USA
[b] LIT Secure and Correct Systems Lab, Johannes Kepler University Linz, Austria
[c] Department of Computer Science, University of Toronto, Canada

ARTICLE INFO

Article history:
Received 21 February 2022
Received in revised form 8 October 2022
Accepted 12 October 2022
Available online 24 November 2022

Keywords:
Dynamic configuration
Small unmanned aerial system
sUAS
Emergency response
Product line

ABSTRACT

In emergency response scenarios, autonomous small Unmanned Aerial Systems (sUAS) must be configured and deployed quickly and safely to perform mission-specific tasks. In this paper, we present *Drone Response*, a Software Product Line for rapidly configuring and deploying a multi-role, multi-sUAS mission whilst guaranteeing a set of safety properties related to the sequencing of tasks within the mission. Individual sUAS behavior is governed by an onboard state machine, combined with coordination handlers which are configured dynamically within seconds of launch and ultimately determine the sUAS' behaviors, transition decisions, and interactions with other sUAS, as well as human operators. The just-in-time manner in which missions are configured precludes robust upfront testing of all conceivable combinations of features — both within individual sUAS and across cohorts of collaborating ones. To ensure the absence of common types of configuration failures and to promote safe deployments, we check vital properties of the dynamically generated sUAS specifications and coordination handlers before sUAS are assigned their missions. We evaluate our approach in two ways. First, we perform validation tests to show that the end-to-end configuration process results in correctly executed missions, and second, we apply fault-based mutation testing to show that our safety checks successfully detect incorrect task sequences.

1. Introduction

Small Unmanned Aerial Systems (sUAS) are increasingly used to support diverse missions such as search-and-rescue operations (Schörner et al., 2021; Doherty and Rudol, 2007; Alotaibi et al., 2019), earthquake relief (Erdelj et al., 2017), bridge inspections, fire and accident surveillance (Sherstjuk et al., 2018), and medical supply delivery (Lv et al., 2021). Many of these missions, especially those related to emergency response, can benefit from, or heavily rely on the simultaneous deployment of multiple sUAS — each charged with an individual task. For example, in a coast guard rescue operation, several sUAS could perform the search, whilst one or more could be on standby to deliver a flotation device, and yet another could provide birds-eye surveillance of the scene.

While such missions typically share a set of common tasks, they also exhibit unique characteristics. Certain tasks, such as launching an sUAS, obstacle and collision avoidance, or planning search routes, are common across many, or even all of the mission deployments; while other tasks, such as collecting water samples, or tracking a moving victim in a river, are unique to a specific type of mission. Efficiently and effectively managing such variabilities in a Cyber–Physical System (CPS) (Krüger et al., 2017), such as our system of collaborating sUAS, is a non-trivial challenge that must take into consideration the context and sequencing of events, while simultaneously addressing safety concerns (Vierhauser et al., 2019; Vattapparamban et al., 2016). Furthermore, given the time-criticality of emergency missions, the overall mission plan, as well as each sUAS' behavior must be configured quickly prior to launch, and mission specifications then pushed to relevant microservices, mobile units, and to individual sUAS.

To address these challenges, we have developed *Drone Response* (Cleland-Huang et al., 2020) a multi-sUAS mission management and control system, which takes a Software Product Line (SPL) approach to support and configure diverse emergency missions. SPLs represent software-intensive systems in which individual products are typically generated from domain-level assets by selecting a set of alternative and optional features

☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail address: janeclelandhuang@nd.edu (J. Cleland-Huang).

https://doi.org/10.1016/j.jss.2022.111543
0164-1212/

(variabilities) and composing them into an individual product on top of a set of common base features (commonalities) (Batory et al., 2006; Pohl et al., 2005; SEI, Software Engineering Institute, 2020).

In the case of *Drone Response*, mission tasks are supported by specific features and carefully choreographed into distinct sequences. It is the sequencing of tasks that consequently determines how each sUAS is configured, and which of the SPL's variable features are included in the mission-specific configuration. This highly dynamic configuration process means that some missions are entirely composed of previously validated combinations of task sequences, whilst others include new, and therefore previously unvalidated combinations. While this approach provides the high degrees of flexibility that are needed in an emergency response platform, it also introduces the possibility of invalid, and potentially unsafe sequences, jeopardizing the success of the mission. For example, dropping a flotation device in a random location before a victim is found, or generating an infinite loop of repeating tasks that prevents the sUAS from returning home when its battery becomes low, can lead to a failed mission.

We address these risks in a number of ways. First, we minimize interactions across modules representing individual features, and then perform rigorous testing of individual features and anticipated feature combinations using unit tests, integration tests, and system-level tests. Additionally, we introduce a global set of safety properties with respect to valid and invalid orderings of tasks and check that all generated mission specifications satisfy these properties. The safety checks are designed to identify problems such as *non-deterministic transitions* that could lead to unexpected behaviors, *non-reachability* of desirable end states which could create *deadlock or livelock* scenarios that prevent an sUAS from successfully completing its mission, and *undesirable sequencing* of tasks that result in failure to achieve mission goals.

Our SPL approach differs from prior work in which sUAS are either programmed to perform a fixed task (Mesar et al., 2019; Claesson et al., 2017; Erdelj et al., 2017; Ezequiel et al., 2014; Doherty and Rudol, 2007), configured in advance of the mission using a domain-specific language (Dragule et al., 2021), or manually deployed — with individual remote pilots as part of a joint mission (Besada et al., 2019). Unlike these approaches, *Drone Response* leverages a set of relatively general-purpose sUAS which can be rapidly configured for diverse roles within seconds of launch, or can be reconfigured dynamically during flight, and can then operate autonomously without manual control.

The capabilities of our SPL are driven by an initial set of seven community-inspired use cases (Drone Response, 2022) for which variability points include software features, hardware capabilities, differing degrees of sUAS autonomy versus human control, and alternate sequencing of tasks for different missions. In our implementation, an operator interactively defines mission tasks through a user interface, selects one or more sUAS for the mission, and then defines tasks to be performed, resulting in a global mission plan. *Drone Response* then deconstructs the mission plan and dynamically assigns specific roles to each sUAS, generates an associated specification for each sUAS, checks the specification to ensure it meets predefined task sequencing properties, and if the specification is valid, sends it to the sUAS. Finally, the sUAS dynamically configures itself according to its validated specification.

This paper significantly extends our earlier work published in the 2020 Systems and Software Product Line Conference (Cleland-Huang et al., 2020). While the earlier paper focused on the Requirements Engineering aspects of our *Drone Response* product line, this paper describes the actual configuration process of a mission and of the participating sUAS. The paper makes two primary contributions. First, it describes the configuration process itself in which a task-based, mission-level specification is used to guide the automated configuration of individual sUAS and their coordination mechanisms. Second, it presents a novel approach for dynamically checking mission specifications against the relevant subset of predefined task sequencing properties prior to deployment.

The remainder of this paper is structured as follows. Section 2 provides background information about the use cases that guided the development of the *Drone Response* product line and introduces the main features of the SPL and the corresponding architecture. Section 3 describes the initial mission planning process, individual sUAS configuration, and mechanisms for supporting sUAS-sUAS and human-sUAS coordination. Section 4 describes our integrated process for validating the task sequencing of each dynamically generated specification prior to its deployment. In Section 5 we describe the experimental evaluation — including unit tests, simulations, and mutation testing. Finally, in Sections 6 to 8, we present threats to validity, related work, and conclusions.

2. The Drone Response SPL

The *Drone Response* SPL includes domain assets such as use cases, a feature model, domain-level activity diagrams, system-level architecture, configurable code, and diverse tests. In this section, we provide a brief overview of the initial use cases that served as a starting point for the development of *Drone Response*, as well as the resulting SPL. Additional details are provided in our prior work (Cleland-Huang et al., 2020).

2.1. Drone Response use cases

In order to guide the development of our SPL, we collected requirements for diverse sUAS mission scenarios from three different sources. First, we analyzed the requirements, features, and architecture of our existing Dronology platform (Cleland-Huang et al., 2018; Vierhauser et al., 2018) which is a single product application that was created and developed through close collaboration with the South Bend Fire Department (Agrawal et al., 2020). Next, we searched for grey literature, including articles, reports, and white papers on sUAS applications in order to find state-of-the-practice descriptions of emergency responders, firefighters, coast guards, and other groups using sUAS in emergency response scenarios. Here our goal was not to retrieve a complete or exhaustive set of use cases, but to select a set of well-described and diverse scenarios for driving the development of our SPL. The references we ultimately used are listed in Table 1. Each of the described scenarios either involves sUAS being controlled manually by a human operator, or the use of an existing off-the-shelf system such as MissionPlanner, QGroundControl, or DroneSense[1] which allows users to plan and execute flight routes. As a final step, we performed an informal literature search of academic publications in order to identify state-of-the-art solutions for sUAS autonomy and their applications to emergency response. We initially used the search terms "UAV & Emergency Response", followed by a snowballing process to retrieve papers referenced by the initial search. Relevant papers are again listed in Table 1.

Textual use cases capture requirements from the perspective of their external actors by describing a primary sequence of actions, as well as alternatives and exception cases (Cockburn, 2000). They are, therefore, able to capture both mandatory and optional features (Fantechi et al., 2004). As an example, a use case for "ice search-and-rescue" is shown in Fig. 1. The use case starts

[1] URLs: ardupilot.org/planner, qgroundcontrol.com, www.dronesense.com.

Table 1
Scenarios were derived from videos recorded by emergency responders, brainstorming meetings, websites, and academic literature.

ID	Use cases	Usage	Papers	Contributing stakeholders
UC1	River search & Rescue	Cleland-Huang and Vierhauser (2018) and Agrawal et al. (2020)	Silvagni et al. (2017)	South Bend Firefighters
UC2	Ice rescue	Rios (2019) and KETV NewsWatch 7 (2018)		News reports
UC3	Defibrillator delivery	Mesar et al. (2019) and Claesson et al. (2017)	Fleck (2016)	DeLive, Cardiac science
UC4	Traffic accidents	Molino et al. (2016) and Pádua et al. (2020)	Kim et al. (2018)	South Bend Firefighters
UC5	Structural fires	Griffith and Wakeham (2015)		South Bend Firefighters
UC6	Water sampling	Lally et al. (2019) and Ore et al. (2015)	Koparan et al. (2018)	Environmental Scientists
UC7	Air sampling	Ruiz-Jimenez et al. (2019) and Chang et al. (2015)	Zhi et al. (2017) and Alvear et al. (2015)	Environmental Scientists

Use Case: Ice Search and Rescue
ID: SPLC-11

Description
UAV(s) dispatched with a flotation device for ice rescue

Primary Actor
Drone Commander

Trigger
The Drone Commander activates the delivery.

Main Success Scenario
1. Emergency responders **plan_area_search** [SPLC-1001]
2. The DroneResponse commander issues a command to start the mission.
3. The UAV(s) **takeoff** [SPLC-1007]
4. The UAVs **perform_search** [SPLC-1002]
5. The UAV **requests_victim_confirmation** [SPLC-1005] from the human operator.
6. The UAV receives confirmation from the human operator that the victim sighting is valid.
7. DroneResponse automatically sends the GPS coordinates to the mobile_rescue system.
8. The UAV switches to **flotation_device_delivery** [SPLC-1006] mode.
9. Human responders reach the victim's location and execute a rescue.
10. The Drone Commander **ends_mission** [SPLC-1007].

Specific Exceptions
1. In step 3, one of the UAVs fails to take-off.
   1.1 If a replacement UAV is flight-ready, it is dispatched in place of the failed UAV.
   1.2 If no replacement is available DroneResponse re-executes **generate-search-plan** [SPLC-1009] for the available UAVs and previously defined search area.

General Exceptions
1. At any time, if communication is lost between the Ground Control Station and a UAV, DroneResponse executes the **Lost Drone-to-GCS Communication** (SPLC-2001) exception case.
2. At any time, a malfunction error is raised by a UAV in flight, DroneResponse executes the **Drone-in-flight Malfunction** (SPLC-2002) exception case.

Fig. 1. A partial view of the "Ice search-and-rescue" use case focusing on the main success scenario and examples of general and specific exceptions. Requirements were derived from news reports and video footage (e.g., Rios (2019) and KETV NewsWatch 7 (2018)).

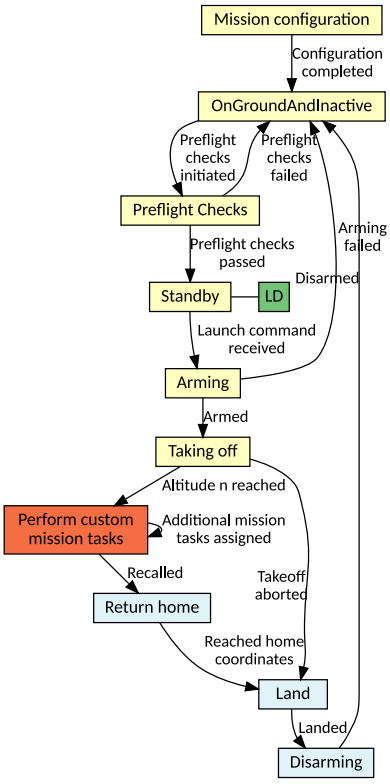
by describing the actors and stakeholders and establishing pre-conditions and post-conditions. It then describes the *main success scenario*, as well as *alternate sequences of events* and *exceptions* that can occur at any time of the mission. Steps that describe common tasks, shared across multiple use cases, are defined as references to supporting use cases, while steps that are specific to the ice-rescue use case are described directly in the text. Due to space limitations, the use case shows only the main scenario steps, with a few examples of specific and general exceptions. A more complete set of use cases is available online[2]

2.2. Creating SPL assets: Activity diagrams and feature models

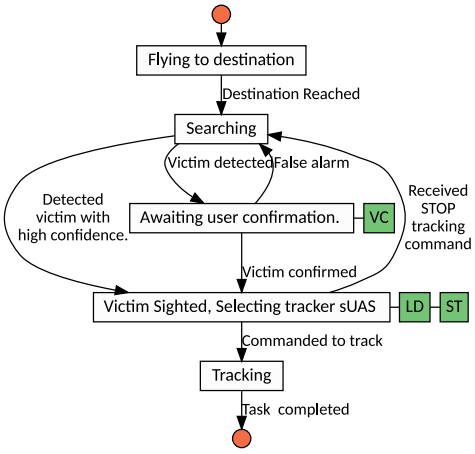
As described in our earlier work (Cleland-Huang et al., 2020), we followed a systematic requirements process to create both an SPL feature model and a domain-level activity diagram describing the sequences of tasks needed to support each of our targeted use cases (Baresi, 2018). This process involved inspecting each of the use case steps to identify specific tasks and then modeling sequences of tasks as activity diagrams with clearly defined transitions between activities.

Based on some initial experimentation, we identified tasks that represented variability points (e.g., track victim, and deliver flotation device) and had a major impact on the sequencing of the mission, whilst hiding internal configuration points such as computer vision models or route planning algorithms that supported activities within specific contexts. Fig. 2 provides examples of activity diagrams for search-detect-track (cf. Fig. 2(b)), item delivery (cf. Fig. 2(d)), and surveillance (cf. Fig. 2(c)), as well as activities common across all missions (cf. Fig. 2(a)). All sUAS share a series of preflight, takeoff, and end-of-mission activities, whilst also performing custom mission tasks, such as "Search, Detect, and

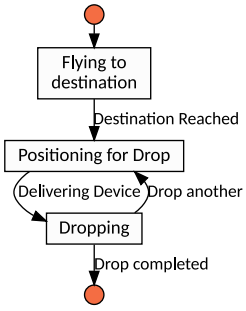
[2] https://github.com/SAREC-Lab/sUAS-UseCases.



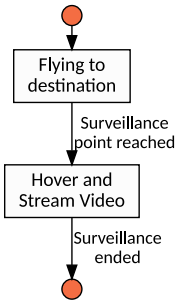
(a) Mission Template: Yellow= Flight initialization and launch nodes; Blue= end-of-flight nodes; Red = Diverse configurable mission tasks.



(b) Search, detect, and track. Coordination Handler 1 links to UI (not shown), while Handler 2 links to Standby node (see Fig 2a).



(c) Item Delivery.



(d) Surveillance.

Fig. 2. A Mission Template (a) with three examples of task sequences (b–d) that plug into the Mission template. Three coordination handlers are shown in Green: 'LD' coordinates launch of the delivery drone, 'ST' selects an sUAS for tracking, and 'VC' elicits victim confirmation from the operator. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Track", "Item Delivery", and "Surveillance". The activity diagrams were modeled iteratively in order to capture sequences that were common across all missions, as well as variable sequences specific to only a subset of the missions. These models are examples of *predefined* sequences. However, new sequences can also be created by rearranging tasks in new ways — for example by inserting "Hover and Stream Video" (cf. Fig. 2(c)) before "Positioning to Drop" (cf. Fig. 2(d)).

Next, we identified the features needed to support each task within the entire set of use cases, classified them as mandatory, optional, or alternative features, and organized them as nodes in a hierarchical feature model in which associations were specified as parent–child relationships between nodes, and cross-tree constraints were structured as tuples (NodeName, NodeName, [requires|excludes]). A partial view of our feature model is shown in Fig. 3, and the complete process for constructing domain-level activity diagrams and feature models are described in more detail in our earlier work (Cleland-Huang et al., 2020).

2.3. Drone Response architecture

Drone Response supports the dynamic configuration of sUAS for diverse emergency response missions. While the main components of the architecture are common among different configurations, the onboard capabilities of each sUAS are subject to variability and thus, are configured dynamically according to the mission purpose and the tasks assigned to each sUAS.

Fig. 4 depicts a high-level overview of the *Drone Response* architecture, which includes support for different *User Interfaces*, a *Ground Control Station* (GCS), and an *Onboard Autonomous Pilot* deployed on each sUAS. We now describe the aspects of the architecture that are particularly relevant to the SPL configuration.

User Interaction: The user-centric *interfaces* allow stakeholders to plan and configure missions, track sUAS, and monitor sUAS status. This flexibility is achieved via a *web application* and a dedicated *database* for storing reusable mission plans and configurations (cf. Section 3), which is hosted either on the cloud or on the local Ground Control Station (GCS). An HTTP server provides a bridge for the UI components to interact with Configuration and Coordination Services and indirectly with the sUAS. The design allows multiple users to work collaboratively in planning and deploying missions, including tasks such as defining search areas and flight routes, configuring sUAS, and tracking missions.

Ground Control Station: The Ground Control station in *Drone Response* adheres to the microservices and publish–subscribe architectural style (Muccini and Moghaddam, 2018) with the aim of achieving scalable and lightweight mission-specific services. Each microservice provides a specific capability such as airspace leasing, multi-sUAS coordination, or safety-checking the mission specification (cf. Section 4). Whilst many microservices are continuously active (e.g., airspace leasing), other services perform functions that are relevant only when specific SPL features are activated. For example, when multiple sUAS are engaged in a joint search, a dedicated microservice is responsible for coordinating victim sightings and subsequent tracking decisions across all of the sUAS. Microservices communicate with sUAS and UIs via the MQTT message broker.

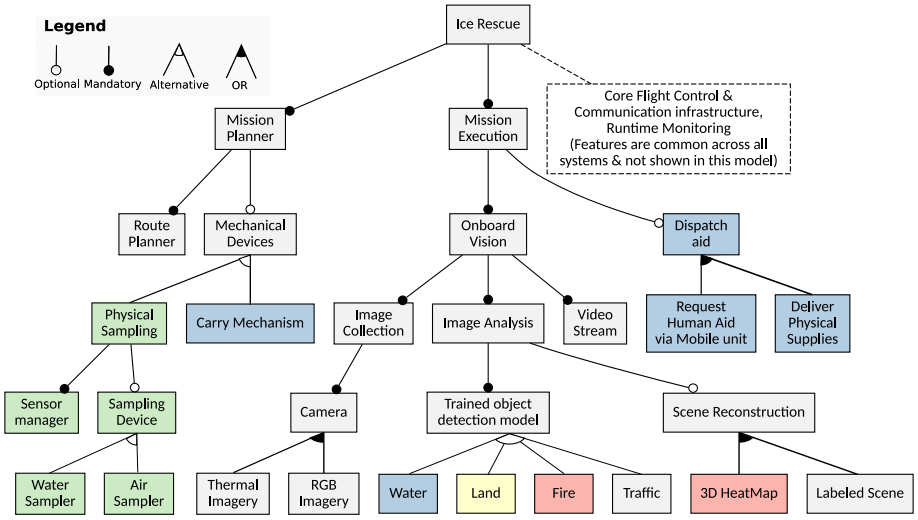


Fig. 3. This subset of the merged Feature Model includes features from river and ice rescue (blue), delivery (yellow), structural fires (red), environmental sampling (green), and traffic surveillance. Features used in multiple scenarios are shown in gray. Cross-tree constraints are not shown in this feature model. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

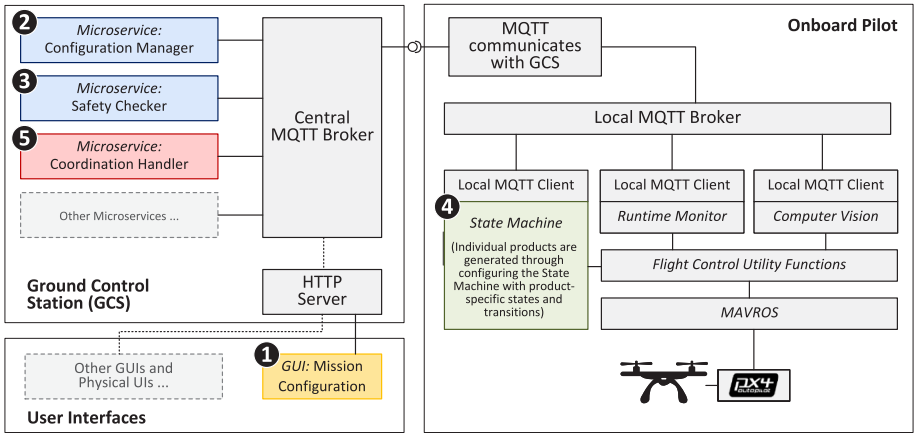


Fig. 4. *Drone Response* Architecture, emphasizing components that contribute to the Product Line Configuration. The user makes mission decisions in the GUI (1-yellow) and a mission specification is generated as a JSON file. The configuration manager (2-blue) generates individual mission specifications as well as coordination handlers (5-red). The Safety Checker (3-blue) checks them against a set of safety properties and allows or prohibits the mission. For supported missions, individualized specifications are configured in the sUAS' onboard state machine (4-green). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Onboard Autonomous Pilot: Much of the SPL configuration process occurs onboard the sUAS, where the *Onboard Autonomous Pilot* (OAP) is responsible for monitoring sensor data, supporting autonomy, and communicating with humans and other sUAS. The OAP interfaces directly with the sUAS Autopilot stack to leverage services of the flight control software and hardware. Onboard configuration occurs when the OAP's *State Machine* component as well as its supporting Computer Vision and monitoring services are configured according to the sUAS' mission specification.

3. Configuring Drone Response for diverse missions

Drone Response configures the OAP for each individual sUAS' intended role at the start of the mission and then reconfigures it dynamically as needed during mission execution. First, the operator utilizes the *Mission Planning UI* to plan the multi-sUAS mission (cf. Section 3.1), then, based on this configuration, *Drone Response* generates individual *mission specifications* for each sUAS (cf. Section 3.2) and checks the validity of task sequences within the mission specifications (cf. Section 4). Should any of the specification validity checks fail, an explanation is generated and the operator is advised to modify the mission plan. When all generated mission specifications pass their validity checks, the specifications are sent to each sUAS, which is then responsible for configuring its own internal state machine. Mission-specific microservices are also configured and activated, and finally, all sUAS are deployed for the mission. We now describe these configuration steps in greater detail.

3.1. Mission planning and specification

At the start of each mission, an operator (for example, the *Incident Commander*), uses *Drone Response*'s UI to plan a mission.

Missions are assembled from workflows composed of sequences of tasks, and workflows that are frequently reused across missions (e.g., search-and-detect, or surveillance), are referred to as 'roles'. *Drone Response* provides an interactive GUI, which allows users to compose tasks into new workflows. A subset of available tasks is shown in Listing 1. Workflows can be named and then used within a wizard to help users construct a mission

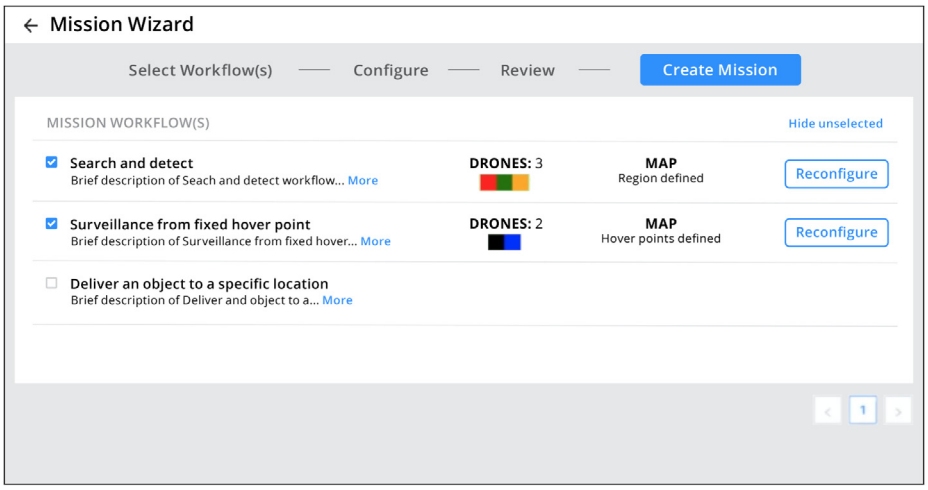


Fig. 5. *Drone Response*'s Mission Planner Wizard guides the operator through a series of steps part of the configuration process. Users can create and integrate entirely new roles, consisting of sequences of available tasks, into the planning process.

```
1  {
2  "states": [{
3    "name": "OnGroundAndActive",
4      "transitions": [{
5              "target": "MissionPreparation",
6            "condition": "MissionReceived"
7      }]
8    },{
9    "name": "FlyingToWayPoints",
10     "transitions": [{
11           "target": "Searching",
12           "condition": "DestinationReached"
13     },{
14           "target": "PositioningForDrop",
15           "condition": "DestinationReached"
16     },{
17           "target": "HoverAndStreamVideo",
18           "condition": "DestinationReached"
19     }],
20     "args:": [{
21       "ComputerVisionMode": [{
22              "option": "CV_ON"
23       }]
24     }]
25   }]
26 }
```

Listing 1: The mission-level library specifies all available mission states and transitions. Examples of 'OnGroundAndActive' and 'FlyingToWayPoints' are depicted.

plan. Mission plans tend to incorporate multiple workflows, each of which is ultimately assigned to one or more sUAS. This is illustrated in Figs. 5 and 6, which show two steps from the Mission Planning Wizard. In Fig. 5, the user selects roles and assigns them to specific sUAS, while in Fig. 6, the user has defined two hover points for a black and blue sUAS respectively. When the user has completed the mission plan, a mission specification is generated using the JSON structure shown for Listing 1, but pruned to reflect only selected roles and their associated tasks. In addition to configuring workflows, individual tasks can also be internally configured. For example, the FlyingToWayPoints task can be configured to have its camera *on* or *off* (cf. Listing 1, rows 20–25), and tasks can be configured with handlers that coordinate sUAS-to-sUAS or sUAS-to-Human communication. We describe these internal configurations in Section 3.3.

The mission specification generated by the wizard is forwarded to the *Configuration Manager* microservice (see component #2 in Fig. 4), responsible for parsing the specification and decomposing it into a set of individual sUAS' specifications and coordination handlers as described in Section 3.3. This set of sUAS specifications are then passed to the Safety Checker microservice (see component #3 in Fig. 4) which performs specification validity checks (cf. Section 4). If all specification validity checks pass, the Configuration Manager activates relevant microservices and sends the individual mission specifications to the OAP of each sUAS.

3.2. sUAS configuration

The sUAS configuration process starts when the OAP on the sUAS receives its individual mission specification. Each task specified in the workflow is mapped to one or more specific states, which in turn are mapped to specific features and capabilities implemented as modules in the sUAS' autopilot. For example, the 'Positioning for Drop' task that is part of the 'Item Delivery' workflow, maps to a concrete Python class entitled 'PositionDrone' which represents a state with clearly defined entry and exit criteria. The sUAS' OAP assumes responsibility for configuring its own onboard state machine. By default, the starting state is always *Mission Configuration*. The sUAS then configures the remainder of its state machine by instantiating relevant states and activating transitions as defined in its individual mission specification. These include the common states and transitions depicted in Fig. 2(a), and additional states and event-driven transitions relevant to the sUAS' specific mission tasks.

In addition to the states relevant for executing a mission and performing specific tasks, a number of additional *failsafe* states exist that are reachable from all other states. Transitions to these states are triggered by specific safety-related events such as *Low Battery*, *Geofence Breach*, or *Human Intervention* and include failsafe states such as RTL (Return to Launch), LAND (Land in Place), and Hover. Failsafe states and transitions do not need to be specified by the user, as they are automatically instantiated in all cases.

Once a mission specification has been successfully applied and the mission commences, the sUAS, at runtime, constantly monitors for events of interest that trigger transitions to new states. For example, many transitions are triggered when an sUAS completes a specific task (e.g., transitioning from *MissionPreparation* to *OnGroundAndInactive*), while others are triggered by changes in sensor data (e.g., vision-based detection of a victim or low battery), or by coordinated decisions made either by the Ground Control Station or collaboratively with other sUAS.

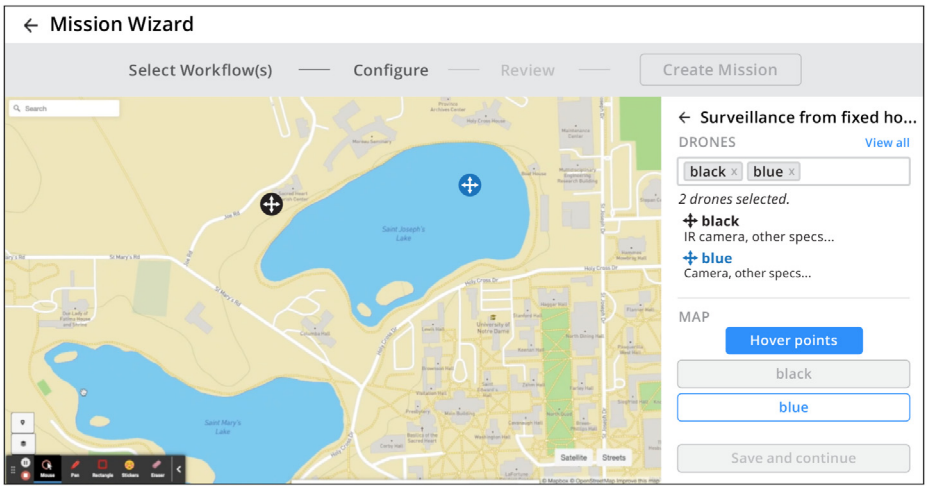


Fig. 6. Each role has a corresponding JSON specification which defines required configuration tasks. In this example, the user has assigned the surveillance role to two sUAS and is, therefore, guided to mark their individual surveillance points on the map. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

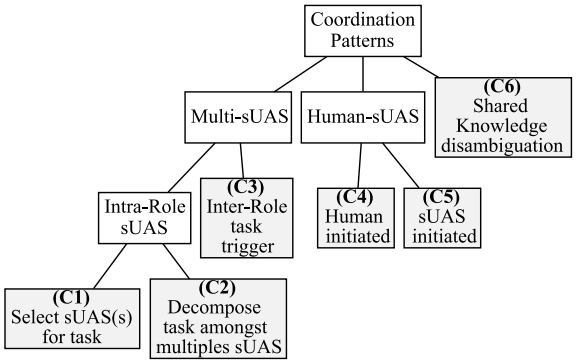


Fig. 7. sUAS-sUAS and sUAS-human coordination types.

3.3. Coordination handlers

While each sUAS has an assigned role during a mission and can perform tasks independently, individual sUAS must coordinate their efforts and work cooperatively with human operators to achieve their mission goals. This sUAS-sUAS or sUAS-human coordination is supported by dedicated *Coordination Handlers* associated with individual runtime states and invoked by a runtime monitor when trigger events are detected. Fig. 7 summarizes six coordination patterns that we identified by analyzing the original set of use cases and also based on own experiences in deploying multi-sUAS missions. Coordination is needed between sUAS assigned to the same role (C1, C2) and different roles (C3), between sUAS and humans (C4, C5), and finally through the exchange of data (C6). Each of these is discussed below.

C1 **Intra-Role: Select sUAS(s) for a task** from a set of available sUAS. An example is the selection of one, and only one, sUAS to track a detected victim (see Algorithm 1).

C2 **Intra-Role: Decompose a task amongst multiple sUAS.** An example is when a large search area is divided into smaller areas and assigned to individual sUAS. This is initially performed when the search area is defined, but could be repeated whenever search areas need to be reallocated. No coordination handler is needed for this task as it is handled by existing code for decomposing a search area into individual routes.

C3 **Inter-Role sUAS triggers:** An example is when an sUAS performing a search detects a victim in open water with a high degree of confidence, thereby triggering the launch of another sUAS charged with delivering a flotation device (see Algorithm 2).

C4 **Human initiated coordination with sUAS:** An example is when a human provides a directive to an sUAS to inspect a specific object during a search. No coordination handler is currently used to support this task. A human operator issues a command from the UI (e.g., by clicking on an sUAS and selecting a command) which is subsequently sent via MQTT to the drone.

C5 **sUAS initiated coordination with human:** An example is when an sUAS asks a human operator to confirm or refute a victim sighting (see Algorithm 3).

C6 **Disambiguating shared information** collected by multiple sUAS and potentially contributed to by humans. An example is determining whether multiple victim sightings by multiple sUAS are of the same, or different victims (see Algorithm 4).

C7 **Hybrid coordination** involving any combination of the above tasks. For example, a task could be decomposed into subtasks, and then a subset of available sUAS could be selected to perform those tasks.

With few exceptions (e.g., splitting a route into multiple parts), coordination handlers are implemented as microservices that instantiate one or more of the patterns. Each handler has a clearly defined trigger (or set of triggers) and is invoked by a runtime monitor which checks for the trigger events. By default, all handlers have access to the mission's knowledge base which, among other things, holds a current snapshot of all sUAS status (e.g., battery, location, and current task). Each handler includes the logic needed to perform its task (e.g., select, disambiguate), and authorization to issue commands that trigger state transitions to sUAS or to human agents via the UI. In certain cases, a more refined conditional check occurs based on detailed sUAS status information. For example, if Pattern 1 (cf. Fig. 7) were applied to selecting an sUAS to track a person, then prior to assigning a tracker, the conditional logic would check whether the person to be tracked has already been detected and tracked or not. Algorithms 1–4 provide listings of the coordination handlers for Patterns 1, 3, 5, and 6, while others are supported in custom ways as described above.

As part of our current implementation, each coordination handler is implemented as a Python program, managed by a microservice (cf. #5 in Fig. 4) responsible for managing all handlers.

Coordination Pattern 1: Select sUAS(s) for Task

```
Actors: Pool of sUAS capable of performing task T
Trigger: Specific sUAS state(s) and sensor values
Result: Zero to many sUAS selected for task T;
Assumption: Num. of available sUAS >= Num. of required
            sUAS;
Retrieve status from each candidate sUAS;
Perform coordination level condition checks;
if any condition checks fail then
    Send "Coordination aborted" message;
    Exit handler;
else
    for all candidates do
        Compute task fitness function;
    end
    Select Winner, notify, and assign task T;
    Notify Losers;
end
```

The handler communicates with multiple sUAS and indirectly with humans via the UI by publishing messages to the MQQT broker and subscribing to topics for which it expects to receive responses. For example, given Coordination Pattern 1, the selection of an sUAS for a specific task is performed by an algorithm implemented in the handler and supported by subsequent notifications to sUAS informing them whether they have been selected for the task or not.

Coordination Pattern 2: Inter-role sUAS Coordination

```
Actors: Two or more sUAS. At least one assigned to each
        role (R1,R2)
Trigger: Triggering event detected by sUAS in Role R1
Result: sUAS in Role R2 initiates its triggered task T;
if sUAS in Role R1 detects trigger event then
    Transmit "start T" command to sUAS in role R2;
end
R2 receives notification and initiates Task T
```

Coordination Pattern 3: sUAS Initiated Human Response

```
Trigger: Event with 'potential human interaction' occurs
Result: Human feedback elicited & potentially provided;
Condition: sUAS checks whether user input is needed
if human input needed then
    Request for human input with supporting information
     sent to UI;
    if human response received within waiting_period then
        Send human response to sUAS;
        sUAS follows human directive;
    else
        Send 'no response' notification to sUAS;
        'human failure to respond' event is logged;
        sUAS makes decision without human input;
    end
end
```

3.4. Extensibility and maintenance

Drone Response is designed with extensibility and maintenance in mind. Due to space constraints, we only briefly lay out the process for introducing new functionality. Entirely new features are created by (1) adding new types of features to the global list of available tasks (see Listing 1) so that they can be included in new workflows and roles, (2) developing (coding and unit testing) new modules in the onboard pilot that can be instantiated as states in the state machine, (3) creating a mapping from the feature name to the new states, so that the sUAS can be configured correctly when it receives a workflow specification including the new feature, and (4) specifying sequencing-related properties for the new feature. These properties and their role in dynamic configuration are described in the next section.

Coordination Pattern 4: Fact disambiguation performed by the runtime knowledge base. Note: This coordination handler is not yet integrated into DroneResponse as further work is needed to determine how to automatically disambiguate conflicting observations.

```
Trigger: sUAS submits a new observation to the knowledge
         base (KB)
Result: KB updates its current beliefs to reflect new data
Condition: KB checks if related facts exist
if related facts exist then
    Check is made whether facts conflict;
    if conflicting facts exist then
        Conflicting facts are reconciled;
    end
Update KB to reflect current belief based on received
 inputs;
```

Coordination Pattern 5: Concrete example for Selecting a tracker sUAS — implements Algorithm 1

```
Trigger: VictimDetected with confidence >= threshold
Result: Select sUAS to track victim if not already tracked;
Additional role: Search-And-Detect || QuickTracker
Additional role: OtherRole
Retrieve status on available tracker sUAS;
Determine if candidate victim is previously detected;
if not previously detected then
    for all available <tracker> sUAS do
        Compute suitability score for tracking;
    end
    Select sUAS with highest suitability;
    for all available <tracker> sUAS do
        if selected then
            Send 'Track' command to <tracker> sUAS
        else if sUAS detected the victim then
            Send 'Continue Search' command to
             <SearchAndDetect> sUAS
        else
            Do nothing
        end
    end
else
    Send 'Already Found' notification to sUAS
end
```

4. Validity checking of mission specifications

The flexibility of our approach allows users to compose diverse mission plans, which, as previously discussed, is important for emergency response scenarios. However, this, in turn, introduces the risk of potentially deploying missions with invalid, and even unsafe, workflows. It is therefore critical to ensure the validity of all task sequences prior to launch. We accomplish this by adopting a formal model-checking approach performed via a dedicated microservice, situated on our Ground Control Station, and

Table 2
Overview of the seven different types of properties that are considered for *Drone Response* safety checks.

	Type	Description	Example
Problem independent properties	P1: Determinism	Preventing multiple transitions from the same state under the same transition condition	The sUAS can transition to *Searching*, *PositioningForDrop*, or *HoverAndStreamVideo* upon the event *DestinationReached*. However, a concrete instance of the state machine must be deterministic and have one and only one possible outcome.
	P2:Absence of deadlocks	Preventing the case when an sUAS gets stuck in a state because there is no transition available from that state to any other state	While configuring the mission, the operator specifies a transition to a particular state but mistakenly does not add a transition out of that state. Such configurations cause deadlock situations and should not be pushed to the sUAS.
Temporal properties	P3: Reachability of key states	The sUAS mission must always have a valid path to reach certain critical states	A sUAS must always have a valid path to reach the *onGround* state from every other state.
	P4: Common temporal properties	Properties checked on every sUAS mission	*PreflightChecks* must be completed before *Takeoff*.
	P5: Mission-specific temporal properties	Mission-specific properties, applied at selected states/roles in the sUAS mission specification	In a delivery mission, *PositionForDrop* precedes *Dropping*.
Cross-role properties	P6: Absence of cross-role starvation	Preventing the case when a process blocks a shared resource, making it unavailable to other processes	An sUAS waiting for human confirmation to go to a next state but human operator is busy/unavailable. Considering human feedback to be a resource, this leads to a starvation situation which must be prevented.
	P7: Cross-role task coordination	Availability of a supporting sUAS with a specific role to complete the mission of the current sUAS	In a delivery mission, when one sUAS is assigned a search task, one other available sUAS must be assigned the role *delivery* with the appropriate hardware equipped.

named the "Specification Validator" (SV). Our approach includes the following steps:

S1 **Specifying sequencing properties:** During the design and development process, we identify and specify a global set of task sequencing properties that, if relevant to a specific mission, must hold in order for the mission specification to be valid. This means that if new features and corresponding tasks are introduced to the SPL, additional properties will need to be specified. We use Linear Temporal Logic (LTL) formulas to formally express a subset of these properties.

S2 **Filtering properties for mission context:** When the system is deployed, a mission specification is generated and subsequently decomposed into a set of individual sUAS specifications. The SV generates a deterministic finite-state automaton (DFA) for each of the sUAS specifications and their associated coordination mechanisms and identifies the subset of properties that are relevant for each DFA.

S3 **Validating Properties:** The SV then uses a model checker to check each DFA against its specified sequencing properties.

S4 **Fly/No-Fly Decision:** Finally, if all DFAs satisfy the sequencing properties, the SV approves the mission. If properties are violated and the mission is therefore not approved, we currently abort the mission. In the future, we will explore ways to recover from violations, for example, by recommending alternative task sequences or role assignments, or by raising warnings for missions with non-critical sequencing flaws without blocking their launch.

4.1. Defining sequencing properties

Several different types of properties are needed for specifying valid sequences of tasks. These are summarized as properties P1–P7 in Table 2 and include Problem Independent Properties related to determinism and general checking for the absence of deadlocks; Temporal Properties, for example, related to the correct ordering of tasks defined in a workflow; and Cross-Role Properties responsible for checking for starvation and cross-role task coordination. We provide examples of each of these.

P1 **Determinism:** The domain-level activity diagram describes all possible task transitions — in some cases showing different transition options for the same event. For example, when the event *destination reached* occurs in the task *FlyingToDestination*, the sUAS can either transition to *Searching*, *PositioningForDrop*, or *HoverAndStreamVideo* (cf. Fig. 2). However, the individual sUAS specifications must be deterministic, meaning that each state must have exit transitions with uniquely defined conditions to support runtime decision-making.

P2 **Absence of Deadlocks:** A missing or misdirected transition can lead to a deadlock which causes the sUAS to get stuck in a specific state with no transitions out of it. Therefore, absence of deadlocks is a crucial safety property.

P3 **Reachability of Key States:** All missions are expected to conclude in a safe end state — the primary example being that there must always exist a valid path to the *onGroundAndInactive* state.

P4 **Common Temporal Properties:** All sUAS flights go through a series of start-of-flight and end-of-flight tasks performed in a specific order. We specify a set of temporal properties to check that these key task sequences are preserved in all specifications. Examples include "*PreFlightChecks* precedes *TakeOff*", and "*OnGround* is a response of *TakeOffFailed*".

P5 **Mission-Specific Temporal Properties:** There are many role-specific sequences of tasks that must be performed in a specific order and so we also specify a set of temporal properties for these individual task sequences. For example, item delivery includes the two tasks of *PositionForDrop* and *Dropping* which must occur in that order. We, therefore, specify the temporal property that *PositionForDrop* precedes *Dropping*.

P6 **Absence of Cross-Role Starvation:** Starvation typically occurs when a required resource becomes unavailable for a long period of time. This occurs within sUAS-sUAS or sUAS-human coordination sequences, for example, when an sUAS is waiting for confirmation from a human, but the human fails to respond (see Patterns P4 and P5 in Fig. 7). From a practical perspective, we differentiate two cases of (1) *required* human intervention, in which the sUAS cannot proceed without human feedback (thereby accepting starvation), and (2) *requested* human intervention — which has built-in time-outs and default decision-making processes. Safety properties, therefore, have to ensure that "requested human intervention" states have alternate exit transitions.

Table 3
Examples of *Drone Response* temporal conditions expressed in both natural language and their corresponding LTL formulas.

Condition	Corresponding LTL formula
LaunchCommandReceived must precede *Armed* (*Precedence*)	$\Diamond Armed \Rightarrow (!Armed\ \mathcal{U}\ (LaunchCommandReceived\ \&\&\ !Armed))$
No *DeliveringDevice* until *PositionedForDrop* (*WeakUntil*)	$\Box !DeliveringDevice\ ||\ (!DeliveringDevice\ \mathcal{U}\ PositionedForDrop)$
If *TakeOffAborted* occurs, in response, *Landed* must occur *(Response)*	$\Box\ (TakeOffAborted \Rightarrow \Diamond Landed)$

P7 **Cross-Role Task Coordination:** When sUAS coordinate activities (cf. Patterns C1 and C2 in Section 3.3) across different roles, we need to check that specifications exist for both roles, and that the specifications include the expected tasks and transition conditions. For example, during a water-based search-and-rescue, when an sUAS is in the search role (S) and detects a victim, it raises a request for an sUAS in the delivery role (D) to takeoff and deliver a flotation device. The desired properties include (a) existence properties at the specification level (i.e., does at least one specification exist for role D in the current mission?), (b) existence properties at the task level (i.e., does the specification for role D include the task *DeliveringItem*, and (c) the existence of correct coordination-related events (i.e., in specification D, is the sUAS transition from *standby* to *takeoff* triggered by the message specified in the relevant coordination handler?

4.2. Specifying sequencing properties

After defining the sequencing properties, we formally express all the properties except the problem independent ones using Linear Time Temporal Logic (LTL) (Baier and Katoen, 2008; Belta et al., 2017; Huth and Ryan, 2000) defined as follows. Let *AP* be the finite set of atomic propositions. LTL formulas are constructed from these atomic propositions and contain Boolean operators, basic logic operators, and temporal operators. From the atomic propositions $\pi \in AP$, LTL formulas can be defined with the following grammar:

$$\varphi ::= \pi \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi \mathcal{U} \varphi$$

The Boolean operators *True* and *False* are defined as $True = \varphi \vee \neg\varphi$ and $False = \neg True$. There are other logical operators such as "conjunction" ($\wedge$), "negation" ($\neg$) and "disjunction" ($\vee$). Additionally, there are temporal operators such as "next" ($\bigcirc$) and "until" ($\mathcal{U}$). Using these operators, other temporal operators can also be derived such as "eventually" $\Diamond\varphi = True\ \mathcal{U}\ \varphi$, "always" $\Box\varphi = \neg\Diamond\neg\varphi$, "implication" $\varphi \Rightarrow \psi = \neg\varphi \vee \psi$, and "equivalence" $\varphi \Leftrightarrow \psi = (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$.

The semantics of an LTL formula φ are defined over infinite traces of atomic propositions $\pi \in AP$ called "words". A word $w = \sigma_0\sigma_1 \ldots$ satisfies φ when the first letter of that word, $(w, 0)$, satisfies φ. This can be written as $w, 0 \vDash \varphi$ or $w \vDash \varphi$. Similarly, $w, i \vDash \varphi$ means that the letter in the ith position of the word w satisfies φ, and formally defined as follows:

$$\begin{aligned}
&w, i \vDash \pi, && \text{iff } \pi \in AP \text{ and } \sigma_i \vDash \pi \\
&w, i \vDash -\varphi, && \text{iff } w, i \nvDash \varphi \\
&w, i \vDash \varphi_1 \wedge \varphi_2, && \text{iff } w, i \vDash \varphi_1 \text{ and } w, i \vDash \varphi_2 \\
&w, i \vDash \varphi_1 \vee \varphi_2, && \text{iff } w, i \vDash \varphi_1 \text{ or } w, i \vDash \varphi_2 \\
&w, i \vDash \bigcirc\varphi, && \text{iff } w, i+1 \vDash \varphi \\
&w, i \vDash \varphi_1 \mathcal{U} \varphi_2, && \text{iff } w, k \vDash \varphi_2 \text{ for some } i \leq k < \infty \text{ and} \\
& && w, j \vDash \varphi_1 \text{ for all } i \leq j < k
\end{aligned}$$

As described above, $\bigcirc\varphi$ intuitively means that φ is true in the "next" step in the sequence. Also, the formula $\varphi_1 \mathcal{U} \varphi_2$ intuitively means that φ_1 is true "until" φ_2 becomes true. Similarly, $\Box\varphi$ means that φ is always true and $\Diamond\varphi$ means that φ eventually becomes true.

Formulating long strings of LTL formulas representing complete functions would limit the scope and flexibility of the mission by only allowing the model checker to validate previously known mission specifications. This in turn would prevent users from validating new mission workflows. We therefore replace long LTL formulas that capture the entire mission, with multiple smaller LTL formulas defining specific temporal properties for small sets of related tasks. Furthermore, we intentionally eliminate the use of the *next*($\bigcirc$) operator as it binds one particular task to be performed immediately following another task, thereby over-constraining our mission specification by preventing both the insertion of new states between the two original ones and the repetition of states (Paun and Chechik, 2002). Rather, we rely on the Dwyer et al. temporal property pattern system (Dwyer et al., 1999) and its *"Precedence"* and *"Response"* patterns. Table 3 shows examples of LTL formulas with their corresponding natural language representations.

4.3. Filtering properties for mission context

Drone Response maintains a predefined list of all available states and transitions. After the mission is configured by the operator, *Drone Response* selects the relevant states and transitions from the predefined list based on the mission specification, and generates a DFA for each of the sUAS specifications.

The DFAs are defined using a formal language–Communicating Sequential Process (CSP) (Hoare, 1978)–and are subsequently processed by the Process Analysis Toolkit (PAT) (Sun et al., 2008, 2009). PAT is particularly well-suited for modeling and simulating concurrent and real-time systems, and can be used to model check safety properties of concern to our system. Further, it supports specification of multiple concurrent processes and their interactions, which is essential in our multi-sUAS environment. *Drone Response* also applies a set of default safety properties to all the sUAS missions. These include the problem independent properties, the reachability properties, and the common temporal properties described above. Finally, mission-specific safety properties are assigned to individual sUAS missions based on the selected states and coordination handlers associated with each specification. This is illustrated in Fig. 8.

4.4. Verifying properties

Different types of properties are checked in different ways. PAT provides a rich library of algorithms for checking diverse properties such as determinism, absence of deadlocks and reachability (Liu et al., 2010). Determinism and absence of deadlock properties are not expressed as LTL formulas, but are checked using the inbuilt features of PAT. The common and mission-specific temporal properties are specified using LTL and then checked directly by PAT. Reachability properties are checked by first defining Boolean variables which can be set to *True* or *False* if the system reaches a particular state, and then by establishing a system *goal* to the targeted (*True* or *False*) value. PAT is then used to determine whether the stated goal is achievable. We use this to verify all reachability properties. For example, in every individual sUAS mission, we declare a Boolean variable $isRTLMission = False$, which becomes *True* upon the event *ReturnHomeCoordinates*. PAT checks whether there is always a way to make this variable *True*. Similar techniques can be used to check cross-role properties such as task coordination and starvation.

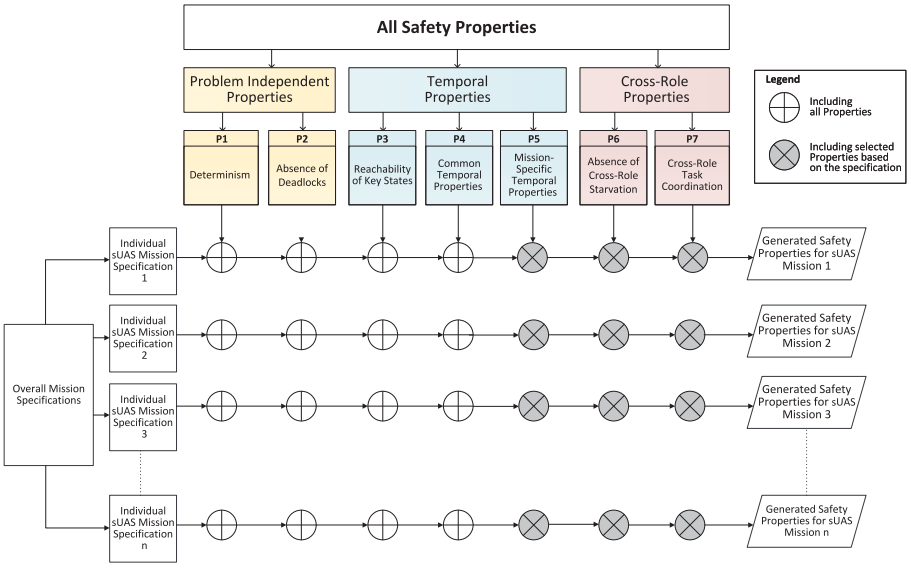


Fig. 8. Overview of the selection of safety properties for each individual sUAS mission.

5. Validation

We validated our dynamic approach to configuring the *Drone Response* SPL through a combination of testing techniques that included standard unit tests, integration tests, acceptance tests, and scalability tests.

All tests were executed in *Drone Response* (cf. Fig. 4) as follows. The UI components were implemented using the MEAN stack (Mongo-Express-Angular-Node) (IBM, 2022), with a NodeJS server responsible for coordinating multiple UI views, and providing access to the MongoDB NoSQL database which stores mission-specific JSON data. This includes, for example, mission configurations, flight routes, labeled search areas, and sUAS route assignments. This data was then used to generate prompts for the user guiding them through the configuration process. In our initial prototype, we supported mission plans that combined three different roles (delivery, surveillance, and search) (cf. Figs. 5 and 6).

We deployed Mosquitto, an open-source MQTT broker (Eclipse Foundation, 2022) on both the GCS and the Onboard Pilot for communication purposes. Microservices were implemented as independent Python modules containing MQTT clients. We utilized Redis (Redis, 2022), a distributed, in-memory database providing real-time data management capabilities, for creating the runtime knowledge base of the Ground Control Station. Each sUAS uses MQTT to share and exchange data with the GCS and the microservices. The Onboard Pilot's state machine uses the SMACH ROS package (Bohren, 2022) to implement its mission-specific finite state machine. The Onboard Pilot further uses YOLO (Redmon and Farhadi, 2018) to support computer vision and object detection, and MAVROS to facilitate communication over the MAVLink protocol (MAVLink, 2022), to send control commands from the Onboard Pilot component sUAS' autopilot. Finally, for simulation purposes, we used the Gazebo 11 (Gazebo, 2022) simulator, a high-fidelity, open-source, 3D, robotics simulator.

5.1. Research questions

We focused the validation around the following research questions:

RQ1: Can Drone Response dynamically configure diverse multi-sUAS missions when given a validated specification?

This research question is designed to assess whether *Drone Response* can take a mission specification generated from our Mission Planner Wizard, process it as described in this paper, configure the system accordingly, and ultimately execute the mission with multiple sUAS. We conducted this experiment in a high-fidelity simulator. However, as *Drone Response* supports both simulated and physical sUAS, demonstrating the configuration process in the simulator is a good proxy for deployment in real-world settings. In the near term, we will deploy and test on physical drones.

RQ2: Can the formal validity checker reject missions with invalid task sequences and approve valid ones for diverse mission specifications? With our second research question, we evaluate if our approach for checking the validity of task sequences generated from diverse mission specifications can reliably identify any sequencing problems.

RQ3: To what extent can the Specification Validity Checker scale to support the anticipated numbers of sUAS for missions that include potentially hundreds tasks and coordination handlers? This question primarily addresses the response time of checking specification validity given that sUAS may need to be deployed within seconds of activation.

5.2. RQ1: Ability to configure diverse multi-sUAS missions

Each part of the *Drone Response* system has been systematically tested through a series of unit, integration, and acceptance tests. These include 131 OAP unit tests, and seven full-system integration tests, each of which tests mandatory states plus three to four mission-specific states. An example of an integration test is available online.[3] Finally, we have three "ecosystem" tests, which each deploy different microservices as well as onboard autonomy. Low-level integration tests for validating software–hardware interfaces have been conducted in both the simulator and with physical sUAS.

At a higher level of integration testing, we focused on scenarios that incorporated the previously described roles of search, detect, and track (S), supply delivery (D), and birds-eye surveillance (V) in which an sUAS flew to a location, searched for victims, detected the victims and hovered in place. Our tests

[3] https://youtu.be/ILAAFOHBGoM.

Table 4
Test cases executed in the simulator. Scenario Key: S = Search-Detect-Track, D = Delivery, V = Surveillance where V1 = Step-based, V2 = Circular, and V3 = Fly-by. Coordination Handlers (C1, C2, C3, C5) are described in Fig. 7.

Test	Scenario					Coordination Patterns				Brief scenario description
	S	D	V1	V2	V3	C1	C2	C3	C5	
T1	●					●	●		●	Multiple search sUAS with HoTL
T2		●								Delivery only (single sUAS)
T3			●							Step-based survey only (single sUAS)
T4				●						Circular survey only (single sUAS)
T5					●					Fly-by survey only (single sUAS)
T6			●			●				Step-based survey only (multiple sUAS)
T7				●		●				Circular only (multiple sUAS)
T8					●	●				Fly-by only (multiple sUAS)
T9	●	●				●	●	●		Coordinated search and delivery
T10	●		●			●				Search & step-based survey, mult. search sUAS
T11	●			●		●				Search & circular survey, mult. search sUAS
T12	●				●	●				Search & fly-by survey, mult. search sUAS
T13		●	●							Basic delivery and step-based survey
T14		●		●						Basic delivery and circular survey
T15		●			●					Basic delivery and fly-by survey

covered different combinations of these three roles, with different numbers of sUAS, and with or without coordination handlers. The surveillance scenario included subvariants for (1) a step-based surveillance pattern that increased distance and altitude whilst maintaining pitch between the target and the drone, (2) circular flights around the object at varying altitudes and distances, and (3) fly-by surveillance with GPS locks moving from across a series of targets. We generated several unique combinations of these options including examples with single or multiple sUAS assigned to each role. Each test started with a valid mission plan from which a set of individual sUAS specifications were generated, sUAS configured accordingly, and appropriate coordination handlers instantiated and activated. The test was considered "passed" if the mission was executed correctly from start to finish.

We conducted a total of 15 integration tests as reported in Table 4. The table depicts the roles (S, D, V) where V1–V3 represent the three specific types of surveillance; and coordination handlers involved in each mission (C1, C2, C3, C5). For example, in the first test case, multiple sUAS were deployed to perform a search mission. Coordination involved decomposing the search area for distributions across collaborating sUAS, assigning the task, and initiating tasks.

Integration Test Results: After fixing some initial inconsistency problems in the JSON templates and the associated parsing routines in the microservices, all tests passed as specified. While these tests are not exhaustive and therefore do not cover every possible niche case, they include representative combinations of the three high-level variants (i.e., S, D, and V), cardinality variations (e.g., single or multiple sUAS in roles for which multiple instances are expected), and fine-grained variations of surveillance roles. They also include an example of integrating a coordination handler. These results, therefore, demonstrate the end-to-end viability of our approach for realistic missions derived from real-world use cases.

5.3. RQ2: Specification Validator (SV)

The second set of tests was designed to evaluate the efficacy of our validity checker using a simplified form of mutation testing (Jia and Harman, 2011). Mutation testing is typically used to detect vulnerabilities in code; however, Ammann describes it more broadly as the use of well-defined rules applied as systematic changes to software artifacts' syntactic structure (Ammann and Offutt, 2008). Small changes are applied against the software artifacts and an existing set of test cases (in our case, validity checks) are run in an iterative manner. Uncaught mutants serve as indications of insufficient or deficient test cases. For more traditional code-based mutation testing, mutants generally include statement deletion, insertion, or duplication, replacement of Boolean expressions (e.g., switch True/False), replacement of arithmetic operations or conditions, replacement of variables, or removal of code segments (e.g., method body).

However, our focus is on the correctness of the generated flight specifications and the coordination mechanisms between sUAS and humans. These experiments were therefore designed to evaluate whether the sequence validity checker was able to expose flaws in the specifications or in their interconnections established by their coordination handlers. We identified the following mutants that were applied directly to the JSON specifications.

- **M1** — Remove a transition between two tasks.
- **M2** — Add a transition between two previously unconnected tasks.
- **M3** — Reverse the direction of a transition so that a transition from Task $T1$ to $T2$ is replaced with a transition from Task $T2$ to $T1$.
- **M4** — Remove a task and all of its incoming and outgoing transitions.

We did not create mutants related to adding entirely new (invalid) tasks because workflows including illegal tasks are non-configurable in practice. Each and every task must be encoded as a feature in the onboard autonomous pilot, and if a task in the specification has no available mapping, then an error will be raised and the mission aborted. This type of error represents a bug rather than a user configuration error. It should be captured via validation tests rather than by the SV.

We also include two faults related to dynamic composition and coordination across sUAS instances by introducing the following mutants to the coordination handlers.

- **M5** — Remove all instances of a specific role (e.g., the mission plan does not include the "Item Delivery" role, even though it is expected by the coordination handler).
- **M6** — Change the number of instances of a specific role.

5.3.1. Testing process

The mutated versions of our specifications were generated using a test harness developed in Java to execute the following steps over ten iterations.

1. Starting with the three task sequences shown in Fig. 2 we generated mutated specifications and used a roulette wheel weighted according to the likelihood of each mutation type, in order to select ten sets of mutated specifications for each iteration. We illustrate how weights were computed with

an example taken from the simple task sequence for Item Delivery depicted in Fig. 2(d). In this case there are five different ways in which Mutant M1 (remove an edge) could be applied, but 20 ways in which M2 (add a transition) could be applied. Therefore, M2 is assigned a portion of the roulette wheel that is four times larger than M1's. Weights were computed based on the possibilities of applying each mutant (M1–M6) to the complete mission specifications depicted in Fig. 2.

2. Each individual test represented a single mission that included one mutated workflow specification and two non-mutated specifications. Mutants could produce either a valid or invalid workflow. Three team members, therefore, inspected each mutated workflow and tagged them as valid or invalid.
3. The SV was then executed against the set of three specifications for each test and resulting violations were reported. We compared these results against our manual answer set and categorized the results into four groups of true positives, false positives, true negatives, and false negatives. True positives meant that the checker raised a violation when a sequencing violation actually occurred, correctly preventing the deployment of sUAS with invalid sequences; whilst false positives meant that the checker incorrectly raised an alert, preventing a valid mission from being deployed. True negatives meant that valid missions would not be allowed to proceed; whilst false negatives represent the case in which an invalid, and potentially unsafe or ineffective mission plan, would have been deployed.
4. At the end of each test iteration, we inspected the results and updated the validity properties to improve the accuracy of the SV in order to address any false positives or false negatives.

Analysis of Results: Results of the mutation testing are reported in Table 5. No violations were raised for the non-mutated specifications. Over the course of the ten iterations, the safety checker correctly raised violations in 95 cases and incorrectly once. This incorrect case occurred due to the fact that the mutant had removed the first edge, and PAT was not able to parse the specification. In addition, the SV incorrectly accepted four invalid sequences due to missing safety properties. In each case, we updated the safety properties to ensure that the safety check would operate correctly in future runs. As a result of the modifications applied during the first four iterations, the final six iterations did not produce any incorrect results. We, therefore, conclude that the SV was able to detect invalid missions in the majority of cases and to correctly approve valid mission sequences.

However, developing and testing components for mission specific tasks, such as those depicted in Figs. 2(b), 2(d), and 2(c), requires significant time and effort, and our currently implemented tasks are therefore dominated by startup (e.g., preflight checks, taking off) and end-of-mission (e.g., land, disarming) actions — all of which have fairly rigid sequencing constraints. This explains why there are few *true negatives* (i.e., valid mutations) in our mutation tests. However, as discussed in Section 3.4, *Drone Response* is highly extensible, and therefore as new features are added, we will repeat the mutation tests as a means of validating that appropriate safety properties have been added to validate correct sequencing of new features.

5.4. Scalability

We also evaluated the SV's ability to scale up in support of larger and/or potentially more complex workflows. This is particularly important, as emergency response missions must often deploy quickly, requiring fast response times for the SV. The performance bottleneck is in the model checking, and so we focus on this aspect in our scalability analysis. Scaling factors include (SF1) the number of sUAS deployed on a mission, (SF2) the number of tasks per workflow, (SF3) the connectedness of tasks in a workflow, measured by the average number of edges per workflow, (SF4) the number of sequencing constraints per workflow, and (SF5) the number of coordination points across workflows. In practice we have found that SF3 and SF4 remain relatively stable, i.e., the number of edges and sequencing constraints increase linearly with the number of tasks, and that SF5 is low. We, therefore, take a typical baseline set of workflows from an existing integration test with 15 states, 20 transitions, and 26 relevant properties per workflow and scale up each factor.

We created a workflow generator that started with the baseline scenario (see row 1 in Table 6), and then used a random state generator to generate additional random states. We scaled up SF2 by $\times 2$, $\times 3$, $\times 4$, $\times 5$, and $\times 10$ using the random state generator, and measured the time taken to run the specification validity checks for each case. Each of these additional states was assigned four random transitions and eight pseudo properties. This is a practical upper bound on what we observed in real workflows. This caused SF3 and SF4 to have much steeper scale-ups than SF2. Experiments were performed on a Standard Desktop Machine with a 3.59 GHz CPU and 16 Gigabytes of memory. Results are reported in Table 6 and Fig. 9.

Based on discussions with stakeholders (e.g, Firefighters), sUAS used for emergency scenarios are ideally deployed within one minute from the point of activation; however, up to two minutes is tolerable. The sUAS needs 30 s to arm, leaving $\leq$30 s. for the ideal case (shown in green), $\geq$30 s. and $<$90 s. in the 'acceptable' case (yellow), and $\geq$90 s. for non-compliant case (orange). Furthermore, for local deployments, most missions will include from 4 to 5 sUAS.

These results indicate that for missions with typical numbers of sUAS, the SV can check the validity of the specifications within the ideal constraint of $\leq$30 s. For larger missions with 10–15 sUAS, SV can process missions with fewer states ($\leq$15) and transitions ($\leq$20) in $<$30 s, but the processing time rapidly increases as the complexity of workflows increases. Many solutions are possible for larger missions including advanced planning with more relaxed constraints, separation of sUAS into distinct validation groups separated by geolocation and/or collaboration groups. We leave this exploration to future work. Fig. 9 shows the logarithmic trend in model checking time for different configurations.

Table 5
Test Results from 10 Iterations of Mutation Testing the Mission Specifications. A True Pos means that a safety violation was detected in response to a mutant. All non-mutated specifications were returned as true negatives with no other outcomes.

Iter.	Mutated				Non-Mutated	
	True Pos.	False Pos.	True Neg.	False Neg.	True Neg.	Other
1	9	1	0	0	20	0
2	8	0	1	1	20	0
3	9	0	0	1	20	0
4	9	0	0	1	20	0
5	10	0	0	0	20	0
6	10	0	0	0	20	0
7	10	0	0	0	20	0
8	10	0	0	0	20	0
9	10	0	0	0	20	0
10	10	0	0	0	20	0

6. Threats to validity

As with any experiment, our evaluation is subject to several threats to validity (Wohlin et al., 2012) related to internal validity, external validity, and construct validity.

Table 6
Performance of the model checker with an increasing number of states (SF2), transitions (SF3), and constraints (SF4).

×	Attribute		3 sUAS		5 sUAS		7 sUAS		10 sUAS		15 sUAS	
	SF2	SF3	SF4	Secs	SF4	Secs	SF4	Secs	SF4	Secs	SF4	Secs
1	15	20	26	1.08	28	1.47	30	2.07	33	2.83	38	7.00
2	30	80	186	2.11	188	3.25	190	6.11	193	27.56	198	2 296.62
3	45	140	306	3.25	308	5.84	310	11.41	313	46.41	318	3 363.23
4	60	200	426	4.65	428	9.21	430	18.86	433	75.92	438	4 804.75
5	75	260	546	6.38	548	14.04	550	29.57	553	110.42	558	6 597.07
10	150	560	1146	20.67	1148	53.55	1150	115.20	1153	406.05	1158	21 007.45

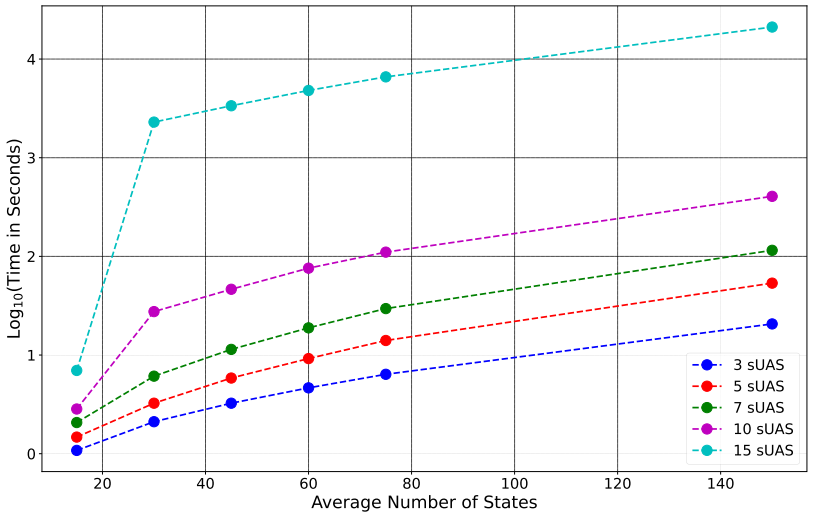


Fig. 9. Logarithmic trend in model checking time required for incrementing the scaling factors. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Internal validity is concerned with the rigor of the experimental design. As the work presented in this paper describes a technique for creating and dynamically configuring a product line, our validation relies to a large extent on demonstrating its feasibility and viability, and, therefore, our experiments are somewhat limited by the current use cases and the *Drone Response* SPL to which it has been applied. Nevertheless, we have shown the viability of our approach in a non-trivial, real-world product line which is currently the basis for a commercial product. We have additionally demonstrated that our sequencing validation checker was able to effectively detect invalid sequences. However, our experiments have so far focused only on the pre-launch configuration and have not included dynamic reconfigurations. *Drone Response* supports simple flight-time reconfigurations, and SV could be rerun prior to these reconfigurations. In future work, we plan to strengthen the evaluation through conducting user studies during field tests with physical sUAS on missions specified by the users, and by introducing additional configurable roles and tasks that can be configured.

External validity refers to how well data and theories from one setting apply to another, i.e., the generalizability of results and findings. While we anticipate that our approach would work in any multi-robotic setting in which task sequencing is a core part of the planning process, we have not yet evaluated it in other settings. Further, we established the specification validity checks, in the form of safety properties, for the tasks and roles that we have already created for *Drone Response*. Many additional tasks are possible, and additional challenges may emerge that we have not yet envisioned. We have so far specified only four coordination handlers; however, they represent diverse types of coordination, and our architecture and supporting processes have provided adequate support for each of these cases.

Construct validity refers to the extent to which a study measures what it claims to be measuring. Our experimental evaluation covers diverse forms of software engineering tests including unit tests, integration tests, scalability tests, and acceptance tests which are widely accepted in the domain of software engineering. While these tests validate the engineering contribution of our work, they also demonstrate the viability of the configuration process and the associated PAT validity checker.

7. Related work

As our approach combines PL concepts, multi-agent systems, and safety assurance, we discuss related work on use case and variability modeling, and Feature Model mappings. We also present related approaches in the domain of multi-agent systems, mission planning, configuration of semi-autonomous sUAS, and safety assurance.

Use Cases and Variability: Modeling commonalities and variabilities of a system, and subsequently deriving features based on use cases and/or requirements has been extensively studied in the past. For example, Halmans and Pohl (Halmans and Pohl, 2003; Pohl et al., 2005) proposed a requirements-driven method for explaining the variability of an SPL. Their method utilized Use Case Diagrams to describe the requirements and to further explain product variability. Bühne et al. (2006) presented a scenario-based procedure on an orthogonal variability model that supported engineers across all stages of the requirements engineering process for an SPL. Bragança and Machado (2005) focused on the architectural design of a PL, proposing an approach for deriving architectural components from requirements. Similar to our work their focus was on the decomposition of use cases and externalizing their variability. Bertolino et al. (2006) extended the standard use case notation to support product lines by

using constraints to document legal versus illegal combinations of variants; however, this approach leads to early formalization of the product line which we found detrimental to the task of exploring and documenting stakeholders' needs. The PLUSS approach (Eriksson et al., 2005) by Eriksson et al. employed Use Case Scenarios, as well as Use Case realizations; however, the focus was on a particular Use Case Model for the entire product line and scenarios were connected via shared features in the Feature Model. While this provided the definition of variants for use in specifying use cases, it did not provide adequate support for diverse forms of variability, such as algorithmic solutions or the selection of computer vision models for configuring specific vision-based tasks in an sUAS mission. In our work, we started with a textual elaboration of the use cases (Drone Response, 2022) (as discussed in Section 2), identified commonalities across use cases describing the different forms of emergency response summarized in Table 1, identified common steps from which features were derived, and then manually merged them into a variability model that served as the foundation for deriving and configuring a particular scenario at runtime.

Feature Model Mappings: Establishing mappings between Feature Models and other forms of artifacts has been the subject of extensive prior research. For example, Czarnecki and Antkiewicz (2005) proposed a template-based approach for mapping feature models to behavioral models or to data specifications. Furthermore, model-driven approaches have been used to automatically establish mappings. For example, Braganca and Machado (2007) created mappings between feature models and Use Case Diagrams, while Griss et al. (1998) integrated a reuse-driven Software Engineering process with FMs and then built a feature model based on requirements elicited from domain experts. However, while we used similar ideas in our approach, *Drone Response* further supports product derivation, incorporates a mission configurator, and provides support for runtime configuration. Regarding product configuration, Hajri et al. proposed PUMConf (Hajri et al., 2016, 2018), a tool-supported method using product line models to aid engineers in configuring products. PUMConf relies on sequentially taken configuration decisions and then automatically creates a use case and domain model based on the specified product. In our approach, we start from the actual use case descriptions created in collaboration with stakeholders, convert them into mission-specific FMs and activity diagrams and then merge them into PL models (Cleland-Huang et al., 2020).

Product Lines of Multi-Agent Systems: Product Line aspects have been introduced to the domain of multi-agent systems and Cyber–Physical Systems by a number of different authors. Cirilo et al. (2009), proposed the use of multi-level models to support automatic product derivation of multi-agent systems and configuration of product lines. They further presented an approach for modeling "Multi-agent System Product Lines (MAS-PLs)" (Nunes et al., 2009). Their approach was based on the PASSI (Process for Agent Societies Specification and Implementation) methodology that supports the specification of software agents. Similarly, Hahn (2008) presented a domain-specific language for modeling multi-agent systems and automatically generated code using model transformation techniques. While these approaches facilitate the specification and code generation of a multi-agent system, in our work, we focus on mission-level configurations as well as safety validation for these dynamically generated missions.

Dehlinger and Lutz (2005) presented an extensible agent-oriented requirements specification template for distributed systems. While their approach is sensitive to dynamic changes within the components of a system, our approach goes beyond this, and by using hierarchical state machines (supported by the SMACH state machine) we can define states and transitions that govern the systems' run-time behavior.

Mission Planning for Semi-autonomous sUAS:

Besada et al. (2019), proposed a management architecture to provide mission planning for drones, specifically focusing on real-time resource allocation. In this work, an application client creates missions via a "Mission Design Service" that supports a human operator in filtering available resources and suggesting optimal options. Similarly, Sampedro et al. (2016), presented a scalable and flexible architecture for real-time mission planning and dynamic agent-to-task assignment for UAV swarms. Their mission planning architecture consists of a "Global Mission Planner" that assigns and monitors different high-level missions and their constituent tasks via an "Agent Mission Planner". In contrast, our approach relies on a graphical user interface that facilitates the configuration of multi-sUAS missions and the resulting mission plans are validated and checked for safety before being deployed. Stecz and Gromada (2020), presented mission planning for a tactical short-range UAV, that determines the flight schedule and recognizes targets using different sensors. Task scheduling and route planning are formulated as a Mixed Integer Linear Problem (MILP). Wei et al. (2013), investigated the problem of dynamic mission planning for UAV swarms, proposing a centralized-distributed control framework for global mission assignment and scheduling.

Safety Assurance for sUAS: Safety assurance is a key aspect of semi-autonomous sUAS. Hägele and Söffker (2017) introduced a simplified real-time environmental situation risk assessment approach for determining the reliability of autonomous systems. They further describe emergency behavior control for autonomous systems' safe behavior assurance. McAree et al. (2016) discussed the development of a semi-autonomous inspection drone that can maintain a fixed distance and relative heading to a particular object, as well as a Model-Based Design (MBD) framework enabling any candidate control system to be tested in a high-fidelity simulation environment prior to any real-world flights. Tzelepi and Tefas (2021) proposed a human crowd detection method that uses deep Convolutional Neural Networks (deep CNN) for drone flight safety. They, however, did not use any state-diagram-based model checking for safety assurance, while in our work, we have ensured that the system is free from unexpected or unprecedented events using the PAT model checker. Safety assurance cases (Kelly and Weaver, 2004) are widely used in a variety of safety-critical domains including aviation and automotive systems. In the domain of sUAS, most notably, Denney and Pai have extensively investigated the use of safety cases for sUAS (Denney et al., 2012; Denney and Pai, 2012; Denney et al., 2017; Denney and Pai, 2013). While their work provides important building blocks and addresses sUAS safety on a general level, they do not provide support for rapidly and dynamically configuring mission-specific safety properties and validating them immediately prior to mission execution.

8. Conclusions

Rapid configuration and deployment of autonomous small Unmanned Aerial Systems is a key aspect in emergency response scenarios. In *Drone Response*, sUAS are assigned specific mission-related workflows dynamically prior to launch and potentially reassigned new workflows during the mission. Upon receiving the mission specification, each sUAS configures its onboard state machine which determines the tasks it can perform and the events or states that trigger transitions between tasks. This approach allows relatively generic sUAS to be configured dynamically to perform very diverse missions. However, this flexibility introduces the potential for previously unforeseen problems in the way workflows are configured. Therefore, to ensure that the workflows assigned to each sUAS, and to the set of sUAS as a whole, are free

from problems such as deadlocks and invalid task sequences, we perform runtime model checking of the dynamically generated mission specifications using the PAT model checker.

To validate our approach, we have conducted a series of end-to-end tests, demonstrating that the configuration process produces valid executable missions. We have applied fault-based testing to check that valid specifications are accepted and invalid ones rejected. The fault-based approach allowed us to detect initial problems and to update the safety properties accordingly.

Future work will primarily focus on increasing the variability of tasks, redesigning the mission planning UI to provide a more intuitive and supportive workflow for users configuring a mission plan, and conducting more extensive field tests and subsequent user studies with emergency responders and physical sUAS.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

The work described in this paper was partially funded by the US National Science Foundation Grant CNS:1931962 and CCF:1513730 and by the Linz Institute of Technology.

References

Agrawal, A., Abraham, S., Burger, B., Christine, C., Fraser, L., Hoeksema, J., Hwang, S., Travnik, E., Kumar, S., Scheirer, W., Cleland-Huang, J., Vierhauser, M., Bauer, R., Cox, S., 2020. The next generation of human-drone partnerships: Co-designing an emergency response system. In: Proc. of the 2020 Conference on Human Factors in Computing Systems.

Alotaibi, E.T., Alqefari, S.S., Koubaa, A., 2019. LSAR: Multi-UAV collaboration for search and rescue missions. IEEE Access 7, 55817–55832.

Alvear, O., Calafate, C.T., Hernández, E., Cano, J.-C., Manzoni, P., 2015. Mobile pollution data sensing using UAVs. In: Proc. of the 13th International Conference on Advances in Mobile Computing and Multimedia. pp. 393–397.

Ammann, P., Offutt, J., 2008. Introduction To Software Testing. Cambridge University Press, http://dx.doi.org/10.1017/CBO9780511809163.

Baier, C., Katoen, J.-P., 2008. Principles of model checking. 26202649.

Baresi, L., 2018. Activity diagrams. In: Encyclopedia of Database Systems, Second Edition.

Batory, D., Benavides, D., Ruiz-Cortes, A., 2006. Automated analysis of feature models: challenges ahead. Commun. ACM 49 (12), 45–47.

Belta, C., Yordanov, B., Gol, E.A., 2017. Formal Methods for Discrete-Time Dynamical Systems, Vol. 15. Springer.

Bertolino, A., Fantechi, A., Gnesi, S., Lami, G., 2006. Product line use cases: Scenario-based specification and testing of requirements. In: Käköla, T., Duenas, J.C. (Eds.), Software Product Lines. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 425–445.

Besada, J.A., Bernardos, A.M., Bergesio, L., Vaquero, D., Campaña, I., Casar, J.R., 2019. Drones-as-a-service: A management architecture to provide mission planning, resource brokerage and operation support for fleets of drones. In: Proc. of the 2019 IEEE International Conference on Pervasive Computing and Communications Workshops. IEEE, pp. 931–936.

Bohren, J., 2022. SMACH. [Last accessed 01-04-2022] http://wiki.ros.org/smach.

Bragança, A., Machado, R.J., 2005. Deriving software product line's architectural requirements from use cases: An experimental approach. In: Proc. of the 2nd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, Rennes, France.

Braganca, A., Machado, R.J., 2007. Automating mappings between use case diagrams and feature models for software product lines. In: Proc. of the 11th Int'L Software Product Line Conf.. IEEE, pp. 3–12.

Bühne, S., Halmans, G., Lauenroth, K., Pohl, K., 2006. Scenario-based application requirements engineering. In: Software Product Lines. Springer, pp. 161–194.

Chang, C.-C., Wang, J.-L., Chang, C.-Y., Liang, M., Lin, M.-R., 2015. Development of a multicopter-carried whole air sampling apparatus and its applications in environmental studies. Chemosphere 144, 484–492.

Cirilo, E., Nunes, I., Kulesza, U., Lucena, C., 2009. Automating the product derivation process of multi-agent systems product lines. In: Proc. of the 2009 XXIII Brazilian Symposium on Software Engineering. IEEE, pp. 12–21.

Claesson, A., Bäckman, A., Ringh, M., Svensson, L., Nordberg, P., Djärv, T., Hollenberg, J., 2017. Time to delivery of an automated external defibrillator using a drone for simulated out-of-hospital cardiac arrests vs emergency medical services. JAMA 317 (22), 2332–2334.

Cleland-Huang, J., Agrawal, A., Islam, M.N.A., Tsai, E., Speybroeck, M.V., Vierhauser, M., 2020. Requirements-driven configuration of emergency response missions with small Aerial Vehicles. In: Proc. of the 24th ACM Conference on Systems and Software Product Line: Volume a. ACM, pp. 26:1–26:12.

Cleland-Huang, J., Vierhauser, M., 2018. Discovering, analyzing, and managing safety stories in agile projects. In: Proc. of the 26th IEEE International Requirements Engineering Conference. pp. 262–273.

Cleland-Huang, J., Vierhauser, M., Bayley, S., 2018. Dronology: an incubator for cyber-physical systems research. In: Proc. of the 40th Int'L Conference on Software Engineering: New Ideas and Emerging Results. pp. 109–112.

Cockburn, A., 2000. Writing Effective Use Cases, first ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Czarnecki, K., Antkiewicz, M., 2005. Mapping features to models: A template approach based on superimposed variants. In: Proc. of the Int'L Conf. on Generative Programming and Component Engineering. Springer, pp. 422–437.

Dehlinger, J., Lutz, R.R., 2005. A product-line requirements approach to safe reuse in multi-agent systems. ACM SIGSOFT Softw. Eng. Notes 30 (4), 7.

Denney, E., Pai, G., 2012. A lightweight methodology for safety case assembly. In: Ortmeier, F., Daniel, P. (Eds.), Computer Safety, Reliability, and Security - 31st International Conference, SAFECOMP 2012, Magdeburg, Germany, September 25-28, 2012. Proceedings, Vol. 7612. In: Lecture Notes in Computer Science, Springer, pp. 1–12. http://dx.doi.org/10.1007/978-3-642-33678-2_1.

Denney, E., Pai, G., 2013. A formal basis for safety case patterns. In: Bitsch, F., Guiochet, J., Kaâniche, M. (Eds.), Computer Safety, Reliability, and Security - 32nd International Conference, SAFECOMP 2013, Toulouse, France, September 24-27, 2013. Proceedings, Vol. 8153. In: Lecture Notes in Computer Science, Springer, pp. 21–32. http://dx.doi.org/10.1007/978-3-642-40793-2_3.

Denney, E., Pai, G., Habli, I., 2012. Perspectives on software safety case development for unmanned aircraft. In: Swarz, R.S., Koopman, P., Cukier, M. (Eds.), IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, Boston, MA, USA, June 25-28, 2012. IEEE Computer Society, pp. 1–8. http://dx.doi.org/10.1109/DSN.2012.6263939, URL http://doi.ieeecomputersociety.org/10.1109/DSN.2012.6263939.

Denney, E., Pai, G., Whiteside, I., 2017. Modeling the safety architecture of UAS flight operations. In: Proc. of the International Conference on Computer Safety, Reliability, and Security. Springer, pp. 162–178.

Doherty, P., Rudol, P., 2007. A UAV search and rescue scenario with human body detection and geolocalization. In: Proc. of the Australasian Joint Conference on Artificial Intelligence. Springer, pp. 1–13.

Dragule, S., Gonzalo, S.G., Berger, T., Pelliccione, P., 2021. Languages for specifying missions of robotic applications. In: Software Engineering for Robotics. Springer, pp. 377–411.

Drone Response, 2022. Use cases. [Last accessed 01-01-2022] https://github.com/SAREC-Lab/sUAS-UseCases.

Dwyer, M.B., Avrunin, G.S., Corbett, J.C., 1999. Patterns in property specifications for finite-state verification. In: Proc. of the 21st International Conference on Software Engineering. Association for Computing Machinery, pp. 411–420.

Eclipse Foundation, 2022. Eclipse mosquitto – an open source MQTT broker. [Last accessed 01-01-2022] https://mosquitto.org.

Erdelj, M., Natalizio, E., Chowdhury, K.R., Akyildiz, I.F., 2017. Help from the sky: Leveraging UAVs for disaster management. IEEE Pervasive Comput. 16 (1), 24–32.

Eriksson, M., Börstler, J., Borg, K., 2005. The PLUSS approach–domain modeling with features, use cases and use case realizations. In: Proc. of the 9th International Software Product Line Conference. Springer, pp. 33–44.

Ezequiel, C.A.F., Cua, M., Libatique, N.C., Tangonan, G.L., Alampay, R., Labuguen, R.T., Favila, C.M., Honrado, J.L.E., Canos, V., Devaney, C., et al., 2014. UAV aerial imaging applications for post-disaster assessment, environmental management and infrastructure development. In: Proc. of the 2014 International Conference on Unmanned Aircraft Systems. IEEE, pp. 274–283.

Fantechi, A., Gnesi, S., John, I., Lami, G., Dörr, J., 2004. Elicitation of use cases for product lines. In: van der Linden, F.J. (Ed.), Software Product-Family Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 152–167.

Fleck, M., 2016. Usability of lightweight defibrillators for UAV delivery. In: Proc. of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems. pp. 3056–3061.

Gazebo, 2022. Open-source 3D robotics simulator. [Last accessed 01-01-2022] http://gazebosim.org.

Griffith, J.C., Wakeham, R.T., 2015. Unmanned aerial systems in the fire service: Concepts and issues. In: Proc. of the Aviation / Aeronautics / Aerospace International Research Conference.

Griss, M.L., Favaro, J., d'Alessandro, M., 1998. Integrating feature modeling with the RSEB. In: Proc. of the 5th Int'L Conf. on Software Reuse. IEEE, pp. 76–85.
Hägele, G., Söffker, D., 2017. A simplified situational environment risk and system reliability assessment for behavior assurance of autonomous and semi-autonomous aerial systems: A simulation study. In: Proc. of the 2017 Int'L Conf. on Unmanned Aircraft Systems. IEEE, pp. 951–960.
Hahn, C., 2008. A domain specific modeling language for multiagent systems. In: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1. pp. 233–240.
Hajri, I., Goknil, A., Briand, L.C., Stephany, T., 2016. PUMConf: a tool to configure product specific use case and domain models in a product line. In: Proc. of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 1008–1012.
Hajri, I., Goknil, A., Briand, L.C., Stephany, T., 2018. Configuring use case models in product families. Softw. Syst. Modeling 17 (3), 939–971.
Halmans, G., Pohl, K., 2003. Communicating the variability of a software-product family to customers. Softw. Syst. Modeling 2 (1), 15–36.
Hoare, C.A.R., 1978. Communicating sequential processes. Commun. ACM 21 (8), 666–677.
Huth, M., Ryan, M., 2000. Logic in computer science - modelling and reasoning about systems.
IBM, 2022. MEAN stack. [Last accessed 01-01-2022] https://www.ibm.com/cloud/learn/mean-stack-explained.
Jia, Y., Harman, M., 2011. An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 37 (5), 649–678.
Kelly, T., Weaver, R., 2004. The goal structuring notation–a safety argument notation. In: Proc. of the Dependable Systems and Networks 2004 Workshop on Assurance Cases. p. 6.
KETV NewsWatch 7, 2018. Council Bluffs firefighters use drones in ice water rescue training, KETC Newswatch 7. [Last accessed 01-04-2022] https://www.youtube.com/watch?v=p2MdbrTgmso.
Kim, H., Mokdad, L., Ben-Othman, J., 2018. Designing UAV surveillance frameworks for smart city and extensive ocean with differential perspectives. IEEE Commun. Mag. 56 (4), 98–104.
Koparan, C., Koc, A.B., Privette, C.V., Sawyer, C.B., Sharp, J.L., 2018. Evaluation of a UAV-assisted autonomous water sampling. Water 10 (5), 655.
Krüger, J., Nielebock, S., Krieter, S., Diedrich, C., Leich, T., Saake, G., Zug, S., Ortmeier, F., 2017. Beyond software product lines: Variability modeling in cyber-physical systems. In: Proc. of the 21st International Systems and Software Product Line Conference. pp. 237–241.
Lally, H., O'Connor, I., Jensen, O., Graham, C., 2019. Can drones be used to conduct water sampling in aquatic environments? A review. Sci. Total Environ. 670, 569–575.
Liu, Y., Sun, J., Dong, J.S., 2010. Developing model checkers using PAT. In: Bouajjani, A., Chin, W.-N. (Eds.), Automated Technology for Verification and Analysis. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 371–377.
Lv, Z., Chen, D., Feng, H., Zhu, H., Lv, H., 2021. Digital twins in unmanned aerial vehicles for rapid medical resource delivery in epidemics. In: IEEE Transactions on Intelligent Transportation Systems. IEEE, pp. 1–9.
MAVLink, 2022. UAV message protocol. [Last accessed 01-01-2022] https://mavlink.io.
McAree, O., Aitken, J.M., Veres, S.M., 2016. A model based design framework for safety verification of a semi-autonomous inspection drone. In: Proc. of the 11th International Conference on Control. IEEE, pp. 1–6.
Mesar, T., Lessig, A., King, D.R., 2019. Use of drone technology for delivery of medical supplies during prolonged field care. J. Spec. Oper. Med. : Peer Rev. J. SOF Med. Prof. 18 4, 34–35.
Molino, A., Brevi, D., Gavilanes, G., Scopigno, R., Sheikh, A., Bagalini, E., 2016. Using drones for automatic monitoring of vehicular accident. In: Proc. of the 2016 AEIT International Annual Conference.
Muccini, H., Moghaddam, M.T., 2018. IoT architectural styles. In: European Conference on Software Architecture. Springer, pp. 68–85.
Nunes, I., Kulesza, U., Nunes, C., Cirilo, E., Lucena, C., 2009. Extending PASSI to model multi-agent systems product lines. In: Proc. of the 2009 ACM Symposium on Applied Computing. pp. 729–730.
Ore, J.-P., Elbaum, S., Burgin, A., Detweiler, C., 2015. Autonomous aerial water sampling. J. Field Robotics 32 (8), 1095–1113.
Pádua, L., Sousa, J., Vanko, J., Hruška, J., Adão, T., Peres, E., Sousa, A., Sousa, J.J., 2020. Digital reconstitution of road traffic accidents: A flexible methodology relying on UAV surveying and complementary strategies to support multiple scenarios. Int. J. Environ. Res. Public Health 17 (6), 1868.
Paun, D.O., Chechik, M., 2002. On closure under stuttering. Form. Asp. Comput. 14, 342–368.
Pohl, K., Böckle, G., van der Linden, F., 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
Redis, 2022. Redis data structures server. [Last accessed 01-01-2022] https://redis.io.
Redmon, J., Farhadi, A., 2018. Yolov3: An incremental improvement. arXiv Preprint arXiv:1804.02767.
Rios, J., 2019. Firefighters practice using drones to assist ice rescues. [Last accessed 01-01-2022] https://www.broomfieldenterprise.com/2019/01/11/firefighters-practice-using-drones-to-assist-ice-rescues.
Ruiz-Jimenez, J., Zanca, N., Lan, H., Jussila, M., Hartonen, K., Riekkola, M.-L., 2019. Aerial drone as a carrier for miniaturized air sampling systems. In: Journal of Chromatography A, Vol. 1597.
Sampedro, C., Bavle, H., Sanchez-Lopez, J.L., Fernández, R.A.S., Rodríguez-Ramos, A., Molina, M., Campoy, P., 2016. A flexible and dynamic mission planning architecture for UAV swarm coordination. In: Proc. of the 2016 International Conference on Unmanned Aircraft Systems. IEEE, pp. 355–363.
Schörner, M., Wanninger, C., Hoffmann, A., Kosak, O., Reif, W., 2021. Architecture for emergency control of autonomous UAV ensembles. In: Proc. of the 3rd IEEE/ACM International Workshop on Robotics Software Engineering.
SEI, Software Engineering Institute, 2020. Software product lines. http://www.sei.cmu.edu/productlines.
Sherstjuk, V., Zharikova, M., Sokol, I., 2018. Forest fire-fighting monitoring system based on UAV team and remote sensing. In: Proc. of the 38th IEEE International Conference on Electronics and Nanotechnology. IEEE, pp. 663–668.
Silvagni, M., Tonoli, A., Zenerino, E., Chiaberge, M., 2017. Multipurpose UAV for search and rescue operations in mountain avalanche events. Geomat., Nat. Hazards Risk 8 (1), 18–33.
Stecz, W., Gromada, K., 2020. UAV mission planning with SAR application. Sensors 20 (4), 1080.
Sun, J., Liu, Y., Dong, J.S., 2008. Model checking CSP revisited: Introducing a process analysis toolkit. In: Proc. of the 3rd Int'L Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Vol. 17. Springer, pp. 307–322.
Sun, J., Liu, Y., Dong, J.S., Pang, J., 2009. PAT: Towards Flexible Verification under Fairness, Vol. 5643. Springer, pp. 709–714.
Tzelepi, M., Tefas, A., 2021. Graph embedded convolutional neural networks in human crowd detection for drone flight safety. IEEE Trans. Emerg. Top. Comput. Intell. 5 (2), 191–204. http://dx.doi.org/10.1109/TETCI.2019.2897815.
Vattapparamban, E., Güvenç, I., Yurekli, A.I., Akkaya, K., Uluağaç, S., 2016. Drones for smart cities: Issues in cybersecurity, privacy, and public safety. In: Proc. of the 2016 International Wireless Communications and Mobile Computing Conference. IEEE, pp. 216–221.
Vierhauser, M., Bayley, S., Wyngaard, J., Xiong, W., Cheng, J., Huseman, J., Lutz, R., Cleland-Huang, J., 2019. Interlocking safety cases for unmanned autonomous systems in shared airspaces. IEEE Trans. Softw. Eng. 47 (5), 899–918.
Vierhauser, M., Cleland-Huang, J., Bayley, S., Krismayer, T., Rabiser, R., Grünbacher, P., 2018. Monitoring CPS at runtime - A case study in the UAV domain. In: Proc. of the 44th Euromicro Conference on Software Engineering and Advanced Applications. pp. 73–80.
Wei, Y., Blake, M.B., Madey, G.R., 2013. An operation-time simulation framework for UAV swarm configuration and mission planning. Procedia Comput. Sci. 18, 1949–1958.
Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A., 2012. Experimentation in Software Engineering. Springer Publishing Company, Incorporated.
Zhi, S., Wei, Y., Yu, Z., 2017. Air quality monitoring platform based on remote unmanned aerial vehicle with wireless communication. In: Proc. of the International Conference on Future Networks and Distributed Systems. pp. 1–7.